C/C++ Programming »

Cross-Compilation

Cross-Development is the process of developing code on one machine - the host, to run on anonther machine - the target. The host will be a powerful machine which supports development better on the target machine which normally has limited resource or unsupported softwares.

#c/c++

Last update: 2022-06-04

Table of Content

Cross-compiler

A cross-compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is running. For example, a compiler that runs on a PC but generates code that runs on an Android smartphone is a cross-compiler.

A cross-compiler is necessary to compile code for multiple platforms from one development host. Direct compilation on the target platform might be infeasible, for example on embedded systems with limited computing resources.

Why use cross-compiler?

1. **Speed** - Target platforms are usually much slower than hosts, by an order of magnitude or more. Most special-purpose embedded hardware is designed for low cost and low power consumption, not high performance. Modern emulators (like `qemu`) are actually faster than a lot of the real world hardware they emulate, by virtue of running on high-powered desktop hardware.
2. **Capability** - Compiling is very resource-intensive. The target platform usually doesn't have gigabytes of memory and hundreds of gigabytes of disk space the way a desktop does; it may not even have the resources to build "hello world", let alone large and complicated packages.
3. **Availability** - Bringing Linux up on a hardware platform it has never run on before requires a cross-compiler. Even on long-established platforms like ARM or MIPS, finding an up-to-date full-featured prebuilt native environment for a given target can be hard. However, you can easily set up a host machine to build a new package for your target machine.
4. **Flexibility** - A fully capable Linux distribution consists of hundreds of packages, but most packages are not used on the target machine. Providing a big system with full-loaded packages clearly is not a good idea on the target system with limited resource. Cross-compilation helps developer to deploy only necessary packages with a small customized system.
5. **Convenience** - The user interface of headless boxes tends to be a bit cramped. On a powerful host machine, you can easily edit, test, and do more work.

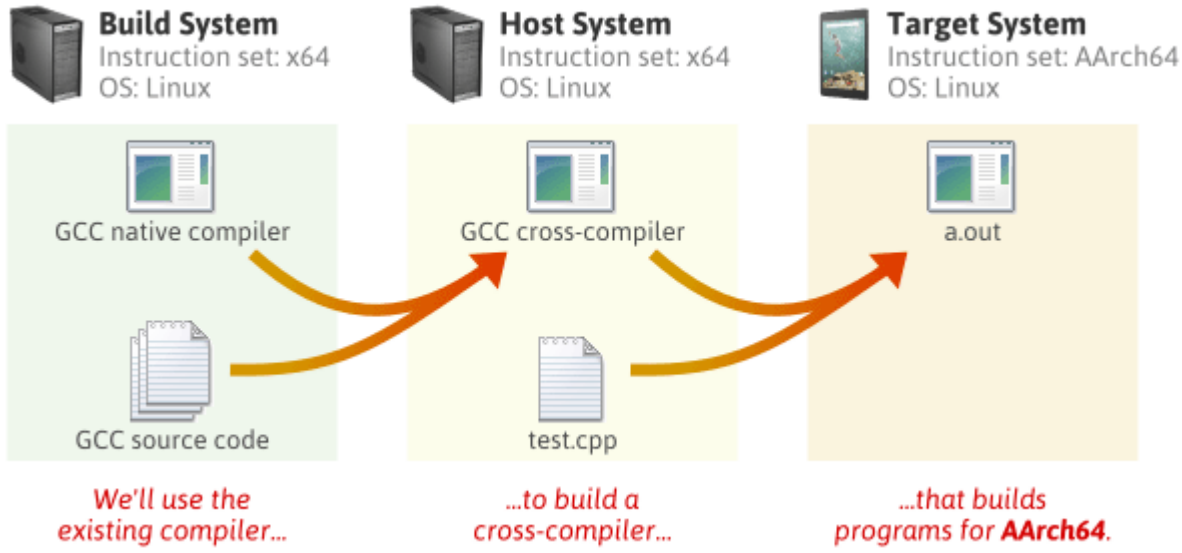
How it works

Let talk about a case:

- Host machine: PC running on x64 Intel hardware (laptop/ desktop)
- Target machine: Embedded device running on ARM X64 hardware (smartphone/ Raspberry Pi)

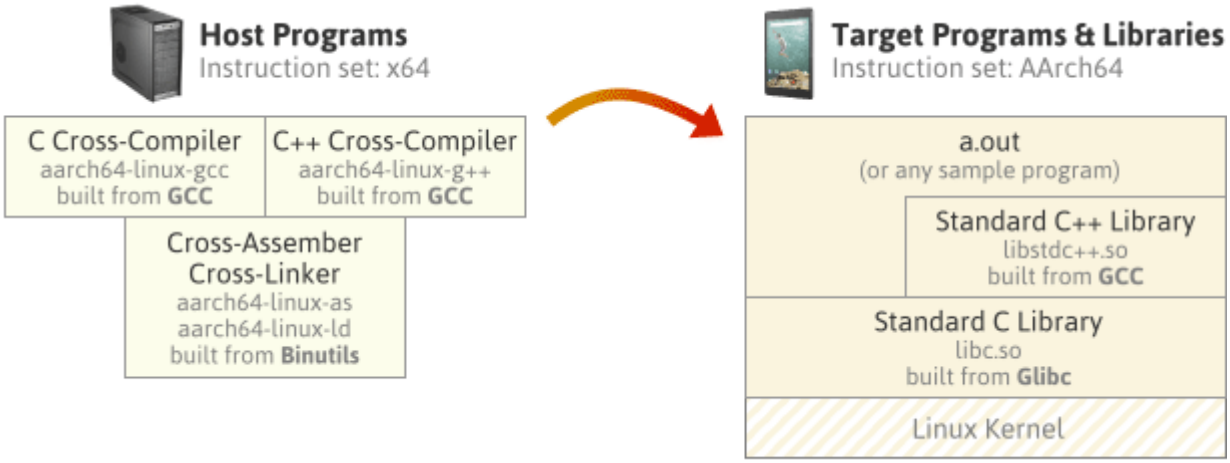
We have to do 2 big steps:

1. Build a cross-compiler from a native-compiler
2. Cross-compile program for target machine



Cross compilation

The diagram on the right represents a sample program, `a.out`, running on the target OS, built using the cross-compiler and linked with the target system's standard C and C++ libraries. The standard C++ library makes calls to the standard C library, and the C library makes direct system calls to the AArch64 Linux kernel.



BinUtils

BinUtils are a collection of binary tools. In Compilation process, you see that you only invoke `gcc` with different options. However, `gcc` is just a driving program which call other programs in a defined order with corresponding inputs, options and outputs.

Let run the example in the Compilation process page, but with `-v` option to see what are actually run behind the scenes.

```
gcc -v source.c mylib.c
```

You will find some interesting that `gcc` invoked, detailed comparison in below section.

- Configuration for default settings
- Compiler call: `/usr/lib/gcc/x86_64-linux-gnu/7/cc1`
- Assembler call: `/usr/bin/as`
- Linker call: `/usr/lib/gcc/x86_64-linux-gnu/7/collect2`

When you build a cross-compiler, you actually build a compiler and a set of binary utility tools. That is the reason, when you install a prebuilt cross-compiler, you will see the package manager will also install `binutils`, and `lib` cross-compiled for the target.

For example:

```
sudo apt install gcc-aarch64-linux-gnu
```

```
binutils-aarch64-linux-gnu              2.30-21ubuntu1~18.04.7          2.8MiB
cpp-7-aarch64-linux-gnu                 7.5.0-3ubuntu1~18.04cross1      5.5MiB
cpp-aarch64-linux-gnu                   4:7.4.0-1ubuntu2.3              3.5KiB
gcc-7-aarch64-linux-gnu                 7.5.0-3ubuntu1~18.04cross1      6.2MiB
gcc-7-aarch64-linux-gnu-base            7.5.0-3ubuntu1~18.04cross1       19KiB
gcc-7-cross-base                        7.5.0-3ubuntu1~18.04cross1       13KiB
gcc-8-cross-base                        8.4.0-1ubuntu1~18.04cross2       14KiB
gcc-aarch64-linux-gnu                   4:7.4.0-1ubuntu2.3              1.4KiB
libc6-arm64-cross                       2.27-3ubuntu1cross1.1           1.1MiB
libc6-dev-arm64-cross                   2.27-3ubuntu1cross1.1           2.0MiB
libgcc1-arm64-cross                     1:8.4.0-1ubuntu1~18.04cross2     34KiB
libgcc-7-dev-arm64-cross                7.5.0-3ubuntu1~18.04cross1      815KiB
libgomp1-arm64-cross                    8.4.0-1ubuntu1~18.04cross2       67KiB
libitm1-arm64-cross                     8.4.0-1ubuntu1~18.04cross2       24KiB
libstdc++6-arm64-cross                  8.4.0-1ubuntu1~18.04cross2      328KiB
linux-libc-dev-arm64-cross              4.15.0-35.38cross1.1            840KiB
```

Example

In this example, we will install a pre-compiled ARM64 cross-compiler.

Install the target cross-compiler

```
sudo apt install \
    build-essential \
    gcc-aarch64-linux-gnu \
```

Hello from machine

cross-compilation.zip

This simple program will show the machine type it is running on.

hello.c

```c
#include <sys/utsname.h>
#include <stdio.h>

int main() {
    struct utsname name;
    uname(&name);
    printf("Hello from %s\n",name.machine);
    return 0;
}
```

Compile using native compiler

```
gcc hello.c -o hello_native
```

This command builds a program for current host machine, run it, and you should see current host machine type is `x86_64`:

```
./hello_native
```

```
Hello from x86_64
```

Compile using ARM64 compiler

```
aarch64-linux-gnu-gcc hello.c -o hello_arm
```

when you run it, it can not execute:

```
./hello_arm
```

```
-bash: ./hello_arm: cannot execute binary file: Exec format error
```

Check the differences with native compiler

> Cross-Compilation:
>
> ```
> aarch64-linux-gnu-gcc -v hello.c
> ```

Target:
`aarch64-linux-gnu`

Configuration:
`--host=x86_64-linux-gnu`
`--target=aarch64-linux-gnu`
`--includedir=/usr/aarch64-linux-gnu/include`

Options:
`-mlittle-endian -mabi=lp64`

Compiler:
`/usr/lib/gcc-cross/aarch64-linux-gnu/7/cc1`

Assembler:
`/usr/aarch64-linux-gnu/bin/as`

Linker:
`/usr/lib/gcc-cross/aarch64-linux-gnu/7/collect2`

Native compilation:

```
gcc -v hello.c
```

Target:
`x86_64-linux-gnu`

Configuration:
`--host=x86_64-linux-gnu`
`--target=x86_64-linux-gnu`

Options:
`-mtune=generic -march=x86-64`

Compiler:
`/usr/lib/gcc/x86_64-linux-gnu/7/cc1`

Assembler:
`/usr/bin/as`

Linker:
`/usr/lib/gcc/x86_64-linux-gnu/7/collect2`

Run ARM64 Binary with QEMU

You can run `hello_cross` on ARM64 machine. However, for testing, you can run an ARM64 emulator on `qemu`.

Install `qemu-user` package which provides `qemu-aarch64` emulator:

```
sudo apt install \
    qemu-user \
```

Then you can run `hello_arm` by providing the dynamic libraries of ARM64 machine using `-L <dir>` option:

```
qemu-aarch64 -L /usr/aarch64-linux-gnu/ ./hello_arm
```

```
Hello from aarch64
```

That's it! You have cross-compiled `hello.c` using `aarch64-linux-gnu-gcc` cross-compiler from a host `x86_64` PC machine to run on a target `aarch64` ARM machine.

Static Libraries

If you want to run `hello_arm` directly without sending it to `qemu-aarch64`, you can do below steps:

- Install `qemu-user-static` package
- Build `hello_arm` with `-static` flag
- Run directly `./hello_arm`