<table>
<thead>
<tr>
<th>Outline</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Visual C++</strong></td>
</tr>
<tr>
<td>- Structured Exception Handling (SEH)</td>
</tr>
<tr>
<td>- C++ Exception Handling (EH)</td>
</tr>
<tr>
<td><strong>GCC</strong></td>
</tr>
<tr>
<td>- RTTI</td>
</tr>
<tr>
<td>- SjLj exceptions</td>
</tr>
<tr>
<td>- Zero-cost (table based)</td>
</tr>
</tbody>
</table>
SEH vs C++ EH
SEH (Structured Exceptions Handling) is the low-level, system layer
Allows to handle exceptions sent by the kernel or raised by user code
C++ exceptions in MSVC are implemented on top of SEH
Keywords __try, __except and __finally can be used for compiler-level SEH support.

The compiler uses a single exception handler per all functions with SEH, but different supporting structures (scope tables) per function.

The SEH handler registration frame is placed on the stack.

In addition to fields required by system (handler address and next pointer), a few VC-specific fields are added.
Visual C++

Frame structure (fs:0 points to the Next member)

```c
// -8 DWORD _esp;
// -4 PEXCEPTION_POINTERS xpointers;
struct _EH3_EXCEPTION_REGISTRATION
{
    struct _EH3_EXCEPTION_REGISTRATION *Next;
    PVOID ExceptionHandler;
    PSCOPETABLE_ENTRY ScopeTable;
    DWORD TryLevel;
};
```
ExceptionHandler points to __except_handler3 (SEH3) or __except_handler4 (SEH4)

The frame set-up is often delegated to compiler helper (__SEH_prolog/__SEH_prolog4/etc)

**ScopeTable** points to a table of entries describing all __try blocks in the function

in SEH4, the scope table pointer is XORed with the security cookie, to mitigate scope table pointer overwrite
Visual C++: Scope Table

- One scope table entry is generated per __try block
- EnclosingLevel points to the block which contains the current one (first table entry is number 0)
- Top level (function) is -1 for SEH3 and -2 for SEH4
- SEH4 has additional fields for cookie checks

<table>
<thead>
<tr>
<th>SEH3</th>
<th>SEH4</th>
</tr>
</thead>
</table>
| struct _SCOPETABLE_ENTRY {
  DWORD EnclosingLevel;
  void* FilterFunc;
  void* HandlerFunc;
}                         | struct _EH4_SCOPETABLE {
  DWORD GSCookieOffset;
  DWORD GSCookieXOROffset;
  DWORD EHCookieOffset;
  DWORD EHCookieXOROffset;
  _EH4_SCOPETABLE_RECORD ScopeRecord[];
}                         |
Visual C++: mapping tables to code

- **FilterFunc** points to the exception filter (expression in the `__except` operator)
- **HandlerFunc** points to the `__except` block body
- If **FilterFunc** is NULL, **HandlerFunc** is the `__finally` block body
- Current try block number is kept in the **TryLevel** variable of the exception registration frame

```asm
; Entering __try block 0
mov [ebp+ms_exc.registration.TryLevel], 0
; Entering __try block 1
mov [ebp+ms_exc.registration.TryLevel], 1
[...]
; Entering __try block 0 again
mov [ebp+ms_exc.registration.TryLevel], 0
```
A few intrinsics are available for use in exception filters and __finally block

- They retrieve information about the current exception
- GetExceptionInformation/GetExceptionCode use the xpointers variable filled in by the exception handler
- AbnormalTermination() uses a temporary variable which is set before entering the __try block and cleared if the __finally handler is called during normal execution of the function
C++ implementation
Visual C++: RTTI

- See openrce.org for more info
Visual C++: EH

- EH is present if function uses try/catch statements or automatic objects with non-trivial destructors are present implemented on top of SEH
- Uses a distinct handler per function, but they all eventually call a common one (_CxxFrameHandler/_CxxFrameHandler3)
- Compiler-generated unwind funclets are used to perform unwinding actions (calling destructors etc) during exception processing
- A special structure (FuncInfo) is generated for the function and contains info about unwinding actions and try/catch blocks
Visual C++ EH: Registration and FuncInfo structure

```c
struct EHRegistrationNode {
    // -4 void *_esp;
    EHRegistrationNode *pNext;
    void *frameHandler;
    int state;
};

typedef const struct _s_FuncInfo {
    unsigned int magicNumber:29;
    unsigned int bbtFlags:3;
    int maxState;
    const struct _s_UnwindMapEntry * pUnwindMap;
    unsigned int nTryBlocks;
    const struct _s_TryBlockMapEntry * pTryBlockMap;
    unsigned int nIPMapEntries;
    void * pIPtoStateMap;
    const struct _s_ESTypeList * pESTypeList;
    int EHFlags;
} FuncInfo;
```
### Visual C++ EH: FuncInfo structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| magicNumber               | 0x19930520: original (pre-VC2005?)  
                           | 0x19930521: pESTypeList is valid  
                           | 0x19930522: EHFlags is valid   |
| maxState/pUnwindMap       | Number of entries and pointer to unwind map                           |
| nTryBlocks/pTryBlockMap   | Number of entries and pointer to try{} block map                      |
| nIPMapEntries/pIPtoStateMap | IP-to-state map (unused on x86)                                      |
| pESTypeList               | List of exceptions in the throw specification (undocumented feature)   |
| EHFlags                   | FI_EHS_FLAG=1: function was compiled /EHs                             |
Visual C++ EH: Unwind map

- Similar to SEH's scope table, but without exception filters
- All necessary actions (unwind funclets) are executed unconditionally
- Action can be NULL to indicate no-action state transition
- Typical funclet destroys a constructed object on the stack, but there may be other variations
- Top-level state is -1

```c
typedef const struct _s_UnwindMapEntry {
    int toState;
    void *action;
} UnwindMapEntry;
```
Visual C++: changes for x64

- SEH changes completely
- Instead of stack-based frame registration, pointers to handlers and unwind info are stored in .pdata section
- Only limited set of instructions are supposed to be used in prolog and epilog, which makes stack walking and unwinding easier
- "Language-specific handlers" are used to implement compiler-level SEH and C++ EH
- However, the supporting SEH/EH structures (scope table, FuncInfo etc) are very similar
x64: .pdata section

- Contains an array of RUNTIME_FUNCTION structures
- Each structure describes a contiguous range of instructions belonging to a function
- Chained entries (bit 0 set in UnwindInfo) point to the parent entry
- All addresses are RVAs

```c
typedef struct _RUNTIME_FUNCTION {
    DWORD BeginAddress;
    DWORD EndAddress;
    DWORD UnwindInfoAddress;
} RUNTIME_FUNCTION;
```
x64: Unwind Info

- Starts with a header, then a number of "unwind codes", then an optional handler and any additional data for it
- Handler is present if Flags have UNW_FLAG_EHANDLER or UNW_FLAG_UHANDLER

```c
typedef struct _UNWIND_INFO {
    unsigned char Version : 3;                 // Version Number
    unsigned char Flags   : 5;                 // Flags
    unsigned char SizeOfProlog;
    unsigned char CountOfCodes;
    unsigned FrameRegister : 4;
    unsigned FrameOffset   : 4;
    UNWIND_CODE UnwindCode[1];
    /*  UNWIND_CODE MoreUnwindCode[ (CountOfCodes+1)&~1 ]-1];
    *  union {
    *      OPTIONAL ULONG ExceptionHandler;
    *      OPTIONAL ULONG FunctionEntry;
    *  };
    *  OPTIONAL ULONG ExceptionData[];
    */
} UNWIND_INFO, *PUNWIND_INFO;
```
### x64: Standard VC Exception Handlers

<table>
<thead>
<tr>
<th>Handler</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>__C_specific_handler</td>
<td>Scope table</td>
</tr>
<tr>
<td>__GSHandlerCheck</td>
<td>GS data</td>
</tr>
<tr>
<td>__GSHandlerCheck_SEH</td>
<td>Scope table + GS data</td>
</tr>
<tr>
<td>__CxxFrameHandler3</td>
<td>RVA to FuncInfo</td>
</tr>
<tr>
<td>__GSHandlerCheck_EH</td>
<td>RVA to FuncInfo + GS data</td>
</tr>
</tbody>
</table>

```c
struct _SCOPE_TABLE_AMD64 {
    DWORD Count;
    struct {
        DWORD BeginAddress;
        DWORD EndAddress;
        DWORD HandlerAddress;
        DWORD JumpTarget;
    } ScopeRecord[1];
};
```

```c
struct _GS_HANDLER_DATA {
    union {
        union {
            unsigned long EHandler:1;
            unsigned long UHandler:1;
            unsigned long HasAlignment:1;
        } Bits;
        int CookieOffset;
    } u;
    long AlignedBaseOffset;
    long Alignment;
};
```
Scope table entries are looked up from the PC (RIP) value instead of using an explicit state variable.

Since they're sorted by address, this is relatively quick.

**Handler** points to the exception filter and **Target** to the __except block body.

However, if Target is 0, then Handler is the __finally block body.

Compiler (always?) emits a separate function for __finally blocks and an inline copy in the function body.

GS cookie data, if present, is placed after the scope table.
x64: VC C++ EH FuncInfo

Pretty much the same as x86, except RVAs instead of addresses and IP-to-state map is used

typedef struct _s_FuncInfo
{
    int magicNumber;          // 0x19930522
    int maxState;             // number of states in unwind map
    int dispUnwindMap;        // RVA of the unwind map
    unsigned int nTryBlocks; // count of try blocks
    int dispTryBlockMap;      // RVA of the try block array
    unsigned int nIPMapEntries; // count of IP-to-state entries
    int dispIPtoStateMap;     // RVA of the IP-to-state array
    int dispUwindHelp;        // rsp offset of the state var
        // (initialized to -2; used during unwinding)
    int dispESTypeList;       // list of exception spec types
    int EHFlags;              // flags
} FuncInfo;
x64: IP-to-state map

- Instead of an explicit state variable on the stack (as in x86), this map is used to find out the current state from the execution address

```c
typedef struct IptoStateMapEntry {
    __int32 Ip; // Image relative offset of IP
    __ehstate_t State;
} IptoStateMapEntry;
```
x64: C++ Exception Record

Since exceptions can be caught in a different module and the ThrowInfo RVAs might need to be resolved, the imagebase of the throw originator is added to the structure.

typedef struct EHExceptionRecord {
    DWORD ExceptionCode; // (= EH_EXCEPTION_NUMBER)
    DWORD ExceptionFlags; // Flags determined by NT
    struct _EXCEPTION_RECORD *ExceptionRecord; // extra record (not used)
    void * ExceptionAddress; // Address at which exception occurred
    DWORD NumberParameters; // Number of extended parameters. (=4)
    struct EHParameters {
        DWORD magicNumber; // = EH_MAGIC_NUMBER1
        void * pExceptionObject; // Pointer to the actual object thrown
        ThrowInfo *pThrowInfo; // Description of thrown object
        void * pThrowImageBase; // Image base of thrown object
    } params;
} EHExceptionRecord;
Visual C++: references

SEH
- http://uninformed.org/index.cgi?v=4&a=1

C++ EH/RTTI
- http://www.codeproject.com/Articles/2126/How-a-C-compiler-implements-exception-handling
- http://www.openrce.org/articles/full_view/21

Wine sources

Visual Studio 2012 RC includes sources of the EH implementation
- see VC\src\crt\ehdata.h and VC\src\crt\eh\n- includes parts of "ARMNT" and WinRT
GCC

C++ in GCC
In the most common case (no virtual inheritance), the virtual table starts with two entries: offset-to-base and RTTI pointer. Then the function pointers follow.

In the virtual table for the base class itself, the offset will be 0. This allows us to identify class vtables if we know RTTI address.

```
`vtable for'SubClass
  dd 0 ; offset to base
  dd offset `typeinfo for'SubClass ; type info pointer
  dd offset SubClass::vfunc1(void) ; first virtual function
  dd offset BaseClass::vfunc2(void) ; second virtual function
```
GCC: RTTI

- GCC's RTTI is based on the Itanium C++ ABI [1]
- The basic premise is: typeid() operator returns an instance of a class inherited from std::type_info
- For every polymorphic class (with virtual methods), the compiler generates a static instance of such class and places a pointer to it as the first entry in the Vtable
- The layout and names of those classes are standardized, so they can be used to recover names of classes with RTTI

For class recovery, we're only interested in three classes inherited from `type_info`

```cpp
class type_info {
    //void *vfptr;
private:
    const char *__type_name;
};

// a class with no bases
class __class_type_info : public std::type_info {}

// a class with single base
class __si_class_type_info: public __class_type_info {
public:
    const __class_type_info *__base_type;
};

// a class with multiple bases
class __vmi_class_type_info : public __class_type_info {
public:
    unsigned int __flags;
    unsigned int __base_count;
    __base_class_type_info __base_info[1];
};
struct __base_class_type_info {
public:
    const __class_type_info *__base_type;
    long __offset_flags;
}
```
GCC: recovery of class names from RTTI

- Find vtables of \_\_class\_type\_info, \_\_si\_class\_type\_info, \_\_vmi\_class\_type\_info
- Look for references to them; those will be instances of typeid classes
- From the \_\_type\_name member, mangled name of the class can be recovered, and from other fields the inheritance hierarchy
- By looking for the pair (0, pTypeInfo), we can find the class virtual table

```
`\_\_class\_type\_info` for 'SubClass
  dd offset `\_\_vtable` for '__\_\_cxxabiv1::__\_\_si\_class\_type\_info+8
  dd offset `\_\_type\_name` name for 'SubClass ; "8SubClass"
  dd offset `\_\_type\_info` for 'BaseClass
```
GCC: RTTI example

```
vtable for 'SubClass

0
(typeinfo for 'SubClass
SubClass::vfunc1(void)
BaseClass::vfunc2(void)

vtable for 'BaseClass

0
(typeinfo for 'BaseClass
__cxa_pure_virtual
BaseClass::vfunc2(void)
```
GCC: exceptions

- Two kinds of implementing exceptions are commonly used by GCC:
  - SjLj (setjump-longjump)
  - "zero-cost" (table-based)
- These are somewhat analogous to VC's x86 stack-based and x64 table-based SEH implementations
- SjLj is simpler to implement but has more runtime overhead, so it's not very common these days
- MinGW used SjLj until GCC 4.3(?), and iOS on ARM currently supports only SjLj
Similar to x86 SEH, a structure is constructed on the stack for each function that uses exception handling. However, instead of using list in fs:0, implementation-specific functions are called at entry and exit (_Unwind_SjLj_Register/_Unwind_SjLj_Unregister). The registration structure can vary but generally uses this layout:

```c
struct SjLj_Function_Context
{
    struct SjLj_Function_Context *prev;
    int call_site;
    _Unwind_Word data[4];
    _Unwind_Personality_Fn personality;
    void *lsda;
    int jbuf[];
};
```
GCC: SjLj exceptions

- **personality** points to the so-called "personality function" which is called by the unwinding routine during exception processing.
- **lsda** points to "language-specific data area" which contains info in the format specific to the personality routine.
- **call_site** is analogous to the state variable of VC's EH and is updated by the compiler on every change which might need an corresponding unwinding action.
- **jbuf** contains architecture-specific info used by the unwinder to resume execution if the personality routine signals that a handler has been found.
- However, usually **jbuf[2]** contains the address of the *landing pad* for the function.
SjLj setup example

```
ADD   R0, SP, #0xA4+_sjlj_ctx
LDR   R3, [R3] ; __gxx_personality_sj0
STR   R3, [SP,#0xA4+_sjlj_ctx.personality]
LDR   R3, =_lsda_sub_14F94
STR   R7, [SP,#0xA4+_sjlj_ctx.jbuf]
STR   R3, [SP,#0xA4+_sjlj_ctx.lsda]
LDR   R3, =lp_sub_14F94
STR   SP, [SP,#0xA4+_sjlj_ctx.jbuf+8]
STR   R3, [SP,#0xA4+_sjlj_ctx.jbuf+4]
BL    __Unwind_SjLj_Register
MOV   R3, #2
STR   R3, [SP,#0xA4+_sjlj_ctx.call_site]

_sjlj_ctx.personality = &__gxx_personality_sj0;
_sjlj_ctx.jbuf[0] = &v11;   // frame pointer
_sjlj_ctx.lsda = &lsda_sub_14F94;
_sjlj_ctx.jbuf[2] = &v5;    // stack pointer
_sjlj_ctx.jbuf[1] = lp_sub_14F94; // landing pad
_Unwind_SjLj_Register(&_sjlj_ctx);
_sjlj_ctx.call_site = 2;
```
SjLj landing pad: unwinding

- The SjLj landing pad handler inspects the **call_site** member and depending on its value performs unwinding actions (destruct local variables) or executes user code in the catch blocks.

```
__lp__sub_1542C
LDR    R3, [SP,#0x114+_sjlj_ctx.call_site]
LDR    R2, [SP,#0x114+_sjlj_ctx.data]
CMP    R3, #1
STR    R2, [SP,#0x114+exc_obj]
BEQ    unwind_from_state1
CMP    R3, #2
BEQ    unwind_from_state2
CMP    R3, #3
BEQ    unwind_from_state3
...
```

```
unwind_from_state3
ADD    R0, SP, #0x114+tmp_str3
MOV    R3, #0
STR    R3, [SP,#0x114+_sjlj_ctx.call_site]
BL     std::string::~string()
MOV    R3, #0
ADD    R0, SP, #0x114+_sjlj_ctx
STR    R3, [SP,#0x114+_sjlj_ctx.call_site]
[...]
MOV    R3, 0xFFFFFFFF
LDR    R0, [SP,#0x114+exc_obj]
STR    R3, [SP,#0x114+_sjlj_ctx.call_site]
BL     __Unwind_SjLj_Resume
```
SjLj landing pad: catch blocks

- If the current state corresponds to a try block, then the landing pad handler checks the **handler switch value** to determine which exception was matched.

```
__lp__GpsRun
LDR  R2, [SP,#0xD4+_sjlj_ctx.data]
LDR  R3, [SP,#0xD4+_sjlj_ctx.call_site]
STR  R2, [SP,#0xD4+exc_obj]
LDR  R2, [SP,#0xD4+_sjlj_ctx.data+4]
CMP  R3, #1
STR  R2, [SP,#0xD4+handler_switch_val]
BEQ  _GpsRun_lp_01
CMP  R3, #2
BEQ  _GpsRun_lp_02

_GpsRun_lp_02
LDR  R2, [SP,#0xD4+handler_switch_val]
CMP  R2, #2
BNE  _catch_ellipsis
LDR  R0, [SP,#0xD4+exc_obj]
BLX  ___cxa_begin_catch
[...]
MOV  R3, #0
STR  R3, [SP,#0xD4+_sjlj_ctx.call_site]
BLX  ___cxa_end_catch
```
GCC exceptions: zero-cost

- "Zero-cost" refers to no code overhead in the case of no exceptions (unlike SjLj which has set-up/tear-down code that is always executed)
- Uses a set of tables encoding address ranges, so does not need any state variables in the code
- Format and encoding is based on Dwarf2/Dwarf3
- The first-level (language-independent) format is described in Linux Standard Base Core Specification [1]
- Second-level (language-specific) is based on HP Itanium implementation [2] but differs from it in some details

GCC exceptions: .eh_frame

- Format of the section is based on Dwarf's debug_frame
- Consists of a sequence of Common Frame Information (CFI) records
- Each CFI starts with a Common Information Entry (CIE) and is followed by one or more Frame Description Entry (FDE)
- Usually one CFI corresponds to one object file and one FDE to a function

```
CFI 0
  CIE 0
  FDE 0-0
  FDE 0-1
  FDE 0-2
  ...

CFI 1
  CIE 1
  FDE 1-0
  FDE 1-1
  ...
```
### .eh_frame: Common Information Entry

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length (uint32)</td>
<td>total length of following data; 0 means end of all records</td>
</tr>
<tr>
<td>Extended Length (uint64)</td>
<td>present if Length==0xFFFFFFFF</td>
</tr>
<tr>
<td>CIE_id (uint32)</td>
<td>Must be 0 for a CIE</td>
</tr>
<tr>
<td>Version (uint8)</td>
<td>1 or 3</td>
</tr>
<tr>
<td>Augmentation (asciiz string)</td>
<td>Indicates presence of additional fields</td>
</tr>
<tr>
<td>Code alignment factor (uleb128)</td>
<td>Usually 1</td>
</tr>
<tr>
<td>Data alignment factor (sleb128)</td>
<td>Usually -4 (encoded as 0x7C)</td>
</tr>
<tr>
<td>return_address_register (uint8 (version 1) or uleb128)</td>
<td>Dwarf number of the return register</td>
</tr>
<tr>
<td>Augmentation data length (uleb128)</td>
<td>Present if Augmentation[0]=='z'</td>
</tr>
<tr>
<td>Augmentation data</td>
<td>Present if Augmentation[0]=='z'</td>
</tr>
<tr>
<td>Initial instructions</td>
<td>Dwarf Call Frame Instructions</td>
</tr>
</tbody>
</table>
### .eh_frame: Augmentation data

- Each string character signals that additional data is present in the "Augmentation data" of the CIE (and possibly FDE)

<table>
<thead>
<tr>
<th>String character</th>
<th>Data</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'z'</td>
<td>uleb128</td>
<td>Length of following data</td>
</tr>
<tr>
<td>'P'</td>
<td>personality_enc (uint8)</td>
<td>Encoding of the following pointer</td>
</tr>
<tr>
<td></td>
<td>personality_ptr</td>
<td>Personality routine for this CIE</td>
</tr>
<tr>
<td>'R'</td>
<td>fde_enc (uint8)</td>
<td>Encoding of initial location and length in FDE (if different from default)</td>
</tr>
<tr>
<td>'L'</td>
<td>lsda_enc</td>
<td>Encoding of the LSDA pointer in FDE's augmentation data</td>
</tr>
</tbody>
</table>
**.eh_frame: Frame Description Entry**

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length (uint32)</td>
<td>total length of following data; 0 means end of all records</td>
</tr>
<tr>
<td>Extended Length (uint64)</td>
<td>present if Length==0xFFFFFFFF</td>
</tr>
<tr>
<td>CIE pointer (uint32)</td>
<td>Distance to parent CIE from this field</td>
</tr>
<tr>
<td>Initial location (fde_encoding)</td>
<td>Address of the first instruction in the range</td>
</tr>
<tr>
<td>Range length (fde_encoding.size)</td>
<td>Length of the address range</td>
</tr>
<tr>
<td>Augmentation data length (uleb128)</td>
<td>Present if CIE.Augmentation[0]=='z'</td>
</tr>
<tr>
<td>Augmentation data</td>
<td>Present if CIE.Augmentation[0]=='z'</td>
</tr>
<tr>
<td>Instructions</td>
<td>Dwarf Call Frame Instructions</td>
</tr>
</tbody>
</table>

- "Augmentation data" contains pointer to LSDA if CIE's augmentation string included the L character
**.eh_frame: pointer encodings**

- Bits 0:3 (0x0F): data format
- Bits 4:6 (0x70): how the value is applied
- Bit 7 (0x80): the value should be dereferenced to get final address
- Common encodings: 0xFF – value omitted; 0x00 – native-sized absolute pointer; 0x1B – self-relative 4-byte displacement; 0x9B – dereferenced self-relative 4-byte displacement

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DW_EH_PE_ptr</td>
<td>0x00</td>
<td>DW_EH_PE_absptr</td>
<td>0x00</td>
</tr>
<tr>
<td>DW_EH_PE_uleb128</td>
<td>0x01</td>
<td>DW_EH_PE_pcrel</td>
<td>0x10</td>
</tr>
<tr>
<td>DW_EH_PE_udata2</td>
<td>0x02</td>
<td>DW_EH_PE_textrel</td>
<td>0x20</td>
</tr>
<tr>
<td>DW_EH_PE_udata4</td>
<td>0x03</td>
<td>DW_EH_PE_datarel</td>
<td>0x30</td>
</tr>
<tr>
<td>DW_EH_PE_udata8</td>
<td>0x04</td>
<td>DW_EH_PE_funcrel</td>
<td>0x40</td>
</tr>
<tr>
<td>DW_EH_PE_sleb128</td>
<td>0x09</td>
<td>DW_EH_PE_aligned</td>
<td>0x50</td>
</tr>
<tr>
<td>DW_EH_PE_sdata2</td>
<td>0x0A</td>
<td>DW_EH_PE_indirect</td>
<td>0x80</td>
</tr>
<tr>
<td>DW_EH_PE_sdata4</td>
<td>0x0B</td>
<td>DW_EH_PE_omit</td>
<td>0xFF</td>
</tr>
<tr>
<td>DW_EH_PE_sdata8</td>
<td>0x0C</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
GCC: .gcc_except_table (LSDA)

- Used by both SjLj and table based implementations to encode unwinding actions and catch handlers for try blocks
- Although LSDA is indeed language-specific and GCC uses different personality functions to parse it, the overall layout is very similar across most implementations
- In fact, the SjLj (_sj0) and table-based (_v0) personality functions use almost identical format
- It also uses Dwarf encoding (LEB128) and pointer encodings from .eh_frame
- Consists of: header, call site table, action table and optional type table
## GCC: LSDA

### LSDA

<table>
<thead>
<tr>
<th>LSDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header</td>
</tr>
<tr>
<td>Call site table</td>
</tr>
<tr>
<td>Action table</td>
</tr>
<tr>
<td>Types table (optional)</td>
</tr>
</tbody>
</table>

### LSDA Header

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>lpstart_encoding (uint8)</td>
<td>Encoding of the following field (landing pad start offset)</td>
</tr>
<tr>
<td>LPStart (encoded)</td>
<td>Present if lpstart_encoding != DW_EH_PE_omit (otherwise default: start of the range in FDE)</td>
</tr>
<tr>
<td>ttype_encoding (uint8)</td>
<td>Encoding of the pointers in type table</td>
</tr>
<tr>
<td>TType (uleb128)</td>
<td>Offset to type table from the end of the header Present if ttype_encoding != DW_EH_PE_omit</td>
</tr>
</tbody>
</table>
### GCC: LSDA

#### LSDA call site table header

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>call_site_encoding (uint8)</td>
<td>Encoding of items in call site table</td>
</tr>
<tr>
<td>call_site_tbl_len (uleb128)</td>
<td>Total length of call site table</td>
</tr>
</tbody>
</table>

#### LSDA call site entry (SjLj)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cs_lp (uleb128)</td>
<td>New &quot;IP&quot; value (call_site variable)</td>
</tr>
<tr>
<td>cs_action (uleb128)</td>
<td>Offset into action table (+1) or 0 for no action</td>
</tr>
</tbody>
</table>

#### LSDA call site entry (table-based)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cs_start (call_site_encoding)</td>
<td>Start of the IP range</td>
</tr>
<tr>
<td>cs_len (call_site_encoding)</td>
<td>Length of the IP range</td>
</tr>
<tr>
<td>cs_lp (call_site_encoding)</td>
<td>Landing pad address</td>
</tr>
<tr>
<td>cs_action (uleb128)</td>
<td>Offset into action table (+1) or 0 for no action</td>
</tr>
</tbody>
</table>
### GCC: LSDA

#### LSDA action table entry

<table>
<thead>
<tr>
<th>Filter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ar_filter (sleb128)</td>
<td>Type filter value (0 = cleanup)</td>
</tr>
<tr>
<td>ar_disp (sleb128)</td>
<td>Self-relative byte offset to the next action (0 = end)</td>
</tr>
</tbody>
</table>

#### LSDA type table

<table>
<thead>
<tr>
<th>T3 typeinfo (ttype_encoding)</th>
<th>Filter value</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1 typeinfo (ttype_encoding)</td>
<td>1</td>
</tr>
<tr>
<td>idx1, idx2, idx3, 0 (uleb128)</td>
<td>-1</td>
</tr>
<tr>
<td>idx4, idx5, idx6, 0 (uleb128)</td>
<td>-N</td>
</tr>
</tbody>
</table>

- **TTBase**: Catches
- **Exception specifications**: ...
The personality function looks up the call site record which matches the current IP value (for SjLj the `call_site` variable is used)

If there is a valid action (non-zero) then it is "executed" – the chain of action records is walked and filter expressions are checked

If there is a filter match or a cleanup record found (ar_filter==0) then the control is transferred to the landing pad

For SjLj, there is a single landing pad handler so the `call_site` is set to the `cs_lp` value from the call site record

For table-based exceptions, the specific landing pad for the call site record is run
GCC: EH optimizations

- The eh_frame structure uses variable-length encodings which (supposedly) saves space at the expense of slower look up at run time
- Some implementations introduce various indexing and lookup optimizations
GCC EH optimizations: .eh_frame_hdr

- A table of pairs (initial location, pointer to the FDE in .eh_frame)
- Table is sorted, so binary search can be used to quickly search
- PT_EH_FRAME header is added to ELF program headers

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>version (uint8)</td>
<td>structure version (=1)</td>
</tr>
<tr>
<td>eh_frame_ptr_enc (uint8)</td>
<td>encoding of eh_frame_ptr</td>
</tr>
<tr>
<td>fde_count_enc (uint8)</td>
<td>encoding of fde_count</td>
</tr>
<tr>
<td>table_enc (uint8)</td>
<td>encoding of table entries</td>
</tr>
<tr>
<td>eh_frame_ptr (enc)</td>
<td>pointer to the start of the .eh_frame section</td>
</tr>
<tr>
<td>fde_count (enc)</td>
<td>number of entries in the table</td>
</tr>
<tr>
<td>initial_loc[0] (enc)</td>
<td>initial location for the FDE</td>
</tr>
<tr>
<td>fde_ptr[0] (enc)</td>
<td>corresponding FDE</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
GCC EH optimizations: __unwind_info

- An additional section added to Mach-O binaries on OS X
- Contains entries for efficient lookup
- More info: compact_unwind_encoding.h in libunwind

```c
struct unwind_info_section_header {
    uint32_t version;        // UNWIND_SECTION_VERSION
    uint32_t commonEncodingsArraySectionOffset;
    uint32_t commonEncodingsArrayCount;
    uint32_t personalityArraySectionOffset;
    uint32_t personalityArrayCount;
    uint32_t indexSectionOffset;
    uint32_t indexCount;
    // compact_unwind_encoding_t[]
    // uintptr_t personalities[]
    // unwind_info_section_header_index_entry[]
    // unwind_info_section_header_lsoa_index_entry[]
};
```
**GCC EH optimizations: ARM EABI**

- `.ARM.exidx` contains a map from program addresses to unwind info instead of `.eh_frame`
- Short unwind encodings are inlined, longer ones are stored in `.ARM.extab`
- EABI provides standard personality routines, or custom ones can be used
- GCC still uses `__gxx_personality_v0`, and the same LSDA encoding
- This kind of exception handling is also used in Symbian EPOC files

<table>
<thead>
<tr>
<th>Word 0</th>
<th>Word 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>+---------------------------------------------------------------+---------------------------------------------------------------+</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>+---------------------------------------------------------------+---------------------------------------------------------------+</td>
<td></td>
</tr>
<tr>
<td>31 30</td>
<td>0</td>
</tr>
</tbody>
</table>

- prel31 offset of the start of the table entry for this function
- The ex table entry itself encoded in 31bit

---

Credit: [https://wiki.linaro.org/KenWerner/Sandbox/libunwind](https://wiki.linaro.org/KenWerner/Sandbox/libunwind)
GCC: references

- http://www.airs.com/blog/archives/460 (.eh_frame)
- http://stackoverflow.com/questions/87220/
- apple/gcc: gcc/gcc-5493/libstdc++-v3/libsupc++/
- http://www.x86-64.org/documentation/abi.pdf
- Exploiting the Hard-Working DWARF (James Oakley & Sergey Bratus)
Conclusions

- RTTI is very useful for reverse engineering
- Helps discover not just class names but also inheritance hierarchy
- Necessary for dynamic_cast, so present in many complex programs
- Not removed by symbol stripping
- Parsing exception tables is necessary to uncover all possible execution paths
- Can improve static analysis (function boundaries etc.)
- Some implementation are very complicated and are likely to have bugs – a potential area for vulnerabilities
Thank you!

Questions?