ABIs and (Dynamic) Linking in C++

Filip Strömbäck

LINKÖPING UNIVERSITY

The Topic for Today

How are C++-features implemented?

In particular:

- Object layout
- Function calls
- Linking

Focus on Linux AMD64, but with hints of Windows, x86-32, and ARM64

The Topic for Today

How are C++-features implemented?

In particular:

- Object layout
- Function calls
- Linking

Focus on Linux AMD64, but with hints of Windows, x86-32, and ARM64

This is *outside* the C++ standard, compiler may do as it wishes.

The Topic for Today

How are C++-features implemented?

In particular:

- Object layout
- Function calls
- Linking

Focus on Linux AMD64, but with hints of Windows, x86-32, and ARM64

This is *outside* the C++ standard, compiler may do as it wishes.

However: different compilers want to be compatible, to use pre-compiled libraries, provide FFI, ...

$\Rightarrow$ they follow the **ABI**

Using different compilers (and even languages) requires a bit of care

The Topic for Today

How are C++-features implemented?

In particular:

- Object layout
- Function calls
- Linking

Focus on Linux AMD64, but with hints of Windows, x86-32, and ARM64

Why?

If you know the implementation...

- ...you can reason about the cost of language features
- ...you can understand why certain things are undefined
- ...you can ~~abuse~~ use the implementation to do fun things

A Note About the Examples

Many code examples are not defined according to the C++ standard. They do, however, fall within the realms of the ABI and the POSIX standard, and are likely to work.

Don't use the approaches here without *understanding* them well, the risks involved, and *isolating* them for *when* they break.

The code examples will be available on the course webpage if you wish to experiment with them.

How to Investigate Further?

Explore by reading the output from the compiler:

- `g++ -S <file>` or `cl /FAs <file>`
- `objdump -d <program>`
- `readelf -a`
- In a debugger (e.g. gdb with command `disas`)
- Compiler Explorer

Understand what falls within the specification:

- OSDev Wiki (`https://wiki.osdev.org/System_V_ABI`)
- System V ABI:
  - AMD64: `https://www.uclibc.org/docs/psABI-x86_64.pdf`
  - ARM: `https://github.com/ARM-software/abi-aa/tree/main`

What is an API (Application Programming Interface)?

Set of operations with well-defined behavior, e.g. header files in C++

Answers the question: How should I write my code to use some library?

```
class Data {
public:
  int a;
  double b;
  Data();
};
void print_val(Data d);
void print_cr(const Data &d);
```

- Specifies how to create `Data` and how to call `print_val`: (`print_val(Data())`)
- Needs some description of the expected behavior as well
- Does *not* specify how they work internally

$\Rightarrow$ Allows for different implementations

What is an API (Application Programming Interface)?

Set of operations with well-defined behavior, e.g. header files in C++

Answers the question: How should I write my code to use some library?

Example: Using the STL

```
std::vector<int> v;
v.push_back(10);
std::cout << v.at(0) << std::endl;
```

What is an ABI (Application Binary Interface)?

Like an API, but at the *binary*, machine-specific, level.

Answers questions like:

- What is the "real" name of `print_data(Data &)`?
- How do I find the address of the function?
- What should I do with parameters to the function?
- How should the members of `Data` be stored in memory?
- How do I throw an exception?

Often specified as: How are features of a language implemented?

Specification vs. Implementation

Different implementations may provide additional functionality

For example:

```
// OK, within the standard
std::type_info &info = typeid(int);
// OK, implementation defined (by the ABI)
std::cout << info.name() << std::endl;
// Outside of the API, may or may not exist.
info.__do_catch(...);
```

Specification vs. Implementation

Different implementations may provide additional functionality

For example:

```
#include <vector>
class Test : public std::vector<int> {
public:
  void examine() {
    cout << "start: " << _M_impl._M_start << endl;
    cout << "finish: " << _M_impl._M_finish << endl;
    cout << "end: " << _M_impl._M_end_of_storage << endl;
  }
};
```

API Compatibility

Consider the following changes to `shared.h` in `01-compatibility`. Do they change the API? (Think: Do they break compilation of existing programs?)

1. Change `int a` to `int64_t a`?
2. Add a member `int c` to `Data`?
3. Add a member `std::string c` to `Data`?
4. Change the order of `int a` and `double b`?
5. Add a new parameter with default value to `print_val`?
6. Add a copy/move-constructor to `Data`?
7. Change `print_val` to accept a const-reference?

ABI Compatibility

Consider the following changes to `shared.h` in `01-compatibility`. Do they change the API? (Think: Do they disallow linking with old code?)

1. Change `int a` to `int64_t a` or `double a`?
2. Add a member `int c` to `Data`?
3. Add a member `std::string c` to `Data`?
4. Change the order of `int a` and `double b`?
5. Add a new parameter with default value to `print_val`?
6. Add a copy/move-constructor to `Data`?
7. Change `print_val` to accept a const-reference?

API vs. ABI Compatibility

Work at different levels of abstraction: source-code vs. machine code

- API: if all (valid) uses of source code compiles, it is OK
- ABI: if all (valid) uses work without re-compilation, it is OK

$\Rightarrow$ Stable ABIs require a bit more thought...

$\Rightarrow$ ...but avoids re-compilation!

Different ABIs

There are two major ABIs:

- System V ABI (Linux, MacOS on AMD64/ARM64)
- Microsoft ABI (Windows)

Variants for many systems:

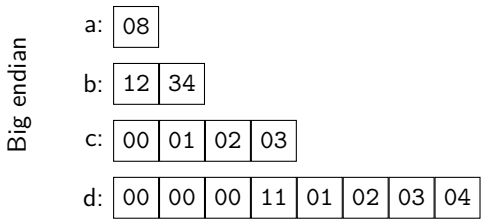
- x86-32
- **AMD64**
- ARM64
- ...

Integer Types and Endianness

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

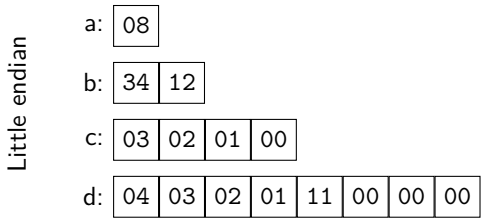
Integer Types and Endianness

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



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};
```

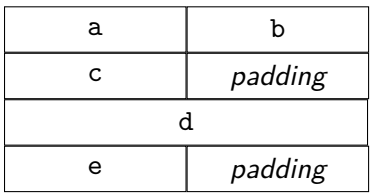
All look the same in memory!

Other Types

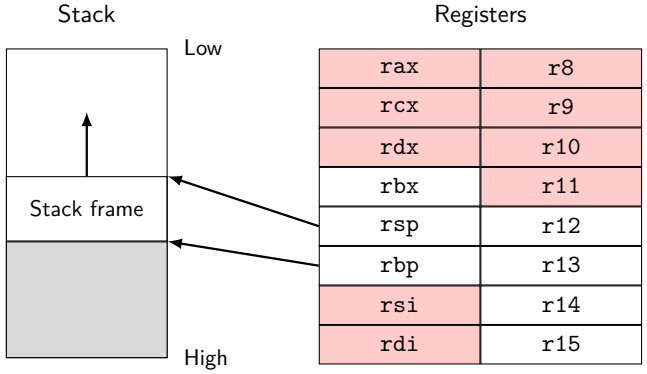
- Each type has a *size* and an *alignment*
- Members are placed sequentially, respecting the alignment

Example:

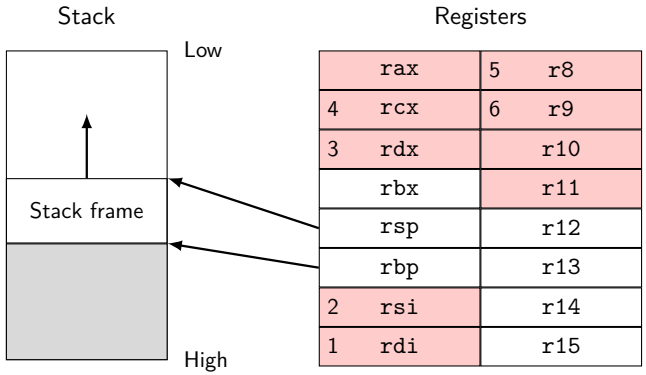
```
struct simple {
  int a{1};
  int b{2};
  int c{3};
  long d{100};
  int e{4};
};
```



Overview


Stack
Registers
Low
Stack frame
High
rax
r8
rcx
r9
rdx
r10
rbx
r11
rsp
r12
rbp
r13
rsi
r14
rdi
r15

Overview


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

Rules (simplified)

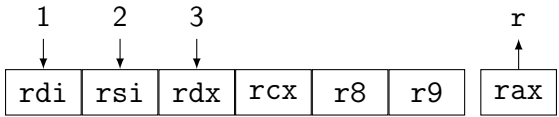
1. If a parameter has a copy constructor or a destructor:
   - Pass by hidden reference
2. If a parameter is larger than 4*8 bytes
   - Pass in memory
3. If a parameter uses more than 2 integer registers
   - Pass in memory
4. Otherwise
   - Pass in appropriate registers (integer/floating-point)

Primitives

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

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

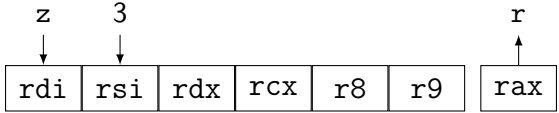
```
mov $1, %edi
mov $2, %esi
mov $3, %edx
call fn
mov %rax, "r"
```



“Simple” Types 1

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

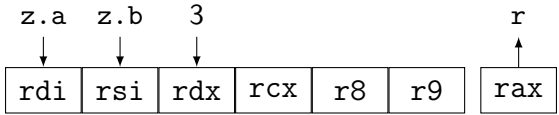
```
mov "z", %rdi
mov $3, %rsi
call fn
mov %rax, "r"
```



“Simple” Types 2

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

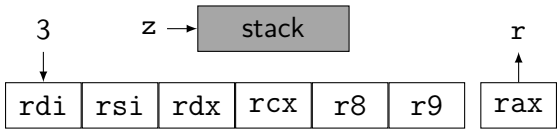
```
mov "z", %rdi
mov $3, %rsi
call fn
mov %rax, "r"
```



“Simple” Types 3

```
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"
```

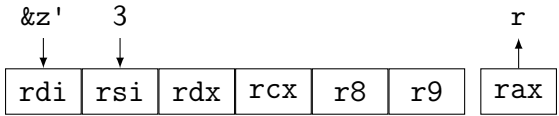


“Complex” Types

```
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"
```

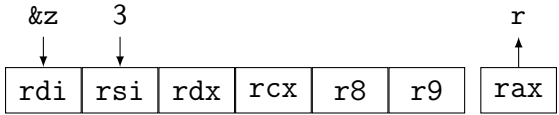
`large` is not trivially copiable, has a destructor or a vtable



By Reference

```
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"
```

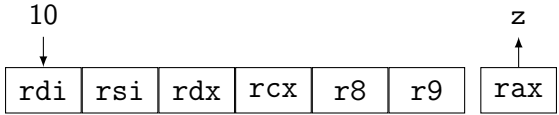


Returning “Simple” Types 1

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

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

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

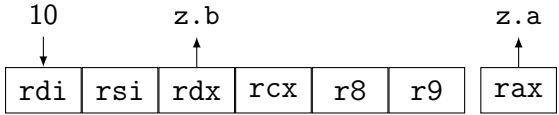


Returning “Simple” Types 2

```
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
```

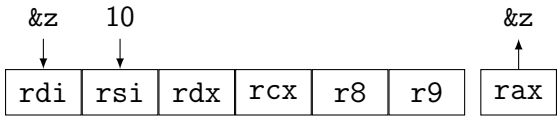


Returning Large or "Complex" Types

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

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

```
lea "z", %rdi
mov $10, %rsi
call fn
```

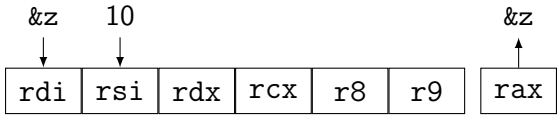


Returning Large or "Complex" Types

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

int main() {
  large z = /* uninitialized */;
  fn(z, 10);
}
```

```
lea "z", %rdi
mov $10, %rsi
call fn
```



Conclusions

- Passing primitives by value is cheap
- Passing simple types by value is cheap
  - 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 usually cheap, even without RVO
- Types that are not trivially copiable are more cumbersome: pass them by reference

Scenario

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

  int data{0x1020};

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

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

Virtual Function Tables – vtables (System V)

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!

Pointers to Members

```
struct MyClass {
  virtual int virtual_member() { return 1; }
};
struct DerivedClass : public MyClass {
  int virtual_member() override { return 2; }
};
int main() {
  int (MyClass::* ptr)() = &MyClass::virtual_member;
  cout << sizeof(ptr) << endl; // How large?
  DerivedClass c;
  return (c.*ptr)(); // Which one is called?
}
```

Pointers to Members

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

  // Object offset (for multiple inheritance)
  size_t offset;
};
```

This is why it is not possible to cast pointers to members into `void *`.

Motivation

Why should we know about *linking*?

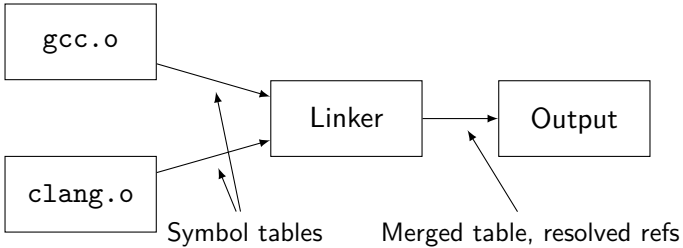
- Understand and fix errors/warnings
- Motivates the design of some parts of C++
  - Easier to remember "rules" when you know why
- Allows reasoning about optimizations (inlining)
- Utilize the powers of dynamic linking (plugin systems, etc.)

What Does the Linker Do?

Remember the first program (`00-different-compilers`)

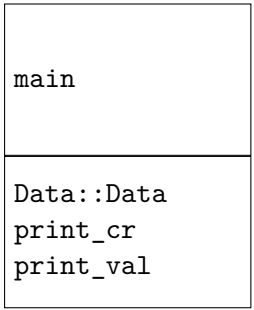
- How does the compiler know where `print_cr` and `print_val` are located?
- Why can't I modify the parameters to the functions?

A Program from the Linker's Perspective

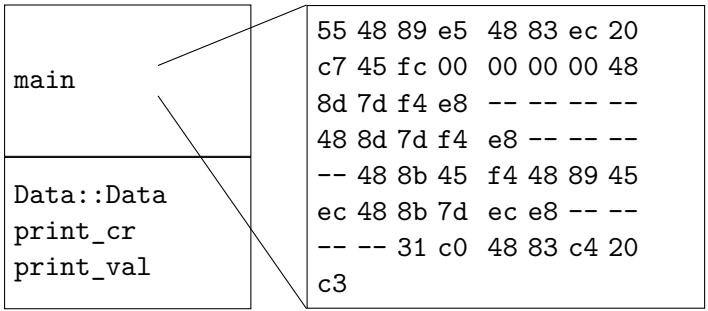


- `objdump -t <file>` – show symbol table
- `objdump -t -C <file>` – show symbol table
- `objdump -d -C <file>` – disassemble code

A Program from the Linker's Perspective



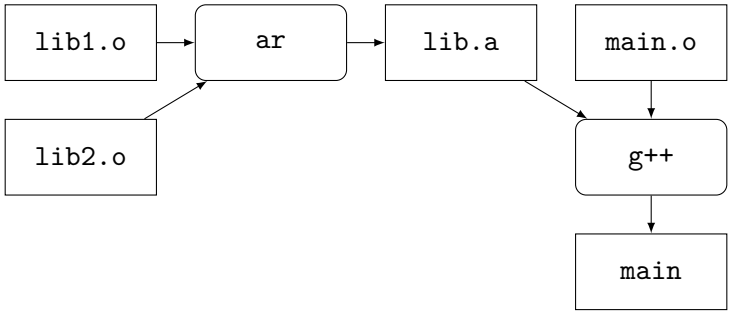
A Program from the Linker's Perspective



LINKÖPING UNIVERSITY

Brief history: Static libraries

What if we have a “large” library? Inconvenient to distribute many `.o` files...



What is an `.a`-file?

Consider the code in `10-static-lib`.

- `ar t <file>` – list members
- `ar x <file>` – extract members

Problems with static linking

Static linking **copies** code into the final binary. This means:

- The final binary becomes larger, both on disk and in RAM
- Fixing a bug in the library requires re-linking all programs using it

Gotchas:

- The `.a`-file must appear after inputs that are using it.
- Only archive members that are used are included!
  - For example: remove call to `print_greeting` from `main.cpp`
- On Windows, libraries (`.lib`) behave more like regular object files.

Dynamic Linking

Idea: leave symbols undefined and resolve them when loading the program

We then let the *dynamic linker* handle linking of *shared libraries*

- Avoids copies of code, both on disk and in RAM
- We can easily update the library
- Makes loading code at runtime easier

Practicalities

- Modelled to work like static linking
- This is what the `-l` flag does. Two forms:
  - `-l<x>` ⇒ finds `lib<x>.so`
  - `-l:<x>` ⇒ finds `<x>`
- Default: system's library path, we can use `-L` to modify this
- The dynamic linker also needs to know where to look
  - `rpath` or `runpath`
- Code must be *position independent*: `-fPIC`

Dynamic linking

Consider the code in `11-shared-lib`

- Now, what happens if we uncomment `print_greeting`?

- `readelf -h <program>` – show headers
- `readelf -l <program>` – show program headers
- `readelf -d <program>` – inspect dependencies
- `objdump -T <program>` – inspect dynamic symbol table
- `ldd <program>` – inspect behavior of dynamic linker

Multiple dynamic libraries

Consider the code in `12-multiple`. We have two libraries, `lib1.so` and `lib2.so` linked to our executable.

- Try uncommenting `lib_name` in `lib1.cpp`
- Try uncommenting `print_greeting` in `lib1.cpp`
- Try uncommenting `check_int` in `main.cpp`

$\Rightarrow$ The symbols in *all* libraries form a *single* namespace!

  - Order based on appearance on command line

- How do we fix this?

Library Isolation

Linux (UNIX in general):

- Symbols have *visibility*:
  - default – visible outside the shared object
  - hidden – only visible inside the shared object
  - internal – hidden, but only called from the same module
  - protected – can not be overridden by another module
- We can set default with `-fvisibility=hidden`
- We can use `static` and anonymous namespaces.

Differences on Windows

Windows takes a different approach:

- Symbols are hidden by default
  - Explicitly export symbols `__declspec(dllexport)`
  - Explicitly import symbols `__declspec(dllimport)`
- Compiling a DLL makes the DLL and a *import library* (`.lib`)
- Link with the *import library* to use the DLL
- Search path typically include executable's path by default

Calling the Dynamic Linker

We can load libraries dynamically by calling the dynamic linker (`-ldl` on Linux):

- `dlopen` or `LoadLibrary` – load a shared library
- `dlclose` or `FreeLibrary` – unload a shared library
- `dlsym` or `GetProcAddress` – get the address of a symbol
  Note: name mangling differs between systems, even for C!

Calling the Dynamic Linker

Consider the code in `13-dlsym`

- Try running `./main ./lib1.so` and `./main ./lib2.so`
- What happens if we specify `RTLD_NOW`?
- Why do we need `RTLD_GLOBAL`?
- Why don't we get an error when linking `lib2.so`?
  - We can add `-Wl,-z,defs`
- Why can't we add `do_fun_stuff` in main executable?
  - We can link with `-rdynamic`

What Does the Compiler Assume?

Consider the code in `14-dynamic-rebind`

- Try running `./main`
  and `LD_PRELOAD=./inject.so ./main`
- Unless visibility is set, symbols may be overwritten

$\Rightarrow$ Compiler is not able to inline/reason about functions

Patch Internal Functions

Consider the code in `15-patch`:

- If you know the internals of a library, it is possible to intercept and patch functionality...

Instrument a program

Consider the code in `16-instrument`:

- We can inject our minimal library anywhere we want
- For example: `LD_PRELOAD=./track.so /usr/bin/echo hello`

Windows

- Stronger guarantees by default: compiler is able to reason about the code to a larger extent
- Strong isolation leads to other peculiarities. In particular, we may have multiple copies of the same thing:
  - metadata: must compare *names* of types, rather than pointers
  - globals: sometimes we have multiple *heaps*, must allocate and free from the same DLL
- All symbols must be resolved, more work to make "pluggable" interfaces
- We can still do "bad things", but they require more work

Implications for library design

There are many things to consider when writing libraries. Good API design is important, and we need to consider how linking works:

- Consider visibility, especially for internal functions
- Memory allocated by your library might need to be freed by your library
- You might have multiple instances of code and/or data

Filip Strömbäck

www.liu.se

LINKÖPING UNIVERSITY