Exceptional C++

VICTOR CIURA

Exceptional C++

CppCon 2021

October 25th

Abstract

When writing code we usually focus our attention on the happy paths - that’s where the interesting stuff happens. But there are also plenty of exciting things happening on the error handling flow, too. Although not universally loved/used, exceptions are a powerful mechanism of maneuvering execution on the unhappy path.

Even if std::exception and related machinery are not your cup of tea, you might care about hardware faults or OS signals like access violations, page errors, ALU overflows.

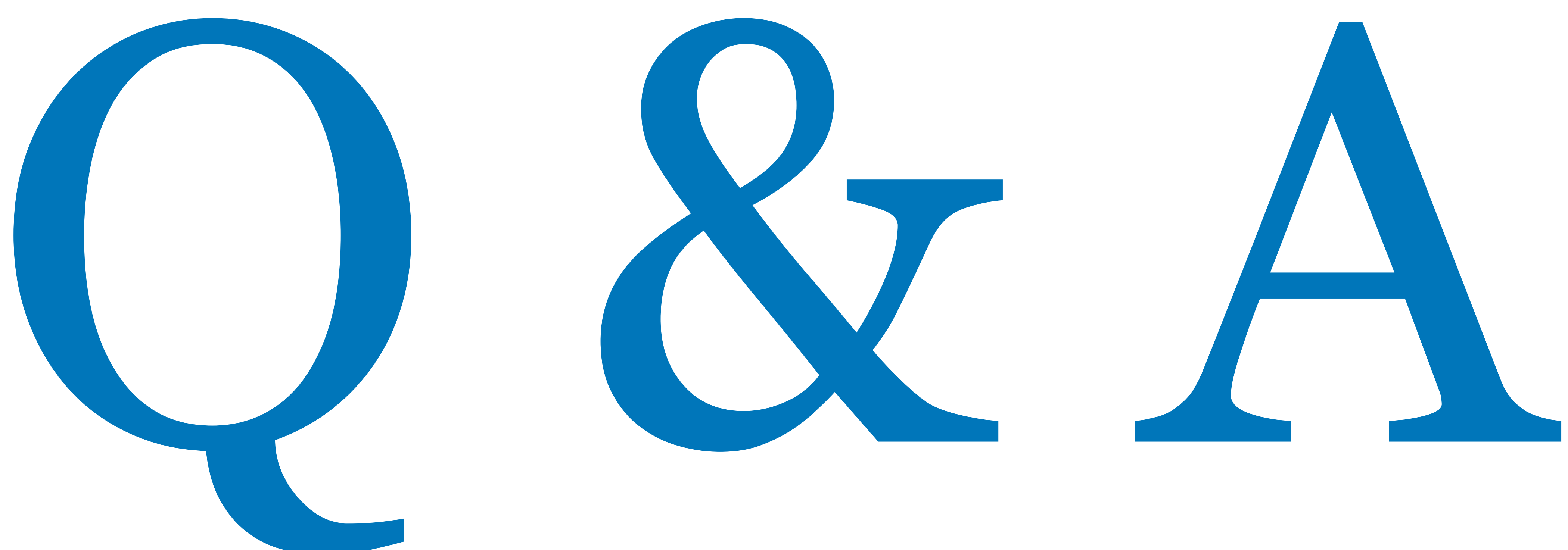
Let’s take a deep dive and explore what happens when an exception occurs, both at the application level and the OS level. We’ll explore the unwind process, the compiler generated code, the CRT hooks available and other exception internals. As we’re taking the scenic Windows route, we’re also going to encounter async exceptions (structured exceptions) on our quest for a better crash. We’ll poke into these mechanisms and see how we can leverage them in our application error handling. Did I mention threads? Routing exceptions between threads… oh my!

About me

Advanced Installer

Clang Power Tools

Use the Q&A tab in Zoom



When writing code we usually focus our attention on the happy paths - that's where the interesting stuff happens

But there are also plenty of exciting things happening on the error handling flow, too

Although not universally loved, exceptions are a powerful mechanism of maneuvering execution on the unhappy path

Motivation

This talk is **not** about:

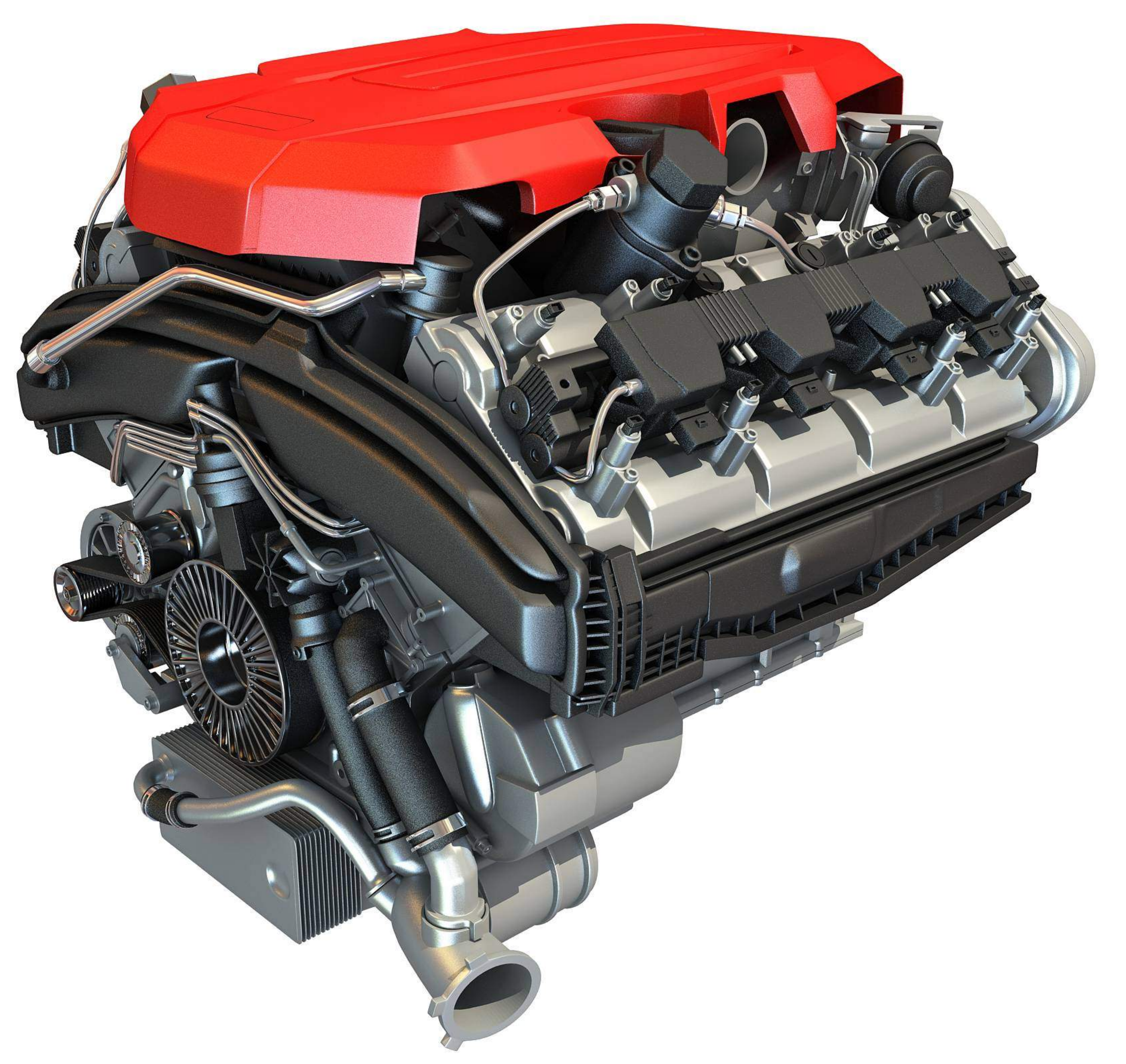
Exceptions vs Error codes vs Expected<T>

Exception safe code

Error handling best practices

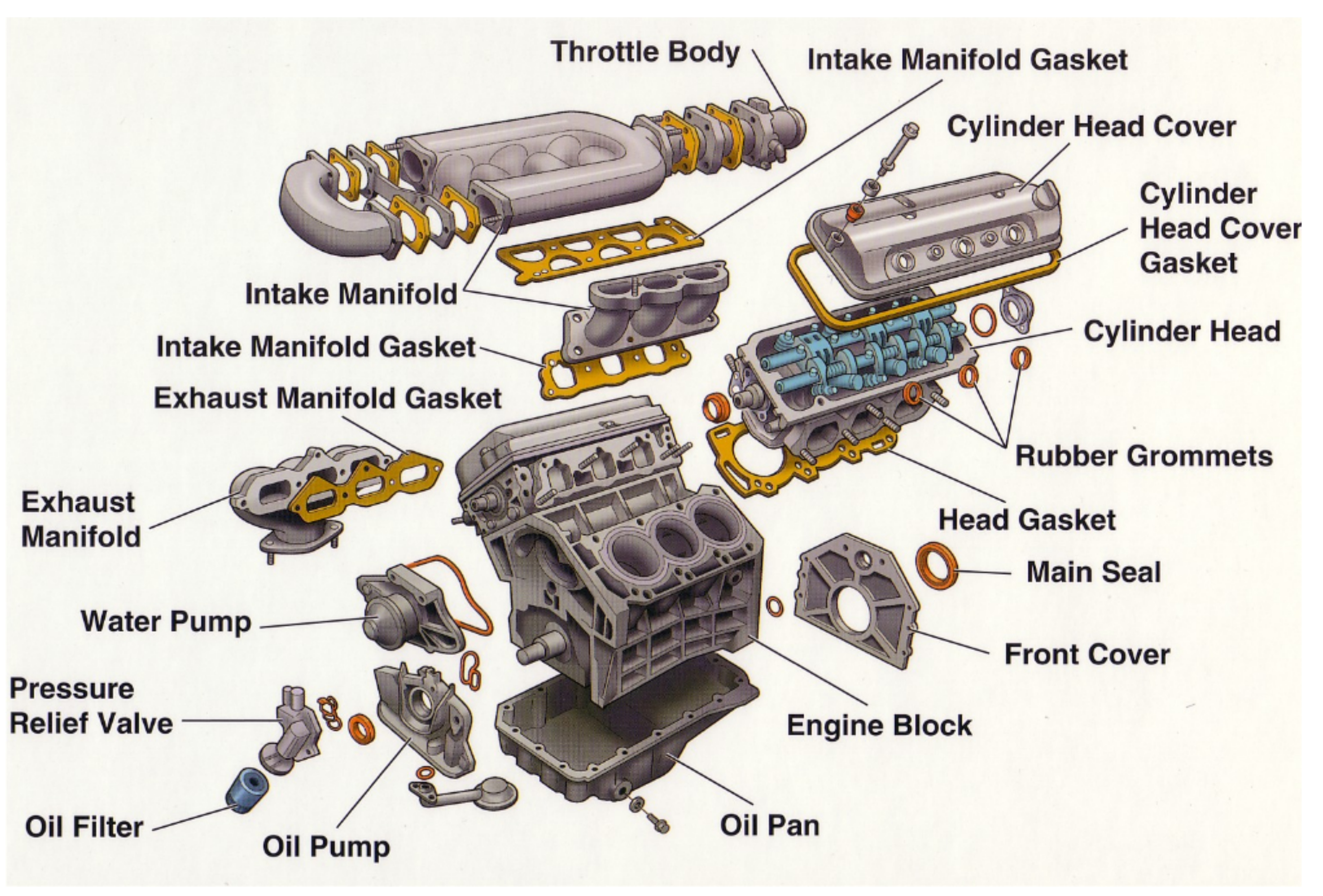
This talk is **not** about

This talk is about



To be honest...

What we actually care about



This talk is about

Windows

x86/x64

Life happens...

```
[SEH_AV_WRITE_NULLPTR] ACCESS_VIOLATION (0xc0000005) at address [0x000000014002772f]

Advanced Repackager (x64) 17.6 build c087f2e6
*** Stack Trace (x64) ***

[0x000000014002772f]  ProductDetailsPage::OnWizardNext() -> productdetailspage.cpp:97
[0x000000014002911c]  WTL::CPropertyPageImpl<ProductDetailsPage>::OnNotify() -> atldlgs.h:4527
[0x0000000140026f86]  ProductDetailsPage::_ProcessWindowMessage() -> productdetailspage.h:36
[0x0000000140026e68]  ProductDetailsPage::ProcessWindowMessage() -> productdetailspage.h:31
[0x0000000140020be8]  ATL::CDialogImplBaseT<WTL::CPropertyPageWindow>::DialogProc() -> atlwin.h:3862

...

[0x000000014003268e]  WTL::CPropertySheetImpl<RepackagerWizard>::OnCommand() -> atldlgs.h:4257
[0x00000001400312d4]  RepackagerWizard::ProcessWindowMessage() -> repackagerwizard.h:48
[0x00000001400338a3]  ATL::CWindowImplBaseT<WTL::CWizard97SheetWindow>::WindowProc() -> atlwin.h:3508
[0x000000014004176e]  Repackager::RunNormal() -> repackager.cpp:192
[0x00000001400429bb]  wWinMain() -> repackager.cpp:250
[0x0000000140089d02]  __tmainCRTStartup() -> crtexe.c:547
[0x0000000076a6652d]  BaseThreadInitThunk()
[0x000000007715c521]  RtlUserThreadStart()
[0x0000000000a00000]  MODULE_BASE_ADDRESS
```

Structured Exceptions

```
__try
{
  // stuff we hope works
}
__except( ExceptionFilter(GetExceptionInformation()) )
{
  // pretend it never happened
}
```

Structured Exception Handling (SEH)

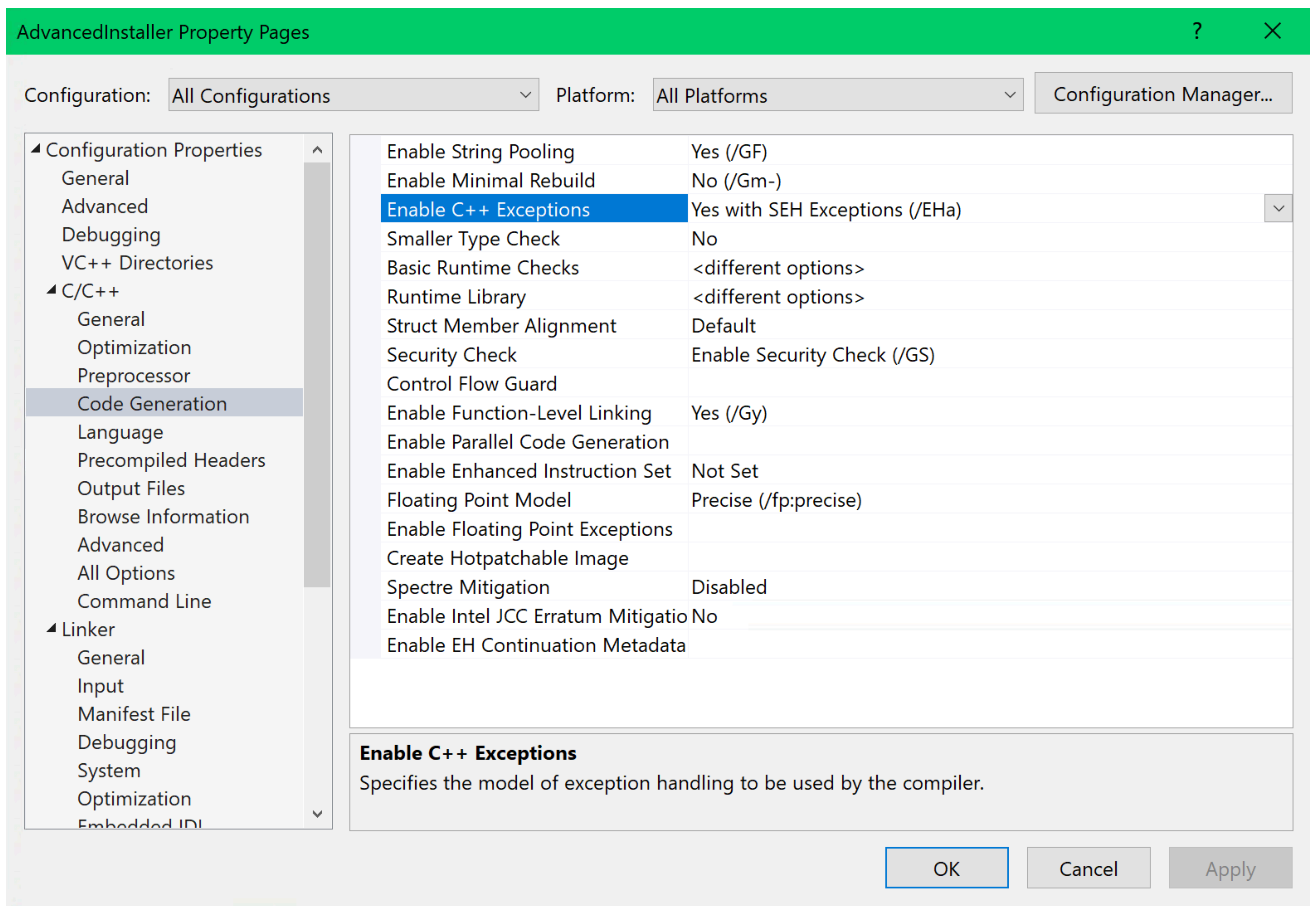
`/EHa`

we use *async* exceptions on all modules

docs.microsoft.com/en-us/windows/win32/debug/structured-exception-handling

docs.microsoft.com/en-us/cpp/cpp/structured-exception-handling-c-cpp?view=msvc-160

Structured Exception Handling (SEH)

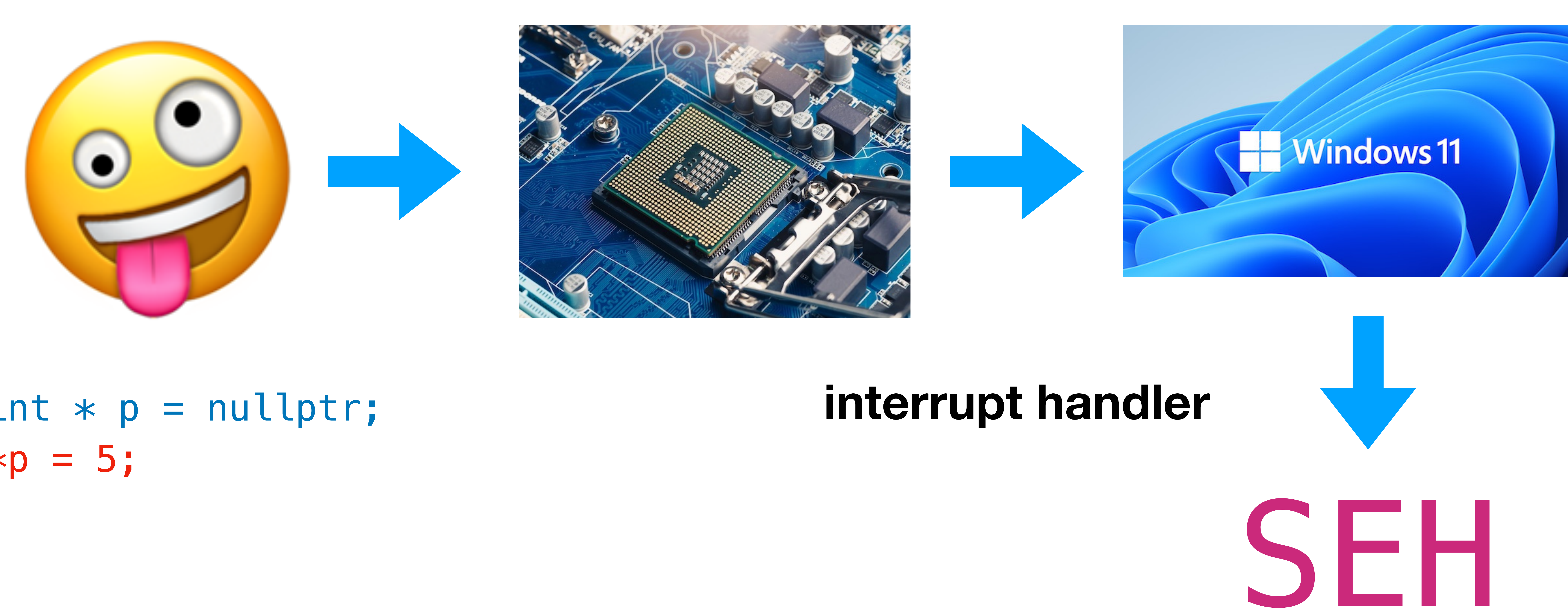


Structured Exception Handling (SEH)

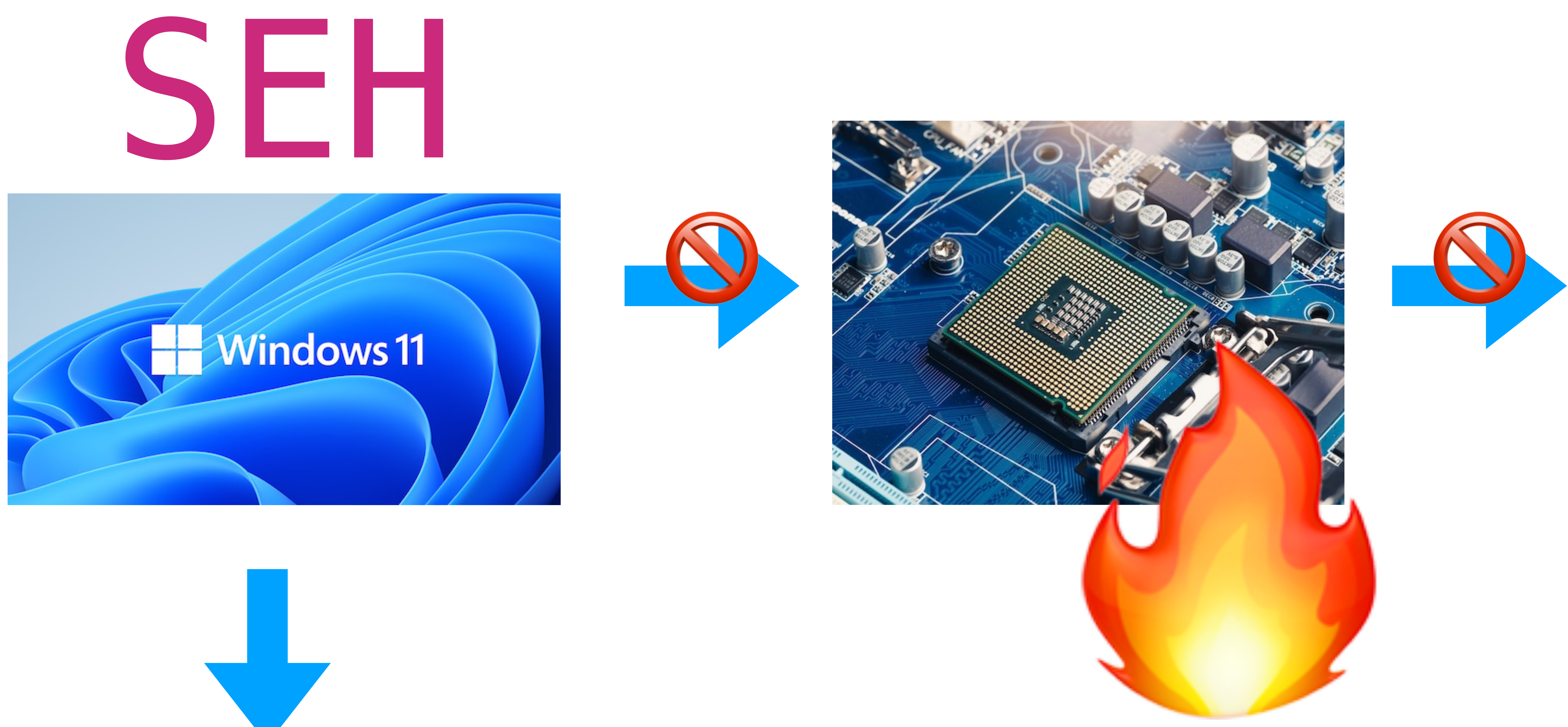
```
<ItemDefinitionGroup>
  <ClCompile>
    <DebugInformationFormat>ProgramDatabase</DebugInformationFormat>
    <ExceptionHandling>Async</ExceptionHandling>
  </ClCompile>
  <Link>
    <GenerateDebugInformation>DebugFull</GenerateDebugInformation>
    <SubSystem>Windows</SubSystem>
  </Link>
</ItemDefinitionGroup>
```

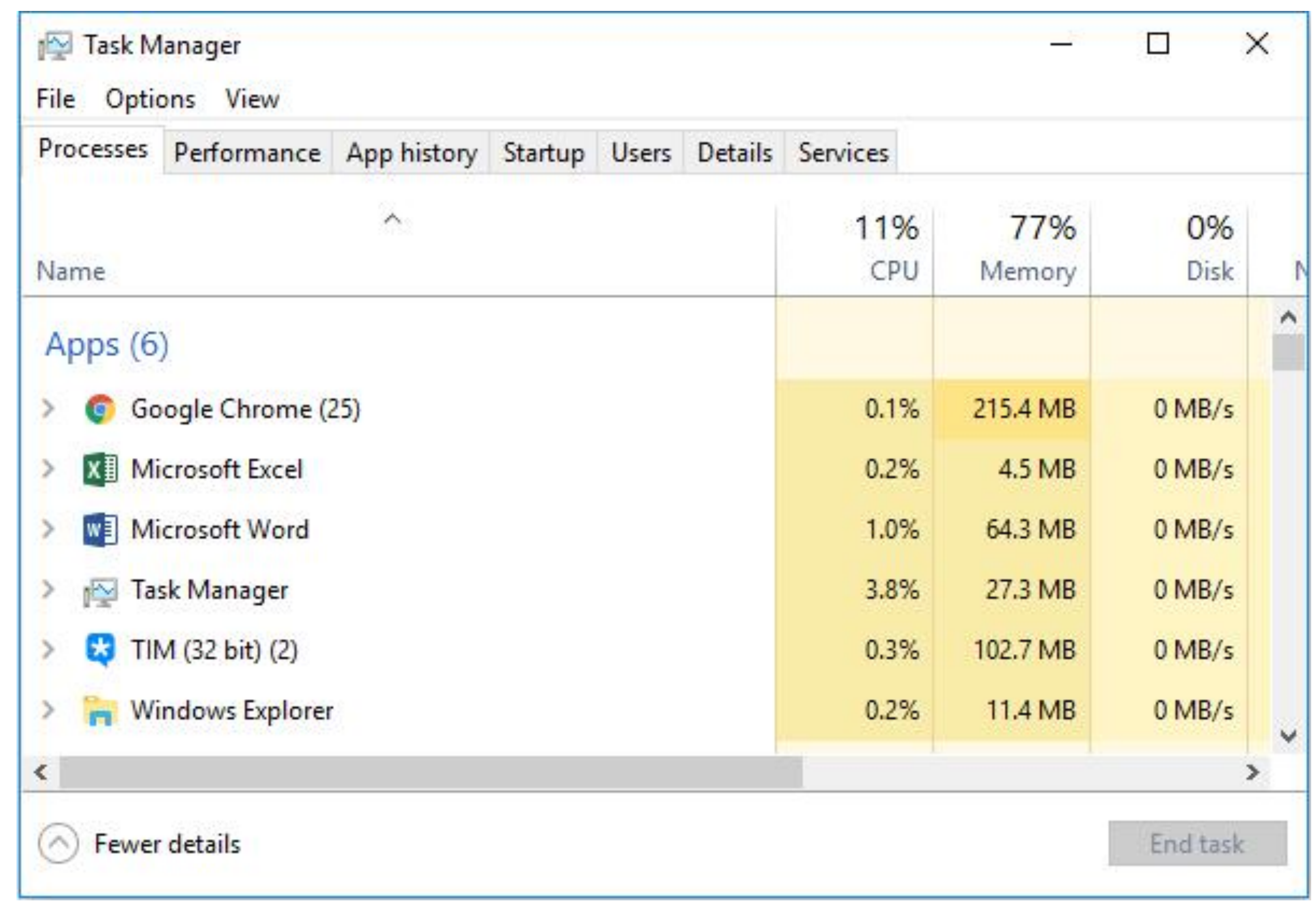
/EHa /DEBUG:FULL /Zi

SEH



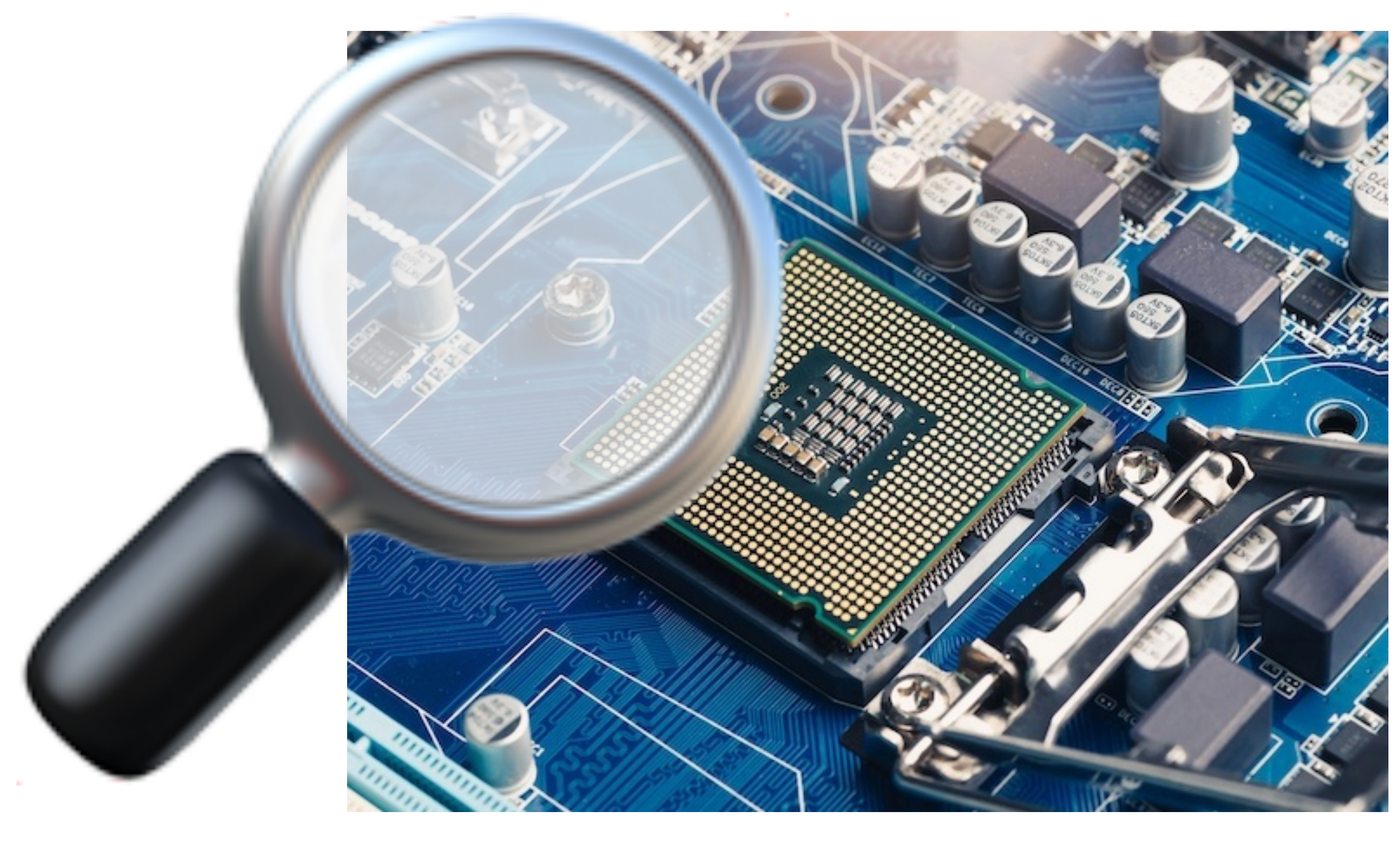
What's next?





React / Recover / Control

Snapshot - context



SEH

CONTEXT

```
struct CONTEXT {
    DWORD ContextFlags;

    // This section is specified/returned if CONTEXT_DEBUG_REGISTERS is
    // set in ContextFlags.  Note that CONTEXT_DEBUG_REGISTERS is NOT
    // included in CONTEXT_FULL.
    DWORD   Dr0;
    DWORD   Dr1;
    DWORD   Dr2;
    DWORD   Dr3;
    DWORD   Dr6;
    DWORD   Dr7;

    // This section is specified/returned if the
    // ContextFlags word contians the flag CONTEXT_FLOATING_POINT.
    FLOATING_SAVE_AREA FloatSave;

    // This section is specified/returned if the
    // ContextFlags word contians the flag CONTEXT_SEGMENTS.
    DWORD   SegGs;
    DWORD   SegFs;
    DWORD   SegEs;
    DWORD   SegDs;

    // This section is specified/returned if the
    // ContextFlags word contians the flag CONTEXT_INTEGER.
    DWORD   Edi;
    DWORD   Esi;
    DWORD   Ebx;
    DWORD   Edx;
    DWORD   Ecx;
    DWORD   Eax;

    // This section is specified/returned if the
    // ContextFlags word contians the flag CONTEXT_CONTROL.
    DWORD   Ebp;
    DWORD   Eip;
    DWORD   SegCs;
    DWORD   EFlags;
    DWORD   Esp;
    DWORD   SegSs;

    // This section is specified/returned if the ContextFlags word
    // contains the flag CONTEXT_EXTENDED_REGISTERS.
    // The format and contexts are processor specific
    BYTE    ExtendedRegisters[MAXIMUM_SUPPORTED_EXTENSION];
};
```

Contains processor-specific **register** data.

The system uses `CONTEXT` structures to perform various internal operations.

docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-context

CONTEXT

```
struct CONTEXT {
...
    DWORD   ContextFlags;

    DWORD   SegGs;
    DWORD   SegFs;
    DWORD   SegEs;
    DWORD   SegDs;

    DWORD   Edi;
    DWORD   Esi;
    DWORD   Ebx;
    DWORD   Edx;
    DWORD   Ecx;
    DWORD   Eax;

    DWORD   Ebp;
    DWORD   Eip;
    DWORD   SegCs;
    DWORD   EFlags;
    DWORD   Esp;
    DWORD   SegSs;

    BYTE    ExtendedRegisters[MAXIMUM_SUPPORTED_EXTENSION];
...
};
```

Contains processor-specific **register** data.

The system uses `CONTEXT` structures to perform various internal operations.

docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-context

EXCEPTION_RECORD

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

docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-exception_record

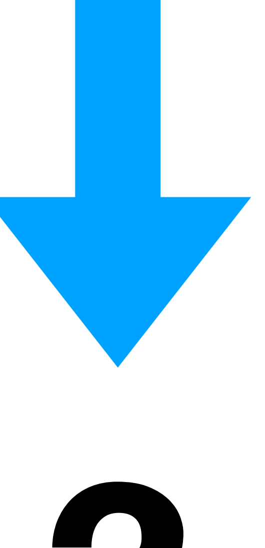
EXCEPTION_RECORD x86/x64

```
struct EXCEPTION_RECORD32 {
    DWORD   ExceptionCode;
    DWORD   ExceptionFlags;
    DWORD   ExceptionRecord;
    DWORD   ExceptionAddress;
    DWORD   NumberParameters;
    DWORD   ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
};


struct EXCEPTION_RECORD64 {
    DWORD   ExceptionCode;
    DWORD   ExceptionFlags;
    DWORD64 ExceptionRecord;
    DWORD64 ExceptionAddress;
    DWORD   NumberParameters;
    DWORD   __unusedAlignment;
    DWORD64 ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
};
```

docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-exception_record

EXCEPTION_RECORD

```
EXCEPTION_RECORD ex;

ex.ExceptionCode            = STATUS_ACCESS_VIOLATION; // 0xc0000005
ex.ExceptionFlags           = 0;
ex.ExceptionRecord          = nullptr; // next exception rec in the chain
ex.ExceptionAddress         = 0x05ED24C0; // WHERE: instruction address (PC)
ex.NumberParameters         = 2;
ex.ExceptionInformation[0] = EXCEPTION_WRITE_FAULT; // 0 = read; 1 = write; 8 = DEP
ex.ExceptionInformation[1] = 0x01ED6F42; // WHAT: the virtual address of the inaccessible data
```



NT_TIB

Win32 Thread Information Block (`TIB`)

a data structure that stores information about the currently running thread

It is accessed from:

- the `FS` segment register on 32-bit Windows => `FS:[18h]`
- the `GS` register on 64-bit Windows => `GS:[30h]`

```
NT_TIB * tib = (NT_TIB*)::NtCurrentTeb();
```

```
void * getTIB()
{
#ifdef _M_IX86
  return (void *)__readfsdword(0x18);
#elif _M_AMD64
  return (void *)__readgsqword(0x30);
#endif
}
```

NT_TIB

Win32 Thread Information Block (`TIB`)

The TIB contains the thread-specific exception handling chain
and pointer to the TLS (thread local storage)

```
FS:[0x00]
GS:[0x00]
```

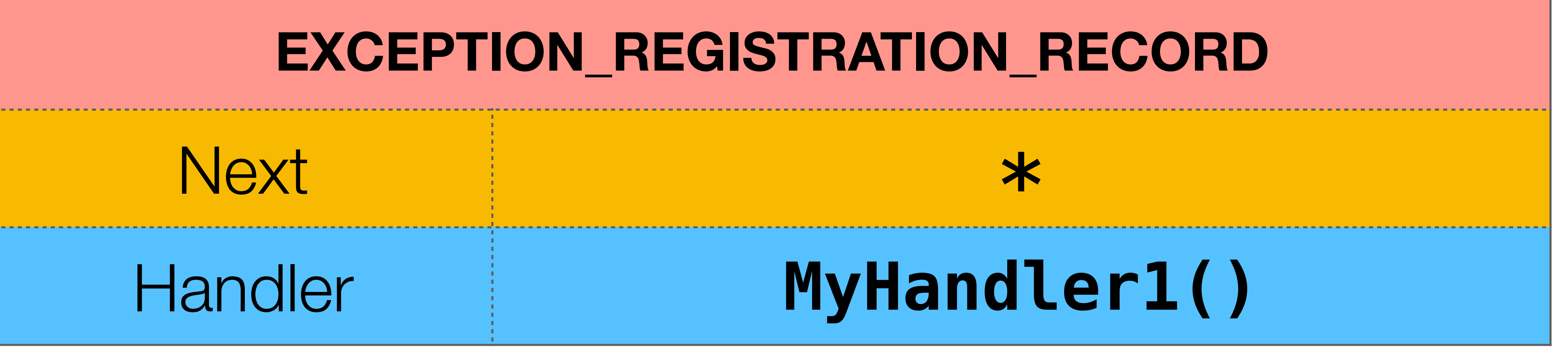
Current Structured Exception Handling (SEH) frame

* 64-bit Windows uses stack unwinding done in kernel mode instead

EXCEPTION_REGISTRATION_RECORD

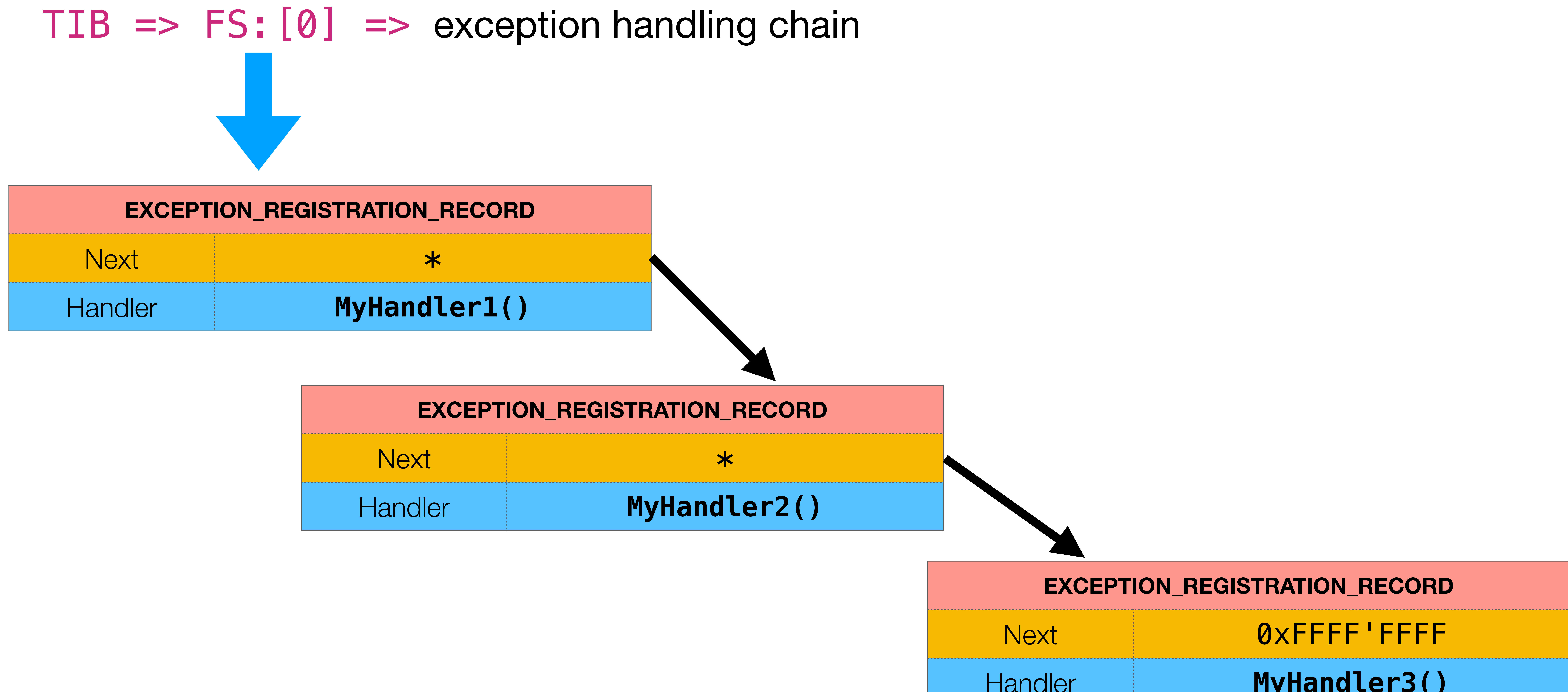
`TIB => FS:[0] =>` exception handling chain

```
struct EXCEPTION_REGISTRATION_RECORD
{
   PEXCEPTION_REGISTRATION_RECORD  Next;
   PEXCEPTION_DISPOSITION          Handler;
}
```



EXCEPTION_REGISTRATION_RECORD

`TIB => FS:[0] =>` exception handling chain



EXCEPTION_REGISTRATION_RECORD

If you don't register any handler and
a structured exception occurs => process crash

Simple Handler

```
void Func()
{
  NT_TIB * tib = (NT_TIB*)::NtCurrentTeb();

  EXCEPTION_REGISTRATION_RECORD reg;
  reg.Handler = &MyExceptionHandler;
  reg.Next    = tib->ExceptionList;

  tib->ExceptionList = &reg;

  int * p = nullptr;
  *p = 5;

  // restore original handler
  tib->ExceptionList = tib->ExceptionList->Next;
}
```

Simple Handler

```
EXCEPTION_DISPOSITION MyExceptionHandler(EXCEPTION_RECORD * aExRecord,
                                         void * aEstablisherFrame,
                                         CONTEXT * aContextRecord,
                                         void * aDispatcherContext)
{
  printf("Exception at address: [0x%p] - ExceptionCode = 0x%08x\n",
         aExRecord->ExceptionAddress,
         aExRecord->ExceptionCode);

  // we don't handle the exception
  return ExceptionContinueSearch;
}
```

Simple Handler

```
EXCEPTION_DISPOSITION MyExceptionHandler(EXCEPTION_RECORD * aExRecord,
                                         void * aEstablisherFrame,
                                         CONTEXT * aContextRecord,
                                         void * aDispatcherContext)
{
  printf("Exception at address: [0x%p] - ExceptionCode = 0x%08x\n",
         aExRecord->ExceptionAddress,
         aExRecord->ExceptionCode);

{ magic fix }

  // if we attempt to handle the exception,
  // stop searching for another handler func
  // and continue execution
  return ExceptionContinueExecution;
}
```

Simple Handler

```
EXCEPTION_DISPOSITION MyExceptionHandler(EXCEPTION_RECORD * aExRecord,
                                         void * aEstablisherFrame,
                                         CONTEXT * aContextRecord,
                                         void * aDispatcherContext)
{
  if (aExRecord->ExceptionCode == EXCEPTION_ACCESS_VIOLATION)
  {
    // the first element of the array contains a read-write flag
    // that indicates the type of operation that caused the access violation
    ULONG_PTR operationType = aExRecord->ExceptionInformation[0];

    // the second array element specifies the virtual address of the inaccessible data
    ULONG_PTR virtualAddress = aExRecord->ExceptionInformation[1];

    if (operationType == EXCEPTION_READ_FAULT)
      seType = virtualAddress ? SehException::SEH_AV_READ_BADPTR  : SehException::SEH_AV_READ_NULLPTR;
    else if (operationType == EXCEPTION_WRITE_FAULT)
      seType = virtualAddress ? SehException::SEH_AV_WRITE_BADPTR : SehException::SEH_AV_WRITE_NULLPTR;

    { handle/fix the issue }
  }

  // if we attempt to handle the exception, stop searching for another handler and continue execution
  return ExceptionContinueExecution;
}
```

Simple Handler

So much boilerplate for such a simple handler...

Simple Handler

```
void Func()
{
  __try
  {
    int * p = nullptr;
    *p = 5;
  }
  __except( ExceptionFilter( GetExceptionInformation() ) )
  {
    // pretend it never happened?
  }
}
```

EXCEPTION_CONTINUE_SEARCH   EXCEPTION_CONTINUE_EXECUTION

GetExceptionInformation()

Retrieves a computer-independent *description* of an exception, and information about the *computer state* that exists for the thread when the exception occurs.

This function can be called only from within the *filter expression* of an exception handler.

```
EXCEPTION_POINTERS * GetExceptionInformation();

DWORD GetExceptionCode();
```

docs.microsoft.com/en-us/windows/win32/debug/getexceptioninformation

docs.microsoft.com/en-us/windows/win32/debug/getexceptioncode

EXCEPTION_POINTERS

We've already seen this (CONTEXT)

```
struct EXCEPTION_POINTERS
{
  EXCEPTION_RECORD * ExceptionRecord;
  CONTEXT * ContextRecord;
};
```

docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-exception_pointers

Simple Handler

```
void Func()
{
  __try
  {
    int * p = nullptr;
    *p = 5;
  }
  __except( ExceptionFilter( GetExceptionInformation() ) )
  {
    // pretend it never happened?
  }
}
```

EXCEPTION_CONTINUE_SEARCH   EXCEPTION_CONTINUE_EXECUTION

Exception Filter

```
int ExceptionFilter( EXCEPTION_POINTERS * aPtrs )
{
  if (aPtrs->ExceptionRecord->ExceptionCode != STATUS_ACCESS_VIOLATION ||
      aPtrs->ExceptionRecord->ExceptionInformation[0] != EXCEPTION_WRITE_FAULT)
  {
    return EXCEPTION_CONTINUE_SEARCH;
  }

  void * writeAddress = aPtrs->ExceptionRecord->ExceptionInformation[1];

  // attempt a fix
  ::VirtualProtect(writeAddress, sizeof(int), PAGE_READWRITE);

  // resume execution from the faulting instruction
  return EXCEPTION_CONTINUE_EXECUTION;
}
```

Why are we bothering to understand all this ?

Why are we bothering to understand all this ?

Because C++ exceptions are built on top of the SEH machinery.

The compiler does all the bookkeeping work.

C++ Exceptions

What does the compiler map to:

C++	SEH
`throw`	`RaiseException()`
`try` `catch`	`__try` `__except`
Local variable destruction	`__try` `__finally`
`_CxxFrameHandler()`	`_except_handler3()`

C++ Exceptions

```
throw MyException{};          =>   MyException exObject{};

[[noreturn]]
void _CxxThrowException(void * aExObject, _ThrowInfo * aThrowInfo);

_CxxThrowException(&exObject, &_ThrowInfoFor<MyException>);
```

C++ Exceptions

```
struct _ThrowInfo
{
  unsigned int          attributes;
  Destructor          * pmfnUnwind;
  CatchableTypeArray * pCatchableTypeArray;
};

struct CatchableTypeArray
{
  int             nCatchableTypes;
  CatchableType * arrayOfCatchableTypes[nCatchableTypes];
};

struct CatchableType
{
  unsigned int      properties;
  std::type_info  * pType;
  PMD               thisDisplacement;
  int               sizeOrOffset;
  CopyConstructor * copyFunction;
};
```

_CxxThrowException()

```
[[noreturn]]
void _CxxThrowException(void * aExObject, _ThrowInfo * aThrowInfo)
{
  EXCEPTION_RECORD exception;

  exception.ExceptionCode  = EH_EXCEPTION_NUMBER; // 0xE06D7363 ('msc'|0xE0000000)
  exception.ExceptionFlags = EXCEPTION_NONCONTINUABLE;

  exception.NumberParameters = 3;
  exception.ExceptionInformation[0] = EH_MAGIC_NUMBER1;
  exception.ExceptionInformation[1] = (ULONG_PTR)aExObject;
  exception.ExceptionInformation[2] = (ULONG_PTR)aThrowInfo;

  ::RaiseException(exception.ExceptionCode,
                   exception.ExceptionFlags,
                   exception.NumberParameters,
                   exception.ExceptionInformation);
}
```

_CxxThrowException()

a C++ exception is born 🎉

Let's flip this around...

```
SEH  =>  C++ exceptions
```

Structured Exception Handling (SEH)

Handle C structured exceptions (Win32) as C++ typed exceptions:

```
_set_se_translator(ExceptionHandling::TransFunc);
```

docs.microsoft.com/en-us/cpp/c-runtime-library/reference/set-se-translator

Structured Exception Handling (SEH)

```
_set_se_translator(ExceptionHandling::TransFunc);
```

Each new thread needs to install its own translator function

=> each thread is in charge of its own translation handling

docs.microsoft.com/en-us/cpp/c-runtime-library/reference/set-se-translator

Structured Exception Handling (SEH)

```
class thread
{
public:
  using id = std::thread::id;

  thread() noexcept {}

  template <
    class FunType,
    class... ArgTypes,
    class = typename enable_if<!is_same<typename decay<FunType>::type, thread>::value>::type>
  explicit thread(FunType && aFun, ArgTypes &&... aArgs)
    : mTh(&ai::thread::shim_proc<typename std::decay<FunType>::type,
                                 typename std::decay<ArgTypes>::type...>,
          std::forward<FunType>(aFun),
          std::forward<ArgTypes>(aArgs)...)
  {
  }
```

Structured Exception Handling (SEH)

```
private:
  template <class FunType, class... ArgTypes>
  static decltype(auto) shim_proc(FunType && aFun, ArgTypes &&... aArgs)
  {
    // handle Win32 exceptions (C structured exceptions) as C++ typed exceptions
    ExceptionHandling::HandleSEH();

    return std::invoke(std::forward<FunType>(aFun), std::forward<ArgTypes>(aArgs)...);
  }

  std::thread mTh;
};
```

```
_set_se_translator(ExceptionHandling::TransFunc);
```

SEH Translator

```
void ExceptionHandling::TransFunc(unsigned int aSECode, EXCEPTION_POINTERS * aExInfo)
{
  // write the exception prolog (type, code, address, etc.)

  switch (aSECode)  // decode SEH exception type
  {
    case EXCEPTION_ACCESS_VIOLATION:
      swprintf_s(buf, MSG_BUFFER_LEN, L"%hs (0x%.8x) at address " ADDRESS_FORMAT SW_EOL,
                 "ACCESS_VIOLATION", EXCEPTION_ACCESS_VIOLATION,
                 aExInfo->ExceptionRecord->ExceptionAddress);
    break;
    case EXCEPTION_DATATYPE_MISALIGNMENT:
    break;
    case EXCEPTION_INT_DIVIDE_BY_ZERO:
    break;
    case EXCEPTION_INT_OVERFLOW:
    break;
    case EXCEPTION_ILLEGAL_INSTRUCTION:
    break;
    case EXCEPTION_STACK_OVERFLOW:
    break;
    ...
```

docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-exception_record

SEH Translator

```
void ExceptionHandling::TransFunc(unsigned int aSECode, EXCEPTION_POINTERS * aExInfo)
{
  ...
  SehException::SEType seType = SehException::SEH_GENERIC;

  // for AV exception, we can determine the type of operation that caused it
  if (aSECode == EXCEPTION_ACCESS_VIOLATION)
  {
    // the first element of the array contains a read-write flag
    // that indicates the type of operation that caused the access violation
    ULONG_PTR operationType = aExInfo->ExceptionRecord->ExceptionInformation[0];

    // the second array element specifies the virtual address of the inaccessible data
    ULONG_PTR virtualAddress = aExInfo->ExceptionRecord->ExceptionInformation[1];

    if (operationType == 0)
      seType = virtualAddress ? SehException::SEH_AV_READ_BADPTR  : SehException::SEH_AV_READ_NULLPTR;
    else if (operationType == 1)
      seType = virtualAddress ? SehException::SEH_AV_WRITE_BADPTR : SehException::SEH_AV_WRITE_NULLPTR;
    else if (operationType == 8)
      seType = virtualAddress ? SehException::SEH_AV_DEP_BADPTR   : SehException::SEH_AV_DEP_NULLPTR;
  }

  // record SEH type info in exception message
  exceptionMsg.insert(0, L"[" + SehException::SeTypeToString(seType) + L"] ");
```

SEH Translator

```
void ExceptionHandling::TransFunc(unsigned int aSECode, EXCEPTION_POINTERS * aExInfo)
{
  // write the exception prolog (type, code, address, etc.)
  // decode SEH exception type
  ...

  // walk the function call stack and gather information about each frame
  StackWalker::TraceFromContext(exceptionMsg, aExInfo->ContextRecord);

  // for AV exception, we can determine the type of operation that caused it
  ... => seType

  // extract SEH exception origin from StackTrace
  SymbolUtil::SrcPos exOrigin = GetExceptionOrigin(aExInfo->ContextRecord);

  // throw a C++ typed exception with the necessary fault information (attached)
  throw SehException(exOrigin.mFile, exOrigin.mLine, seType, exceptionMsg);
}
```

So we end up with a regular C++ exception wrapping the **SEH** info

What's the catch ?

What about an exception **in flight** ?

Get the stack trace for the raised exception on the *current thread*.

What's the catch ?

What about an exception **in flight** ?

Get the stack trace for the raised exception on the *current thread*.

```
wstring ExceptionHandling::GetStackTraceForCurrentException()
{
  wstring stackTrace;
  StackWalker::TraceFromContext(stackTrace, ExceptionHandling::GetCurrentExceptionContext());

  return stackTrace;
}
```

What's the catch ?

What about an exception **in flight** ?

Get the stack trace for the raised exception on the *current thread*.

```
wstring ExceptionHandling::GetStackTraceForCurrentException()
{
  wstring stackTrace;
  StackWalker::TraceFromContext(stackTrace, ExceptionHandling::GetCurrentExceptionContext());

  return stackTrace;
}
```

PCONTEXT

We've already seen this (PCONTEXT)

```
void ExceptionHandling::TransFunc(unsigned int aSECode, EXCEPTION_POINTERS * aExInfo)
{
  StackWalker::TraceFromContext(exceptionMsg, aExInfo->ContextRecord);
  ...
```

```
struct EXCEPTION_POINTERS
{
  PEXCEPTION_RECORD ExceptionRecord;
  PCONTEXT ContextRecord;
};
```

docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-exception_pointers

Where to start ?

So, how do we get this `PCONTEXT` ?

#if _MSC_VER >= 1900
(Visual Studio 2015-19)

So, how do we get this PCONTEXT ?

#if _MSC_VER >= 1900
(Visual Studio 2015-19)

```
PCONTEXT ExceptionHandling::GetCurrentExceptionContext()
{
  __vcrt_ptd * pTid = nullptr;

#ifdef _DLL  // Multi-Threaded DLL /MD or /MDd

  pTid = (__vcrt_ptd *)(((BYTE *)__current_exception_context())
                        - offsetof(__vcrt_ptd, _curcontext));

#else        // Multi-Threaded  /MT or /MTd

  pTid = __vcrt_getptd();

#endif

  return (CONTEXT *)pTid->_curcontext;
}
```

Where to start ?

So, how do we get this `PCONTEXT` ?

#if _MSC_VER < 1900
(Visual Studio 2013)

Where to start ?

So, how do we get this `PCONTEXT` ?

#if _MSC_VER < 1900
(Visual Studio 2013)

```
PCONTEXT ExceptionHandling::GetCurrentExceptionContext()
{
  _tiddata * pTid = nullptr;

#ifdef _DLL  // Multi-Threaded DLL /MD or /MDd

  pTid = (_tiddata *)(((BYTE *)__pxcptinfoptrs())
                     - offsetof(_tiddata, _tpxcptinfoptrs));

#else        // Multi-Threaded  /MT or /MTd

  pTid = _getptd();

#endif

  return (CONTEXT *)pTid->_curcontext;
}
```

CRT Power

```
#include <eh.h>

#include <signal.h>  // for use of API void ** __pxcptinfoptrs()

#if _MSC_VER >= 1900

  #include <../CRT/src/vcruntime/vcruntime_internal.h>

  extern "C" __vcrt_ptd * __cdecl __vcrt_getptd();
  extern "C" void ** __cdecl __current_exception_context();

#else

  // for use of (private) API _tiddata * _getptd()
  #include <../CRT/src/mtdll.h>

#endif
```

CRT Power

```
// per-thread data
struct __vcrt_ptd  // #include <../CRT/src/vcruntime/vcruntime_internal.h>
{
  // C++ Exception Handling (EH) state
  unsigned long      _NLG_dwCode;       // Required by NLG routines
  unexpected_handler _unexpected;       // unexpected() routine
  void *             _translator;       // S.E. translator
  void *             _purecall;         // called when pure virtual happens
  void *             _curexception;     // current exception
  void *             _curcontext;       // current exception context
  int                _ProcessingThrow;  // for uncaught_exception
  void *             _curexcspec;       // for handling exceptions thrown from std::unexpected
  int                _cxxReThrow;       // true if it's a rethrown C++ exception

#if defined _M_X64 || defined _M_ARM || defined _M_ARM64
  void *    _pExitContext;
  void *    _pUnwindContext;
  void *    _pFrameInfoChain;
  uintptr_t _ImageBase;
  uintptr_t _ThrowImageBase;
  void *    _pForeignException;
#elif defined _M_IX86
  void * _pFrameInfoChain;
#endif
};
```

PCONTEXT à-la carte

What if we want to get the current StackTrace from the *context of the caller* ?
(on demand - eg. assertions, logging)

When **no exception** is in flight!

How to get the **caller**'s `PCONTEXT` ?

Caller PCONTEXT ...

```
void StackWalker::TraceFromCaller(wstring & aStackMsg)
{
  using PF_RtlCaptureContext = void(WINAPI *)(PCONTEXT aContextRecord);

  // dynamically load the RtlCaptureContext() kernel API
  static auto CaptureCtx = (PF_RtlCaptureContext)::GetProcAddress(
      ::LoadLibraryA("Kernel32.dll"), "RtlCaptureContext");

  CONTEXT context;
  ::ZeroMemory(&context, sizeof(context));

  // retrieve the context record of the caller function
  CaptureCtx(&context);

  StackWalker::TraceFromContext(aStackMsg, &context);
}
```

PCONTEXT ...

So now we know how to get this `PCONTEXT`

How do we *walk the stack* ?

```
wstring stackTrace;
StackWalker::TraceFromContext(stackTrace, GetCurrentExceptionContext());
```

Walk the stack - init

```
void StackWalker::TraceFromContext(wstring & aStackMsg, PCONTEXT aContext, int MaxFrameCount)
{
  // All <DbgHelp> functions, such as StackWalk(), are single threaded.
  // (calls from more than one thread to this function will likely result
  //  in unexpected behavior or memory corruption)
  // => we must synchronize all concurrent calls to this function
  SyncGuard guard(sEHSyncSupport);

  // Copy the given machine CONTEXT structure because the StackWalk() API
  // might modify it and subsequent calls needing the CONTEXT will fail
  CONTEXT context;
  ::CopyMemory(&context, aContext, sizeof(context));

  HANDLE hProcess = ::GetCurrentProcess();
  HANDLE hThread  = ::GetCurrentThread();

  // create a symbol explorer
  SymbolUtil symMgr;
  if (!symMgr.Init(hProcess))
    return;

  ...
```

Walk the stack

```
void StackWalker::TraceFromContext(wstring & aStackMsg, PCONTEXT aContext, int MaxFrameCount)
{
  ...

  // initialize the STACKFRAME according to the platform we are working on (PE type)
  STACKFRAME sf;
  DWORD imageType = InitStackFrameFromContext(&sf, &context);

  for (int frmIndex = 0; frmIndex < MaxFrameCount; frmIndex++)
  {
    // get the current frame info
    BOOL result = ::StackWalk(imageType, hProcess, hThread, &sf, &context, nullptr,
                              SymFunctionTableAccess, SymGetModuleBase, nullptr);
    if (!result)
      break;

    aStackMsg += symMgr.ComposeStackFrame(sf.AddrPC.Offset);
  }

  // write the module load address - needed because of ASLR (Address Space Layout Randomization)
  aStackMsg += symMgr.ComposeModuleBaseAddress();
}
```

Walk the stack - frame setup

```
DWORD InitStackFrameFromContext(LPSTACKFRAME aStackFrame, PCONTEXT aContext)
{
  ::ZeroMemory(aStackFrame, sizeof(STACKFRAME));

#if defined _M_IX86

  DWORD imageType = IMAGE_FILE_MACHINE_I386;

  aStackFrame->AddrStack.Offset = aContext->Esp;
  aStackFrame->AddrStack.Mode   = AddrModeFlat;

  aStackFrame->AddrFrame.Offset = aContext->Ebp;
  aStackFrame->AddrFrame.Mode   = AddrModeFlat;

  aStackFrame->AddrPC.Offset    = aContext->Eip;
  aStackFrame->AddrPC.Mode      = AddrModeFlat;

#elif defined _M_X64

  .....

#endif

  return imageType;
}
```

Walk the stack - frame setup

```
DWORD InitStackFrameFromContext(LPSTACKFRAME aStackFrame, PCONTEXT aContext)
{
  ::ZeroMemory(aStackFrame, sizeof(STACKFRAME));

#if defined _M_IX86

  DWORD imageType = IMAGE_FILE_MACHINE_I386;

  aStackFrame->AddrStack.Offset = aContext->Esp;
  aStackFrame->AddrStack.Mode   = AddrModeFlat;

  aStackFrame->AddrFrame.Offset = aContext->Ebp;
  aStackFrame->AddrFrame.Mode   = AddrModeFlat;

  aStackFrame->AddrPC.Offset    = aContext->Eip;
  aStackFrame->AddrPC.Mode      = AddrModeFlat;

#elif defined _M_X64

  .....

#endif

  return imageType;
}
```

```
#elif defined _M_X64

  DWORD imageType = IMAGE_FILE_MACHINE_AMD64;

  aStackFrame->AddrStack.Offset = aContext->Rsp;
  aStackFrame->AddrStack.Mode   = AddrModeFlat;

  aStackFrame->AddrFrame.Offset = aContext->Rbp;
  aStackFrame->AddrFrame.Mode   = AddrModeFlat;

  aStackFrame->AddrPC.Offset    = aContext->Rip;
  aStackFrame->AddrPC.Mode      = AddrModeFlat;

#endif
```

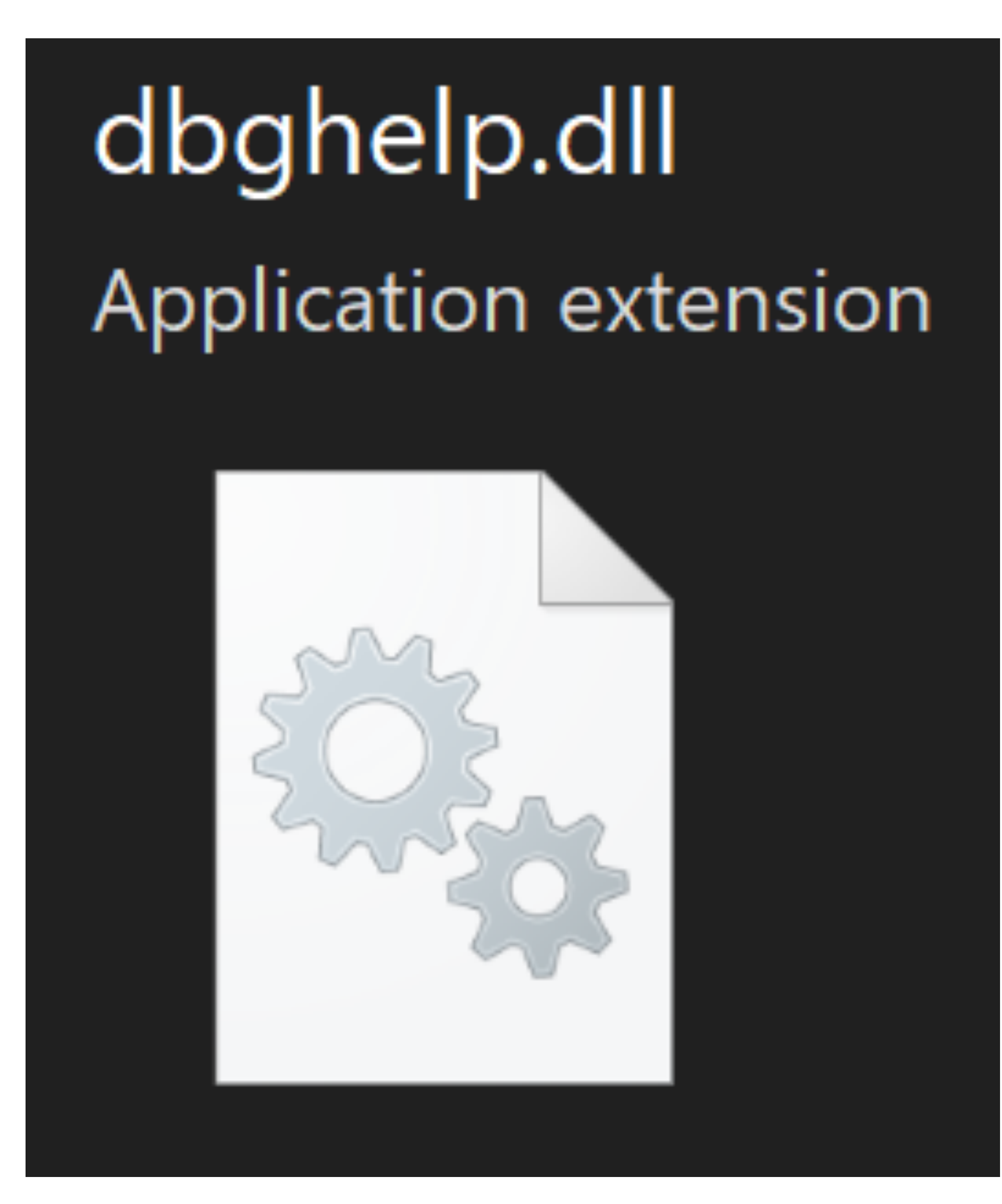
Walk the stack - SymbolUtil

```
wstring SymbolUtil::SymbolNameFromAddress(DWORD_PTR aAddress) const
{
  ...

  auto pSymbol = reinterpret_cast<PSYMBOL_INFO>(mSymMemBuffer);
  pSymbol->SizeOfStruct = sizeof(SYMBOL_INFO);

  // get symbol name (de-mangled function name)
  if (DynSymFromAddr(mProcess, aAddress, nullptr, pSymbol))
    return pSymbol->Name;
  else
    return SW_NO_SYMBOL;
}
```

```
PF_SymFromAddr DynSymFromAddr()
{
  static auto symProc = (PF_SymFromAddr)::GetProcAddress(
    ::LoadLibraryA("Dbghelp.dll"), "SymFromAddr");

  return symProc;
}
```

Walk the stack - SymbolUtil

```
wstring SymbolUtil::SymbolSourceFromAddress(DWORD_PTR aAddress) const
{
  DWORD displacement = 0;

  IMAGEHLP_LINE line;
  ::ZeroMemory(&line, sizeof(line));
  line.SizeOfStruct = sizeof(line);

  // get location information for symbol "sourceFile:lineNo"
  if (::SymGetLineFromAddr(mProcess, aAddress, &displacement, &line))
  {
    wchar_t origin[MAX_PATH];
    swprintf_s(origin, MAX_PATH, L"%s:%ld", line.FileName, line.LineNumber);
    return origin;
  }
}
```



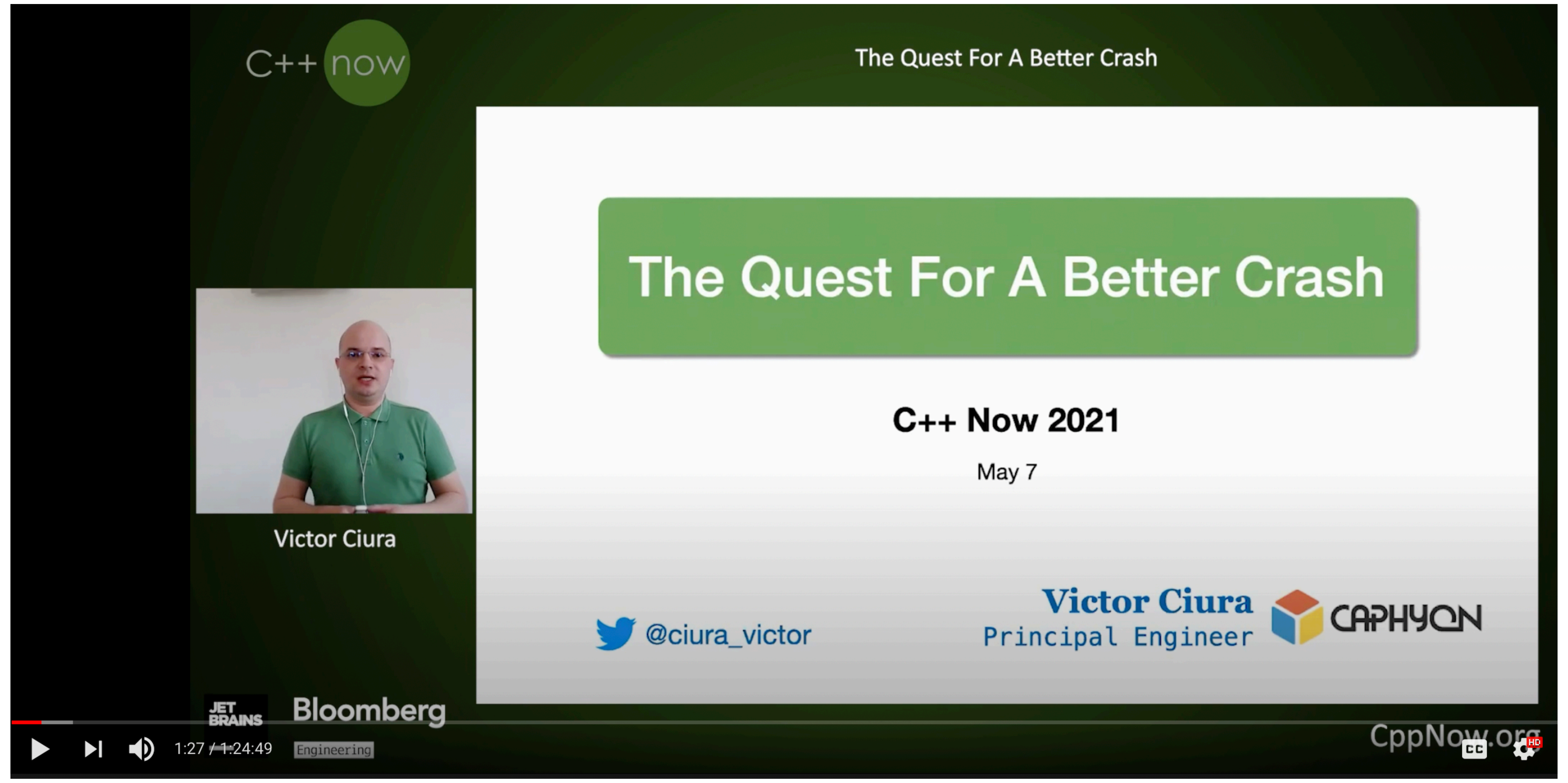
`x86/x64`

```
#include <dbghelp.h>
```

/LINK `Dbghelp.lib`

Dynamic dependency on `Dbghelp.dll`

My Quest For A Better Crash



youtube.com/watch?v=pJPRdNTxL-E

Handle the unhandled...

JUST WHEN
I THOUGHT
I WAS OUT
THEY PULL ME BACK IN
imgflip.com

No matching catch block ☹️

```
void __scrt_set_unhandled_exception_filter()
{
  ::SetUnhandledExceptionFilter(__scrt_unhandled_exception_filter);
}

LONG __scrt_unhandled_exception_filter(EXCEPTION_POINTERS * aPointers)
{
  if (aPointers->ExceptionRecord->ExceptionCode == EH_EXCEPTION_NUMBER)
    std::terminate();

  return EXCEPTION_CONTINUE_SEARCH;
}
```

Unhandled Exceptions

```
static bool installedFilter = false;
if (!installedFilter)
{
  ::SetUnhandledExceptionFilter(ExceptionHandling::UnhandledException);
  installedFilter = true;
}
```

If an exception occurs in a process that is not being debugged,
and the exception makes it to the **Unhandled** exception filter => we intercept it

This replaces the existing top-level exception filter for ALL existing and
ALL future threads in the calling process.

Unhandled Exceptions

```
LONG ExceptionHandling::UnhandledException(EXCEPTION_POINTERS * aExceptionInfo)
{
  wstring message(L"[EXCEPTION_UNHANDLED] ");

  wchar_t buf[MSG_BUFFER_LEN];
  swprintf_s(buf, MSG_BUFFER_LEN, L"(0x%.8x) at address " ADDRESS_FORMAT SW_EOL,
             aExceptionInfo->ExceptionRecord->ExceptionCode,
             aExceptionInfo->ExceptionRecord->ExceptionAddress);
  message += buf;

  StackWalker::TraceFromContext(message, aExceptionInfo->ContextRecord);

  ErrMsgPresenter::Message(message);

  return EXCEPTION_EXECUTE_HANDLER;
}
```

Bonus Slides

ISO C++ Next

P0881

A Proposal to add stacktrace library

- Alexey Gorgurov, Antony Polukhin

First draft: 2018-01-23 [R0] based on Boost.Stacktrace

....

Didn't make into **C++20** 🙁

...

[R7] wg21.link/P0881 `#include <stacktrace>`

Tweet

Alisdair Meredith
@AlisdairMered

The first major library feature for #Cpp23 will be a stack trace library

7:34 PM · Nov 9, 2020 · Twitter Web App

1 Quote Tweet **18** Likes

Kilian @kilian_ukilele · Nov 10
Replying to @AlisdairMered
Awesome! Will std::exception also get a function to query the callstack from where it got thrown?

twitter.com/alisdairmered/status/1325854252338716672?s=21

C++ 23 <stacktrace>

Key features (desired):

C++ 23 <stacktrace>

Key features (desired):

- all functions are **lazy**: do not query the stacktrace entry info without explicit request

C++ 23 <stacktrace>

Key features (desired):

- all functions are **lazy**: do not query the stacktrace entry info without explicit request
- **dynamic size** for trace - all the available invokers must be stored in a stacktrace

C++ 23 <stacktrace>

Key features (desired):

- all functions are **lazy**: do not query the stacktrace entry info without explicit request
- **dynamic size** for trace - all the available invokers must be stored in a stacktrace
- implementations: allow to **disable/enable** gathering stacktraces by a linker switch

C++ 23 <stacktrace>

Key features (desired):

- all functions are **lazy**: do not query the stacktrace entry info without explicit request
- **dynamic size** for trace - all the available invokers must be stored in a stacktrace
- implementations: allow to **disable/enable** gathering stacktraces by a linker switch
- stacktracing shouldn't prevent any of **optimizations**

C++ 23 <stacktrace>

Key features (desired):

- all functions are **lazy**: do not query the stacktrace entry info without explicit request
- **dynamic size** for trace - all the available invokers must be stored in a stacktrace
- implementations: allow to **disable/enable** gathering stacktraces by a linker switch
- stacktracing shouldn't prevent any of **optimizations**
- stacktrace should be **usable** in contract violation handler, coroutines, handler functions, parallel algorithms

C++ 23 <stacktrace>

Key features (desired):

C++ 23 <stacktrace>

Key features (desired):

C++ 23 <stacktrace>

Key features (desired):

- `stacktrace_entry::description()` should return a **demangled** function signature

Key features (desired):

- `stacktrace_entry::description()` should return a **demangled** function signature
- `to_string(stacktrace)` should query information from **debug symbols**, symbol export tables and any other sources, returning *demangled* signatures

C++ 23 <stacktrace>

Key features (desired):

- `stacktrace_entry::description()` should return a **demangled** function signature
- `to_string(stacktrace)` should query information from **debug symbols**, symbol export tables and any other sources, returning *demangled* signatures
- information about **inlined functions** that have no separate stacktrace entries is welcomed -> `to_string(stacktrace)`

C++ 23 <stacktrace>

Key features (desired):

- `stacktrace_entry::description()` should return a **demangled** function signature
- `to_string(stacktrace)` should query information from **debug symbols**, symbol export tables and any other sources, returning *demangled* signatures
- information about **inlined functions** that have no separate stacktrace entries is welcomed -> `to_string(stacktrace)`
- avoid doing **heavy** operations in `basic_stacktrace` constructors or `stacktrace_entry::current()`

C++ 23 <stacktrace>

```
class stacktrace_entry
{
  public:
    using native_handle_type = implementation-defined;
    ...
    constexpr native_handle_type native_handle() const noexcept;
    constexpr explicit operator bool() const noexcept;
    ...
    string   description() const;
    string   source_file() const;
    uint32_t source_line() const;
};
```

`stacktrace_entry` models concepts:

regular and `three_way_comparable<strong_ordering>`

C++ 23 <stacktrace>

```
template<class Allocator>
class basic_stacktrace
{
public:
  using value_type = stacktrace_entry;
  using allocator_type = Allocator;
  ...
  const_iterator begin() const noexcept;
  const_iterator end() const noexcept;
  const_reverse_iterator rbegin() const noexcept;
  const_reverse_iterator rend() const noexcept;
  ...

private:
  vector<value_type, allocator_type> m_frames;
};
```

C++ 23 <stacktrace>

```
static basic_stacktrace current(const allocator_type& alloc = allocator_type()) noexcept;

static basic_stacktrace current(size_type skip,
                                const allocator_type& alloc = allocator_type()) noexcept;

static basic_stacktrace current(size_type skip, size_type max_depth,
                                const allocator_type& alloc = allocator_type()) noexcept;
```

=> `basic_stacktrace` object with `m_frames` storing the stack trace of the current evaluation in the *current thread* of execution

`alloc` is passed to the constructor of the `m_frames` object.

C++ 23 <stacktrace>

```
namespace std {

  using stacktrace = basic_stacktrace<allocator<stacktrace_entry>>;

  string to_string(const stacktrace_entry& f);

  template<class Alloc>
  string to_string(const basic_stacktrace<Alloc>& st);

  template<class charT, class traits>
  basic_ostream<charT, traits>&
  operator<<(basic_ostream<charT, traits>& os, const stacktrace_entry& f);

  template<class charT, class traits, class Alloc>
  basic_ostream<charT, traits>&
  operator<<(basic_ostream<charT, traits>& os, const basic_stacktrace<Alloc>& st);
```

```
description()
source_file()
source_line()
```

C++ 23 Example

```
auto trace = basic_stacktrace::current();

for (stacktrace_entry frame : trace)
{
  std::cerr << frame.description() << " at "
            << frame.source_file() << ":" << frame.source_line() << "\n";
}
```

C++ 23 Example

```
auto trace = basic_stacktrace::current();

for (stacktrace_entry frame : trace)
{
  std::cerr << std::to_string(frame) << "\n";
}
```

C++ 23 Example

```
auto trace = basic_stacktrace::current();

for (stacktrace_entry frame : trace)
{
  std::cerr << frame << "\n";
}
```

C++ 23 Example

```
auto trace = basic_stacktrace::current();

std::cerr << std::to_string(trace);
```

C++ 23 Example

```
auto trace = basic_stacktrace::current();

std::cerr << trace;
```

C++ 23 <stacktrace>

It can't get any simpler than that.

C++ 23 <stacktrace>

It can't get any simpler than that.

I can't wait to see **early** implementations from our standard library providers!

Exceptional C++

CppCon 2021

October 25th