What does the compiler actually do with my code?

An introduction to the C++ ABI

Filip Strömbäck

LINKÖPING UNIVERSITY

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 -masm=intel <file>` or `cl /FAs <file>`
  - `objdump -d -M intel <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!

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 ⇒ 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
- ...

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{0x110102030 4}; // = 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
```

Little endian (x86)

a: | 08 |

b: | 34 | 12 |

c: | 03 | 02 | 01 | 00 |

d: | 04 | 03 | 02 | 01 | 11 | 00 | 00 | 00 |

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};
};
```

a	b
c	*padding*
d	
e	*padding*

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!

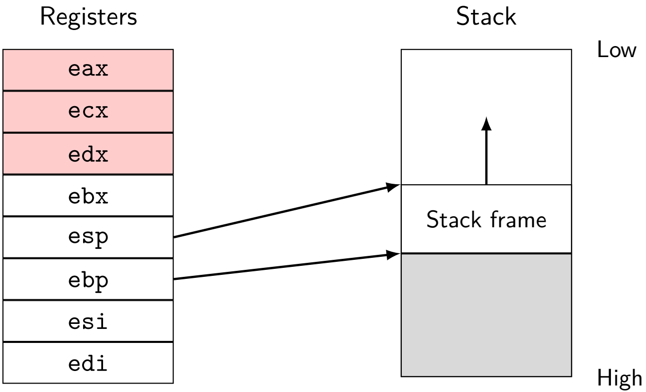
```
struct foo {
  int a, b, c;
};

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!

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);
}
```

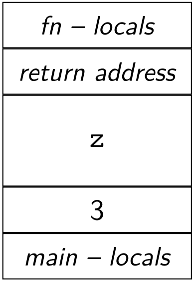
```
  push 3
  push 2
  push 1
  call fn
  add esp, 12
  mov "r", eax
```

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);
}
```

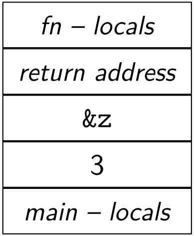
```
  push 3
  sub esp, 8
  ;; initialize z at esp
  call fn
  add esp, 12
  mov "r", eax
```



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 10
  lea eax, "z"
  push eax
  call fn
  add esp, 8
  mov "r", eax
```

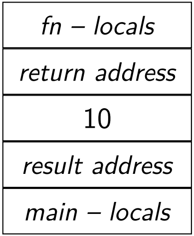


The default on x86 – `cdecl`

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

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

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



The default on x86 – cdecl

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

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

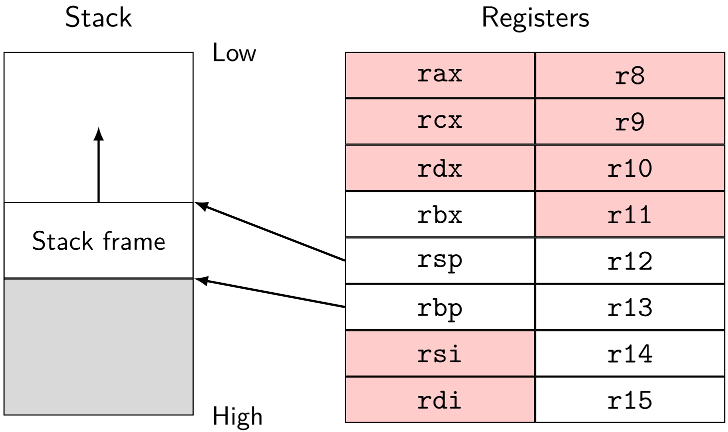
```
  push 10
  lea eax, "z"
  push eax
  call fn
  add esp, 8
```

fn – locals
return address
10
result address
main – locals

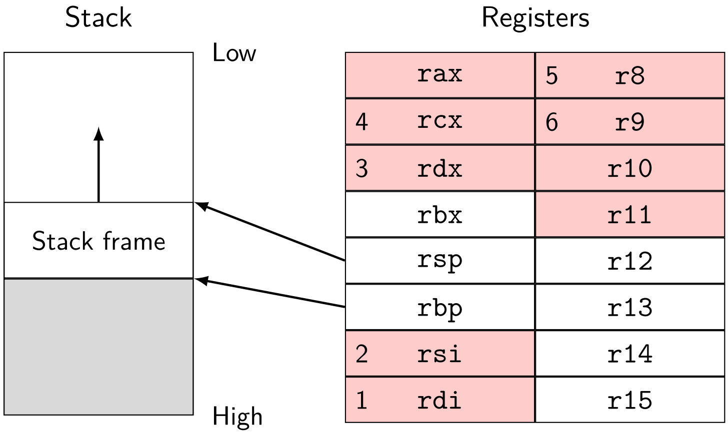
More advanced – AMD64

This is where the fun begins!

More advanced – AMD64



More advanced – AMD64



Rules (simplified)

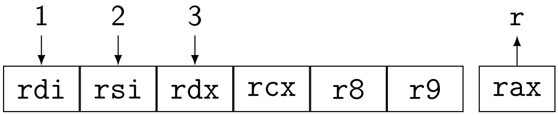
1. If a parameter has a copy constructor or a destructor:
   - Pass by hidden reference
2. If a parameter is larger than 4*8 bytes
   - Pass in memory
3. If a parameter uses more than 2 integer registers
   - Pass in memory
4. Otherwise
   - Pass in appropriate registers (integer/floating-point)

AMD64

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

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

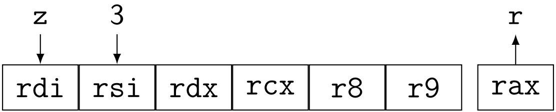
```
mov edi, 1
mov esi, 2
mov edx, 3
call fn
mov "r", rax
```



AMD64

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

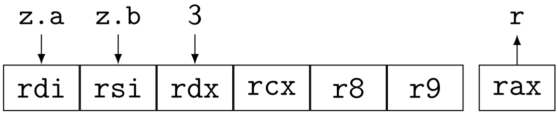
```
mov rdi, "z"
mov rsi, 3
call fn
mov "r", rax
```



AMD64

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

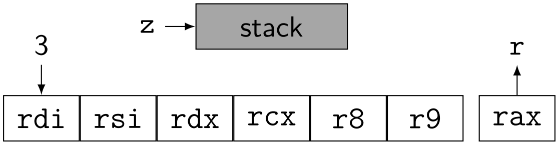
```
mov rdi, "z"
mov rsi, 3
call fn
mov "r", rax
```



AMD64

```
struct large { long a, b, c; };
int fn(large a, long b);
int main() {
  large z{ 1, 2, 3 };
  int r = fn(z, 4);
}
```

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

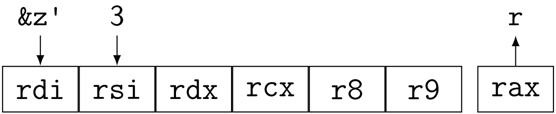


AMD64

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

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

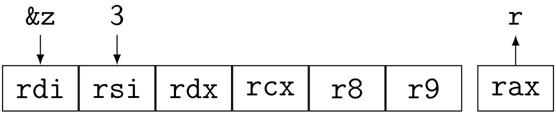
large is not trivially copiable, has a destructor or a vtable



AMD64

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

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

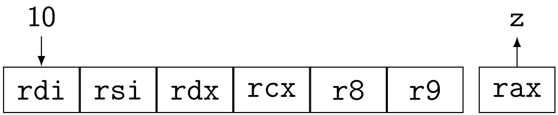


AMD64

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

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

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

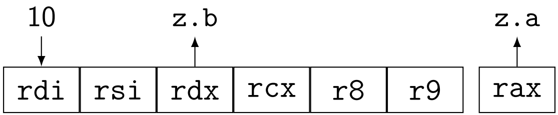


AMD64

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

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

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

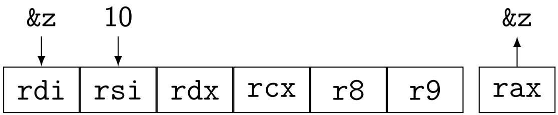


AMD64

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

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

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



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

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

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

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 rdi, "x"         ; Put x in a register
mov rax, [rdi]       ; Read vtable
mov rax, [rax+16]    ; Read slot #2
mov rsi, 100         ; Add parameter
call [rax]           ; Call the function
```

Pointers to members

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

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 rax, "ptr.ptr"
  and rax, 1
  test rax, rax
  jne .L12
  mov rax, "ptr.ptr"
  jmp .L13
```

```
.L12:
  mov rax, "ptr.offset"
  add rax, "&c"
  mov rdx, [rax]
  mov rax, "ptr"
  mov rax, [rax+rdx-1]
.L13:
  mov rdi, "ptr.offset"
  add rdi, "&c"
  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     rbp, rsp
    mov     rax, rdi                ; First parameter
    mov     rax, QWORD PTR [rax]
    mov     rax, QWORD PTR [rax-8]
    pop     rbp                     ; Function epilog
    ret
```

Let's look at the code!

```
_Z13find_typeinfoR4base:
    push    rbp                 ; Function prolog
    mov     rbp, rsp
    mov     rax, rdi            ; First parameter
    mov     rax, QWORD PTR [rax]
    mov     rax, QWORD PTR [rax-8]
    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)`	

LINKÖPING UNIVERSITY

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

When an exception is thrown:

1. Traverse `eh_stack` and ask each `handler` if they handle the current exception.
2. Traverse `eh_stack` again and ask each `handler` to perform any cleanup required.
3. Continue execution as specified by `handler` in the function that caught the exception.

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 - System V

Exception handling works similar to SEH, however traversing the stack requires:

1. Find the current function's entry by binary searching the tables
2. Call the personality function and determine what to do next
3. Interpret the "program" describing how to undo the functions manipulation of the stack and registers and undo the changes
4. Repeat until a handler is found

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

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