C++ ABI: the only thing that is more important than performance

iCSC 2020

Nis Meinert

German Aerospace Center

Deutsches Zentrum für Luft- und Raumfahrt

CERN School of Computing

Reading x86-64 Assembly

…for fun and profit

Function Prologue & Epilogue

- Few lines of code at the beginning (*prologue*) and end (*epilogue*) of a function, which **prepares** (and eventually restores)
  - the **stack** and
  - **registers**
- Not part of assembly: **convention** (defined & interpreted differently by different OS and compilers)

Prologue

```
push rbp      ; rbp: frame pointer
mov rbp, rsp ; rsp: stack pointer
sub rsp, N
```

alternatively

```
enter N, 0
```

(reserve N bytes on stack for local use)

Epilogue

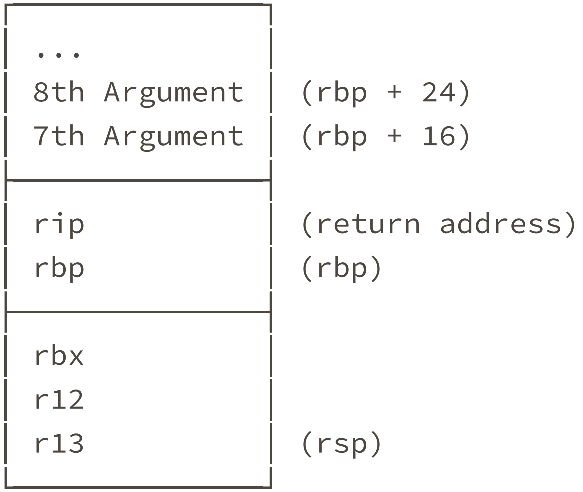
```
mov rsp, rbp
pop rbp
ret
```

alternatively

```
leave
ret
```

Stack frame for function call

- → `CALL` = `PUSH` *address of next instruction* + `JMP` *target*
- → `RET` pops return address and transfers control there
- → pass arguments 1 …6 in registers (`rsi`, `rdx`, …)



```
 ...
 8th Argument    (rbp + 24)
 7th Argument    (rbp + 16)

 rip             (return address)
 rbp             (rbp)

 rbx
 r12
 r13             (rsp)
```

(stack frame for function call with 8 arguments and local registers `rbx`, `r12` and `r13`)

Reading assembly for fun and profit

```cpp
int f(int x, int y, int z) {
    int sum = x + y + z;
    return sum;
}
```

godbolt.org/z/MaWcP9

```
# g92 -O0
  | _Z1fiii:
 1|   push rbp
 1|   mov rbp, rsp
 1|   mov DWORD PTR [rbp-20], edi
 1|   mov DWORD PTR [rbp-24], esi
 1|   mov DWORD PTR [rbp-28], edx
 2|   mov edx, DWORD PTR [rbp-20]
 2|   mov eax, DWORD PTR [rbp-24]
 2|   add edx, eax
 2|   mov eax, DWORD PTR [rbp-28]
 2|   add eax, edx
 2|   mov DWORD PTR [rbp-4], eax
 3|   mov eax, DWORD PTR [rbp-4]
 4|   pop rbp
 4|   ret
```

godbolt.org/z/MaWcP9

Reading assembly for fun and profit

```
1  int f(int x) {
2      return x + 1;
3  }
4
5  int g(int x) {
6      return f(x + 2);
7  }
```

godbolt.org/z/87GK4q

```
# g92 -O0
  | _Z1fi:
 1|   push rbp
 1|   mov rbp, rsp
 1|   mov DWORD PTR [rbp-4], edi
 2|   mov eax, DWORD PTR [rbp-4]
 2|   add eax, 1
 3|   pop rbp
 3|   ret
  | _Z1gi:
 5|   push rbp
 5|   mov rbp, rsp
 5|   sub rsp, 8
 5|   mov DWORD PTR [rbp-4], edi
 6|   mov eax, DWORD PTR [rbp-4]
 6|   add eax, 2
 6|   mov edi, eax
 6|   call _Z1fi
 7|   leave
 7|   ret
```

godbolt.org/z/87GK4q

Reading assembly for fun and profit

```cpp
1  void side_effect();
2
3  int f(int x) {
4      auto a = x;
5      side_effect();
6      return a - x;
7  }
```

godbolt.org/z/5xq5n5

```
# g92 -O0
  | _Z1fi:
 3|   push rbp
 3|   mov rbp, rsp
 3|   sub rsp, 32
 3|   mov DWORD PTR [rbp-20], edi
 4|   mov eax, DWORD PTR [rbp-20]
 4|   mov DWORD PTR [rbp-4], eax
 5|   call _Z11side_effectv
 6|   mov eax, DWORD PTR [rbp-4]
 6|   sub eax, DWORD PTR [rbp-20]
 7|   leave
 7|   ret
```

godbolt.org/z/5xq5n5

Name mangling: C++ vs C

```cpp
1  int f(int x) {
2      return x * x;
3  }
4
5  extern "C" int g(int x) {
6      return x * x;
7  }
```

godbolt.org/z/cj7bqx

```
# g92 -O0
  | _Z1fi:
 1|   push rbp
 1|   mov rbp, rsp
 1|   mov DWORD PTR [rbp-4], edi
 2|   mov eax, DWORD PTR [rbp-4]
 2|   imul eax, eax
 3|   pop rbp
 3|   ret
  | g:
 5|   push rbp
 5|   mov rbp, rsp
 5|   mov DWORD PTR [rbp-4], edi
 6|   mov eax, DWORD PTR [rbp-4]
 6|   imul eax, eax
 7|   pop rbp
 7|   ret
```

godbolt.org/z/cj7bqx

Name mangling: C++ vs C

```cpp
1  int f(int x) {
2      return x * x;
3  }
4
5  extern "C" int g(int x) {
6      return x * x;
7  }
```

godbolt.org/z/cj7bqx

Why?

- → overloading
- → namespaces
- → templating

(Name of function doesn't suffice to resolve JMP location)

```
# g92 -O0
  | _Z1fi:
 1|   push rbp
 1|   mov rbp, rsp
 1|   mov DWORD PTR [rbp-4], edi
 2|   mov eax, DWORD PTR [rbp-4]
 2|   imul eax, eax
 3|   pop rbp
 3|   ret
  | g:
 5|   push rbp
 5|   mov rbp, rsp
 5|   mov DWORD PTR [rbp-4], edi
 6|   mov eax, DWORD PTR [rbp-4]
 6|   imul eax, eax
 7|   pop rbp
 7|   ret
```

godbolt.org/z/cj7bqx

Name mangling in C++

```cpp
void f(int) {}

void f(double) {}

namespace my_fancy_namespace {
void f(int) {}
} // my_fancy_namespace
```

godbolt.org/z/jWY14x

```
# g92 -O2
   | _Z1fi:
  1|   ret
   | _Z1fd:
  3|   ret
   | _ZN18my_fancy_namespace1fEi:
   |   ret
```

godbolt.org/z/jWY14x

→ C++ does not standardize name mangling

→ *Annotated C++ Reference Manual* even actively discourages usage of common mangling schemes. (Prevent linking when other aspects of ABI are incompatible.)

What is ABI?

What is ABI (***A**pplication **B**inary **I**nterface*)?

Specifies interaction of functions and types across TUs[†] (translation units)

- → Platform-specific (*e.g.*, Linux)
- → Vendor-specified (*e.g.*, gcc)
- → not controlled by WG21

Similar to a binary network protocol (Titus Winters)

Photo by Spencerian at `en.wikipedia.org` (2005)

† *TU*: ultimate input to the compiler from which an object file is generated (*i.e.*, typically the `.cpp` file)

What is ABI (***A**pplication **B**inary **I**nterface*)?

Specifies interaction of functions and types across TUs[†] (translation units) covering:

→ Name mangling of functions

→ Name mangling of types

→ `sizeof` and alignment of objects

→ Bytes semantics of the binary representation of objects

→ Calling convention

Similar to a binary network protocol (Titus Winters)

Photo by Spencerian at `en.wikipedia.org` (2005)

[†] *TU*: ultimate input to the compiler from which an object file is generated (*i.e.*, typically the `.cpp` file)

Why should I care?

…do you depend on any pre-compiled shared library?

Why should I care?

Why should I care?

- → **Linking** different TUs requires usage of same ABI
- → Typically a problem at API boundaries when combining TUs (*e.g.*, shared libraries) that were compiled at different **time**s
- → Similar to binary network protocols: ABI tells you how to interpret bytes

Why should I care? $\Leftrightarrow$ Why do network protocols have versions?

(Problem: ABI does not encode version number)

ABI: the problem

ABI does not encode version number

- → **Q**: How to check if a given TU uses a compatible ABI?
- → **A**: You can’t!
- → What happens if ABI is incompatible?
  - (a) Linking fails during compile time (good)
  - (b) Program spectacularly dies during run time (bad)
- → Why isn’t this a common problem?
  - → Itanium ABI is mostly stable since C++11

History

ABI breakage of `std::string`

- Before C++11: libstdc++ relied on copy-on-write (COW)
- C++11 disallows COW
  - fewer indirections
  - short string optimization (SSO)
- Problem: passing COW string to impl that expects SSO **may link** (same mangled name)!
  - one word passed
  - three words read
- *Solution*[†]: gcc changed mangled name

```cpp
// until C++11
struct string {
    struct control_block {
        /* ... */
    };
    control_block *data;
};

// since C++11
struct string {
    char *data;
    std::size_t size;
    std::size_t capacity;
}
```

godbolt.org/z/KM5Tvq

↪ Take-away for compiler vendors: ABI break was a huge disaster

† RHEL 7 still uses old `std::string` ABI to provide compatibility for older `.so`

Quiz time