
Core C++ 2026
20-21 Oct. 2026 :: Tel-Aviv
Running C++ on Bare Metal
Building an aarch64 cross-toolchain
from configure to first boot
Ron Shabi
ronsh.net

Most of the C++ we write runs on top of **machinery we never see.**

What do you assume is already there?

```
#include <iostream>


int main()
{
    std::cout << "Hello world\n";
    return 0;
}
```

- Intro Blabber
- Checklist
- Building the Toolchain
- Booting QEMU
- C++ Runtime
- Payoff

Building Vortex

an ARMv8-A bare-metal kernel with C++

Ron Shabi ronsh.net

Intro Blabber
Checklist
Building the Toolchain
Booting QEMU
C++ Runtime
Payoff

THE PROCESSOR

- 64-bit ARM
- Same architecture family as most smartphones
- **ARM Cortex A-72**
- Implements **ARMv8-A** 64-bit instruction set



Broadcom BCM2711 (SoC)
4 cores of the Cotex-A72!

Intro Blabber
Checklist
Building the Toolchain
Booting QEMU
C++ Runtime
Payoff

THE PROCESSOR

- 64-bit ARM
- Same architecture family as most smartphones
- **ARM Cortex A-72**
- Implements **ARMv8-A** 64-bit instruction set



QEMU

Ron Shabi ronsh.net

Intro Blabber
Checklist
Building the Toolchain
Booting QEMU
C++ Runtime
Payoff

THE TOOLCHAIN

- The one on our machine builds programs for **Linux on x86-64**
- It assumes:
  - an operating system
  - a C library
  - a loader
- Even the common ARM cross toolchain, **`aarch64-linux-gnu`** still assumes Linux on the other end.

Intro Blabber
Checklist
Building the Toolchain
Booting QEMU
C++ Runtime
Payoff

WE NEED

- A **cross toolchain**
- We could compile binaries for different platforms and ISAs on our machine
- Our target is **aarch64-elf**
- **`binutils`**
- **`gcc`** and **`g++`**
- **`gdb`** (would be nice)

Intro Blabber
Checklist
Building the Toolchain
Booting QEMU
C++ Runtime
Payoff

Why even build it ourselves!?

- Prebuilt toolchains are great when...
  - one exists for our target
  - in the version we need
  - with the configuration we need
- Natural instinct is to `apt install *gcc`
- But what if we don't have it built for us?

Intro Blabber



Checklist

Building the Toolchain

Booting QEMU

C++ Runtime

Payoff

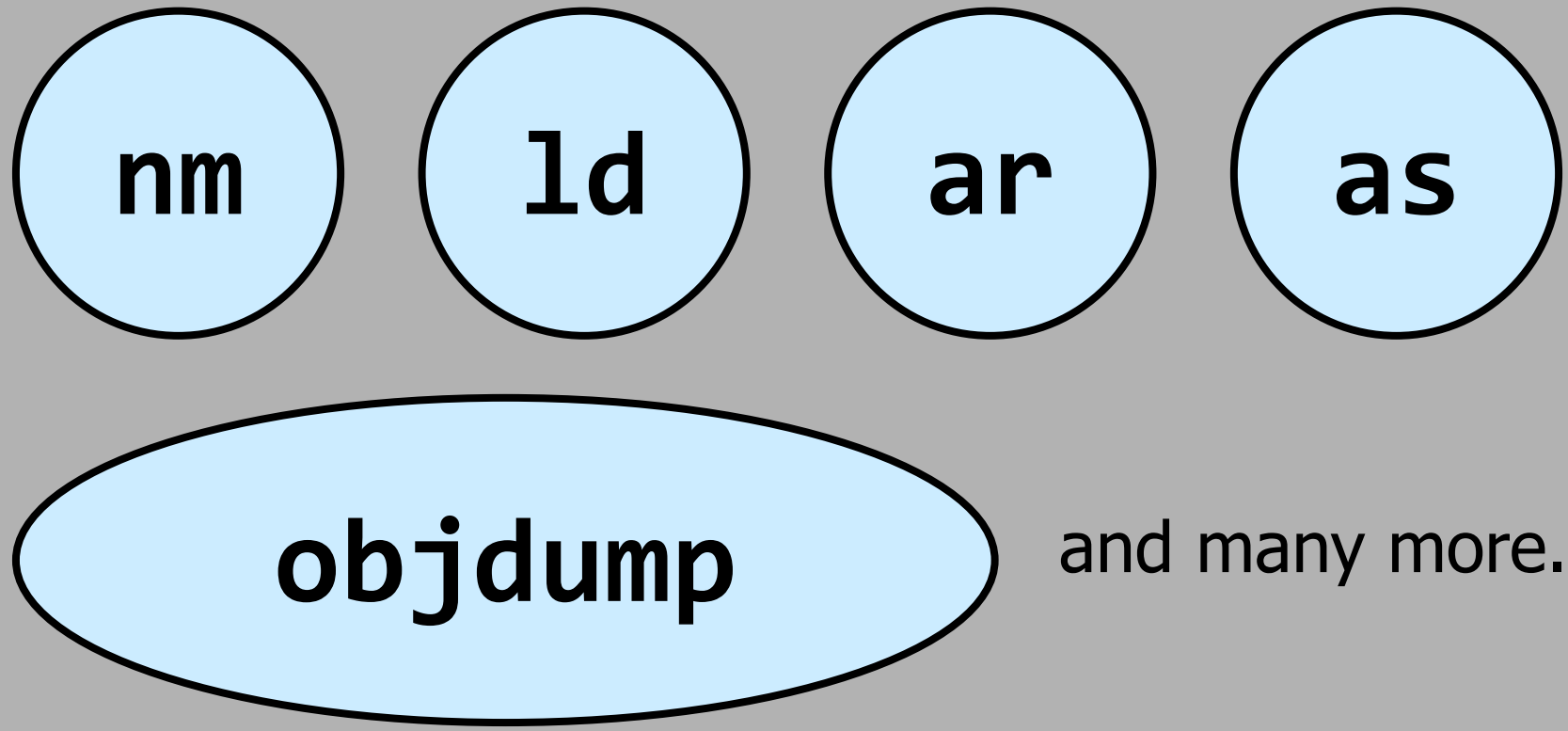
Well then...

We build it ourselves

Intro Blabber
Checklist
Building the Toolchain
Booting QEMU
C++ Runtime
Payoff

Step 1: binutils

- needed to build gcc and gdb

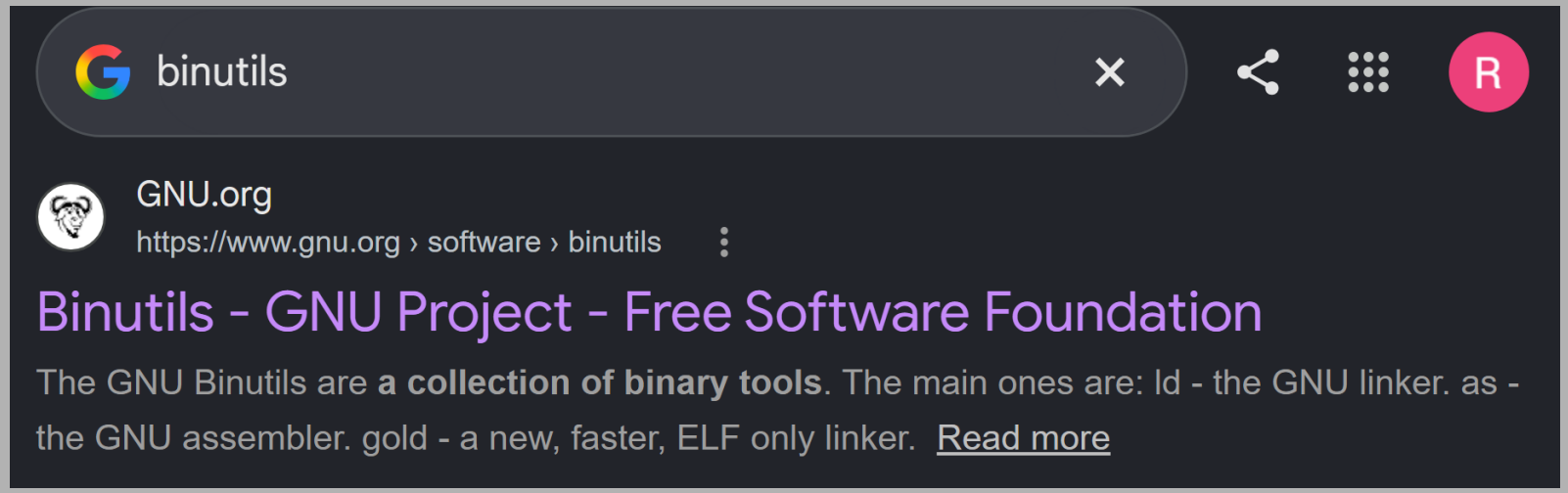


and many more...

Intro Blabber
Checklist
Building the Toolchain
Booting QEMU
C++ Runtime
Payoff

Step 1: `binutils`

- Getting the source tarball



Intro Blabber
Checklist
Building the Toolchain
Booting QEMU
C++ Runtime
Payoff

Step 1: binutils

- Getting the source tarball

Obtaining binutils

The latest release of GNU binutils is 2.47. The various NEWS files (binutils, gas, and ld) have details of what has changed in this release.

See the SOFTWARE page for information on obtaining releases of GNU binutils and other GNU software. The current release can be downloaded from https://ftp.gnu.org/gnu/binutils or https://sourceware.org/pub/binutils/releases/.

If you plan to do active work on GNU binutils, you can ac the development source tree by anonymous git:

```
git clone git://sourceware.org/git/binutils-gdb.git
```

Alternatively, you can use the gitweb interface.

There is also a mirror of the repository at sourcehut.

Snapshots of the sources are taken after every commit and are made available as compressed tar files from https://snapshots.sourceware.org/binutils/trunk/.

Intro Blabber
Checklist
Building the Toolchain
Booting QEMU
C++ Runtime
Payoff

Step 1: binutils

- Getting the source tarball



Index of /gnu/binutils

Name	Last modified	Size	Description
Parent Directory		-	
binutils-2.47.tar.zst.sig	2026-07-26 13:30	833	
binutils-2.47.tar.zst	2026-07-26 13:30	30M	
binutils-2.47.tar.xz.sig	2026-07-26 13:29	833	
binutils-2.47.tar.xz	2026-07-26 13:29	28M	
binutils-2.47.tar.lz.sig	2026-07-26 13:28	833	
binutils-2.47.tar.lz	2026-07-26 13:28	27M	
binutils-2.47.tar.gz.sig	2026-07-26 13:28	833	
binutils-2.47.tar.gz	2026-07-26 13:28	52M	
binutils-2.47.tar.bz2.sig	2026-07-26 13:26	833	
binutils-2.47.tar.bz2	2026-07-26 13:26	39M	

Intro Blabber
Checklist
Building the Toolchain
Booting QEMU
C++ Runtime
Payoff

Step 1: binutils

- Preparing our environment
- **TARGET**: Target triplet
  - `EXPORT TARGET="aarch64-elf"`
- **PREFIX**: where should we install?
  - `EXPORT PREFIX="/home/ron/armcompiler"`
- Add Prefix to **PATH**: gcc needs binutils
  - `EXPORT PATH="$PREFIX/bin:$PATH"`
- Binutils likes out-of-tree builds

Intro Blabber
Checklist
Building the Toolchain
Booting QEMU
C++ Runtime
Payoff

Step 1: binutils

configure

- --target="$TARGET" ← aarch64-elf
- --prefix="$PREFIX" ← /home/ron/armcompiler
- --with-sysroot
- --disable-nls
- --disable-Werror

- make
- make install ← No sudo!

Intro Blabber
Checklist
Building the Toolchain
Booting QEMU
C++ Runtime
Payoff

Step 2: gcc and g++

- Getting the source tarball

GCC Releases

Download

GCC releases may be downloaded from our mirror sites.

Release	Release date
GCC 13.5	Septemb
GCC 16.2	August 7, 2026
GCC 14.4	June 26, 2026
GCC 15.3	June 12, 2026

Intro Blabber
Checklist
Building the Toolchain
Booting QEMU
C++ Runtime
Payoff

Step 2: gcc and g++

configure

- `--target="$TARGET"` ← `aarch64-elf`
- `--prefix="$PREFIX"` ← `/home/ron/armcompiler`
- `--disable-nls`
- `--enable-languages=c,c++`
- `--without-headers` ← Build **gcc** and **libgcc** without a standard C library
- `--enable-initfini-array` ← `.init` → `.init_array`, `.fini` → `.fini_array`
- `--disable-hosted-libstdcxx`

C++ library should be freestanding instead of hosted

Intro Blabber
Checklist
Building the Toolchain
Booting QEMU
C++ Runtime
Payoff

Step 2: gcc and g++

- compile
  - `make all-gcc`
  - `make all-target-libc`
  - `make all-target-libstdc++-v3`
- install
  - `make install-gcc`
  - `make install-target-libgcc`
  - `make install-target-libstdc++-v3`

Again, no sudo!

Intro Blabber
Checklist
Building the Toolchain
Booting QEMU
C++ Runtime
Payoff

Step 3: profit

```
ron@vbox:~/aarch64/bin$ ls
aarch64-elf-addr2line  aarch64-elf-g++
aarch64-elf-ar         aarch64-elf-gcc
aarch64-elf-as         aarch64-elf-gcc-16.2.0
aarch64-elf-c++        aarch64-elf-gcc-ar
aarch64-elf-c++filt    aarch64-elf-gcc-nm
aarch64-elf-cpp        aarch64-elf-gcc-ranlib
aarch64-elf-elfedit    aarch64-elf-gcov
```

```
aarch64-elf-gcov-dump      aarch64-elf-ld.bfd     aarch64-elf-size
aarch64-elf-gcov-tool      aarch64-elf-lto-dump   aarch64-elf-strings
aarch64-elf-gdb            aarch64-elf-nm         aarch64-elf-strip
aarch64-elf-gdb-add-index  aarch64-elf-objcopy
aarch64-elf-gprof          aarch64-elf-objdump
aarch64-elf-gstack         aarch64-elf-ranlib
aarch64-elf-ld             aarch64-elf-readelf
```

Intro Blabber
Checklist
Building the Toolchain
Booting QEMU
C++ Runtime
Payoff

The Machine

- QEMU
- `virt` board
- Doesn't model any real product
- With a Cortex-A72 CPU

```
qemu-system-aarch64
    -machine virt
    -cpu cortex-a72
```

Intro Blabber

Checklist

Building the Toolchain



Booting QEMU

C++ Runtime

Payoff

ARM Exception Levels

- ARM vocabulary
- Privilege levels = "Exception levels"
- Applications runs at EL0
- Kernels run at EL1
- CPU starts at EL1

Intro Blabber
Checklist
Building the Toolchain
Booting QEMU
C++ Runtime
Payoff

QEMU as the Loader

- QEMU is our bootloader!
- It reads our ELF file
- Copies it to memory
- Jumps to its entry point
- We still decide on the addresses

```
-device loader,file=vortex.elf,
addr=0x40100000,cpu-num=0
```

- Intro Blabber
- Checklist
- Building the Toolchain
- Booting QEMU
- C++ Runtime
- Payoff

```
ENTRY(_start)
SECTIONS {
    . = 0x40100000;

    .text : {
        *(.text .text.*)
    }

    .rodata : {
        *(.rodata .rodata.*)
    }

    .init_array : ALIGN(8) {
        __init_array_start = .;
        KEEP(*(SORT_BY_INIT_PRIORITY(.init_array.*)))
        KEEP(*(.init_array))
        __init_array_end = .;
    }

    .data : {
        *(.data .data.*)
    }

    .bss : {
        . = ALIGN(8);
        __bss_start = .;
        *(.bss .bss.* COMMON)
        . = ALIGN(8);
        __bss_end = .;
    }

    . = ALIGN(8);
    . += 0x1000; /* stack */
    stack_top = .;
}
```

Linker Script

We decide those addresses with a linker script

Ron Shabi ronsh.net

Intro Blabber
Checklist
Building the Toolchain
Booting QEMU
C++ Runtime
Payoff

The Boot Stub

- Our entry point is _start
- No stack yet
- Zero .bss then jump to C++ world

```
_start:
    ldr x30, =stack_top
    mov sp, x30

    bl cxx_run_global_ctors
    bl kmain
```

Intro Blabber
Checklist
Building the Toolchain
Booting QEMU
C++ Runtime
Payoff

Extern "C"

- kmain is declared extern "C"
- We call it from boot.S
- Avoids C++ mangling (_Z5kmainv)

```
extern "C" void kmain(void)
{
    log_buffer_init(&printk_log_buffer);
    printk_log_buffer_ptr = &printk_log_buffer;

    uart_init();

    printk("-*- Vortex -*-\n");
```

Intro Blabber

Checklist

Building the Toolchain



Booting QEMU

C++ Runtime

Payoff

But wait!

C++ requires more

Ron Shabi ronsh.net

Intro Blabber
Checklist
Building the Toolchain
Booting QEMU
C++ Runtime
Payoff

Global Constructors

- Here is a global that has to run before **kmain**
- On Linux, the C library startup code does that before **main**(argc, argv)

```
#define KMEM_BASE 0x40200000
#define KMEM_SIZE 0x100000

// global ctor lol
vtx::Array<int, 67> g_arr{67};

extern "C" void kmain(void) {
```

Intro Blabber
Checklist
Building the Toolchain
Booting QEMU
C++ Runtime
Payoff



.init_array

- An array of function pointers to global constructors!
- Basically, a TODO list.

- Intro Blabber
- Checklist
- Building the Toolchain
- Booting QEMU
- C++ Runtime
- Payoff

Linker Script to the Rescue

```
using TCtorFn = void(*)();
extern "C"  TCtorFn __init_array_start[];
extern "C"  TCtorFn __init_array_end[];
```

```
    .init_array : ALIGN(8) {
        __init_array_start = .;
        KEEP(*(SORT_BY_INIT_PRIORITY(.init_array.*)))
        KEEP(*(.init_array))
        __init_array_end = .;
    }
```

- Intro Blabber
- Checklist
- Building the Toolchain
- Booting QEMU
- C++ Runtime
- Payoff

Runtime Initialization

```
extern "C" void cxx_run_global_ctors(void)
{
    for (TCtorFn *fn = __init_array_start; fn != __init_array_end; ++fn) {
        (*fn)();
    }
}
```

- Intro Blabber
- Checklist
- Building the Toolchain
- Booting QEMU
- C++ Runtime
- Payoff

Destructors

```cpp
// The kernel will never exit so I don't care about global dtors
extern "C" {
    void *__dso_handle = nullptr;
    int __cxa_atexit(void(*)(void *), void *, void *)
    {
        return 0;
    }

    int __cxa_pure_virtual(void)
    {
        // We should panic here, but I should prepare an exception info for that
        // For now just halt
        for (;;) {}
    }
}
```

Intro Blabber

Checklist

Building the Toolchain

Booting QEMU

C++ Runtime

Payoff

Exceptions? RTTI?

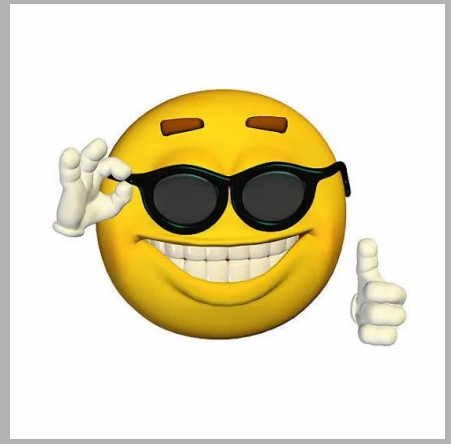
- We compile without exceptions and RTTI
- We have no throw machinery to support
  - No unwinder
  - No personality routine
  - No runtime type objects
- We give up **throw**, **catch**, **dynamic_cast**, **typeid**
- Out of scope for this talk

- Intro Blabber
- Checklist
- Building the Toolchain
- Booting QEMU
- C++ Runtime
- Payoff

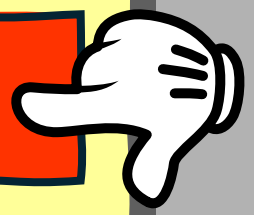
Quick Round!

What about **function local statics?**

`-fno-threadsafe-statics`

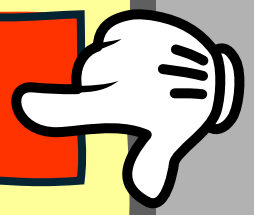


- Intro Blabber
- Checklist
- Building the Toolchain
- Booting QEMU
- C++ Runtime
- Payoff



Wrapping up…

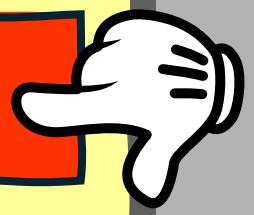
Intro Blabber
Checklist
Building the Toolchain
Booting QEMU
C++ Runtime
Payoff



What does all of it buy us?

- **Ordinary C++**
  - A class `template`
  - Global objects, constructed before `kmain`
  - A container that lives on the stack

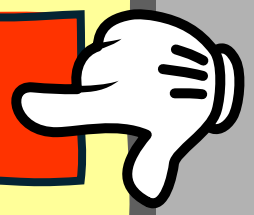
Intro Blabber
Checklist
Building the Toolchain
Booting QEMU
C++ Runtime
Payoff



What does it not?

- A heap
- Any memory management schemes or routines
- The standard C library
- The standard C++ library
- `std::cout`

Intro Blabber
Checklist
Building the Toolchain
Booting QEMU
C++ Runtime
Payoff



What does it not?

But now **we** know who did every job that used to be **invisible**.

Intro Blabber

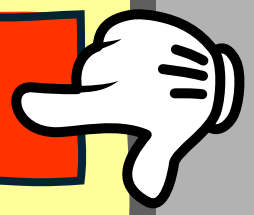
Checklist

Building the Toolchain

Booting QEMU

C++ Runtime

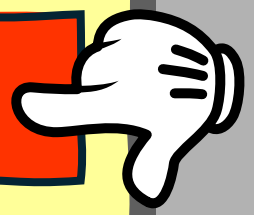
Payoff



Where to go next?

- Interrupts
- Setting up virtual memory
- Memory allocation schemes
- Implementing a standard C library
- Supporting floating point (ARM)

- Intro Blabber
- Checklist
- Building the Toolchain
- Booting QEMU
- C++ Runtime
- Payoff



Where to go next?

We just built a **cross toolchain**,
and **barely scratched the surface**

But we got from **nothing...**

To **something**

Source code available at

ronsh.net