What does the compiler actually do with my code?

An introduction to the C++ ABI

Filip Strömbäck

LINKÖPING UNIVERSITY

1 Introduction
2 What is an ABI?
3 What is in my binary?
4 Object layout
5 Function calls
6 Virtual functions
7 Exceptions

LINKÖPING UNIVERSITY

The topic for today

How are parts of C++ realized on x86 and AMD64?

- Object layout
- Function calls
- Virtual function calls
- Exceptions

Why?

If you know the implementation...

- ...you can reason about the efficiency of your solution
- ...you can see why some things are undefined behaviour
- (...you can abuse undefined behaviour and do *really* strange things)

Note: Everything discussed here is *highly* system specific, and most likely undefined behavior according to the standard!

How?

- Read the assembler output from the compiler!
  - `g++ -S <file>` or `cl /FAs <file>`
  - `objdump -d <program>`
  - In a debugger
  - Compiler Explorer
- Figure out why it does certain things:
  - OSDev Wiki (https://wiki.osdev.org/)
  - System V ABI (https://www.uclibc.org/docs/psABI-x86_64.pdf)
  - x86 instruction reference (http://ref.x86asm.net/)
- Lots of tinkering and thinking!

1 Introduction
2 What is an ABI?
3 What is in my binary?
4 Object layout
5 Function calls
6 Virtual functions
7 Exceptions

LINKÖPING UNIVERSITY

What is an ABI (Application Binary Interface)?

Specifies how certain aspects of a language are realized on a particular CPU

Language specification $+$ ABI $\Rightarrow$ compiler

Specifies:

- Size of built-in types
- **Object layout**
- **Function calls** (calling conventions)
- Exception handling
- Name mangling
- ...

Different systems use different ABIs

There are two major ABIs:

- System V ABI (Linux, MacOS on AMD64)
- Microsoft ABI (Windows)

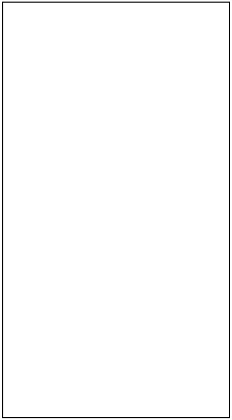
Variants for many systems:

- **x86**
- **AMD64**
- ARM
- ...

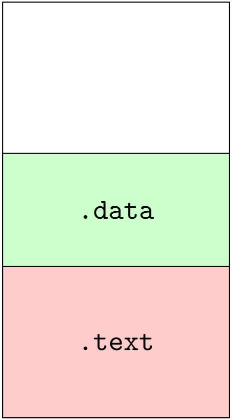
1 Introduction
2 What is an ABI?
3 What is in my binary?
4 Object layout
5 Function calls
6 Virtual functions
7 Exceptions

LINKÖPING UNIVERSITY

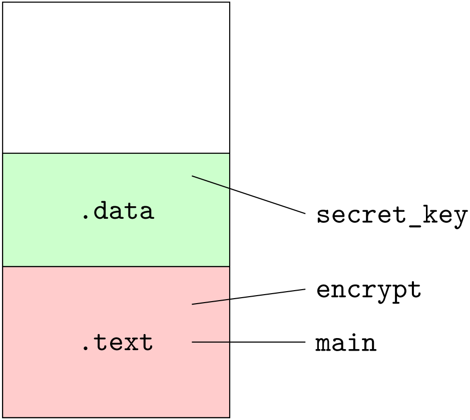
Memory



Memory



Memory



1 Introduction
2 What is an ABI?
3 What is in my binary?
4 Object layout
5 Function calls
6 Virtual functions
7 Exceptions

LINKÖPING UNIVERSITY

Integer types and endianness

```
char  a{0x08};
short b{0x1234};      // = 4660
int   c{0x00010203};  // = 66051
long  d{0x1101020304}; // = 73031353092
```

Integer types and endianness

```
char   a{0x08};
short  b{0x1234};        // = 4660
int    c{0x00010203};    // = 66051
long   d{0x1101020304};  // = 73031353092
```

Big endian (ARM)

a: 08

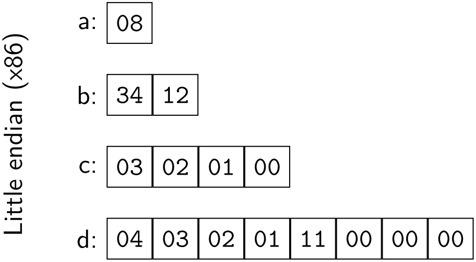
b: 12 34

c: 00 01 02 03

d: 00 00 00 11 01 02 03 04

Integer types and endianness

```
char  a{0x08};
short b{0x1234};      // = 4660
int   c{0x00010203};  // = 66051
long  d{0x1101020304}; // = 73031353092
```



The type system

The type system is not present in the binary! It just helps us to keep track of how to *interpret* bytes in memory!

```
struct foo {
  int a, b, c;
};

foo x{1, 2, 3};
int y[3] = {1, 2, 3};
short z[6] = {1, 0, 2, 0, 3, 0};
```

All look the same in memory!

Other types

- Each type has a *size* and an *alignment*
- Members are placed sequentially, respecting the alignment

Example:

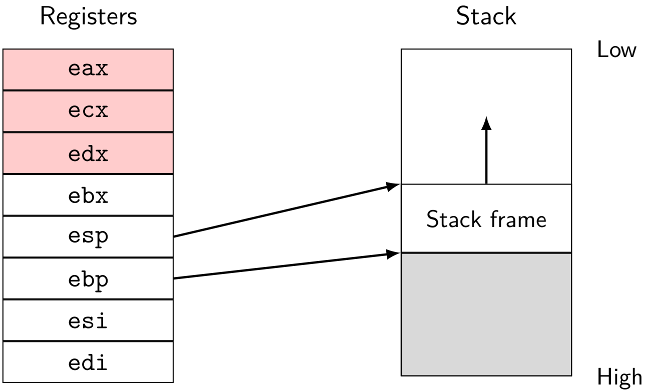
```
struct simple {
  int a{1};
  int b{2};
  int c{3};
  long d{100};
  int e{4};
};
```

<table>
<tr><td>a</td><td>b</td></tr>
<tr><td>c</td><td>padding</td></tr>
<tr><td colspan="2">d</td></tr>
<tr><td>e</td><td>padding</td></tr>
</table>

1 Introduction
2 What is an ABI?
3 What is in my binary?
4 Object layout
5 Function calls
6 Virtual functions
7 Exceptions

LINKÖPING UNIVERSITY

Starting simple – x86



The default on x86 – `cdecl`

```
int fn(int a, int b, int c);

int main() {
  int r = fn(1, 2, 3);
}
```

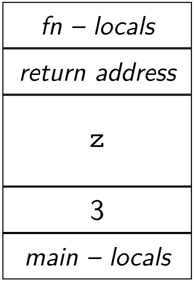
```
  push $3
  push $2
  push $1
  call fn
  add $12, %esp
  mov %eax, "r"
```

fn – locals
return address
1
2
3
main – locals

The default on x86 – `cdecl`

```
struct large { int a, b; };
int fn(large a, int b);
int main() {
  large z{ 1, 2 };
  int r = fn(z, 3);
}
```

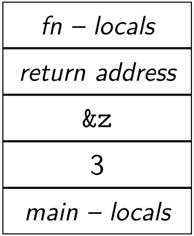
```
  push $3
  sub $8, %esp
  ;; initialize z at esp
  call fn
  add $12, %esp
  mov %eax, "r"
```



The default on x86 – cdecl

```
struct large { int a, b; };
int fn(large &a, int b);
int main() {
  large z{ 1, 2 };
  int r = fn(z, 3);
}
```

```
  push $3
  lea "z", %eax
  push %eax
  call fn
  add $8, %esp
  mov %eax, "r"
```



The default on x86 – cdecl

```
struct large { int a, b; };
large fn(int a);

int main() {
  large z = fn(10);
}
```

```
  push $10
  lea "z", %eax
  push %eax
  call fn
  add $8, %esp
```

fn – locals
return address
10
result address
main – locals

The default on x86 – cdecl

```
struct large { int a, b; };
large *fn(large *result, int a);

int main() {
  large z = fn(10);
}
```

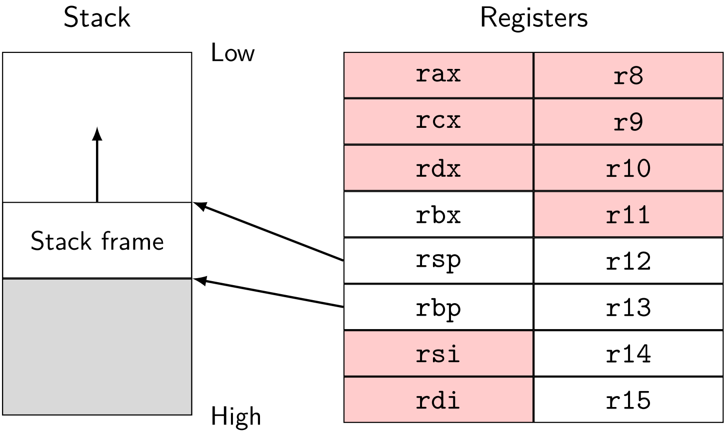
```
  push $10
  lea "z", %eax
  push %eax
  call fn
  add $8, %esp
```

fn – locals
return address
10
result address
main – locals

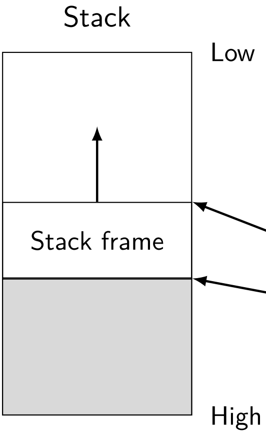
More advanced – AMD64 (SystemV)

This is where the fun begins!

More advanced – AMD64 (SystemV)



More advanced – AMD64 (SystemV)



Registers

	rax	5	r8
4	rcx	6	r9
3	rdx		r10
	rbx		r11
	rsp		r12
	rbp		r13
2	rsi		r14
1	rdi		r15

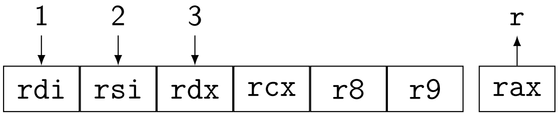
Rules (simplified)

1. If a parameter has a copy constructor or a destructor:
   - Pass by hidden reference
2. If a parameter is larger than 4*8 bytes
   - Pass in memory
3. If a parameter uses more than 2 integer registers
   - Pass in memory
4. Otherwise
   - Pass in appropriate registers (integer/floating-point)

AMD64 (SystemV)

```
int fn(int a, int b, int c);

int main() {
  int r = fn(1, 2, 3);
}
```

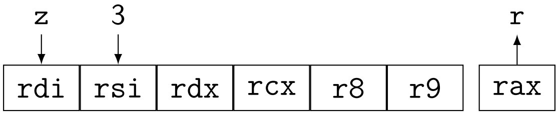
```
mov $1, %edi
mov $2, %esi
mov $3, %edx
call fn
mov %rax, "r"
```



AMD64 (SystemV)

```
struct large { int a, b; };
int fn(large a, int b);
int main() {
  large z{ 1, 2 };
  int r = fn(z, 3);
}
```

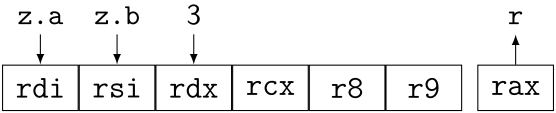
```
mov "z", %rdi
mov $3, %rsi
call fn
mov %rax, "r"
```



AMD64 (SystemV)

```
struct large { long a, b; };
int fn(large a, long b);
int main() {
  large z{ 1, 2 };
  int r = fn(z, 3);
}
```

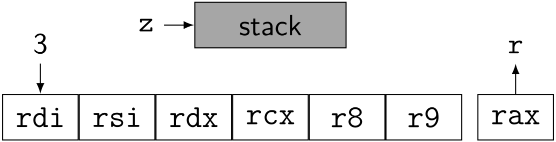
```
mov "z", %rdi
mov $3, %rsi
call fn
mov %rax, "r"
```



AMD64 (SystemV)

```
struct large { long a, b, c; };
int fn(large a, long b);
int main() {
  large z{ 1, 2, 3 };
  int r = fn(z, 3);
}
```

```
push "z.c"
push "z.b"
push "z.a"
mov $3, rdi
call fn
mov %rax, "r"
```

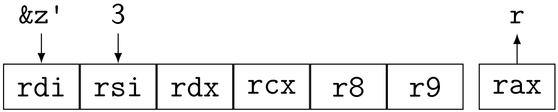


AMD64

```
struct large { /*...*/ };
int fn(large a, long b);
int main() {
  large z{ 1, 2 };
  int r = fn(z, 3);
}
```

```
;; Copy z into z'
lea "z'", %rdi
mov $3, %rsi
call fn
mov %rax, "r"
```

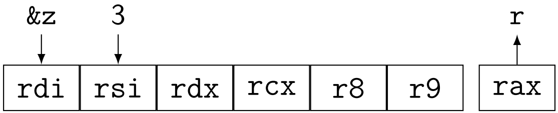
`large` is not trivially copiable, has a destructor or a vtable



AMD64 (SystemV)

```
struct large { int a, b; };
int fn(large &a, int b);
int main() {
  large z{ 1, 2 };
  int r = fn(z, 3);
}
```

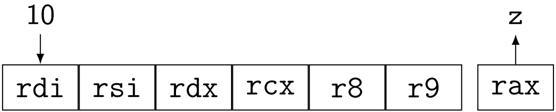
```
lea "z", $rdi
mov $3, %rsi
call fn
mov %rax, "r"
```



AMD64 (SystemV)

```c
struct large { int a, b; };
large fn(int a);

int main() {
  large z = fn(10);
}
```

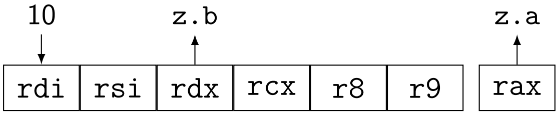
```
mov $10, %rdi
call fn
mov %rax, "z"
```



AMD64 (SystemV)

```
struct large { long a, b; };
large fn(int a);

int main() {
  large z = fn(10);
}
```

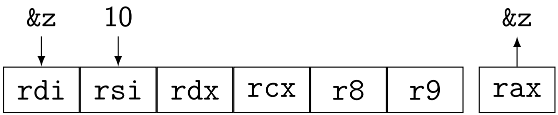
```
mov $10, %rdi
call fn
mov %rax, "z"
mov %rdx, "z"+8
```



AMD64 (SystemV)

```c
struct large { long a, b, c; };
large fn(int a);

int main() {
  large z = fn(10);
}
```

```
mov $10, %rdi
call fn
mov %rax, "z"
mov %rdx, "z"+8
```



Conclusions

- Passing primitives by value is cheap
- Passing simple types by value is cheap (sometimes cheaper than passing multiple parameters)
  - As long as they are trivially copiable and destructible
  - As long as they are below about 4 machine words or about 64 bytes
- Returning small simple types by value is cheap on AMD64, even without RVO
- Types that are not trivially copiable are more cumbersome: pass them by reference

1 Introduction
2 What is an ABI?
3 What is in my binary?
4 Object layout
5 Function calls
6 Virtual functions
7 Exceptions

LINKÖPING UNIVERSITY

Scenario

```
struct base {
  virtual ~base() = default;

  int data{0x1020};

  virtual void fun(int x) = 0;
};

void much_fun(base &x) {
  x.fun(100);
}
```

How do we know what to call here?

Virtual function tables – vtables (SystemV)

Idea: Put some type info in the objects!

This is called a *virtual function table* or *vtable*:

Offset	Symbol
0	`derived::~derived()`
8	`derived::~derived()`
16	`derived::fun(int)`

Note: More complex for multiple and virtual inheritance!

Virtual function tables – vtables (SystemV)

Idea: Put some type info in the objects!

This is called a *virtual function table* or *vtable*:

Offset	Symbol	
0	`derived::~derived()`	doesn't call `delete`
8	`derived::~derived()`	calls `delete`
16	`derived::fun(int)`	

Note: More complex for multiple and virtual inheritance!

Virtual dispatch

```
void much_fun(base &x) {
  x.fun(100);
}
```

```
mov "x", %rdi          ; Put x in a register
mov (%rdi), %rax       ; Read vtable
mov 16(%rax), %rax     ; Read slot #2
mov $100, %rsi         ; Add parameter
call *%rax             ; Call the function
```

Pointers to members

```
class MyClass {
  virtual int virtual_member() { return 1; }
};
class DerivedClass : public MyClass {
  int virtual_member() override { return 1; }
};
int main() {
  auto ptr = &MyClass::virtual_member();
  DerivedClass c;
  return (c.*ptr)(); // Which one is called?
}
```

Pointers to members (SystemV)

Function pointers are fairly straight forward... What about pointers to members?

```
plain_ptr  x = &MyClass::static_member;
member_ptr y = &MyClass::normal_member;
member_ptr z = &MyClass::virtual_member;
```

Let's look at their sizes:

```
sizeof(x) == ?;
sizeof(y) == ?;
sizeof(z) == ?;
```

Pointers to members (SystemV)

Function pointers are fairly straight forward... What about pointers to members?

```
plain_ptr  x = &MyClass::static_member;
member_ptr y = &MyClass::normal_member;
member_ptr z = &MyClass::virtual_member;
```

Let's look at their sizes:

```
sizeof(x) == sizeof(void *);
sizeof(y) == sizeof(void *)*2;
sizeof(z) == sizeof(void *)*2;
```

What?

Let's look at the code!

```
call_member:
  mov "ptr.ptr", %rax
  and $1, %rax
  test %rax, %rax
  jne .L12
  mov "ptr.ptr", %rax
  jmp .L13
```

```
.L12:
  mov "ptr.offset", %rax
  add "&c", %rax
  mov (%rax), %rdx
  mov "ptr", %rax
  mov (%rax+%rdx-1), %rax
.L13:
  mov "ptr.offset", %rdi
  add "&c", %rdi
  call *%rax
```

Let's look at the code!

```
struct member_ptr {
  // Pointer or vtable offset
  size_t ptr;

  // Object offset
  size_t offset;
};
```

Let's look at the code!

```
void member_call(MyClass &c, member_ptr ptr) {
  void *obj = (void *)&c + ptr.offset;
  void *target = ptr.ptr;
  // Is it a vtable offset?
  if (ptr.ptr & 0x1) {
    void *vtable = *(void **)obj;
    target = *(size_t *)(vtable + ptr - 1);
  }
  // Call the function!
  (obj->*target)();
}
```

Pointers to members

- This is realized differently on x86 on Windows
  - There, *thunks* are used instead.
- This is one of the reasons why you can't just cast member function pointers to `void *`!
- Pointers to member variables are simpler, they're just the offset of the variable.

What about typeid?

```
const type_info &find_typeinfo(base &var) {
  return typeid(var);
}
```

How does the compiler know the actual type of var?

Let's look at the code!

```
_Z13find_typeinfoR4base:
    push    %rbp              ; Function prolog
    mov     %rsp, %rbp
    mov     %rdi, %rax        ; First parameter
    movq    (%rax), %rax
    mov     -8(%rax), %rax
    pop     %rbp              ; Function epilog
    ret
```

Let’s look at the code!

```
_Z13find_typeinfoR4base:
    push    %rbp                ; Function prolog
    mov     %rsp, %rbp
    mov     %rdi, %rax          ; First parameter
    movq    (%rax), %rax
    mov     -8(%rax), %rax
    pop     %rbp                ; Function epilog
    ret
```

There is something at offset -8 of the vtable!

A closer look at the vtable

```
_ZTV7derived:
  .quad 0
  .quad _ZTI7derived
  .quad _ZN7derivedD1Ev
  .quad _ZN7derivedD0Ev
  .quad _ZN7derived3funEi
```

A closer look at the vtable

Offset	Symbol	
-16	(offset)	
-8	`typeinfo for derived`	
0	`derived::~derived()`	doesn't call `delete`
8	`derived::~derived()`	calls `delete`
16	`derived::fun(int)`	

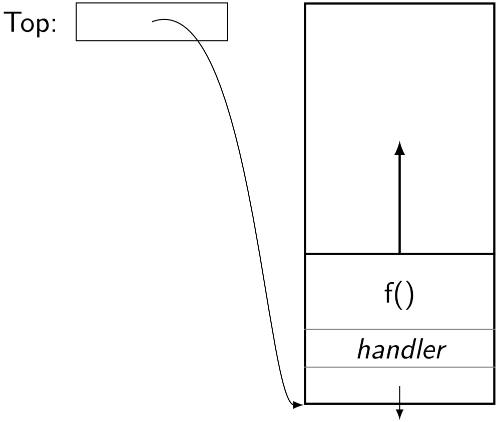
1 Introduction
2 What is an ABI?
3 What is in my binary?
4 Object layout
5 Function calls
6 Virtual functions
7 Exceptions

SEH – x86, Win32

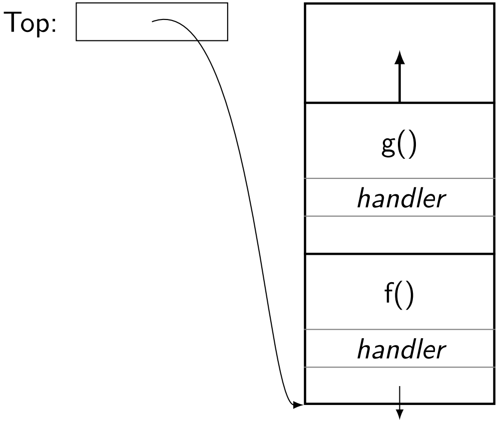
Idea: Functions in need of handling exceptions store an entry in a per-thread list of handlers. Essentially:

```
void function() {
  eh_entry entry;
  entry.next = eh_stack;
  entry.handler = &handle_exception;
  eh_stack = &entry;

  // Code as normal

  eh_stack = entry.next;
}
```

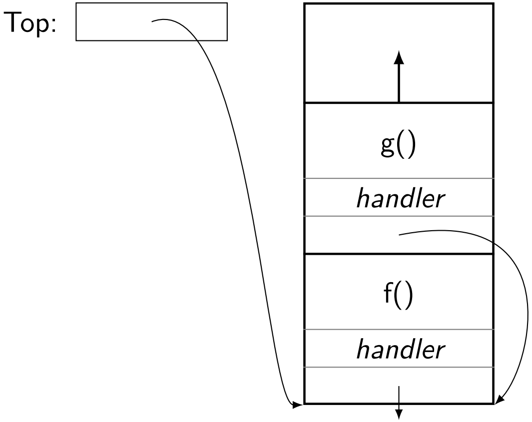
SEH – x86, Win32



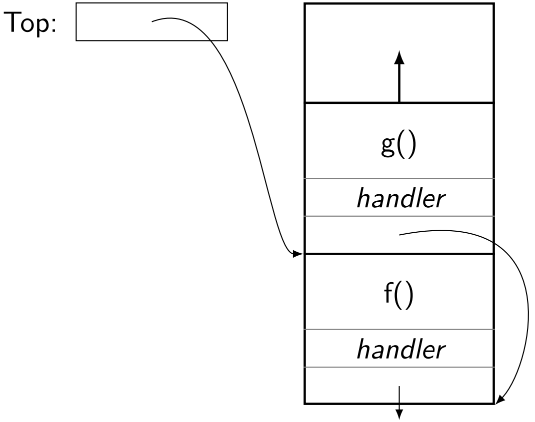
SEH – x86, Win32



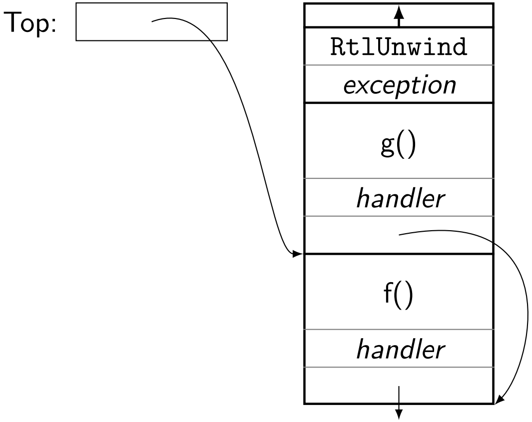
SEH – x86, Win32



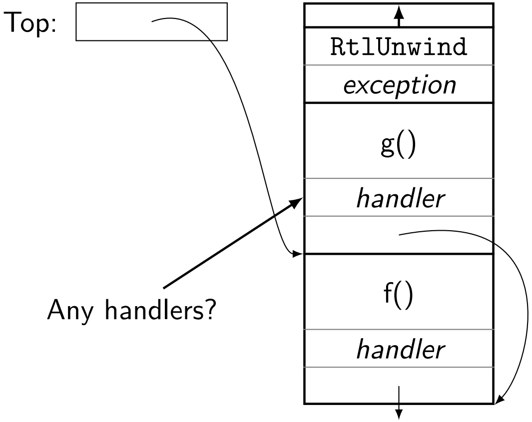
SEH – x86, Win32



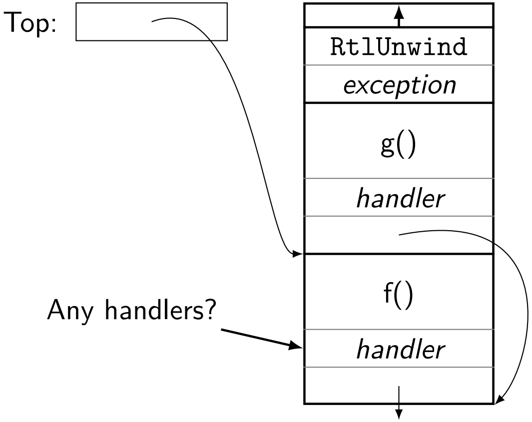
SEH – x86, Win32



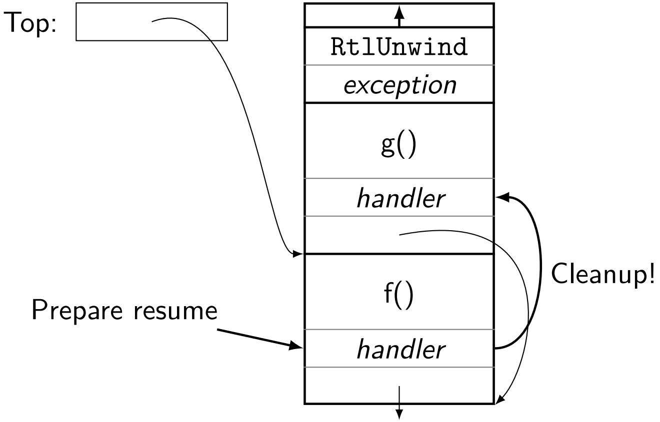
SEH – x86, Win32



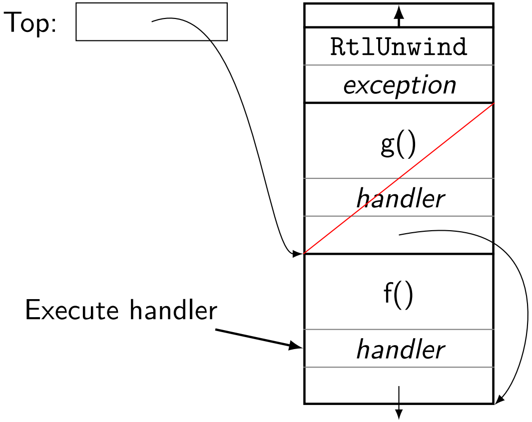
SEH – x86, Win32



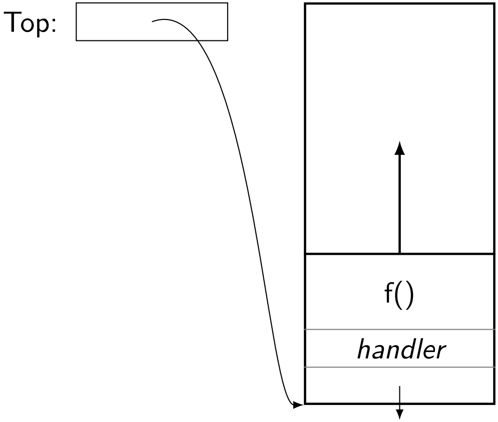
SEH – x86, Win32



SEH – x86, Win32



SEH – x86, Win32



What was thrown?

Table of `typeinfo`-objects in metadata:

```
class A {};

class B :
  public A {};
class C :
  public B {};

void f() {
  try {
    throw C();
  } catch (const A &) {}
}
```

What was thrown?

Table of `typeinfo`-objects in metadata:

```
class A {};

class B :
  public A {};
class C :
  public B {};

void f() {
  try {
    throw C();
  } catch (const A &) {}
}
```

```
typeinfo *options[] = {
  &typeid(C),
  &typeid(B),
  &typeid(A),
}
```

SEH – x86, Win32

Benefits:

- Language agnostic – almost no pre-defined data structures
- Straightforward unwinding

Drawbacks:

- Overhead in all cases – not only when throwing exceptions
- Storing function pointers on the stack...

For AMD64, a solution similar to DWARF is used

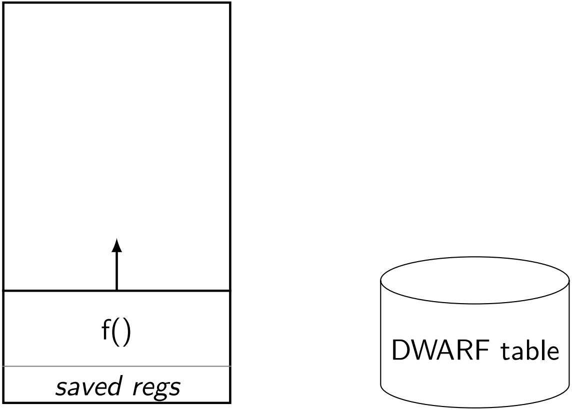
DWARF – System V

Idea: Store unwinding information in big tables somewhere!

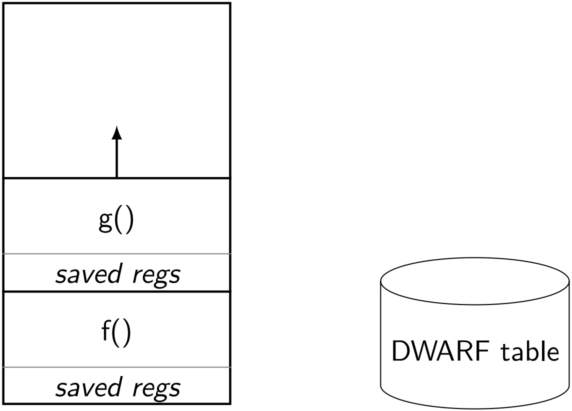
Each function has an entry containing:

- Unwinding information – How to undo any changes to the stack and/or registers done by the function at any point in the function.
- Personality function – Like in SEH, function that determines if a particular exception is handled and hanles cleanup.
- Additonal data – Any additional information required by the personality function.

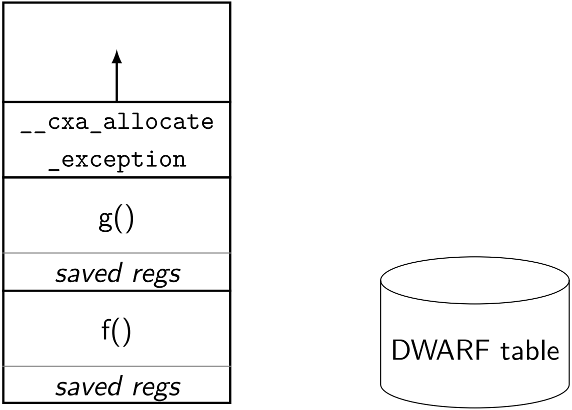
DWARF – SystemV



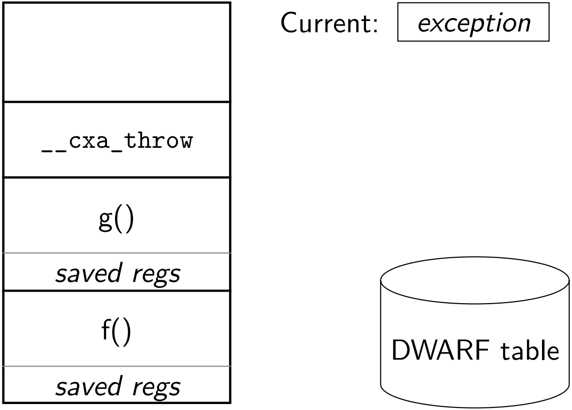
DWARF – SystemV



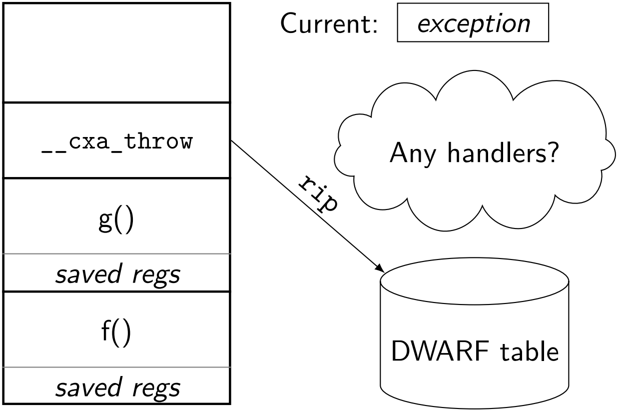
DWARF – SystemV



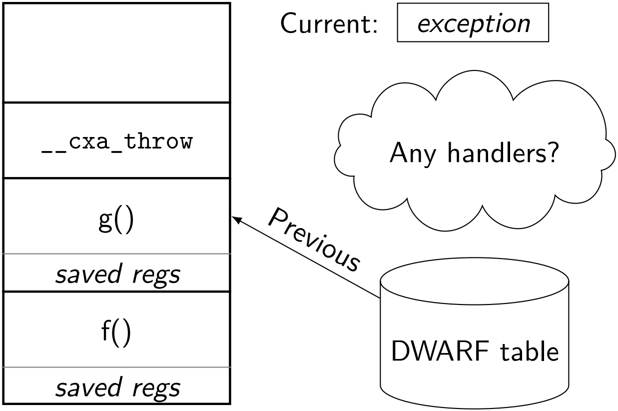
DWARF – SystemV



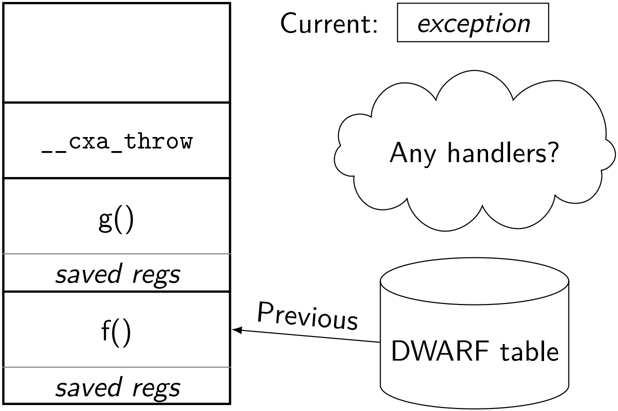
DWARF – SystemV



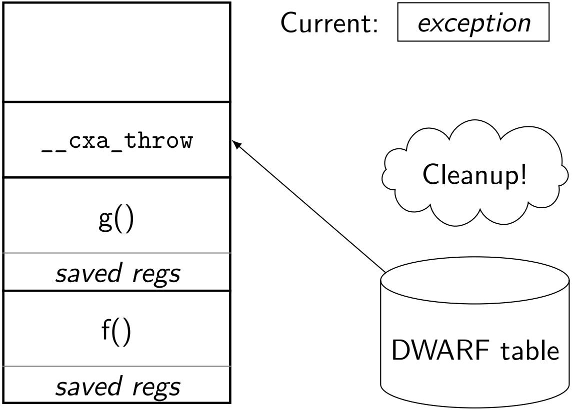
DWARF – SystemV



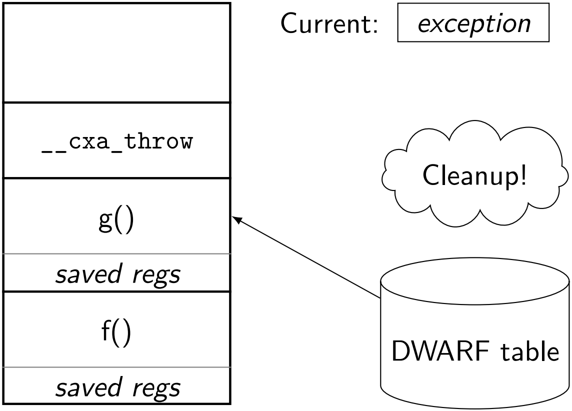
DWARF – SystemV



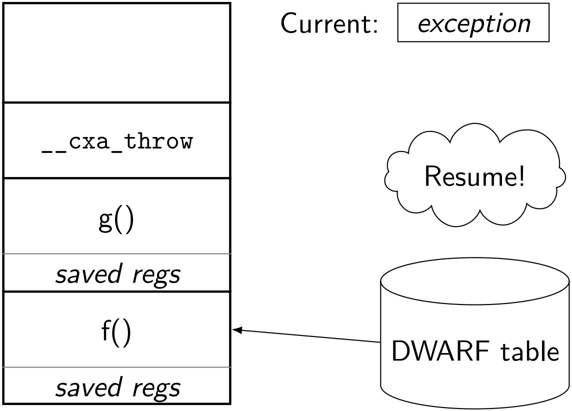
DWARF – SystemV



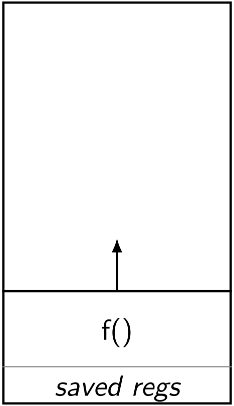
DWARF – SystemV



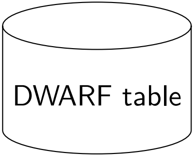
DWARF – SystemV



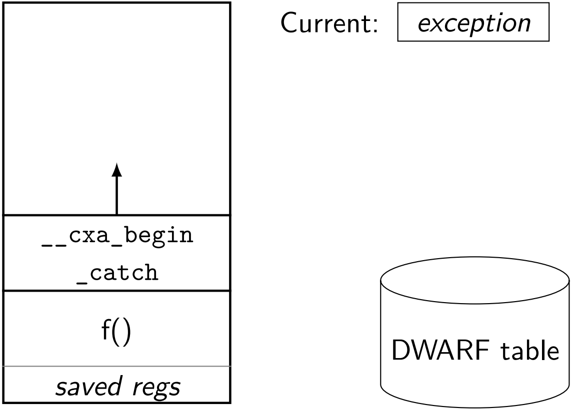
DWARF – SystemV



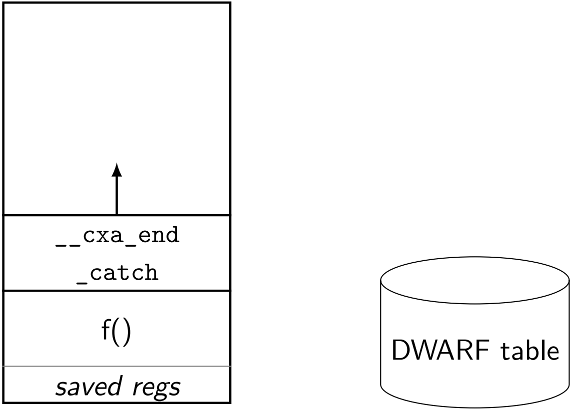
Current: *exception*



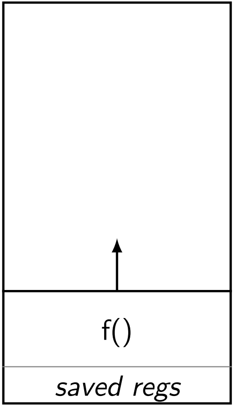
DWARF – SystemV

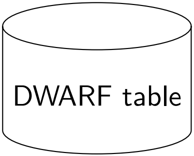


DWARF – SystemV



DWARF – SystemV





What was thrown?

Well, `std::typeinfo` is a polymorphic class...

https://itanium-cxx-abi.github.io/cxx-abi/abi.html

```
bool matches(_Unwind_Exception *data) {
  std::type_info *type = /* data->type */;
  // perhaps
  return __dynamic_cast(..., type, &typeid(A), -1);

  // not in the ABI:
  return typeid(A).__do_catch(type, ...);
}
```

DWARF - System V

Benefits:

- Low cost (almost zero) unless exceptions are actually thrown
- Difficult to utilize during buffer overflows

Drawbacks:

- Most functions need to provide unwind information (difficult when doing JIT compilation)
- High cost of actually throwing exceptions

Some interesting functions here:
`https://libcxxabi.llvm.org/spec.html`

Conclusions

- There are many ways of implementing exceptions
- Most are expensive, hopefully only when used!
- Don't use exceptions for normal control-flow!

Filip Strömbäck

www.liu.se

LINKÖPING UNIVERSITY