

# PhD Diary

Wei Minn  
Singapore Management University

Version 1.0.5  
Last compiled: December 21, 2023



# Contents

<b>I</b>	<b>2023</b>	<b>1</b>
<b>1</b>	<b>November</b>	<b>3</b>
1.1	November 14, 2023 . . . . .	3
1.1.1	AOSP . . . . .	3
1.1.2	C++ Primer . . . . .	6
1.2	November 15, 2023 . . . . .	8
1.2.1	C++ Primer . . . . .	8
1.2.2	AOSP . . . . .	11
1.3	November 17, 2023 . . . . .	13
1.3.1	AOSP Hardware Abstraction Layer . . . . .	13
1.4	November 18, 2023 . . . . .	15
1.4.1	Configuring AOSP for Acme Device on Ubuntu 23 . . . . .	15
1.5	November 19, 2023 . . . . .	16
1.5.1	Building AOSP HAL . . . . .	16
1.6	November 20, 2023 . . . . .	17
1.6.1	const Qualifier . . . . .	17
1.6.2	Rudimentary AOSP HAL Application . . . . .	20
1.7	November 21, 2023 . . . . .	21
1.7.1	Fixing SELinux Policy to Start System Service . . . . .	21
1.7.2	Memory Dump . . . . .	21
1.8	November 22, 2023 . . . . .	22
1.8.1	C++ Data Structure . . . . .	22
1.9	November 23, 2023 . . . . .	24
1.9.1	C++ Types and Data Structures . . . . .	24
1.10	November 25, 2023 . . . . .	25
1.10.1	Android Memory Dump . . . . .	25
1.11	November 27, 2023 . . . . .	26
1.11.1	Android Profile Guided Compilation . . . . .	26
1.12	November 29, 2023 . . . . .	27
1.12.1	Android OAT Dump Parser . . . . .	27

<b>2</b>	<b>December</b>	<b>31</b>
2.1	December 01, 2023 . . . . .	31
2.1.1	C++ Strings and Vectors . . . . .	31
2.2	December 03, 2023 . . . . .	37
2.2.1	C++ Iterator, and Arrays . . . . .	37
2.3	December 04, 2023 . . . . .	43
2.3.1	C++ Expressions I . . . . .	43
2.4	December 05, 2023 . . . . .	44
2.4.1	C++ Expressions II . . . . .	44
2.5	December 06, 2023 . . . . .	50
2.5.1	C++ Statements . . . . .	50
2.6	December 10, 2023 . . . . .	52
2.6.1	Learn about SELinux . . . . .	53
2.7	December 13, 2023 . . . . .	59
2.7.1	Android Testing . . . . .	59
2.8	December 19, 2023 . . . . .	60
2.8.1	SELinux Practical . . . . .	60

**Part I**

**2023**



# Chapter 1

## November

### 1.1 November 14, 2023

#### 1.1.1 AOSP

##### **Zygote**

Zygote initializes by pre-loading the entire Android framework. Unlike desktop Java, it does not load the libraries lazily; it loads all of them as part of system start up. After completely initializing, it enters a tight loop, waiting for connections to a socket. When the system needs to create a new application, it connects to the Zygote socket and sends a small packet describing the application to be started. Zygote clones itself, creating a new kernel-level process.

Memory is organized into uniformly sized **pages**. When the application refers to memory at a particular address, the device hardware reinterprets the address as an index into a **page table**. Newly cloned Zygote processes for newly started applications are simply clone of Zygote's page table, pointing to the exact same pages of physical memory. Only the pages the new application uses for its own purposes are not shared:

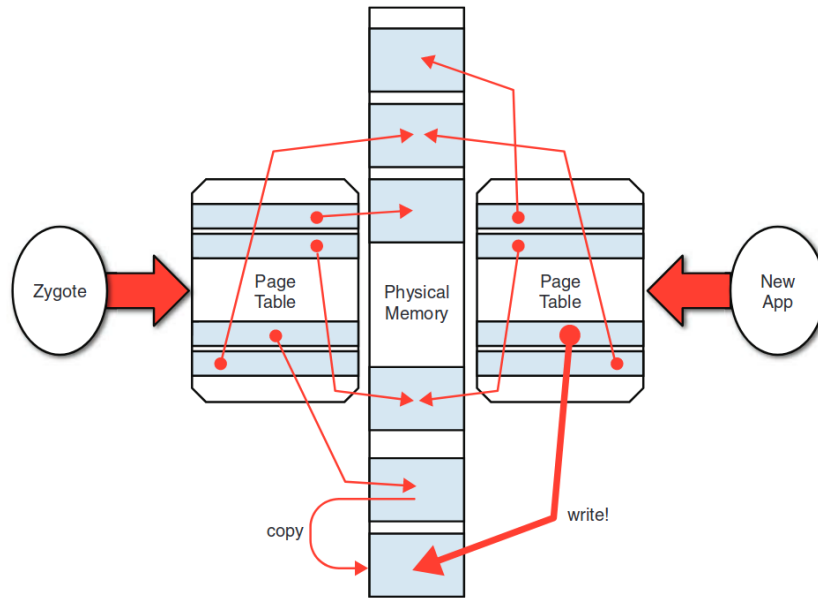


Figure 1.1: Zygote Copy-on Write

## Zygote Initialization

Zygote is started by `init`. `ro.zygote` system variable set at platform build time decides which of four types of Zygotes are started and which one is "primary". Both the `init` and Zygote scripts are stored inside `$AOSP/system/core/rootdir`. In the following `init.zygote64_32.rc`, 2 Zygote processes, primary and secondary, are started at 2 different sockets:

```

1  service zygote /system/bin/app_process64 -Xzygote \
2      /system/bin --zygote --start-system-server --socket-name=zygote
3      class main
4      priority -20
5      user root
6      group root readproc reserved_disk
7      socket zygote stream 660 root system
8      socket usap_pool_primary stream 660 root system
9      onrestart exec_background - system system -- /system/bin/vdc volume abort_fuse
10     onrestart write /sys/power/state on
11     onrestart restart audioserver
12     onrestart restart cameraserver
13     onrestart restart media
14     onrestart restart media.tuner
15     onrestart restart netd
16     onrestart restart wificond
17     task_profiles ProcessCapacityHigh MaxPerformance
18     critical window=${zygote.critical_window.minute:-off} target=zygote-fatal
19
20  service zygote_secondary /system/bin/app_process32 -Xzygote \
21      /system/bin --zygote --socket-name=zygote_secondary --enable-lazy-preload
22      class main

```



```

23     priority -20
24     user root
25     group root readproc reserved_disk
26     socket zygote_secondary stream 660 root system
27     socket usap_pool_secondary stream 660 root system
28     onrestart restart zygote
29     task_profiles ProcessCapacityHigh MaxPerformance
30     disabled

```

The actual application that is started as user root at the very highest priority by init is /system/bin/app\_process64. The script requests that init create a stream socket for the process and catalog it as /dev/socket/zygote\_secondary which will be used by the system to start new Android applications.

Zygote is only started once during the system startup, by app\_process64 and app\_process32, and is simply cloned to start subsequent applications. Zygote initialization sequence is described below:

Method	Description	Source
init.rc	Imports the init.zygote64_32.rc that contains the script that starts Zygote service.	\$AOSP/system/core/rootdir
init.zygote64_32.rc	Runs app_process64 and app_process32 which will initialize the starting of Zygote service.	\$AOSP/system/core/rootdir
app_process	Creates AppRuntime, a subclass of AndroidRuntime, that does bookkeeping, naming the process, setting up parameter, and the name of the class to run when not running Zygote, and then calls AndroidRuntime.start() to invoke the runtime.	\$AOSP/frameworks/base/cmds/app_process
AppRuntime::start	Invokes startVM.startVM which invokes JNI_CreateJavaVM.	\$AOSP/frameworks/base/core/jni/AndroidRuntime.cpp
JNI_CreateJavaVM	Calls Runtime::Create.	\$AOSP/art/runtime/jni/java_vm_ext.cc
Runtime::Create	Initializes the ART runtime, loading the system OAT files and the libraries they contain.	\$AOSP/art/runtime/runtime.cc

**Table 1.1:** Zygote Initialization Sequence

The argument that app\_process passed to start is com.android.internal.os.Zygote.Init, the source for which is in \$AOSP/frameworks/base/core/java/com/android/internal/os/ZygoteInit.java. app\_process is the launcher for all Java programs (not

apps!) in the Android system, and Zygote is one example of the programs (system service) to be launched.

### Zygote System Service

Zygote has 3 major tasks, on startup:

1. Register the socket to which the system will connect to start new application. Handled by `registerServerSocket` method which creates socket using the named passed as parameter for init script.
2. Preload Android resources (classes, libraries, resources and even WebViews) with a call to `preload` method. After preload is finished, Zygote is fully initialized and ready to clone to new applications very quickly.
3. Start Android System Server. Thus, `SystemServer` is the first application to be cloned by Zygote.

After it has completed these three tasks, it enters a loop, waiting for connections to the socket.

### 1.1.2 C++ Primer

#### Primitive Built-in Types

Includes **arithmetic types** and a special type named **void** which has no associated values and can be used in only a few circumstances, most commonly as the return type for functions that do not return a value.

The arithmetic types are divided into two categories: **integral types** (which include character and boolean types) and floating-point types.

Type	Meaning	Minimum Size
<code>bool</code>	boolean (true or false)	NA
<code>char</code>	character	8 bits
<code>w_char_t</code>	wide character	16 bits
<code>char16_t</code>	Unicode character	16 bits
<code>char32_t</code>	Unicode character	32 bits
<code>short</code>	short integer	16 bits
<code>int</code>	integer	16 bits
<code>long</code>	long integer	32 bits
<code>long long</code>	long integer	64 bits
<code>float</code>	single-precision floating-point	6 significant digits
<code>double</code>	double-precision floating-point	10 significant digits

long double	extended-precision floating-point	10 significant digits
-------------	-----------------------------------	-----------------------

**Table 1.2:** Zygote Initialization Sequence

Except for `bool` and extended character types, the integral types may be **signed** (can represent negative or positive numbers) or **unsigned**. By default, `int`, `short`, `long`, `long long` are all signed. To declare unsigned type, prepend `unsigned` to the type. `char` is signed on some machine and unsigned on others, and `unsigned int` is abbreviated as `unsigned`.

Conversions happen automatically when we use an object of one type where an object of another type is expected.

```
1 unsigned u = 10;
2 int i = -42;
3 std::cout << i + i << std::endl; // prints -84
4 std::cout << u + i << std::endl; // if 32-bit ints, prints 4294967264
```

In the above snippet, converting a negative number to unsigned will cause the value to "wrap around" because unsigned values can never be less than 0. Thus, extra care should be taken if we want to write loops with unsigned values and stopping conditions at negative values like the snippet below:

```
1 // WRONG: u can never be less than 0; the condition will always succeed
2 for (unsigned u = 10; u >= 0; --u)
3     std::cout << u << std::endl;
```

As such it is always advisable to not mix signed and unsigned types. By default, integer literals (42) are signed, while octal (024) and hexadecimal (0x14) may be signed or unsigned.

Escape sequences are used as if they were single characters:

```
1 std::cout << '\n'; // prints a newline
2 std::cout << "\tHi!\n"; // prints a tab followed by "Hi!" and a newline
```

## Variables

Initialization is not assignment. Initialization happens when a variable is given a value when it is created. Assignment obliterates an object's current value and replaces that value with a new one.

Four different ways to initialize:

```
1 int units_sold = 0;
2 int units_sold = {0}; // list initialization; does not work for built-in types if
    data loss is likely
3 int units_sold{0};
4 int units_sold(0);
```

Variables defined outside any function body are initialized to zero by default. Variables of built-in type defined inside a function are **uninitialized** and therefore undefined. Objects of class type that we do not explicitly initialize have a value that is defined by the class.

A **declaration** makes a name known to the program. A file that wants to use a name defined elsewhere includes a declaration for that name. A **definition** creates the associated entity. A definition involves declaration, allocates storage and may provide the variable with an initial value.

```

1  extern int i;    // declares but not define j
2  int j;          // declares and defines j
3  int k = 12;     // declares, defines and initializes j
4  extern double pi = 3.14;    // definition

```

Variables must be defined only once but can be declared several times. To use a variable in more than one file requires declarations that are separate from the variable's definition. To use the same variable in multiple files, we must define that in one - and only one - file. Other files that use that variable must declare - but not define - that variable.

## 1.2 November 15, 2023

### 1.2.1 C++ Primer

#### Scopes of Names

Most scopes in C++ are delimited by curly braces.

```

1  #include <iostream>
2  int main() {
3      int sum = 0;
4      // sum values from 1 through 10 inclusive
5      for (int val = 1; val <= 10; ++val)
6          sum += val; // equivalent to sum=sum+val
7      std::cout << "Sum of 1 to 10 inclusive is "
8          << sum << std::endl;
9      return 0;
10 }

```

In above program, main - like most names defined outside a function - has **global scope** and thus, is accessible throughout the program. sum has **block scope** and is accessible from its point of declaration throughout the rest of the main function. val is defined in the scope of the for statement and can be used in that statement but not elsewhere in main.

Names declared in the outer scope can also be redefined in an inner scope although it is always a bad idea:

```

1  #include <iostream>
2  // Program for illustration purposes only: It is bad style for a function
3  // to use a global variable and also define a local variable with the same name
4  int reused = 42; // reused has global scope
5  int main() {
6      int unique = 0; // unique has block scope
7
8      // output #1: uses global reused; prints 42 0

```

```

9      std::cout << reused << "␣" << unique << std::endl;
10
11      int reused = 0; // new, local object named reused hides global reused
12      // output #2: uses local reused; prints 0 0
13      std::cout << reused << "␣" << unique << std::endl;
14
15      // output #3: explicitly requests the global reused; prints 42 0
16      std::cout << ::reused << "␣" << unique << std::endl;
17
18      return 0;
19  }

```

When the scope operator (`::` **operator**) has an empty LHS, it is a request to fetch the name on the RHS from the global scope.

## References

A **reference** defines an alternative name for an object. A reference type can be defined by writing a declarator of the form `&d` where `d` is the name being declared:

```

1  int ival = 1024;
2  int &refVal = ival; // refVal refers to (is another name for) ival
3  int &refVal2; // error: a reference must be initialized

```

When we define a reference, instead of copying the initializer's value, we bind the reference to its initializer. Once initialized, a reference remains bound to its initial object. There is no way to rebind a reference to refer to a different object. Because there is no way to rebind a reference, references must be initialized.

A reference is not an object. Instead, a reference is just another name for an already existing object. Thus, *all* operation on that reference are actually operations on the object to which the reference is bound:

```

1  refVal = 2; // assigns 2 to the object to which refVal refers, i.e., to ival
2  int ii = refVal; // same as ii=ival

```

Because references are not objects, we may not define a reference to a reference. We can define references in a single definition with each identifier that is a reference being preceded by the `&` symbol:

```

1  int i = 1024, i2 = 2048; // i and i2 are both ints
2  int &r = i, r2 = i2; // r is a reference bound to i; r2 is an int
3  int i3 = 1024, &ri = i3; // i3 is an int; ri is a reference bound to i3
4  int &r3 = i3, &r4 = i2; // both r3 and r4 are references

```

## Pointers

Like references, pointers are used for indirect access to other objects. Unlike a reference, a pointer is an object in its own right. Pointers can be assigned and copied; a single pointer can point to several different objects over its lifetime. Unlike a reference, a pointer need not be initialized at the time it is defined. Like other built-in types, pointers defined at block scope have undefined value if they are not initialized.

We define a pointer type by writing a declarator of the form `*d`, where `d` is the name being defined. The `*` must be repeated for each pointer variable:

```
1 int *ip1, *ip2; // both ip1 and ip2 are pointers to int
2 double dp, *dp2; // dp2 is a pointer to double; dp is a double
```

A pointer holds the address of another object. We get the address of an object by using the address-of operator (**& operator**):

```
1 int ival = 42;
2 int *p = &ival; // p holds the address of ival; p is a pointer to ival
3
4 double dval;
5 double *pd = &dval; // ok: initializer is the address of a double
6 double *pd2 = pd; // ok: initializer is a pointer to double
7 int *pi = pd; // error: types of pi and pd differ
8 pi = &dval; // error: assigning the address of a double to a pointer to int
```

We can use the dereference operator (**\* operator**) to access that object:

```
1 int ival = 42;
2 int *p = &ival; // p holds the address of ival; p is a pointer to ival
3 cout << *p; // * yields the object to which p points; prints 42
4
5 *p = 0; // * yields the object; we assign a new value to ival through p
6 cout << *p; // prints 0
```

When we assign to `*p`, we are assigning to the object to which `p` points. We may dereference only a valid pointer that points to an object.

`void*` is a special pointer type that can hold the address of any object. Its useful for when the type of the object at that address is unknown:

```
1 double obj = 3.14, *pd = &obj; // ok: void* can hold the address value of any data
   // pointer type
2 void *pv = &obj; // obj can be an object of any type
3 pv = pd; // pv can hold a pointer to any type
```

The modifiers `*` and `&` do not apply to all variables defined in a single statement:

```
1 int* p1, p2; // p1 is a pointer to int; p2 is an int
2 int *p1, *p2; // both p1 and p2 are pointers to int
```

As pointers are objects in memory, they also have addresses of their own. Therefore, we can store the address of a pointer in another pointer:

```
1 int ival = 1024;
2 int *pi = &ival; // pi points to an int
3 int **ppi = &pi; // ppi points to a pointer to an int
```

We indicate each pointer level by its own `*`. Dereferencing a pointer to a pointer yields the pointer. So in this case, you must dereference twice to access the underlying object.

## 1.2.2 AOSP

### Starting Android System Server and Other Apps using Zygote

During its initialization, Zygote will check for start-system-server flag, and if set, will bring up SystemServer in the following sequence:

Method	Description	Source
ZygoteInit. forkSystemServer	Runs after the Zygote process has been initialized. It is hardcoded with System Server classpath <sup>1</sup> as one of the arguments to call Zygote.forkSystemServer to spawn SystemServer process.	AOSP/framework/ base/core/java/com/ android/internal/os/ ZygoteInit.java
Zygote. forkSystemServer	Zygote class wraps native methods that communicate with Android Runtime, one of whom is com_android_internal_os_nativeForkSystemServer.	AOSP/framework/base/ core/java/com/android/ internal/os/Zygote. java
com_android_in- ternal_os_native- ForkSystemServer SpecializeCommon	Calls zygote::ForkCommon and SpecializeCommon which does the actual forking.  Looks at the flags and Process ID for setting up sandboxing, configuring the correct SE Linux context, and process capabilities. Afterwards, it will call Zygote methods for post-fork procedures.	AOSP/frameworks/base/ core/jni/com_android_ internal_os_Zygote.cpp  AOSP/frameworks/base/ core/jni/com_android_ internal_os_Zygote.cpp
Zygote. callPostForkSystemServerHooks	Calls ZygoteHooks.java at the end of specialization procedures. Only applicable for SystemSever.	AOSP/framework/base/ core/java/com/android/ internal/os/Zygote. java
Zygote. callPostForkChildHooks	Calls ZygoteHooks.java at the end of specialization procedures. Applicable to all applications and services including SystemServer	AOSP/framework/base/ core/java/com/android/ internal/os/Zygote. java

<sup>1</sup>com.android.server.SystemServer, the source for which is stored in AOSP/frameworks/base/services/java/com/android/server/SystemServer.java.

Method	Description	Source
ZygoteHooks. postForkSystemService and ZygoteHooks. postForkSystemService	Wrappers for ZygoteHooks inside the Android Runtime. They call their respective native code inside the ART via Java Native Interface.	AOSP/libcore/dalvik/ src/main/java/dalvik/ system/ZygoteHooks. java
ZygoteHooks_ nativePostForkSystemService	Loads the specialized class libraries to start the the System Server.	AOSP/art/runtime/ native/dalvik_system_ ZygoteHooks.cc
ZygoteHooks_ nativePostForkChild	Loads the specialized class libraries to start the service/application.	AOSP/art/runtime/ native/dalvik_system_ ZygoteHooks.cc
handleSystemServiceProcess	After control returns to ZygoteInit, and finish remaining work for the newly forked system server process, and calls ZygoteInit.zygoteInit.	AOSP/framework/ base/core/java/com/ android/internal/os/ ZygoteInit.java
ZygoteInit. zygoteInit	The main function called when started through the zygote process, which calls RuntimeInit.applicationInit	AOSP/framework/ base/core/java/com/ android/internal/os/ ZygoteInit.java
RuntimeInit. applicationInit	Calls the public static void main method of the application	AOSP/framework/ base/core/java/com/ android/internal/os/ RuntimeInit.java
ZygoteServer. runSelectLoop	After forking has finished, the control enters ZygoteServer which starts an endless loop that handles incoming connections with ZygoteConnection.processCommand.	AOSP/framework/ base/core/java/com/ android/internal/os/ ZygoteServer.java
ZygoteConnection. processCommand	Calls Zygote.forkAndSpecialize which is a version of ZygoteInit.forkSystemService for the masses.	AOSP/framework/ base/core/java/com/ android/internal/os/ ZygoteConnection.java
Zygote. forkAndSpecialize	A version of Zygote.forkSystemService for the masses.	AOSP/frameworks/base/ core/jni/com_android_ internal_os_Zygote.cpp



Method	Description	Source
<code>com_android_internal_os_native-ForkAndSpecialize</code>	Calls <code>zygote::ForkCommon</code> and <code>SpecializeCommon</code> which does the actual forking, and returns to <code>ZygoteInit</code> and immediately enters <code>ZygoteServer</code> .	<code>AOSP/frameworks/base/core/jni/com_android_internal_os_Zygote.cpp</code>

**Table 1.3:** System Server and Applications Initialization Sequence

## 1.3 November 17, 2023

### 1.3.1 AOSP Hardware Abstraction Layer

The interface to the hardware is a device drivers which are usually device specific and sometimes proprietary. A single set of C header files describes the functionality that a HAL provides to the Android system. HAL Code for a particular device is the implementation of the API defined by those header files, so that no code above the HAL needs to be changed to port Android to use the new device.

## HAL Code Structure

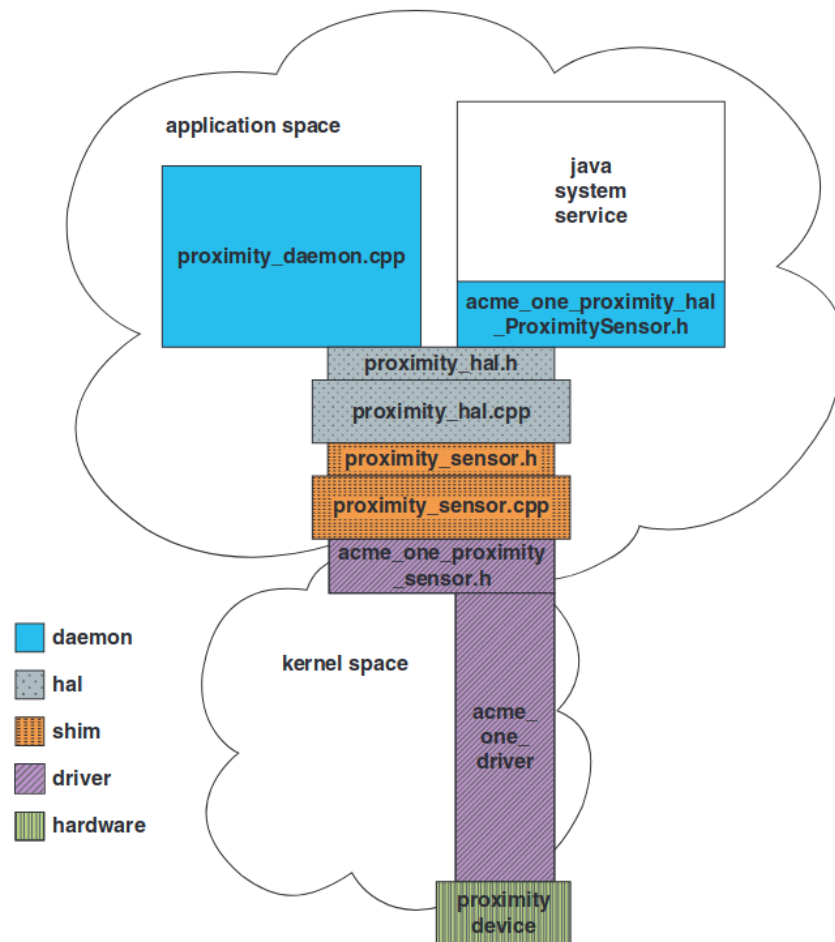


Figure 1.2: HAL Layer Structure

The code consists of four functional components as show in Figure 1.2:

1. **HAL code (dotted boxes):** Abstraction that separates the capabilities of hardware from its specific implementations. The `.h` file defines the HAL interface, and the implementation (`.cpp` file) specializes the Android HAL API for the target hardware.
2. **Shim code (dashed boxes):** Glue code that connects the HAL to a specific device hardware/driver. This code adapts the Android HAL API to the device driver for the hardware.
3. **Daemon (blue):** Stand-alone application that interacts with the hardware through the HAL.

#### 4. **Java System Service (white):** System Service that Android applications will use to access the custom hardware.

The source for those components are structured like in the directory tree below:

```

one
├── app
├── native_daemon
│   └── ...
├── java_daemon
│   └── ...
├── proximity
│   ├── include
│   │   ├── dev
│   │   │   └── ...
│   │   └── ...
│   ├── dev
│   │   └── ...
│   ├── hal
│   │   └── ...
│   └── jni
│       └── ...

```

where one is the device folder of the AOSP project. All the code implementing the HAL for the proximity sensors goes into a new subdirectory `proximity`.<sup>2</sup>

## 1.4 November 18, 2023

### 1.4.1 Configuring AOSP for Acme Device on Ubuntu 23

#### Repo Manifest

Top-level subdirectory named `.repo` contains the manifests repository inside `manifests` subdirectory. The `manifests` repo contain one or more manifest files named as the argument of the `-m` command line option. `.repo/manifest` file controls the structure of the rest of the repository, and includes `.repo/manifests/default.xml`<sup>3</sup> which is a list of git repositories. `repo` program parses `manifest.xml`, and thus `default.xml`, and clone each repository into a location specified inside `default.xml`.

<sup>2</sup>To be a "real" HAL, the interface `proximity/include/proximity_hal.h` would have to be promoted from its current directory specifically for the One device, up into the Android source tree to a location that would make it visible to other code that needed to use it. Here, it is only shared by Acme devices, so it is put under the subdirectory of the Acme device directory. If it's visible across device from multiple vendors, it might be promoted into the device directory itself.

<sup>3</sup><https://gerrit.googlesource.com/git-repo/+master/docs/manifest-format.md>

Each project element in the XML identifies a git repository by its name, relative to some base URL, its remote; and where that repository should be placed in the local workspace, its path. If the full URL for the repository is not specified, repo will use the default remote specified in the default element near the top of the manifest:

```
1 <default revision="refs/tags/android-13.0.0_r11"
2   remote="aosp"
3   sync-j="4" />
```

where the remote, aosp, is defined likewise in the top of of the default.xml:

```
1 <remote name="aosp"
2   fetch=".."
3   review="https://android-review.googlesource.com/" />
```

Instead of including a URL as its attribute, it includes the fetch attribute which indicates the URL for this remote should be derived from the URL used to initialize the workspace (the argument to the -u option).

```
1 git ls-remote -h https://android.googlesource.com/platform/manifest.git
2 repo init -u https://android.googlesource.com/platform/manifest -b android-10.0.0_r33
3 git config --global user.email "mg.weiminn@gmail.com"
4 git config --global user.name "weiminn"
5 repo init -u https://android.googlesource.com/platform/manifest -b android-10.0.0_r33
6 repo sync -j31
7 source build/envsetup.sh
8 lunch sdk_phone_x86_64-userdebug
9 make -j31
```

Ubuntu 23 does not have the repository for libncurses5 because they already have libncurses6, so you have to add old focal repository to your /etc/apt/sources.list:

```
1 deb http://security.ubuntu.com/ubuntu focal-security main universe
```

After including the old repo, update the repository index and install libncurses5:

```
1 sudo apt update
2 sudo apt install libncurses5
```

## 1.5 November 19, 2023

### 1.5.1 Building AOSP HAL

#### Implementing the HAL

Device driver and its API are usually provided by a third-party hardware provider. Shim code include .h files for one or more device drivers.

New devices can communicate with the processor via USB which is popular. Even without driver, the device can be accessed with generic USB commands. *libusb*<sup>4</sup> is

---

<sup>4</sup><https://libusb.info>

a portable, user-mode, and USB-version agnostic library for using USB devices, and supports Android. If the device has a driver, it is likely to be a specialization of the USB command.

```

1  #ifndef PROXIMITY_HAL_H
2  #define PROXIMITY_HAL_H
3
4  #include <hardware/hardware.h>
5
6  #define ACME_PROXIMITY_SENSOR_MODULE "libproximityhal"
7
8  typedef struct proximity_sensor_device proximity_sensor_device_t;
9
10 struct value_range {
11     int min; int range;
12 };
13
14 typedef struct proximity_params {
15     struct value_range precision;
16     struct value_range proximity;
17 }
18
19 proximity_params_t;
20
21 struct proximity_sensor_device { hw_device_t common;
22     int fd;
23     proximity_params_t params;
24     int (*poll_sensor)(proximity_sensor_device_t *dev, int precision);
25 }; #endif // PROXIMITY_HAL_H

```

## 1.6 November 20, 2023

### 1.6.1 const Qualifier

When we have variables whose value we know cannot be changed, and We want to prevent code from inadvertently giving a new value to the variable, we define the variable's type as `const`:

```

1  const int bufSize = 512; // input buffer size
2  bufSize = 512; // error: attempt to write to const object

```

Because we can't change the value of a `const` object after we create it, it must be initialized:

```

1  const int i = get_size(); // ok: initialized at run time
2  const int j = 42; // ok: initialized at compile time
3  const int k; // error: k is uninitialized const

```

#### `const` is Local to the File

When a `const` object is initialized from a compile-time constant as in:

```

1  const int bufSize = 512;

```

the compiler will usually replace uses of the variable with its corresponding value during compilation. Thus, when we split a program to multiple files, every file that uses that `const` must have a access to its initializer. In order to see the initializer, the variable must be defined in every file that wants to use the variable's value. To support this, we define the `const` in one file, and declare it in other files that use that object.

```
1 // file_1.cc defines and initializes a const that is accessible to other files
2 extern const int bufSize = fcn();
3
4 // file_1.h
5 extern const int bufSize; // same bufSize as defined in file_1.cc
```

Because `bufSize` is `const`, we must specify `extern` in order for `bufSize` to be used in other files. `extern` signifies that `bufSize` is not local to this file and that its definition will occur elsewhere.

### References to `const`

Unlike an ordinary reference, a reference to `const` cannot be used to change the object to which the reference is bound:

```
1 const int ci = 1024;
2 const int &r1 = ci; // ok: both reference and underlying object are const
3 r1 = 42; // error: r1 is a reference to const int &r2 =
4 ci; // error: nonconst reference to a const object
```

We can bind a reference to a `const` to a nonconst object, literals, or a more general expression:

```
1 int i = 42;
2 const int &r1 = i; // we can bind a const int& to a plain int object
3 const int &r2 = 42; // ok: r1 is a reference to const
4 const int &r3 = r1 * 2; // ok: r3 is a reference to
5 const int &r4 = r * 2; // error: r4 is a plain, nonconst reference
```

It is important to realize that a reference to `const` restricts only what we can do through that reference. Binding a reference to `const` to an object says nothing about whether the underlying object itself is `const`.

### Pointers and `const`

Like a reference to `const`, a pointer to `const` may not be used to change the object to which the pointer points. We may store the address of a `const` object only in a pointer to `const`, where we can modify the pointer itself (change the reference stored inside) but not the object (value) pointed to:

```
1 const double pi = 3.14; // pi is const; its value may not be changed
2 double *ptr = &pi; // error: ptr is a plain pointer
3 const double *cptr = &pi; // ok: cptr may point to a double that is const
4 *cptr = 42; // error: cannot assign to *cptr
```

Like a reference to `const`, a pointer to `const` says nothing about whether the object to which the pointer points is `const`, that there is no guarantee that an object pointed to by a pointer to `const` won't change.

We can have a pointer that is itself `const`, and the address it holds cannot be changed. We indicate that the pointer is `const` by putting the `const` by the `const`:

```

1  int errNumb = 0;
2  int *const curErr = &errNumb; // curErr will always point to errNumb
3  const double pi = 3.14159;
4  const double *const pip = &pi; // pip is a constpointer to a const object
5
6  *pip = 2.72; // error: pip is a pointer to const and cannot be changed as the
               // pointer is const
7  // if the object to which curErrpoints (i.e., errNumb) is nonzero
8  if (*curErr) {
9      errorHandler();
10     *curErr = 0; // ok: reset the value of the object to which curErr is bound
                  // because errNumb is not const
11 }
```

## Constant Expressions

A constant expression is an expression whose value cannot change and that can be evaluated at compile time. A literal is a constant expression. A `const` object that is initialized from a constant expression is also a constant expression:

```

1  const int max_files = 20; // max_files is a constant expression
2  const int limit = max_files + 1; // limit is a constant expression
3  int staff_size = 27; // staff_size is not a constant expression
4  const int sz = get_size(); // sz is not a constant expression
```

Although `staff_size` is initialized from a literal, it is not a constant expression because it is a plain `int`, not a `constint`. Even though `sz` is a `const`, the value of its initializer is not known until run time, and thus, `sz` is not a constant expression.

We can ask the compiler to verify that a variable is a constant expression by declaring the variable in a `constexpr` declaration. Variables declared as `constexpr` are implicitly `const` and must be initiated by constant expressions:

```

1  constexpr int mf = 20; // 20 is a constant expression
2  constexpr int limit = mf + 1; // mf+1 is a constant expression
3  constexpr int sz = size(); // ok only if size is a constexpr function
```

Because a constant expression is one that can be evaluated at compile time, only literal types (arithmetic, reference, and pointer) types can be defined as constant expressions. Custom classes, library IO and string types are not literal types. We can point (or bind) to an object that remains at a fixed address.

`constexpr` declaration applies to the pointer, not the type to which the pointer points:

```

1  const int *p = nullptr; // p is a pointer to a constint
2  constexpr int *q = nullptr; // q is a constpointer to int
```

`p` is a pointer to `const` (low-level), whereas `q` is a constant pointer (top-level).

## 1.6.2 Rudimentary AOSP HAL Application

Create Rudimentary Service source file, `rudi.cpp`, in `AOSP/device/generic/goldfish/app/wei_daemon`

```

1  #include <unistd.h>
2  #include <stdio.h>
3  #include <android/log.h>
4
5  #define DELAY_SECS 2
6  #define ALOG(msg) __android_log_write(ANDROID_LOG_DEBUG, "WEIMINN_PROJECT", msg)
7
8  int main(int argc, char *argv[]) {
9      ALOG("STARTING_WEIMINN_PROJECT");
10
11     int n = 0;
12     while (true) {
13         sleep(DELAY_SECS);
14         n++;
15
16         ALOG("TESTING");
17     }
18 }

```

And create Soong build file, `Android.bp` in the same folder:

```

1  cc_binary {
2      name: "weiminn_daemon",
3      relative_install_path: "hw",
4      init_rc: ["init.weiminn.rc"],
5      header_libs: [
6          "liblog_headers",
7      ],
8      srcs: [
9          "weiminn.cpp"
10     ],
11     shared_libs: [
12         "liblog",
13         "libcutils",
14     ],
15     static_libs: [
16     ],
17     vendor: true,
18     proprietary: true,
19 }

```

Add `wei_rudi_daemon` to the `PRODUCT_PACKAGES+=\` attribute of `AOSP/device/generic/goldfish/vendor.mk`

Add the startup script below to the end of the `init.rc` file:

```

1  service wei_rudi_daemon /vendor/bin/hw/wei_rudi_daemon
2      class main
3      user system
4      group system
5      oneshot

```

But the service cannot start due to SE Policy not being implemented for the service yet:



```

1 11-21 02:13:54.242 1862 1862 W cp : type=1400 audit(0.0:231): avc: denied {
    getattr } for path="/vendor/bin/hw/wei_rudi_daemon" dev="dm-3" ino=110
    scontext=u:r:shell:s0 tcontext=u:object_r:vendor_file:s0 tclass=file
    permissive=0

```

## 1.7 November 21, 2023

### 1.7.1 Fixing SELinux Policy to Start System Service

Add seclabel to the startup script:

```

1 service weiminn_daemon /vendor/bin/hw/weiminn_daemon
2     class main
3     user system
4     group system
5     seclabel u:r:ueventd:s0

```

Add start weiminn\_daemon under on early-init right after start ueventd.

Got a new permission denied error this time:

```

1 11-21 09:45:47.732 0 0 E init : cannot execv('/vendor/bin/hw/weiminn_daemon'). See
    the 'Debugging init' section of init's README.md for tips: Permission denied
2 11-21 09:45:47.733 0 0 I init : Service 'weiminn_daemon' (pid 1402) exited with
    status 127
3 11-21 09:45:47.733 0 0 I init : Sending signal 9 to service 'weiminn_daemon' (pid
    1402) process group...

```

Adding device/generic/goldfish/sepolicy/x86/weiminn.te with following contents:

```

1 type weiminn, domain;
2 permissive weiminn;
3 type weiminn_exec, vendor_file_type, exec_type, file_type;
4
5 init_daemon_domain(weiminn)

```

and changed seclabel of the startup script to seclabel u:r:weiminn\_exec:s0. And it reverts to the previous error:

```

1 11-21 10:19:43.865 0 0 I init : starting service 'weiminn_daemon'...
2 11-21 10:19:43.866 0 0 F init : cannot setexeccon('u:r:weiminn_exec:s0') for
    weiminn_daemon: Invalid argument
3 11-21 10:19:43.867 0 0 I init : Service 'weiminn_daemon' (pid 1827) exited with
    status 6
4 11-21 10:19:43.868 0 0 I init : Sending signal 9 to service 'weiminn_daemon' (pid
    1827) process group...
5 11-21 10:19:43.868 0 0 I libprocessgroup : Successfully killed process cgroup uid
    1000 pid 1827 in 0ms

```

### 1.7.2 Memory Dump

#### Try on Android Emulator

Set up emulator environment by appending the following inside ~/.bashrc:

```

1 export ANDROID_SDK_ROOT=~/.Android/Sdk
2 export ANDROID_HOME=~/.Android/Sdk
3 export ANDROID_AVD_HOME=~/.android/avd
4
5 PATH=$PATH:$ANDROID_SDK_ROOT/emulator

```

Register the path:

```
1 source ~/.bashrc
```

Create new AVD, Pixel 7 Pro with Tramisu (Android 13 with Google APIs, not Google Play), via Android Studio.

Run emulator using command:

```
1 emulator -avd Pixel_7_Pro_API_33
```

Load APK into emulator:

```

1 adb devices # to get id of running instance
2 adb root
3 adb install -r de.drmaxnix.birthdaycountdown.apk

```

Clone Fridump<sup>5</sup>:

```

1 git clone https://github.com/Nightbringer21/fridump
2 code fridump
3 pip install frida frida-tools

```

Download and Run Fridump dependencies:

```

1 git clone https://github.com/Nightbringer21/fridump
2 code fridump
3 adb push frida-server /data/local/tmp
4 sudo sysctl kernel.yama.pttrace_scope=0 && frida-ps -D emulator-5554
5 frida-ps -D emulator-5554 | grep Birth # to get process ID of the Birthday app
6
7 # inside emulator shell
8 ./data/local/tmp/frida-server

```

Found out that Fridump doesn't support the latest Android 13 and API 34, so download Android 10 with API 29 with Pixel 3a.

Run Fridump:

```
1 python fridump.py -U -s Birthday\ Countdown
```

and it works now!

## 1.8 November 22, 2023

### 1.8.1 C++ Data Structure

#### Sales\_item Struct

The data structure does not support any operations<sup>6</sup> any requires the user to implement the operations themselves:

<sup>5</sup><https://pentestcorner.com/introduction-to-fridump>

<sup>6</sup>Basically, class with only attributes, and no methods.

```

1 struct Sales_data {
2     std::string bookNo;
3     unsigned units_sold = 0;
4     double revenue = 0.0;
5 };

```

The names defined inside the class must be unique within the class but can reuse names defined outside the class. We define data members the same way that we define normal variables: We specify a base type followed by a list of one or more declarators.

The close curly that ends the class body must be followed by a semicolon. The semicolon is needed because we can define variables after the class body:

```

1 struct Sales_data { /* ... */ } accum, trans, *salesptr;
2 // equivalent, but better way to define these objects
3 struct Sales_data { /* ... */}; Sales_data accum, trans, *salesptr;

```

The semicolon marks the end of the (usually empty) list of declarators. Ordinarily, it is a bad idea to define an object as part of a class definition. Doing so obscures the code by combining the definitions of two different entities—the class and a variable—in a single statement.

We use the dot operator (.) to read into the member attributes of the object:

```

1 Sales_data data1, data2;
2 double price = 0; // price per book, used to calculate total revenue
3
4 // read the first transactions: ISBN, number of books sold, price per book
5 std::cin >> data1.bookNo >> data1.units_sold >> price;
6
7 // calculate total revenue from price and units_sold
8 data1.revenue = data1.units_sold * price;
9
10 // read the second transaction
11 std::cin >> data2.bookNo >> data2.units_sold >> price;
12 data2.revenue = data2.units_sold * price;

```

## Preprocessor

In order to ensure that the class definition is the same in each file, classes are usually defined in header files. Typically, classes are stored in headers whose name derives from the name of the class. Thus, we will define our `Sales_data` class in a header file named `Sales_data.h`.

Headers often need to use facilities from other headers. For example, because our `Sales_data` class has a string member, `Sales_data.h` must `#include` the string header. As we've seen, programs that use `Sales_data` also need to include the string header in order to use the `bookNo` member. As a result, programs that use `Sales_data` will include the string header twice: once directly and once as a side effect of including `Sales_data.h`. Because a header might be included more than once, we need to write our headers in a way that is safe even if the header is included multiple times.

The **preprocessor** is a program that runs before the compiler and changes the source text of our programs. Our programs already rely on one preprocessor facility, `#include`. When the preprocessor sees a `#include`, it replaces the `#include` with the contents of the specified header.

Header guards can be defined using the preprocessor. Preprocessor variables have one of two states: defined and not defined. `#define` directive takes a name as a preprocessor variable. `#ifdef` is true if the variable has been defined, and `#ifndef` is true if the variable has not been defined. If the test is true, then everything following the `#ifdef` or `#ifndef` is processed up to the matching `#endif`:

```

1  #ifndef SALES_DATA_H
2
3  #define SALES_DATA_H
4  #include <string>
5
6  struct Sales_data {
7      std::string bookNo;
8      unsigned units_sold = 0;
9      double revenue = 0.0;
10 };
11
12 #endif

```

The first time `Sales_data.h` is included, the `#ifndef` test will succeed. The preprocessor will process the lines following `#ifndef` up to the `#endif`. As a result, the preprocessor variable `SALES_DATA_H` will be defined and the contents of `Sales_data.h` will be copied into our program. If we include `Sales_data.h` later on in the same file, the `#ifndef` directive will be false. The lines between it and the `#endif` directive will be ignored.

Preprocessor variables, including names of header guards, must be unique throughout the program. Typically we ensure uniqueness by basing the guard's name on the name of a class in the header. To avoid name clashes with other entities in our programs, preprocessor variables usually are written in all uppercase.

## 1.9 November 23, 2023

### 1.9.1 C++ Types and Data Structures

#### Aliases

Traditionally, we use `typedef` for synonym for another type:

```

1  typedef double wages; // wages is a synonym for double
2  typedef wages base, *p; // base is a synonym for double, p for double*

```

The new standard introduced a second way to define **alias declaration** type alias:

```

1  using SI = Sales_item; // SI is a synonym for Sales_item

```

It is also possible to declare "pointer to" alias:

```

1 typedef char *pstring;
2 const pstring cstr = 0; // equivalent to char *const cstr = 0;
3 const pstring *ps; // equivalent to const char *ps;

```

## auto Type Specifier

We can let the compiler deduce the type from the initializer for us by using the auto type specifier:

```

1 // the type of item is deduced from the type of the result of adding val1 and val2
2 auto item = val1 + val2; // item initialized to the result of val1+val2

```

The initializer for all the variables must have types that are consistent with each other:

```

1 auto i = 0, *p = &i; // ok: i is int and p is a pointer to int
2 auto sz = 0, pi = 3.14; // error: inconsistent types for sz and pi

```

auto ignores top-level consts, and only low-level initializer are usually kept:

```

1 auto i = 0, *p = &i; // ok: i is int and p is a pointer to int
2 auto sz = 0, pi = 3.14; // error: inconsistent types for sz and pi

```

## 1.10 November 25, 2023

### 1.10.1 Android Memory Dump

#### Heap Dump

```

1 adb shell am dumpheap <PID> <HEAP-DUMP-FILE-PATH>
2 adb shell cat <HEAP-DUMP-FILE-PATH> > <LOCAL-FILE-PATH>
3 strings <LOCAL-FILE-PATH> <LOCAL-FILE-PATH-FOR-STRINGS>
4
5 pip install objection
6 frida-ps -Uai
7 # objection -g de.drmaxnix.birthdaycountdown explore
8 objection -g bloodpressure.bpdinary explore
9
10 android hooking list classes #List all loaded classes, As the target application
    gets usedmore, this command will return more classes.
11
12 android hooking search classes bloodpressure.bpdinary
13 android hooking search methods bloodpressure.bpdinary recordDbActivity
14
15 android hooking watch class bloodpressure.bpdinary.recordDbActivity --dump-args --
    dump-return
16
17 android hooking watch class_method bloodpressure.bpdinary.recordDbActivity.
    showWeight --dump-args --dump-backtrace --dump-return

```

## 1.11 November 27, 2023

### 1.11.1 Android Profile Guided Compilation

7 8

#### Baseline Profiles

```

1 adb shell cmd package dump-profiles bloodpressure.bpdiaary
2 adb shell cat /data/misc/profman/bloodpressure.bpdiaary-primary.prof.txt >
  bpdiaary_profile.txt

```

#### Perfetto

##### Simpleperf

9

```

1 git clone https://android.googlesource.com/platform/system/extras
2 cd extras/simpleperf/demo/scripts/
3 python3 app_profiler.py -p simpleperf.example.java
4 python3 report_html.py --add_source_code --source_dirs ../demo --add_disassembly
5 sudo apt-get install python3-tk
6 ./report.py

1 python3 app_profiler.py -p bloodpressure.bpdiaary
2 adb shell /data/local/tmp/simpleperf record -o /data/local/tmp/perf.data -e task-
  clock:u -f 1000 -g --duration 10 --log info --app bloodpressure.bpdiaary

```

#### Profcollct

10

This is only supported by Coresight ETM-enabled ARM devices, so emulator doesn't work.

Inside ADB shell:

```

1 device_config put profcollect_native_boot enabled true
2 setprop persist.device_config.profcollct_native_boot.collection_interval 60
3 setprop persist.device_config.profcollct_native_boot.sampling_period 1000
4 setprop persist.device_config.profcollct_native_boot.max_trace_limit 53687091200
5 setprop persist.device_config.profcollct_native_boot.enabled true
6 setprop ctl.stop profcollectd
7 setprop ctl.start profcollectd
8 ps -e | grep profcollectd

```

---

<sup>7</sup><https://developer.android.com/games/agde/pgo-overview>

<sup>8</sup><https://newandroidbook.com/files/ArtOfDalvik.pdf>

<sup>9</sup><https://android.googlesource.com/platform/system/extras/+/main/simpleperf/demo/README.md>

<sup>10</sup><https://android.googlesource.com/platform/system/extras/+/master/profcollectd/>

## 1.12 November 29, 2023

### 1.12.1 Android OAT Dump Parser

The format for OAT files is described in AOSP/art/dex2oat/linker/oat\_writer.h.

.bss<sup>11</sup> is the portion of an object file, executable, or assembly language code that contains statically allocated variables that are declared but not have been assigned a value yet.

```

1  import argparse
2
3  # helper function
4  def findLineWith(arr, s):
5      for i in range(len(arr)):
6          if s in arr[i]:
7              return i
8
9  def parse_method(method_arr):
10     method_sig_raw = method_arr.pop(0)
11     method_sig_raw_ = method_sig_raw[0: method_sig_raw.index(',')]+1]
12     method_sig_raw__ = method_sig_raw_.split(':')[1:]
13     method_sig = method_sig_raw__[0].strip()
14
15     method_arr.pop(0) # discard "DEX CODE:"
16     endOfDex = findLineWith(method_arr, "OatMethodOffsets")
17     dexCode = method_arr[0: endOfDex]
18
19     startOfOat = findLineWith(method_arr, "CODE:")
20     nativeCode = method_arr[startOfOat+1:] # Exclude "CODE: "
21
22     parsed_method = {'method_sig': method_sig, 'dex': dexCode, 'native':
23         nativeCode}
24
25     # print("Parsed", method_sig)
26
27     return parsed_method
28
29 def parse_type(type_arr):
30     method_indexes = [i for i in range(len(type_arr)) if 'method_idx' in type_arr[
31         i]]
32     if len(method_indexes) > 0:
33         method_indexes.append(len(type_arr))
34     methods_raw = [type_arr[method_indexes[i]:method_indexes[i+1]] for i in range
35         (0, len(method_indexes)-1)]
36
37     methods = []
38     for mraw in methods_raw:
39         methods.append(parse_method(mraw))
40
41     return {'type_name': type_arr[0].split()[1], 'methods': methods}
42
43 def load_oat_dump(p):
44     f = open(p)
45     lines = f.readlines()
46

```

---

<sup>11</sup><https://en.wikipedia.org/wiki/.bss>

```

44     # determine the start and end of DEX code
45     startOfOatDex = lines.index('OatDexFile:\n')
46     endOfOatDex = lines.index('OAT_FILE_STATS:\n')
47     OatDex = lines[startOfOatDex:endOfOatDex]
48
49     # extract the classes and types
50     type_indexes = [i for i in range(len(OatDex)) if 'type_idx' in OatDex[i]]
51     type_indexes.append(len(OatDex))
52     all_types = [OatDex[type_indexes[i]:type_indexes[i+1]] for i in range(0, len(
        type_indexes)-1)]
53
54     all_methods = []
55     all_methods_dict = {}
56
57     for t in all_types:
58         parsed = parse_type(t)
59         all_methods += parsed['methods']
60
61         for m in parsed['methods']:
62             all_methods_dict[m['method_sig']] = {}
63             all_methods_dict[m['method_sig']]['dex'] = m['dex']
64             all_methods_dict[m['method_sig']]['native'] = m['native']
65
66     # return all_methods
67     return all_methods_dict
68
69 def load_schema(p):
70     f = open(p)
71     lines = f.readlines()
72
73     schema = []
74     for l in lines:
75         split = l.split('|||')
76         schema.append(split[1][:-1]) # -1 to remove \n character
77         # print("Added schema", schema[-1])
78     return schema
79
80 parser = argparse.ArgumentParser(description='OAT_Debloatation_Checker')
81 parser.add_argument('--oat', type=str, help='OAT_Dump_TXT_file')
82 parser.add_argument('--schema', type=str, help='Debloating_schema_file')
83
84 try:
85     args = parser.parse_args()
86     # print("Options:", args.OAT, args.schema)
87
88     # Load Debloated OAT Dump
89     print("Debloated_OAT_Dump:", args.oat)
90     normal_methods = load_oat_dump(args.oat)
91     # normal_methods = load_oat_dump('prelim_results/com.storiestime/10
        min_normal_com.storiestime_oat_dump.txt')
92
93     # Load Schema
94     print("Schema_File:", args.schema)
95     schema = load_schema(args.schema)
96     # schema = load_schema('Money_script/success_schema/169_com.
        storiestime_removed_methods.txt')
97
98     # Check if debloated
99     for m in schema:

```



```
100         retrieve = normal_methods[m]
101         if 'NO_CODE!' in retrieve['native'][0]:
102             print(m, 'is debloated!')
103         else:
104             print(m, 'is not debloated!')
105
106         # print('test')
107     except Exception as e:
108         print("Exception!", str(e))
```



# Chapter 2

# December

## 2.1 December 01, 2023

### 2.1.1 C++ Strings and Vectors

#### Namespace using Declarations

The scope operator (::) says that the compiler should look in the scope of the left-hand operand for the name of the right-hand operand. using declaration lets us use a name from a namespace without qualifying the name with namespace:: prefix as below:

```
1 #include <iostream>
2
3 // when we use the name cin, we get the one from the namespace std
4 using namespace::name;
5
6 int main(){
7     int i;
8     cin >> i;
9     cout << i;
10    std::cout << i;
11    return 0;
12 }
```

Headers should NOT include using declarations, as the contents of a header are copied into the including program's text. As a result, a program that didn't intend to use the specified library name might encounter unexpected name conflicts.

#### string Type

If you include a header that includes <string> you may not have to do so explicitly. However, it is bad practice to count on this, and well-written headers include guards against multiple inclusion, so assuming you're using well-written header files, there is

no harm in including a header that was included via a previous include.<sup>1</sup>:

```
1 #include <string> // You need to include the <string> header to use std::string.
2 using std::string;
```

Most common ways to initialize strings:

```
1 // direct initialization
2 string s1; // default initialization; s1 is the empty string
3 string s4(10, 'c'); // s4 is cccccccccc
4 string s6("hiya");
5
6 // copy initialization
7 string s2 = s1; // s2 is a copy of s1
8 string s3 = "hiya"; // equivalent to string s3("hiya");
9 string s8 = string(10, 'c') // copying initialization; s8 is cccccccccc
```

When we initialize a variable using `=`, we are asking the compiler to copy initialize the object by copying the initializer on the right-hand side into the object being created. When we omit the `=`, we use direct initialization.

string Operations:

Statement	Description
<code>os&lt;&lt;s</code>	Writes <code>s</code> onto output stream <code>os</code> . Return <code>os</code> .
<code>is&gt;&gt;s</code>	Reads whitespace-separated string from <code>is</code> to <code>s</code> . Return <code>is</code> .
<code>getline(is,s)</code>	Reads a line of input from <code>is</code> into <code>s</code> . Returns <code>is</code> .
<code>s.empty()</code>	Returns true if <code>s</code> is empty; otherwise return false.
<code>s.size()</code>	Returns the number of characters in <code>s</code> .
<code>s[n]</code>	Returns a reference to the char at position <code>n</code> in <code>s</code> .
<code>s1 + s2</code>	Returns a string that is the concatenation of <code>s1</code> and <code>s2</code> .
<code>s1 += s2</code>	Equivalent to <code>s1 = s1 + s2</code> .
<code>s1 = s2</code>	Replaces characters in <code>s1</code> with a copy of <code>s2</code> .
<code>s1 == s2</code> and <code>s1 != s2</code>	The strings in <code>s1</code> and <code>s2</code> are equal if they contain the same characters. The equality is case-sensitive.
<code>&lt;, &lt;=, &gt;, &gt;=</code>	Comparisons are case-sensitive and use dictionary ordering.

Table 2.1: string Operations

To check individual character:

Function	Description
<code>isalnum(c)</code>	true if <code>c</code> is a letter or a digit.
<code>isalpha(c)</code>	true if <code>c</code> is a letter.
<code>isctrl(c)</code>	true if <code>c</code> is a control character.

<sup>1</sup><https://stackoverflow.com/a/73640984>

Function	Description
<code>digit(c)</code>	true if <code>c</code> is a digit.
<code>isgraph(c)</code>	true if <code>c</code> is not a space but is printable.
<code>islower(c)</code>	true if <code>c</code> is a lowercase letter.
<code>isprint(c)</code>	true if <code>c</code> is a printable character (i.e., a space or a character that has a visible representation).
<code>ispunct(c)</code>	true if <code>c</code> is a punctuation character.
<code>isspace(c)</code>	true if <code>c</code> is whitespace (i.e., a space, tab, vertical tab, return, newline, or formfeed).
<code>isupper(c)</code>	true if <code>c</code> is an uppercase letter.
<code>isxdigit(c)</code>	true if <code>c</code> is a hexadecimal digit.
<code>tolower(c)</code>	If <code>c</code> is an uppercase letter, returns its lowercase equivalent; otherwise returns <code>c</code> unchanged.
<code>toupper(c)</code>	If <code>c</code> is a lower letter, returns its uppercase equivalent; otherwise returns <code>c</code> unchanged.

Table 2.2: ctype Functions

### Reading an Unknown Number of strings

If the stream is valid - it hasn't hit end-of-file or encountered an invalid input - then the body of while is executed:

```

1  int main() {
2      string word;
3      while (cin >> word) // read until end-of-file
4          cout << word << endl; // write each word followed by a new line
5
6      // Reads the given stream up to and including the first newline
7      string line; // read input a line at a time until end-of-file
8      while (getline(cin, line))
9          // The newline that causes getline to return is discarded; the newline is
           not stored in the string.
10         cout << line << endl;
11
12     while (getline(cin, line))
13         // Only print lines that are not empty
14         if (!line.empty())
15             cout << line << endl;
16
17     return 0;
18 }
```

### Library strings and String Literals

When we mix strings and string or character literals, at least one operand to each + operator must be of string type:

```

1 string s4 = s1 + ",_"; // ok: adding a string and a literal
2 string s5 = "hello" + ",_"; // error: no string operand
3 string s6 = s1 + ",_" + "world"; // ok: each + has a string operand
4 string s7 = "hello" + ",_" + s2; // error: can't add string literals

```

For compatibility reasons with C, string literals are NOT standard library strings. It is important to remember that these types differ when you use string literals and library strings.

### Characters in strings

string expression represent a sequence of characters, and to traverse every character, we can use a range for that follows the syntax:

```

1 for (declaration: expression)
2     statement

```

A simple example:

```

1 string str("some_string");
2 // print the characters in str one character to a line
3 for (auto c : str) // for every char in str
4     cout << c << endl; // print the current character followed by a newline

```

We use auto to let compiler deduce the type of c, which in this case will be char.

If we want to change the value of the character in a string, we must define the loop variable as a reference type:

```

1 string str("some_string");
2 // convert s to uppercase
3 for (auto &c : str) // for every char ref in str
4     c = toupper(s); // c is a reference, so the assignment changes the char in s

```

The subscript operator (the [] operator) takes a string::size\_type value that denotes the position of the character we want to access. The operator returns a reference to the character at the given position:

```

1 string str("some_string");
2 if (!s.empty()) // make sure there's a character to print
3     cout << s[0] << endl; // print the first character in s

```

To iterate using subscript:

```

1 // process characters in s until we run out of characters or we hit a whitespace
2 for (
3     decltype(s.size()) index = 0;
4     index != s.size() && !isspace(s[index]); // the operator yields true if both
        operands are true
5     ++index)
6     s[index] = toupper(s[index]); // capitalize the current character

```

## vector Type

A vector is a collection<sup>2</sup> of objects, all of which have the same type, and each of which has an associated index that gives access to that object. Below is the headers to use a vector:

```
1 #include <vector>
2 using std::vector;
```

vector is not a class/type but a class template, which is a set of instructions for the compiler for generating classes/types, a process called instantiation. To specify what kind of class (what type of object we want the vector to hold) we want to instantiate, we supply additional information inside a pair of angle brackets following the template's name:

```
1 vector<int> ivec; // ivec holds objects of type
2 int  vector<Sales_item> Sales_vec; // holds Sales_items
3 vector<vector<string>> file; // vector whose elements are vectors
```

Here, vector<int>, vector<Sales\_item>, and vector<vector<string>> are the types generated types by the compiler.

Because references are not objects, we cannot have a vector of references.

## Defining and Initializing vectors

The most common way of Initializing a vector is to initialize an empty vector. We can also perform direct and copy initialization, but the objects must be the same type:

```
1 vector<string> svec; // default initialization; svec has no elements
2
3 // direct and copy initialization
4 vector<int> ivec2(ivec); // copy elements of ivec into ivec2
5 vector<int> ivec3 = ivec; // copy elements of ivec into ivec3
6 vector<string> svec(ivec2); // error: svec holds strings, not ints
7
8 // list initialization
9 vector<string> articles = {"a", "an", "the"};
10 vector<string> articles2{"a", "an", "the"};
11 vector<string> articles3("a", "an", "the"); // error
12
13 vector<int> ivec(10, -1); // ten int elements, each initialized to -1
14 vector<string> svec(10, "hi!"); // ten strings; each element is "hi!"
15
16 vector<int> ivec(10); // ten elements, each initialized to 0
17 vector<string> svec(10); // ten elements, each an empty string
```

Some classes require that we always supply an explicit initializer, and cannot be default initialized, in which case, we must supply the initial value/

---

<sup>2</sup>Often referred to as container because it "contains" other objects.

## vector Operations

Two vectors are equal if they have the same number of elements, and if the corresponding elements all have the same value. If the vectors have differing sizes, but the elements that are in common are equal, then the vector with fewer elements is less than the one with more elements. If the elements have differing values, then the relationship between the vectors is determined by the relationship between the first elements that differ. We can compare two vectors only if we can compare the element in those vectors.

Methods	Description
<code>v.empty()</code>	Returns true if <code>v</code> is empty; otherwise returns false.
<code>v.size()</code>	Returns the number of elements in <code>v</code> .
<code>v.push_back(t)</code>	Adds an element with value <code>t</code> to the end of <code>v</code> .
<code>v[n]</code>	Returns a reference to the element at position <code>n</code> in <code>v</code> .
<code>v1 = v2</code>	Replaces the elements in <code>v1</code> with a copy of the elements in <code>v2</code> .
<code>v1 = {a, b, c, ...}</code>	Replaces the elements in <code>v1</code> with a copy of the elements in the comma-separated list.
<code>v1 == v2</code> and <code>v1 != v2</code>	<code>v1</code> and <code>v2</code> are equal if they have the same number of elements and each element in <code>v1</code> is equal to corresponding element in <code>v2</code> .
<code>&lt;</code> , <code>&lt;=</code> , <code>&gt;</code> , <code>&gt;=</code>	Have their normal meanings using dictionary ordering.

**Table 2.3:** vector Methods

As with strings, subscript for vector start at 0; the type of a subscript is the corresponding `size_type`; and we can write to the element returned by the subscript operator.

Subscripting a vector does NOT add elements:

```

1 vector<int> ivec; // empty vector
2 for (decltype(ivec.size()) ix = 0; ix != 10; ++ix)
3     ivec[ix] = ix; // disaster: ivec has no elements

```

It is an error to subscript an element that doesn't exist, but it is an error that the compiler is unlikely to detect. Instead, the value we get at run time is undefined<sup>3</sup>. A good way to ensure that subscripts are in range is to avoid subscripting altogether by using a range for whenever possible.

<sup>3</sup>Buffer overflow errors are the result of subscripting elements that don't exist. Such bugs are the most common cause of security problems in PC and other applications.



## 2.2 December 03, 2023

### 2.2.1 C++ Iterator, and Arrays

#### Iterators

Iterators are more general mechanism than subscript operators. All of the library containers have iterators, but only a few of them support the subscript operator. Like pointers, iterators give us indirect access to an object.

```
1 // the compiler determines the type of b and e;
2 // b denotes the first element and e denotes one past the last element in v
3 auto b = v.begin(), e = v.end(); // b and e have the same type
```

The begin member returns an iterator that denotes the first element (if there is one). The end member returns an iterator positioned "one past the end" of the associated container (or string), also referred to as the off-the-end iterator. If the container is empty, begin returns the same iterator as the one returned by end.

We do not know (or need to care about) the precise type that an iterator has, so we use auto to define b and e.

We compare two valid iterators using == or !=. Iterators are equal if they denote the same element or if they are both off-the-end iterators for the same container. Otherwise, they are unequal.

Like pointers, we can dereference a valid iterator to obtain the element denoted by an iterator. Dereferencing an invalid iterator or an off-the-end iterator has undefined behavior:

```
1 string s("some_string");
2 if (s.begin() != s.end()) { // make sure s is not empty
3     auto it = s.begin(); // it denotes the first character in s
4     *it = toupper(*it); // make that character uppercase
5 }
```

Iterators also support a few other operations:

Methods	Description
*iter	Returns a reference to the element denoted by the iterator iter.
iter->mem	Dereferences iter and fetches the member named mem from the underlying element. Equivalent to (*iter).mem.
++iter	Increments iter to refer to the next element in the container.
--iter	Decrements iter to refer to the previous element in the container.
iter1 == iter2 and iter1 != iter2	Compares two iterators for equality. Two iterators are equal if they denote the same element or if they are off-the-end iterator for the same container.

Table 2.4: Iterator operations

The increment (`++`) operator to move from one element to the next:

```
1 string s("some_string");
2 for (auto it = s.begin(); it != s.end() && !isspace(*it); ++it) {
3     *it = toupper(*it); // make that character uppercase
4 }
```

Because the iterator returned by `end` does not denote an element, it may not be incremented or dereferenced.

When we need to read but not write to an object, we ask specifically for `const_iterator` type:

```
1 vector<int> v;
2 auto b = v.cbegin(); // b has type vector<int>::const_iterator
3 auto e = v.cend(); // e has type vector<int>::const_iterator
```

Regardless of whether the container is `const`, they return a `const_iterator`.

Iterators for `string` and `vector` support additional operations that can move an iterator multiple elements at a time, often referred to as iterator arithmetic:

Iter. Arithmetic	Description
<code>iter + n</code> and <code>iter - n</code>	Adding (subtracting) an integral value <code>n</code> to (from) an iterator yields an iterator that many elements forward (or backward) within the container. The resulting iterator must denote elements in, or one past the end of, the same container.
<code>iter += n</code> and <code>iter -= n</code>	Compound-assignment for iterator addition and subtraction. Assigns to <code>iter1</code> the value of adding <code>n</code> to, or subtracting <code>n</code> from, <code>iter1</code> .
<code>iter1 - iter2</code>	Subtracting two iterators yields the number that when added to the right-hand iterator yields the left-hand iterator. The iterators must denote elements in, or one past the end of, the same container.
<code>&gt;, &gt;=, &lt;, &lt;=</code>	Relational operators on iterators. One iterator is less than another if it refers to an element that appears in the container before the one referred to by the other iterator. The iterators must denote elements in, or one past the end of, the same container.

Table 2.5: Iterator Arithmetics

A classic algorithm that uses iterator arithmetic is binary search:

```
1 vector<int> v;
2 // text must be sorted
3 // beg and end will denote the range we're searching
4 auto beg = text.begin(), end = text.end();
```

```

5  auto mid = text.begin() + (end - beg)/2; // original midpoint
6
7  // while there are still elements to look at and we haven't yet found sought
8  while (mid != end && *mid != sought) {
9      if (sought < *mid) // is the element we want in the first half?
10         end = mid; // if so, adjust the range to ignore the second half
11     else // the element we want is in the second half
12         beg = mid + 1; // start looking with the element just after mid
13     mid = beg + (end - beg)/2; // new midpoint
14 }

```

## Arrays

Similar to library vector type, an array is a container of unnamed objects of a single type that we access by position. Unlike a vector, arrays have fixed size; we cannot add elements to an array, in order to attain better runtime performance for specialized applications at the cost of flexibility.

An array declarator has the form `a[d]`, where `a` is the name being defined and `d` is the dimension of the array which specifies the number of elements and must be greater than zero. The dimension must be known at compile time, which means that the dimension must be a `constexpr`:

```

1  unsigned cnt = 42; // not a constant expression
2  constexpr unsigned sz = 42; // constant expression
3  int arr[10]; // array of ten ints
4  int *parr[sz]; // array of 42 pointers to int
5  string bad[cnt]; // error: cnt is not a constant expression
6  string strs[get_size()]; // ok if get_size is constexpr, error otherwise

```

By default, the elements in an array are default initialized. As with vector, arrays hold objects. Thus, there are no arrays of references.

We can list initialize an array which allows us to omit the dimension as the compiler infers it from the number of initializers. If we specify, the number of initializers must not exceed the specified size:

```

1  const unsigned sz = 3;
2  int ia1[sz] = {0,1,2}; // array of three ints with values 0, 1, 2
3  int a2[] = {0, 1, 2}; // an array of dimension 3
4  int a3[5] = {0, 1, 2}; // equivalent to a3[] = {0, 1, 2, 0, 0}
5  string a4[3] = {"hi", "bye"}; // same as a4[] = {"hi", "bye", ""}
6  int a5[2] = {0,1,2}; // error: too many initializers

```

Character arrays can also be initialized from a string literal. It's important to remember that string literals end with a null character:

```

1  char a1[] = {'C', '+', '+'}; // list initialization, no null
2  char a2[] = {'C', '+', '+', '\0'}; // list initialization, explicit null
3  char a3[] = "C++"; // null terminator added automatically
4  const char a4[6] = "Daniel"; // error: no space for the null!

```

We cannot initialize an array as a copy of another array, nor is it legal to assign one array to another<sup>4</sup>:

```
1 int a[] = {0, 1, 2}; // array of three ints
2 int a2[] = a; // error: cannot initialize one array with another
3 a2 = a; // error: cannot assign one array to another
```

Defining arrays that hold pointers is fairly straightforward, defining a pointer or reference to an array is a bit more complicated:

```
1 int *ptrs[10]; // ptrs is an array of ten pointers to int
2 int &refs[10] = /* ? */; // error: no arrays of references
3 int (*Parray)[10] = &arr; // Parray points to an array of ten ints
4 int (&arrRef)[10] = arr; // arrRef refers to an array of ten ints
```

The parentheses around `*Parray` means that `Parray` is a pointer. Looking right, we see that `Parray` points to an array of size 10. Looking left, we see that the elements in the array are ints. Thus, `Parray` is a pointer to an array of ten ints. Similarly, `&arrRef` says that `arrRef` is a reference.

We can use range for or the subscript operator to access elements of an array:

```
1 // count the number of grades by clusters of ten: 0--9, 10--19, . . . 90--99, 100
2 unsigned scores[11] = {}; // 11 buckets, all value initialized to 0
3 unsigned grade;
4 while (cin >> grade) {
5     if (grade <= 100)
6         ++scores[grade/10]; // increment the counter for the current cluster
7 }
8
9 for (auto i : scores) // for each counter in scores
10     cout << i << " "; // print the value of that counter
11 cout << endl;
```

We have to use a variable to have type `size_t` (defined in `cstdint` header) which is a machine-specific unsigned type that is guaranteed to be large enough to hold the size of any object in memory.

The most common source of security problems are buffer overflow bugs. Such bugs occur when a program fails to check a subscript and mistakenly uses memory outside the range of an array or similar data structure. Nothing stops a program from stepping across an array boundary except careful attention to detail and thorough testing of the code.

We obtain a pointer to an array element by taking the address of that element:

```
1 string nums[] = {"one", "two", "three"}; // array of strings
2 string *p = &nums[0]; // p points to the first element in nums
3 string *p2 = nums; // equivalent to p2=&nums[0]
```

When we use an object of array type, we are really using a pointer to the first element in that array, as the compiler automatically substitutes a pointer to the first element.

When we use an array as an initializer for a variable defined using `auto`, the deduced type is a pointer, not an array:

---

<sup>4</sup>Some compilers allow array assignment as a compiler extension. It is usually a good idea to avoid using nonstandard features. Programs that use such features, will not work with a different compiler.

```

1 int ia[] = {0,1,2,3,4,5,6,7,8,9}; // ia is an array of ten ints
2 auto ia2(ia); // ia2 is an int* that points to the first element in ia
3 ia2 = 42; // error: ia2 is a pointer, and we can't assign an int to a pointer

```

Although `ia` is an array of ten ints, when we use `ia` as an initializer, the compiler treats that initialization as if we had written:

```

1 auto ia2(&ia[0]); // now it's clear that ia2 has type int*

```

This conversion does not happen when we use `decltype`:

```

1 // ia3 is an array of ten ints
2 decltype(ia) ia3 = {0,1,2,3,4,5,6,7,8,9};
3 ia3 = p; // error: can't assign an int* to an array
4 ia3[4] = i; // ok: assigns the value of i to an element in ia3

```

Pointers to array elements support the same operations as iterators on vectors or strings:

```

1 // ia3 is an array of ten ints
2 int arr[] = {0,1,2,3,4,5,6,7,8,9};
3 int *p = arr; // p points to the first element in arr
4 ++p; // p points to arr[1]
5 int *e = &arr[10]; // pointer just past the last element in arr

```

`arr` has 10 elements, so the last element in `arr` is at index 9. Like the off-the-end iterator, off-the-end pointer does not point to an element. As a result, we may not dereference or increment an off-the-end pointer.

To be safer and less error-prone, we can use `begin` and `end` functions that act like similarly named container members. However, as arrays are not class types, these are not member functions, so they take an argument that is an array:

```

1 int ia[] = {0,1,2,3,4,5,6,7,8,9}; // ia is an array of ten ints
2 int *beg = begin(ia); // pointer to the first element in ia
3 int *last = end(ia); // pointer one past the last element in ia

```

Pointers that address array elements can use all iterator operations in Table 2.4 and Table 2.5. When we add an integral value to or from a pointer, the result is a new pointer. That new pointer points to the element the given number ahead of the original pointer:

```

1 constexpr size_t sz = 5;
2 int arr[sz] = {1,2,3,4,5};
3 int *ip = arr; // equivalent to int*ip=&arr[0]
4 int *ip2 = ip + 4; // ip2 points to arr[4], the last element in arr
5
6 // ok: arr is converted to a pointer to its first element; p points one past the end
  of arr int *p = arr + sz; // use caution -- do not dereference! int *p2 = arr
  + 10; // error: arr has only 5 elements; p2 has undefined value

```

When we add `sz` to `arr`, the compiler converts `arr` to a pointer to the first element in `arr`. As a result, we can dereference the resulting pointer:

```

1 int ia[] = {0,2,4,6,8}; // array with 5 elements of type int
2 int last = *(ia + 4); // ok: initializes last to 8, the value of ia[4]
3 last = *ia + 4; // ok: last=4, equivalent to i

```

As with iterators, subtracting two pointers gives us the distance between those pointers. The pointers must point to elements in the same array:

```
1 auto n = end(arr) - begin(arr); // n is 5, the number of elements in arr
```

The result of subtracting two pointers is a library type named `ptrdiff_t` which is a machine-specific type and is defined in the `cstdint` header.

When we subscript an array, we are subscripting a pointer to an element in that array:

```
1 int i = ia[2]; // ia is converted to a pointer to the first element in ia
2 // ia[2] fetches the element to which (ia+2) points
3 int *p = ia; // p points to the first element in ia
4 i = *(p + 2); // equivalent to i = ia[2]
5 int k = p[-2]; // p[-2] is the same element as ia[0]
```

Unlike subscripts for vector and string, the index of the built-in subscript operator is not an unsigned type.

Modern C++ programs should use vectors and iterators instead of built-in arrays and pointers, and use strings rather than C-style array-based character strings. Pointers are used for low-level manipulations and it is easy to make bookkeeping mistakes. Other problems arise because of the syntax, particularly the declaration syntax used with pointers.

## Multidimensional Arrays

Multidimensional arrays in C++ are actually arrays of arrays:

```
1 int ia[3][4]; // array of size 3; each element is an array of ints of size 4
2 // array of size 10; each element is a 20-element array whose elements are arrays
  // of 30 ints
3 int arr[10][20][30] = {0}; // initialize all elements to 0
4
5 int ia[3][4] = { // three elements; each element is an array of size 4
6     {0, 1, 2, 3}, // initializers for the row indexed by 0
7     {4, 5, 6, 7}, // initializers for the row indexed by 1
8     {8, 9, 10, 11} // initializers for the row indexed by 2
9 };
10
11 // equivalent initialization without the optional nested braces for each row
12 int ia[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
13
14 // explicitly initialize only element 0 in each row
15 int ia[3][4] = {{ 0 }, { 4 }, { 8 }};
16
17 // explicitly initialize row 0; the remaining elements are value initialized
18 int ix[3][4] = {0, 3, 6, 9};
19
20 // assigns the first element of arr to the last element in the last row of ia
21 ia[2][3] = arr[0][0][0];
22
23 int (&row)[4] = ia[1]; // binds row to the second four-element array in ia
```

## 2.3 December 04, 2023

### 2.3.1 C++ Expressions I

#### **lvalues and rvalues**

Every expression is either an rvalue or an lvalue. lvalues could stand on the left-hand side of an assignment where as rvalue could not. Roughly speaking, when we use an object as an rvalue, we use the object's value (its contents). When we use an object as an lvalue, we use the object's identity (its location in memory). We can use an lvalue when an rvalue is required, but we cannot use an rvalue when an lvalue (i.e., a location) is required. When we use an lvalue in place of an rvalue, the object's contents (its value) are used:

- Assignment requires a (non const) lvalue as its left-hand operand and yields its left-hand operand as an lvalue.
- The address-of operator requires an lvalue operand and returns a pointer to its operand as an rvalue.
- The built-in dereference and subscript operators and the iterator dereference and string and vector subscript operator all yield lvalues.
- The built-in and iterator increment and decrement operators require lvalue operands and the prefix versions also yield lvalues.

Lvalues and rvalues also differ when used with `decltype`. When we apply `decltype` to an expression, the result is a reference type if the expression yields an lvalue.

#### **Arithmetic Operators**

Division between integers returns an integer. If the quotient contains a fractional part, it is truncated toward zero:

```
1 int ival1 = 21/6; // ival1 is 3; result is truncated; remainder is discarded int
2 ival2 = 21/7; // ival2 is 3; no remainder; result is an integral value
```

For most operators, operands of type `bool` are promoted to `int`. In this case, the value of `b` is `true`, which promotes to the `int` value `1`. That (promoted) value is negated, yielding `-1`. The value `-1` is converted back to `bool` and used to initialize `b2`. This initializer is a nonzero value, which when converted to `bool` is `true`. Thus, the value of `b2` is `true`!

The operands to `%` must have integral type:

```

1  int ival = 42;
2  double dval = 3.14;
3
4  ival % 12; // ok: result is 6
5  ival % dval; // error: floating-point operand

```

## 2.4 December 05, 2023

### 2.4.1 C++ Expressions II

#### Logical and Relational Operators

The relational operators take operands of arithmetic or pointer type; the logical operators take operands of any type that can be converted to `bool`. The operands to those operators are rvalues and the result is an rvalue.

Associativity	Operator	Function	Use
Right	!	logical NOT	!expr
Left	<	less than	expr < expr
Left	<=	less than or equal	expr <= expr
Left	>	greater than	expr > expr
Left	>=	greater than or equal	expr >= expr
Left	==	equality	expr == expr
Left	!=	inequality	expr != expr
Left	&&	logical AND	expr && expr
Left		logical OR	expr    expr

**Table 2.6:** Logical and Relational Operators

Because relational operators return `bool`s, the result of chaining these operators together is likely to be surprising:

```

1  // oops! this condition compares k to the bool result of i<j
2  if(i<j<k) // true if k is greater than 1!

```

The compiler converts `val` to `bool`:

```

1  if (val) { /* ... */} // true if val is any nonzero value
2  if (!val) { /* ... */} // true if val is zero
3  if (val == true) { /* ... */} // true only if val is equal to 1!
4  if (val == 1) { /* ... */}

```

If `val` is not `bool`, then `true` is converted to the type of `val` before the `==` operator is applied.



## Assignment Operators

The left-hand operand of an assignment operator must be a modifiable lvalue. For example, given:

```
1 int i = 0, j = 0, k = 0;    // initializations, not assignment
2 const int ci = i;         // initialization, not assignment
3 1024 = k;                  // error: literals are rvalues
4 i + j = k;                // error: arithmetic expressions are rvalues
5 ci = k;                   // error: ci is a const(nonmodifiable) lvalue
6 k = 0;                    // result: type int, value 0
7 k = 3.14159;              // result: type int, value 3
8 k = {3.14};               // error: narrowing conversion
9 vector<int> vi;            // initially empty
10 vi = {0,1,2,3,4,5,6,7,8,9}; // vi now has ten elements, values 0 through 9
```

Unlike the other binary operators, assignment is right associative. The right-most assignment, `jval = 0`, is the right-hand operand of the left-most assignment operator:

```
1 int ival, jval;
2 ival = jval = 0; // ok: each assigned 0
3 int ival, *pval; // ival is an int; pval is a pointer to int
4 ival = pval = 0; // error: cannot assign the value of a pointer to an int
5 string s1, s2;
6 s1 = s2 = "OK"; // string literal "OK" converted to string
```

Each object in a multiple assignment must have the same type as its right-hand neighbor or a type to which that neighbor can be converted.

Assignment often occurs in conditions. Because assignment has relatively low precedence, we usually must parenthesize the assignment for the condition to work properly:

```
1 // a verbose and therefore more error-prone way to write this loop
2 int i = get_value(); // get the first value
3 while (i != 42) {
4     // do something . . .
5     i = get_value();
6     // get remaining values
7 }
8
9
10 int i;
11 // a better way to write our loop---what the condition does is now clearer
12 while ((i = get_value()) != 42) {
13     // do something . . .
14 }
```

## Assignment Operators

The dot and arrow operators provide for member access. The dot operator fetches a member from an object from an object of class type; arrow is defined so that `ptr->mem` is a synonym for `(*ptr).mem`:

```
1 string s1 = "a_string", *p = &s1;
2 auto n = s1.size(); // run the sizemember of the strings1
```

```

3 n = p->size();           // equivalent to (*p).size()
4 n=(*p).size();          // run size on the object to which p points
5
6 // run the size member of p, then dereference the result!
7 *p.size(); // error: p is a pointer and has no member named size

```

Because dereference has a lower precedence than dot, we must parenthesize the dereference subexpression.

## Conditional Operator

The conditional (the `?:` operator) lets us embed simple if-else logic inside an expression:

```

1 // cond ? expr1: expr2;
2 string final grade = (grade < 60) ? "fail": "pass";

```

where `expr1` and `expr2` are expressions of the same type.

An incompletely parenthesized conditional operator in an output expression can have surprising results:

```

1 cout << ((grade < 60) ? "fail" : "pass"); // prints pass or fail
2 cout << (grade < 60) ? "fail" : "pass"; // prints 1 or 0!
3 cout << grade < 60 ? "fail" : "pass"; // error: compares cout to 60

```

The second expression uses the comparison between `grade` and `60` as the operand to the `<<` operator.

## Bitwise Operator

Because there are no guarantees for how the sign bit is handled, it is strongly recommended to use unsigned types with the bitwise operators.

Operator	Function	Use
	bitwise NOT	<code>expr</code>
<code>&lt;&lt;</code>	left shift	<code>expr1 &lt;&lt; expr2</code>
<code>&gt;&gt;</code>	right shift	<code>expr1 &gt;&gt; expr2</code>
<code>&amp;</code>	bitwise AND	<code>expr1 &amp; expr2</code>
<code>^</code>	bitwise XOR	<code>expr1 ^ expr2</code>
<code> </code>	bitwise OR	<code>expr1   expr2</code>

Table 2.7: Bitwise Operators

The built-in meaning of the shift operators is to perform a bitwise shift on their operands. They yield a value that is a copy of the left-hand operand with the bits

shifted as directed by the right-hand operand. The right-hand operand must not be negative and must be a value that is strictly less than the number of bits in the result. Otherwise, the operation is undefined. The bits are shifted left ( $\ll$ ) or right ( $\gg$ ). Bits that are shifted off the end are discarded:

*These illustrations have the low-order bit on the right*  
*These examples assume char has 8 bits, and int has 32*  
*// 0233 is an octal literal (§ 2.1.3, p. 38)*  
 unsigned char bits = 0233; 1 0 0 1 1 0 1 1

bits  $\ll$  8 // bits promoted to int and then shifted left by 8 bits  
0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 | 1 0 0 1 1 0 1 1 | 0 0 0 0 0 0 0 0

bits  $\ll$  31 // left shift 31 bits, left-most bits discarded  
1 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0

bits  $\gg$  3 // right shift 3 bits, 3 right-most bits discarded  
0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 | 0 0 0 1 0 0 1 1

**Figure 2.1:** Bitwise Shift Operation

Shift operators have midlevel precedence (lower than the arithmetic operators but higher than the relational, assignment, and conditional operators):

```
1 cout << 42 + 10; // ok: + has higher precedence, so the sum is printed
2 cout << (10 < 42); // ok: parentheses force intended grouping; prints 1
3 cout << 10 < 42; // error: attempt to compare cout to 42!
```

The bitwise NOT operator generates a new value with the bits of its operand inverted:

unsigned char bits = 0227; 1 0 0 1 0 1 1 1

~bits  
1 1 1 1 1 1 1 1 | 1 1 1 1 1 1 1 1 | 1 1 1 1 1 1 1 1 | 0 1 1 0 1 0 0 0

**Figure 2.2:** Bitwise NOT Operation

The AND, OR, and XOR operators generate new values with the bit pattern composed from its two operands:

```

unsigned char b1 = 0145;    0 1 1 0 0 1 0 1
unsigned char b2 = 0257;    1 0 1 0 1 1 1 1

b1 & b2  24 high-order bits all 0 0 0 1 0 0 1 0 1
b1 | b2  24 high-order bits all 0 1 1 1 0 1 1 1 1
b1 ^ b2  24 high-order bits all 0 1 1 0 0 1 0 1 0

```

Figure 2.3: Bitwise AND, OR, and XOR Operation

### sizeof Operator

The `sizeof` operator returns the size, in bytes, of an expression or a type name. The operator is right associative. The result of `sizeof` is a constant expression of type `size_t`. The operator takes one of two forms:

```

1 sizeof(type)
2 sizeof expr

```

The `sizeof` operator is unusual in that it does not evaluate its operand:

```

1 Sales_data data, *p;
2 sizeof(Sales_data); // size required to hold an object of type Sales_data
3 sizeof data; // size of data's type, i.e., sizeof(Sales_data)
4 sizeof p; // size of a pointer
5 sizeof *p; // size of the type to which p points, i.e., sizeof(Sales_data)
6 sizeof data.revenue; // size of the type of Sales_data's revenue member
7 sizeof Sales_data::revenue; // alternative way to get the size of revenue

```

Dereferencing an invalid pointer as the operand to `sizeof` is safe because the pointer is not actually used, because `sizeof` does not need to dereference the pointer to know what type it will return.

The result of applying `sizeof` depends in part on the type involved:

- `sizeof char` or an expression of type `char` is guaranteed to be 1.
- `sizeof` a reference type returns the size of an object of the referenced type.
- `sizeof` a pointer returns the size needed to hold a pointer.
- `sizeof` a dereferenced pointer returns the size of an object of the type to which the pointer points; the pointer need not be valid.
- `sizeof` an array is the size of the entire array. It is equivalent to taking the `sizeof` the element type times the number of elements in the array. Note that `sizeof` does not convert the array to a pointer.
- `sizeof` a string or a vector returns only the size of the fixed part of these types; it does not return the size used by the object's elements.

Because `sizeof` returns the size of the entire array, we can determine the number of elements in an array by dividing the array size by the element size.

## Comma Operator

The comma operator takes two operands, which it evaluates from left to right. Like the logical AND and logical OR and the conditional operator guarantees the order in which its operands are evaluated. Most common use for the comma operator is in a for loop:

```
1 vector<int>::size_type cnt = ivec.size();
2 // assign values from size...1 to the elements in ivec
3 for(vector<int>::size_type ix = 0;
4     ix != ivec.size(); ++ix, --cnt)
5     ivec[ix] = cnt;
```

The left-hand expression is evaluated and its result is discarded. The result of a comma expression is the value of its right-hand expression. The result is an lvalue if the right-hand operand is an lvalue.

## Type Conversions

Implicit conversions are carried out automatically without programmer intervention, and are defined to preserve precision, if possible.

- In most expressions, values of integral types smaller than `int` are first promoted to an appropriate larger integral type.
- In conditions, nonbool expressions are converted to `bool`.
- In initializations, the initializer is converted to the type of the variable; in assignments, the right-hand operand is converted to the type of the left-hand.
- In arithmetic and relational expressions with operands of mixed types, the types are converted to a common type.

Conversions also happen during function calls.

Arithmetic conversions:

```
1 bool flag;      char cval;
2 short sval;     unsigned short usval;
3 int ival;       unsigned int uival;
4 long lval;      unsigned long ulval;
5 float fval;     double dval;
6
7 3.14159L + 'a'; // 'a' promoted to int, then that int converted to long double
8 dval + ival;    // ival converted to double
9 dval + fval;    // fval converted to double
10 ival = dval;    // dval converted (by truncation) to int
11 flag = dval;    // if dval is 0, then flag is false, otherwise true
12 cval + fval;    // cval promoted to int, then that int converted to float
```

```

13 sval + cval; // sval and cval promoted to int
14 cval + lval; // cval converted to long
15 ival + ulval; // ival converted to unsigned long
16 usval + ival; // promotion depends on the size of unsigned short and int
17 uival + lval; // conversion depends on the size of unsigned int and long

```

Array to pointer conversion:

```

1 int ia[10]; // array of ten ints
2 int* ip = ia; // convert ia to a pointer to the first element

```

This conversion is not performed when an array is used with `decltype` or as the operand of the address-of(`&`), `sizeof`, or `typeid` operators. The conversion is also omitted when we initialize a reference to an array. A similar pointer conversion happens when we use a function type in an expression.

A constant integral value of 0 and the literal `nullptr` can be converted to any pointer type; a pointer to any nonconst type can be converted to `void*`, and a pointer to any type can be converted to a const `void*`.

There is an automatic conversion from arithmetic or pointer types to `bool`. If the pointer or arithmetic value is zero, the conversion yields `false`; any other yields `true`:

```

1 char *cp = get_string();
2 if (cp) /* ... */ // true if the pointer cp is not zero
3 while (*cp) /* ... */ // true if *cp is not the null character

```

We can convert a pointer to a nonconst type to a pointer to the corresponding const type, and similarly for references. That is, if `T` is a type, we can convert a pointer or a reference to `T` into a pointer or a reference to const `T`:

```

1 int i;
2 const int &j = i; // convert a non const to a reference to const int
3 const int *p = &i; // convert address of a non const to the address of a const
4 int &r = j, *q = p; // error: conversion from const to nonconst not allowed

```

The reverse conversion - removing a low-level const - does not exist.

## 2.5 December 06, 2023

### 2.5.1 C++ Statements

An expression becomes an expression statement when it is followed by a semicolon. Expression statements cause the expression to be evaluated and its result discarded:

```

1 ival + 5; // rather useless expression statement
2 cout << ival; // useful expression statement

```

Null statement is a single semicolon, and is legal anywhere a statement is expected:

```

1 ; // null statement
2 ival = v1 + v2;; // ok: second semicolon is a superfluous null statement
3
4 // disaster: extra semicolon: loop body is this null statement
5 while (iter != svec.end()) ; // the while body is the empty statement
6 ++iter; // increment is not part of the loop

```

A compound statement, usually referred to as a block, is a sequence of statements and declarations surrounded by a pair of curly braces. Names introduced inside a block are accessible only in that block and in blocks nested inside that block.

## Conditional Statements

An if statement conditionally executes another statement based on whether a specified condition is true:

```
1 if (condition)
2     statement
3 else
4     statement2
```

We use a block to enclose multiple statements:

```
1 // if failing grade, no need to check for a plus or minus
2 if (grade < 60)
3     lettergrade = scores[0];
4 else {
5     lettergrade = scores[(grade - 50)/10]; // fetch the letter grade
6     if (grade != 100) // add plus or minus only if not already an A++
7         if (grade % 10 > 7) lettergrade += '+'; // grades ending in 8 or 9 get a +
8         else if (grade % 10 < 3)
9             lettergrade += '-'; // grades ending in 0, 1, or 2 get a -
10 }
```

It is a common mistake to forget the curly braces when multiple statements must be executed as a block.

Dangling else is resolved by specifying that each else matched with the closest preceding unmatched if:

```
1 // WRONG: execution does NOT match indentation; the else goes with the inner if
2 if (grade % 10 >= 3)
3     if (grade % 10 > 7)
4         lettergrade += '+'; // grades ending in 8 or 9 get a +
5 else
6     lettergrade += '-'; // grades ending in 3, 4, 5, 6, or 7 get a minus!
7
8 // add a plus for grades that end in 8 or 9 and a minus for those ending in 0, 1,
9 // or 2
10 if (grade % 10 >= 3) {
11     if (grade % 10 > 7)
12         lettergrade += '+'; // grades ending in 8 or 9 get a +
13 } else // curly braces force the else to go with the outer if
14     lettergrade += '-'; // grades ending in 0, 1, or 2 will get a minus
```

We can make the else part of the outer if by enclosing the inner if in a block.

## Iterative Statements

The syntactic form of the for statement is:

```
1 // for (initializer; condition; expression)
2 //     statement
```

```

3
4 // process characters in s until we run out of characters or we hit a whitespace
5 for (decltype(s.size()) index = 0;
6     index != s.size() && !isspace(s[index]); ++index)
7     s[index] = toupper(s[index]); // capitalize the current character

```

The order of evaluation of for loop:

1. *init-statement* is executed once at the start of the loop.
2. Next, *condition* is evaluated.
3. If the condition is true, the for body executes.
4. Finally, *expression* is evaluated.

*init-statement* can define several objects in a single declaration statement:

```

1 // remember the size of v and stop when we get to the original last element
2 for (decltype(v.size()) i = 0, sz = v.size(); i != sz; ++i)
3     v.push_back(v[i]);

```

A for header can omit any (or all) of *init-statement*, *condition*, or *expression*, by replacing them with null statements:

```

1 auto beg = v.begin();
2 for ( /* null */; beg != v.end() && *beg >= 0; ++beg)
3     ; // no work to do
4
5 // Omitting condition is equivalent to writing true as the condition
6 for (int i = 0; /* no condition */ ; ++i) {
7     // process i; code inside the loop must stop the iteration!
8 }
9
10 // If we omit expression for the for header, either the condition or the body must
    do something to advance the iteration
11 vector<int> v;
12 for (int i; cin >> i; /* no expression */)
13     v.push_back(i);

```

The syntactic form of range for statement to iterate through elements of a container or other sequence:

```

1 // for (declaration: expression)
2 //     statement

```

*expression* must represent a sequence such as braced initializer list, array or object of type such as vector or string that has *begin* and *end* members that return iterators. *declaration* defines a variable. It must be possible to convert each element of the sequence to the variable's type. The easiest way to make sure the types match is to use the *auto* type specifier.

## 2.6 December 10, 2023



### 2.6.1 Learn about SELinux

#### SELinux Architecture

SELinux consists of four main components: object managers (OM), access vector cache (AVC), security server, and security policy as show below:

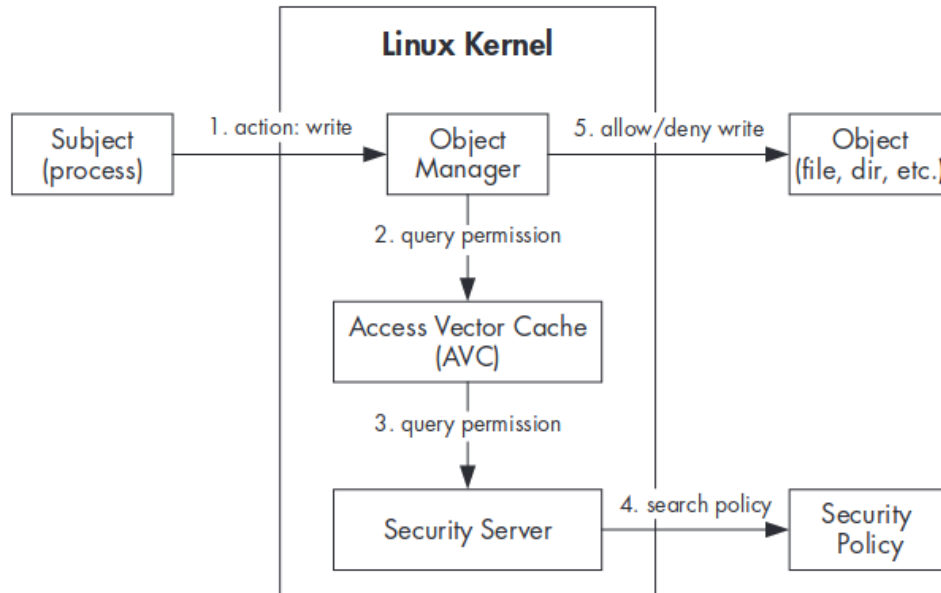


Figure 2.4: SELinux Components

When a subject asks to perform an action on an SELinux object, the associated object manager queries the AVC to see if the attempted action is allowed. If the AVC contains a cached security decision for the request, the AVC returns it to the OM which enforces the decision by allowing or denying the action. If the cache does not contain a matching security decision based on the currently loaded policy and returns it to the AVC, which caches it. The AVC in turn returns it to the OM which ultimately enforces the decision. The security server is part of the kernel, while the policy is loaded from userspace via a series of functions contained in the supporting userspace library.

#### SELinux Modes

SELinux has 3 modes:

- **Disabled.** No policy is loaded and only the default DAC security is enforced.
- **Permissive.** The policy is loaded and object access is checked, but access denial is only logged - not enforced.

- **Enforcing.** The security policy is both loaded and enforced, with violations logged.

SELinux mode can be checked and changed with the `getenforce` and `setenforce` commands:

```
1 # getenforce
2 Enforcing
3 # setenforce 0
4 # getenforce
5 Permissive
```

The mode set with `setenforce` is not persistent and will be reset to the default mode when the device reboots.

### Mandatory Access Control

- **Subjects** are usually running processes that perform actions on objects,
- **Objects** are OS-level resources managed by the kernel (processes can also be objects), and
- **Actions** are carried out only if the security policy allows it.

Both subjects and objects have a set of security attributes (collectively known as the security context) which the OS queries in order to decide whether the requested action should be allowed or not. When SELinux is enabled, subjects cannot bypass or influence policy rules; therefore, the policy is mandatory. The MAC policy is only consulted if the DAC allows access to the resource. If the DAC denies access, the denial is taken as the final security decision.

SELinux support two forms of MAC: *type enforcement (TE)* and *multi-level security (MLS)*. MLS is used to enforce different levels of access to restricted information and is not used in Android. TE implemented in SELinux requires that all subjects and objects have an associated type and SELinux uses this type to enforce the rules of its security policy. A *type* is simply a string that's defined in the policy and associated with objects or subjects. Subject types references processes or groups of processes and are also referred to as *domains*. Types referring to objects usually specify the role an object plays within a policy, such as system file, application data file, and so on. The type (or domain) is an integral part of the security context.

### Security Contexts

A *security context* (also referred to as a *security label*, or just *label*) is a string with four fields delimited with colons: username, role, type, and an optional MLS security range.

An SELinux username is typically associated with a group of class of users; for example `user_u` for unprivileged users and `admin_u` for administrators. Users can be associated with one or more domain type. The type is used to group processes in a domain or to specify an object logical type. In Android context, the user is fixed to `u`.

The security range (or level) is used to implement ML and specifies the security levels a subject is allowed to access. In Android context, the security range is fixed to `s0`.

By specifying the option `-Z`, we can see the security context of the processes running:

```
1 # ps -Z
2 u:r:su:s0    root      1847      1834      10842820    3384      sigsuspe+
3 u:r:su:s0    root      1878      1847      10800932    3572      0
```

Subjects inherit the security context of their parent process, or they can change their context via *domain transition* which can be made automatic. For example, all system daemons are started by the *init* process, which has `u:r:init:s0` security context, they would normally inherit this context, but Android's SELinux policy uses automatic domain transitions to set a dedicated domain to each daemon as need.

Similarly the context of files can be revealed using the `-Z` option:

```
1 # ls -Z
2 lrwxr-xr-x 1 root shell u:object_r:system_file:s0 6 2023-11-19 00:50 uuidgen ->
   toybox
3 -rwxr-xr-x 1 root shell u:object_r:vdc_exec:s0 101920 2023-11-19 00:50 vdc
4 -rwxr-xr-x 1 root shell u:object_r:viewcompiler_exec:s0 277472 2023-11-19 00:50
   viewcompiler
5 lrwxr-xr-x 1 root shell u:object_r:system_file:s0 6 2023-11-19 00:50 vmstat ->
   toybox
6 -rwxr-xr-x 1 root shell u:object_r:vold_exec:s0 994368 2023-11-19 00:50 vold
7 -rwxr-xr-x 1 root shell u:object_r:vold_prepare_subdirs_exec:s0 38576 2023-11-19
   00:50 vold_prepare_subdirs
8 -rwxr-xr-x 1 root shell u:object_r:system_file:s0 169 2023-11-19 01:02 vr
9 lrwxr-xr-x 1 root shell u:object_r:system_file:s0 6 2023-11-19 00:50 watch ->
   toybox
10 -rwxr-xr-x 1 root shell u:object_r:watchdogd_exec:s0 10760 2023-11-19 00:50
   watchdogd
11 lrwxr-xr-x 1 root shell u:object_r:system_file:s0 6 2023-11-19 00:50 wc -> toybox
12 lrwxr-xr-x 1 root shell u:object_r:system_file:s0 6 2023-11-19 00:50 which ->
   toybox
13 lrwxr-xr-x 1 root shell u:object_r:system_file:s0 6 2023-11-19 00:50 whoami ->
   toybox
14 -rwxr-xr-x 1 root shell u:object_r:wificond_exec:s0 393248 2023-11-19 00:50
   wificond
```

For objects, the security context is persistent and is usually stored as an extended attribute in the file's metadata. Objects typically inherit the type label of their parent (their directory), and can change to a different label via *type transition*.

## Security Policy

Security policies are used by the security server in the kernel to allow or disallow access to kernel objects at runtime. For performance reasons, the policy is typically in binary form generated by compiling a number of policy source files. *Statements* define policy entities such as types, users, and roles. *Rules* allow or deny access to objects (access vector rules); and designate how default users, roles, and types are assigned (default rules).<sup>5</sup>

The listing below declares `file_type` and domain attributes, declares `system_data_file` type and associates it with `file_type` and `data_file_type` attributes, declares `untrusted_app` type and associate it with domain attribute:

```
1 attribute file_type;
2 attribute domain;
3
4 type system_data_file, file_type, data_file_type;
5 type untrusted_app, domain;
```

`user` statement declares an SELinux user identifier, associates it with its role(s), and optionally specifies its default security level and the range of security levels that user can access:

```
1 user u roles { r } level s0 range s0 - mls_systemhigh;
```

The `u` user is associated with the `r` role (inside the braces), which in turn is declared using the role statement as show below:

```
1 role r;
2 role r types domain;
```

The second statement associates the `r` role with the domain attribute, which marks it as a role assigned to processes (domains).

`permissive` statement allows a named domain to run in permissive mode<sup>6</sup>:

```
1 type adbd, domain;
2 permissive adbd;
3 --snip--
```

The `class` statement defines an SELinux object class. Object classes and their associated permissions are determined by the respected object manager implementations in Linux kernel, and are static within a policy. Object classes are usually defined in the `security_classes` policy source file:

```
1 --snip-
2 # file-related classes
3 class filesystem
4 class file
5 class dir
6 class fd
7 class lnk_file
```

---

<sup>5</sup>Type, attribute and permission statements make up the bulk of a security policy.

<sup>6</sup>Most domains in Android's current base policy are permissive.

```

8 class chr_file
9 class blk_file
10 class sock_file
11 class fifo_file
12 --snip--

```

Access vectors are usually defined and associated with object classes in a policy source file called *access\_vectors*. Permissions can be either class-specific or inheritable by one or more object classes, in which case they're defined with the `common` keyword. Below is the definition of the set of permissions common to all file objects, and the association of the `dir` class (which represents directories), and a set of directory-specific permissions (*add\_name*, *remove\_name*, and so on):

```

1 --snip--
2 common file
3 {
4     ioctl
5     read
6     write
7     create
8     getattr
9     setattr
10    lock
11    --snip--
12 }
13 --snip--
14 class dir
15 inherits file
16 {
17     add_name
18     remove_name
19     reparent
20     search
21     rmdir
22     --snip--
23 }
24 --snip--

```

## Type Transition Rules

Type enforcement rules and access vector rules typically make the bulk of an SELinux policy. The most commonly used type of enforcement rule is the `type_transition` rule, which specifies when domain and type transitions are allowed:

```

1 # from wpa_suppl. te
2
3 # wpa - wpa supplicant or equivalent
4 type wpa, domain;
5 permissive wpa;
6 type wpa_exec, exec_type, file_type;
7
8 init_daemon_domain(wpa)
9 unconfined_domain(wpa)
10

```

```

11 # wpa_supplicant daemon uses the type transition rule to associate the control
    sockets it creates in /data/misc/wifi directory with wpa_socket type
12 type_transition wpa                                wifi_data_file: sock_file  wpa_socket;
13 #           source type      target type      class      type of object after
    the transition

```

## Domain Transition Rules

Most daemons are associated with a dedicated and use domain transitions to switch their domain when started. This is typically accomplished using the `init_daemon_domain()` macro, which under the hood is implemented using the `type_transition` keyword. The `init_daemon_domain()` macro takes one parameter and is defined in the `te_macros` file using two other macros: `domain_trans()` and `domain_auto_trans()` which are used to allow transition to a new domain and to execute the transition automatically, respectively:

```

1 # Domain transition macros definition int the te_macros file
2
3 # domain_trans(olddomain, type, newdomain)
4 define('domain_trans', '
5 allow $1 $2:file { getattr open read execute };
6 allow $1 $3:process transition;
7 allow $3 $2:file { entrypoint read execute };
8 allow $3 $1:process sigchld;
9 dontaudit $1 $3:process noatsecure;
10 allow $1 $3:process { siginh rlimitinh }; ')
11 # domain_auto_trans(olddomain, type, newdomain)
12 define('domain_auto_trans', '
13 domain_trans($1,$2,$3)
14 type_transition $1 $2:process $3; ')
15 # init_daemon_domain(domain)
16 define('init_daemon_domain', '
17 domain_auto_trans(init, $1_exec, $1)
18 tmpfs_domain($1) ')
19 --snip--

```

The lines beginning with the `allow` keyword are access vector (AV) rules.

## Access Vector Rules

AV rules define what privileges processes have at runtime by specifying the set of permissions they have over their target objects:

```

1 # Format of AV rules
2 rule_name source_type target_type : class perm_set;

```

The `rule_name` can be `allow`, `dontallow`, `auditallow`, `neverallow`. `allow` specifies the operations that a subject (process) of the specified source type is allowed to perform on an object of the target type and class specified in the rule. `auditallow` rule is used with `allow` to record audit events when an operation is allowed. `dontaudit` rule is used to suppress the auditing of denial messages when a specified event is known

to be safe. `neverallow` rule says that the declared operation should never be allowed even if an explicit `allow` rule that allows it exists.

To form a rule, `source_type` and `target_type` elements are replaced with one or more previously defined type or attribute identifiers, where `source_type` is the identifier of a subject (process), and `target_type` is the identifier of an object the process is trying to access. The `class` element is replaced with the object class of the target, and `perm_set` specifies the set of permissions that the source process has over the target object. You can specify multiple types, classes, and permissions by enclosing them in braces (`{}`). In addition, so rules support use of the wildcard (`*`) and complement (`~`) operators, which allow you to specify that all types should be included or that all types except those explicitly listed should be included, respectively:

```

1 type vold, domain;
2 type vold_exec, exec_type, file_type;
3 init_daemon_domain(vold)
4
5 # allows daemons running in vold domain to mount, unmount, and remount filesystems
  of sdcard_type
6 allow vold sdcard_type:filesystem { mount remount unmount };
7
8 # allows daemons running in vold domain to use the CAP_SYS_PTRACE and CAP_KILL
  Linux capabilities
9 # self means that target domain is same as source (vold in this case)
10 allow vold self:capability { sys_ptrace kill };
11
12 type installd, domain;
13
14 # no audit log will be created if the installd daemon is denied the CAP_SYS_ADMIN
  capability
15 dontaudit installd self:capability sys_admin;
16
17 # forbids all domains but the init domain to load the SELinux policy
18 neverallow { domain -init } kernel:security load_policy;
```

## 2.7 December 13, 2023

### 2.7.1 Android Testing

#### Meeting with Mariano and Biniam

Yan Nai is encountering exceptions in DeepGUI<sup>7</sup>'s final step: MonkeyTest.

DeepGUI has 3 steps:

1. Data Collection of interactions and events
2. Training
3. Monkey Test

---

<sup>7</sup><https://github.com/Feri73/deep-gui>

## 2.8 December 19, 2023

### 2.8.1 SELinux Syntax

SELinux Tutorials from <sup>8</sup>.

#### The security context of a process

The security context, together with the run-time user that the process is in, would define what the process is allowed to do.

```
1 ps -ef # -e gives all processes, -f gives more info, -Z gives security context
2
3 ls -ld dir # shows the permissions related to this directory
4 ls -ldZ dir # shows the permissions and also SELinux context
```

- **Domain.** The context of the process that is acting upon something.
- **Type.** The context of the resource on which the process is acting.
- **Class.** The object class of the resource (e.g. *file* or *socket*).
- **Permissions.** The permissions that are allowed given the *domain*, *type* and *class*.

SELinux rule syntax:

```
1 allow <domain> <type>:<class> { <permissions> };
```

#### Decoding Permission Denial Message

Message:

```
1 type=AVC msg=audit(1363289005.532:184): avc: denied { read } for pid=29199 comm
   ="Trace"
2 name="online" dev="sysfs" ino=30 scontext=staff_u:staff_r:googletalk_plugin_t
3 tcontext=system_u:object_r:sysfs_t tclass=file
```

Log part	Name	Description
type=AVC	Log type	Only in the audit.log file; it informs the user what kind of audit log type this is.
msg=audit(1363289005.532:184)	Timestamp	Timestamp in seconds since epoch, meaning the number of seconds since January 1st, 1970. You can convert this to a more human readable format using date -d @ followed by the number, like so: date -d @1363292159.532.

<sup>8</sup><https://wiki.gentoo.org/wiki/SELinux/Tutorials>



Log part	Name	Description
avc:	Log type (again)	
denied	State (if enforced)	What SELinux did, which can be either denied or granted. Note that, if SELinux is in permissive mode, then it will still log as denied even though it was enforced.
{ read }	Permission	The permission that was requested or executed. In this case, it is a read operation. Sometimes the permission contains a set like { read write } but in most cases, it is a single permission request.
for pid=29199	Process PID	The process identifier of the process that took the action.
comm="Trace"	Process CMD	The process command (without arguments, and limited to 15 characters), which helps users identify what the process was in case the process is already gone (a PID is only useful if the process is still running)
name="online"	Target name	The name of the target (in this case, file name). This field depends heavily on the target itself; it can also be path=, capability=, src= and more. But in those cases, its purposes should be clear from the rest of the log.
dev="sysfs"	Device	Device on which the target (in case of a file or file system). In this case, the device is sysfs so we have the hint immediately that this is for something inside /sys. Other valid examples are dev=md-0, dev=sda1, or dev=tmpfs.
ino=30	inode number	The inode number of the target file. In this case, since we know it is on the sysfs file system, we can look for this file using: <code>find /sys -xdev -inum 30</code>
scontent=staff_u:staff_r:googletalk_plugin_t	Source context	The security context of the process (the domain)
tcontext=system_u:object_r:sysfs_t	Target context	The security context of the target resource (in this case the file)
tclass=file	Target class	The class of the target.

Table 2.8: Permission Denied Syntax

## 2.9 December 21, 2023

### 2.9.1 SELinux Controlling Contexts

#### File

Objects are mapped to classes<sup>9</sup> (e.g., a file, a directory, a symbolic link, a socket) and the different kinds of access for each class are represented by permissions. While types and attributes are regularly updated as part of Android SELinux policy, permissions and classes are statically defined and rarely updated as part of a new Linux release.

---

<sup>9</sup>[https://android.googlesource.com/platform/system/sepolicy/+/refs/heads/main/private/security\\_classes](https://android.googlesource.com/platform/system/sepolicy/+/refs/heads/main/private/security_classes)