PhD Diary

Wei Minn
Singapore Management University

Version 1.0.5
Last compiled: December 11, 2023

Contents

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:

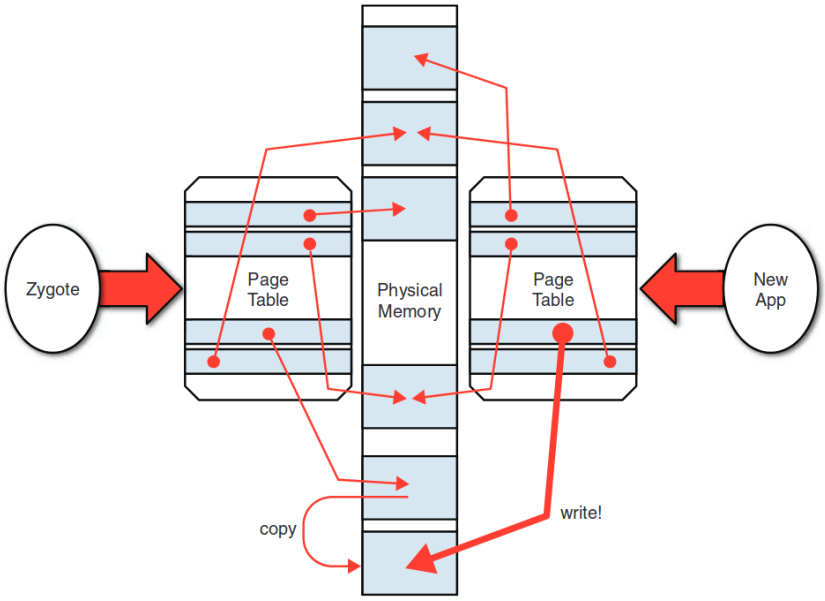

Figure 1.1: Zygote Copy-on Write

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/Andro`
`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
`bool`	boolean (`true` or `false`)	NA
`char`	character	8 bits
`w_char_t`	wide character	16 bits
`char16_t`	Unicode character	16 bits
`char32_t`	Unicode character	32 bits
`short`	short integer	16 bits
`int`	integer	16 bits
`long`	long integer	32 bits
`long long`	long integer	64 bits
`float`	single-precision floating-point	6 significant digits
`double`	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.

```
unsigned u = 10;
int i = -42;
std::cout << i + i << std::endl; // prints -84
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:

```
// WRONG: u can never be less than 0; the condition will always succeed
for (unsigned u = 10; u >= 0; --u)
    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:

```
std::cout << '\n'; // prints a newline
std::cout << "\tHi!\n"; // prints a tab followd 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:

```
int units_sold = 0;
int units_sold = {0}; // list initialization; does not work for built-in types if
    data loss is likely
int units_sold{0};
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.

```
extern int i;   // declares but not define j
int j;          // declares and defines j
int k = 12;     // declares, defines and initializes j
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 declara - 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.

```
#include <iostream>
int main() {
    int sum = 0;
    // sum values from 1 through 10 inclusive
    for (int val = 1; val <= 10; ++val)
        sum += val; // equivalent to sum=sum+val
    std::cout << "Sum␣of␣1␣to␣10␣inclusive␣is␣"
        << sum << std::endl;
    return 0;
}
```

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:

```
#include <iostream>
// Program for illustration purposes only: It is bad style for a function
// to use a global variable and also define a local variable with the same name
int reused = 42; // reused has global scope
int main() {
    int unique = 0; // unique has block scope

    // 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
```

Becuase 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; r2is an int
3  int i3 = 1024, &ri = i3; // i3 is an int; riis a reference bound to i3
4  int &r3 = i3, &r4 = i2; // both r3and r4are 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,whered is the name being defined. The * must be repeated for each pointer variable:

```
int *ip1, *ip2; // both ip1 and ip2 are pointers to int
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**):

```
int ival = 42;
int *p = &ival; // p holds the address of ival; p is a pointer to ival

double dval;
double *pd = &dval; // ok: initializer is the address of a double
double *pd2 = pd; // ok: initializer is a pointer to double
int *pi = pd; // error: types of pi and pd differ
pi = &dval; // error: assigning the address of a doubletoapointertoint
```

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

```
int ival = 42;
int *p = &ival; // p holds the address of ival; p is a pointer to ival
cout << *p; // * yields the object to which p points; prints 42

*p = 0; // * yields the object; we assign a new value to ival through p
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:

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

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

```
int* p1, p2; // p1 is a pointer to int; p2is an int
int *p1, *p2; // both p1and p2are 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:

```
int ival = 1024;
int *pi = &ival; // pi points to an int
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[1] 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_internal_os_nativeForkSystemServer`	Calls `zygote::ForkCommon` and `SpecializeCommon` which does the actual forking.	`AOSP/frameworks/base/core/jni/com_android_internal_os_Zygote.cpp`
`SpecializeCommon`	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`
`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`

[1] `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.postForkSystemServer and ZygoteHooks.postForkSystemServer	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_nativePostForkSystemServer	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
handleSystemServerProcess	The 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.forkSystemServer for the masses.	AOSP/framework/base/core/java/com/android/internal/os/ZygoteConnection.java
Zygote.forkAndSpecialize	A version of Zygote.forkSystemServer for the masses.	AOSP/frameworks/base/core/jni/com_android_internal_os_Zygote.cpp

Method	Description	Source
`com_android_internal_os_native_ForkAndSpecialize`	Calls `zygote::ForkCommon` and `SpecializeCommon` which does the actual forking, and returns to `ZygoteInit` and immediately enters `ZygoteServer`.	`AOSP/frameworks/base/core/jni/com_android_internal_os_Zygote.cpp`

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

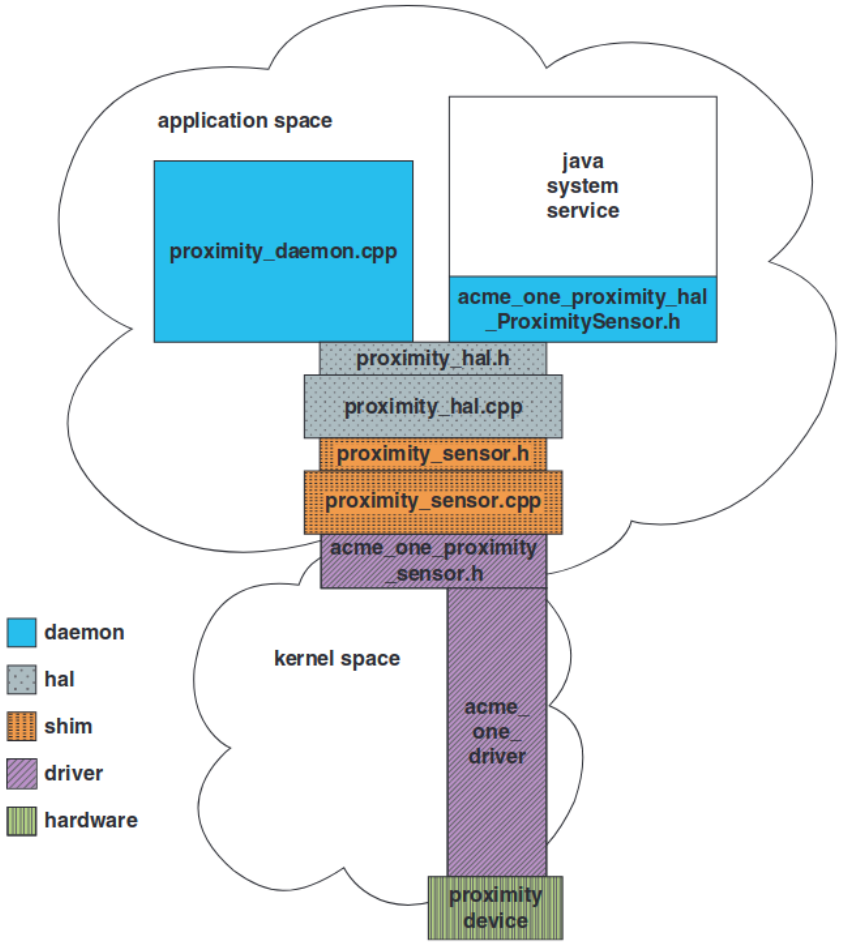

Figure 1.2: HAL Layer Structure

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`.[2]

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`[3] which is a list of git repositories. `repo` program parses `manifest.xml`, and thus `default.xml`, and clone each repository into a location specfied inside `default.xml`.

[2] To be a "real" HAL, the interface `proxmity/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.

[3] `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:

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

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

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

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

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

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

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

```
sudo apt update
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*[4] is

[4] `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.

```
#ifndef PROXIMITY_HAL_H
#define PROXIMITY_HAL_H

#include <hardware/hardware.h>

#define ACME_PROXIMITY_SENSOR_MODULE "libproximityhal"

typedef struct proximity_sensor_device proximity_sensor_device_t;

struct value_range {
    int min; int range;
};

typedef struct proximity_params {
    struct value_range precision;
    struct value_range proximity;
}

proximity_params_t;

struct proximity_sensor_device { hw_device_t common;
    int fd;
    proximity_params_t params;
    int (*poll_sensor)(proximity_sensor_device_t *dev, int precision);
}; #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`:

```
const int bufSize = 512; // input buffer size
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:

```
const int i = get_size(); // ok: initialized at run time
const int j = 42; // ok: initialized at compile time
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:

```
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: r1is a reference to const int &r2 =
4 ci; // error: nonconstreference to a constobject
```

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: r4is 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 by changed. We indicate that the pointer is const by putting the const by the const:

```
int errNumb = 0;
int *const curErr = &errNumb; // curErr will always point to errNumb
const double pi = 3.14159;
const double *const pip = &pi; // pip is a constpointer to a const object

*pip = 2.72; // error: pip is a pointer to const  and cannot be changed as the
    pointer is const
// if the object to which curErrpoints (i.e., errNumb) is nonzero
if (*curErr) {
    errorHandler();
    *curErr = 0; // ok: reset the value of the object to which curErr is bound
        because errNumb is not const
}
```

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:

```
const int max_files = 20; // max_files is a constant expression
const int limit = max_files + 1; // limit is a constant expression
int staff_size = 27; // staff_size is not a constant expression
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:

```
constexpr int mf = 20; // 20 is a constant expression
constexpr int limit = mf + 1; // mf+1 is a constant expression
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:

```
const int *p = nullptr; // pis a pointer to a constint
constexpr int *q = nullptr; // qis 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`

```
#include <unistd.h>
#include <stdio.h>
#include <android/log.h>

#define DELAY_SECS 2
#define ALOG(msg) __android_log_write(ANDROID_LOG_DEBUG, "WEIMINN␣PROJ", msg)

int main(int argc, char *argv[]) {
    ALOG("STARTING␣WEIMINN␣PROJECT");

    int n = 0;
    while (true) {
        sleep(DELAY_SECS);
        n++;

        ALOG("TESTING");
    }
}
```

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

```
cc_binary {
    name: "weiminn_daemon",
    relative_install_path: "hw",
    init_rc: ["init.weiminn.rc"],
    header_libs: [
        "liblog_headers",
    ],
    srcs: [
        "weiminn.cpp"
    ],
    shared_libs: [
        "liblog",
        "libcutils",
    ],
    static_libs: [
    ],
    vendor: true,
    proprietary: true,
}
```

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:

```
service wei_rudi_daemon /vendor/bin/hw/wei_rudi_daemon
    class main
    user system
    group system
    oneshot
```

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

```
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:

```
service weiminn_daemon /vendor/bin/hw/weiminn_daemon
    class main
    user system
    group system
    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:

```
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
11-21 09:45:47.733 0 0 I init : Service 'weiminn_daemon' (pid 1402) exited with
    status 127
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:

```
type weiminn, domain;
permissive weiminn;
type weiminn_exec, vendor_file_type, exec_type, file_type;

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:

```
11-21 10:19:43.865 0 0 I init : starting service 'weiminn_daemon'...
11-21 10:19:43.866 0 0 F init : cannot setexeccon('u:r:weiminn_exec:s0') for
    weiminn_daemon: Invalid argument
11-21 10:19:43.867 0 0 I init : Service 'weiminn_daemon' (pid 1827) exited with
    status 6
11-21 10:19:43.868 0 0 I init : Sending signal 9 to service 'weiminn_daemon' (pid
    1827) process group...
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`:

```
export ANDROID_SDK_ROOT=~/Android/Sdk
export ANDROID_HOME=~/Android/Sdk
export ANDROID_AVD_HOME=~/.android/avd

PATH=$PATH:$ANDROID_SDK_ROOT/emulator
```

Register the path:

```
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:

```
emulator -avd Pixel_7_Pro_API_33
```

Load APK into emulator:

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

Clone Fridump[5]:

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

Download and Run Fridump dependencies:

```
git clone https://github.com/Nightbringer21/fridump
code fridump
adb push frida-server /data/local/tmp
sudo sysctl kernel.yama.ptrace_scope=0 && frida-ps -D emulator-5554
frida-ps -D emulator-5554 | grep Birth # to get process ID of the Birthday app

# inside emulator shell
./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:

```
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[6] any requires the user to implement the operations themselves:

[5] `https://pentestcorner.com/introduction-to-fridump`

[6] Basically, class with only attributes, and no methods.

```
struct Sales_data {
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
```

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:

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

```
Sales_data data1, data2;
double price = 0; // price per book, used to calculate total revenue

// read the first transactions: ISBN, number of books sold, price per book
std::cin >> data1.bookNo >> data1.units_sold >> price;

// calculate total revenue from price and units_sold
data1.revenue = data1.units_sold * price;

// read the second transaction
std::cin >> data2.bookNo >> data2.units_sold >> price;
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`:

```
#ifndef SALES_DATA_H

#define SALES_DATA_H
#include <string>

struct Sales_data {
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};

#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:

```
typedef double wages; // wages is a synonym for double
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:

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

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

```
typedef char *pstring;
const pstring cstr = 0; // equivalent to char *const cstr = 0;
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:

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

The initializaer for all the variables muxt have types that are consistent with each other:

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

`auto` ignores top-level `const`s, and only low-level initializer are usually kept:

```
auto i = 0, *p = &i; // ok: i is int and p is a pointer to int
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

```
adb shell am dumpheap <PID> <HEAP-DUMP-FILE-PATH>
adb shell cat <HEAP-DUMP-FILE-PATH> > <LOCAL-FILE-PATH>
strings <LOCAL-FILE-PATH> <LOCAL-FILE-PATH-FOR-STRINGS>

pip install objection
frida-ps -Uai
# objection -g de.drmaxnix.birthdaycountdown explore
objection -g bloodpressure.bpdiary explore

android hooking list classes #List all loaded classes, As the target application gets usedmore, this command will return more classes.

android hooking search classes bloodpressure.bpdiary
android hooking search methods bloodpressure.bpdiary recordDbActivity

android hooking watch class bloodpressure.bpdiary.recordDbActivity --dump-args --dump-return

android hooking watch class_method bloodpressure.bpdiary.recordDbActivity.showWeight --dump-args --dump-backtrace --dump-return
```

1.11 November 27, 2023

1.11.1 Android Profile Guided Compilation

7 8

Baseline Profiles

```
adb shell cmd package dump-profiles bloodpressure.bpdiary
adb shell cat /data/misc/profman/bloodpressure.bpdiary-primary.prof.txt >
    bpdiary_profile.txt
```

Perfetto

Simpleperf

9

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

```
python3 app_profiler.py -p bloodpressure.bpdiary
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.bpdiary
```

Profcollect

10

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

Inside ADB shell:

```
device_config put profcollect_native_boot enabled true
setprop persist.device_config.profcollect_native_boot.collection_interval 60
setprop persist.device_config.profcollect_native_boot.sampling_period 1000
setprop persist.device_config.profcollect_native_boot.max_trace_limit 53687091200
setprop persist.device_config.profcollect_native_boot.enabled true
setprop ctl.stop profcollectd
setprop ctl.start profcollectd
ps -e | grep profcollectd
```

[7] https://developer.android.com/games/agde/pgo-overview

[8] https://newandroidbook.com/files/ArtOfDalvik.pdf

[9] https://android.googlesource.com/platform/system/extras/+/main/simpleperf/demo/README.md

[10] 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`[11] is the portion of an ojbect file, executable, or assembly language code that contains statically allocated variables that are declared but not have been assigned a value yet.

```
import argparse

# helper function
def findLineWith(arr, s):
    for i in range(len(arr)):
        if s in arr[i]:
            return i

def parse_method(method_arr):
    method_sig_raw = method_arr.pop(0)
    method_sig_raw_ = method_sig_raw[0: method_sig_raw.index(')')+1]
    method_sig_raw__ = method_sig_raw_.split(':')[1:]
    method_sig = method_sig_raw__[0].strip()

    method_arr.pop(0) # discard "DEX CODE:"
    endOfDex = findLineWith(method_arr, "OatMethodOffsets")
    dexCode = method_arr[0: endOfDex]

    startOfOat = findLineWith(method_arr, "CODE:")
    nativeCode = method_arr[startOfOat+1:] # Exclude "CODE: "

    parsed_method = {'method_sig': method_sig, 'dex': dexCode, 'native':
        nativeCode}

    # print("Parsed", method_sig)

    return parsed_method

def parse_type(type_arr):
    method_indexes = [i for i in range(len(type_arr)) if 'method_idx' in type_arr[
        i]]
    if len(method_indexes) > 0:
        method_indexes.append(len(type_arr))
    methods_raw = [type_arr[method_indexes[i]:method_indexes[i+1]] for i in range
        (0, len(method_indexes)-1)]

    methods = []
    for mraw in methods_raw:
        methods.append(parse_method(mraw))

    return {'type_name': type_arr[0].split()[1], 'methods': methods}

def load_oat_dump(p):
    f = open(p)
    lines = f.readlines()

```

[11] 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␣Debloation␣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.[1]:

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

Most common ways to initialize strings:

```
// direct initialization
string s1; // default initialization; s1 is the empty string
string s4(10, 'c'); // s4 is ccccccccc
string s6("hiya");

// copy initialization
string s2 = s1; // s2 is a copy of s1
string s3 = "hiya"; // equivalent to string s3("hiya");
string s8 = string(10, 'c') // copying initialization; s8 is ccccccccc
```

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

string Operations:

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

Table 2.1: string Operations

To check individual character:

Function	Description
isalnum(c)	true if c is a letter or a digit.
isalpha(c)	true if c is a letter.
iscntrl(c)	true if c is a control character.

[1] https://stackoverflow.com/a/73640984

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

Table 2.2: `cctype` Functions

Reading an Unknown Number of `string`s

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 `string`s and String Literals

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

```
string s4 = s1 + ", "; // ok: adding a string and a literal
string s5 = "hello" + ", "; // error: no string operand
string s6 = s1 + ", " + "world"; // ok: each + has a string operand
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:

```
for (declaration: expression)
    statement
```

A simple example:

```
string str("some string");
// print the characters in str one character to a line
for (auto c : str) // for every char in str
    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:

```
string str("some string");
// convert s to uppercase
for (auto &c : str) // for every char ref in str
    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:

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

To iterate using subscript:

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

`vector` Type

A `vector` is a collection[2] 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`:

```
#include <vector>
user 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:

```
vector<int> ivec; // ivec holds objects of type
int vector<Sales_item> Sales_vec; // holds Sales_items
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:

```
vector<string> svec; // default initialization; svechas no elements

// direct and copy initialization
vector<int> ivec2(ivec); // copy elements of ivec into ivec2
vector<int> ivec3 = ivec; // copy elements of ivec into ivec3
vector<string> svec(ivec2); // error: svec holds strings, not ints

// list initialization
vector<string> articles = {"a", "an", "the"};
vector<string> articles2{"a", "an", "the"};
vector<string> articles3("a", "an", "the"); // error

vector<int> ivec(10, -1); // ten int elements, each initialized to -1
vector<string> svec(10, "hi!"); // ten strings; each element is "hi!"

vector<int> ivec(10); // ten elements, each initialized to 0
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/

[2]Often referred to as container because it "contains" other objects.

`vector` Operations

Two `vectors` are equal if they have the same numnber 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
`v.empty()`	Returns `true` if `v` is empty; otherwise returns `false`.
`v.size()`	Returns the number of elements in `v`.
`v.push_back(t)`	Adds an element with value `t` to the end of `v`.
`v[n]`	Returns a reference to the element at position `n` in `v`.
`v1 = v2`	Replaces the elements in `v1` with a copy of the elements in `v2`.
`v1 = {a, b, c, ...}`	Replaces the elements in `v1` with a copy of the elements in the comma-separated list.
`v1 == v2` and `v1 != v2`	`v1` and `v2` are equal if they have the same number of elements and each element in `v1` is equal to corresponding element in `v2`.
`<, <=, >, >=`	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[3]. A good way to ensure that subscripts are in range is to avoid subscripting altogether by using a range `for` whenever possible.

[3] *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 acess to an object.

```
// the compiler determines the type of b and e;
// b denotes the first element and e denotes one past the last element in v
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 of 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 elemnt denoted by an iterator. Dereferencing an invalid iterator or an off-the-end iterator has undefined behavior:

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

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:

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

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:

```
vector<int> v;
auto b = v.cbegin(); // b has type vector<int>::const_iterator
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
`iter + n` and `iter - n`	Adding (subtracting) an integral value `n` 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.
`iter += n` and `iter -= n`	Compound-assignment for iterator addition and subtraction. Assigns to `iter1` the value of adding `n` to, or subtracting `n` from, `iter1`.
`iter1 - iter2`	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.
>,>=,<,<=	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:

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

```
auto mid = text.begin() + (end - beg)/2; // original midpoint

// while there are still elements to look at and we haven't yet found sought
while (mid != end && *mid != sought) {
    if (sought < *mid) // is the element we want in the first half?
        end = mid; // if so, adjust the range to ignore the second half
    else // the element we want is in the second half
        beg = mid + 1; // start looking with the element just after mid
    mid = beg + (end - beg)/2; // new midpoint
}
```

Arrays

Similar to library `vector` type, an `array` is a container of unnamed objectis 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 specialize 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`:

```
unsigned cnt = 42; // not a constant expression
constexpr unsigned sz = 42; // constant expression
int arr[10]; // array of ten ints
int *parr[sz]; // array of 42 pointers to int
string bad[cnt]; // error: cnt is not a constant expression
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 allow 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:

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

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

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

We cannot initialization an array as a copy of another array, nor is it legal to assign one array to another[4]:

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

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

```
// count the number of grades by clusters of ten: 0--9, 10--19, . . . 90--99, 100
unsigned scores[11] = {}; // 11 buckets, all value initialized to 0
unsigned grade;
while (cin >> grade) {
    if (grade <= 100)
        ++scores[grade/10]; // increment the counter for the current cluster
}

for (auto i : scores) // for each counter in scores
    cout << i << "␣"; // print the value of that counter
cout << endl;
```

We have to use a variable to have type `size_t` (defined in `cstddef` 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:

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

[4]Some compilers allow array assignment as a compiler extension.Itis usually a good idea to avoid using nonstandard features. Programs that use such features, will not work with a different compiler.

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

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

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

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

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

Pointers to array elements support the same operations as iterators on `vector`s or `string`s:

```
// ia3 is an array of ten ints
int arr[] = {0,1,2,3,4,5,6,7,8,9};
int *p = arr; // p points to the first element in arr
++p; // ppoints to arr[1]
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:

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

```
constexpr size_t sz = 5;
int arr[sz] = {1,2,3,4,5};
int *ip = arr; // equivalent to int*ip=&arr[0]
int *ip2 = ip + 4; // ip2points to arr[4], the last element in arr

// ok: arris converted to a pointer to its first element; ppoints one past the end
     of arr int *p = arr + sz; // use caution -- do not dereference! int *p2 = arr
     + 10; // error: arrhas only 5 elements; p2has 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:

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

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

```
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 `cstddef` header.

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

```
int i = ia[2];  // ia is converted to a pointer to the first element in ia
                // ia[2]fetches the element to which (ia+2)points
int *p = ia; // p points to the first element in ia
i=*(p + 2); // equivalent to i=ia[2]
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:

```
int ia[3][4]; // array of size 3; each element is an array of ints of size 4
// array of size 10; each element is a 20-element array whose elements are arrays
    of 30 ints
int arr[10][20][30] = {0}; // initialize all elements to 0

int ia[3][4] = { // three elements; each element is an array of size 4
    {0, 1, 2, 3}, // initializers for the row indexed by 0
    {4, 5, 6, 7}, // initializers for the row indexed by 1
    {8, 9, 10, 11} // initializers for the row indexed by 2
};

// equivalent initialization without the optional nested braces for each row
int ia[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};

// explicitly initialize only element 0 in each row
int ia[3][4] = {{ 0 }, { 4 }, { 8 }};

// explicitly initialize row 0; the remaining elements are value initialized
int ix[3][4] = {0, 3, 6, 9};

// assigns the first element of arr to the last element in the last row of ia
ia[2][3] = arr[0][0][0];

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:

- Assignement 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 buit-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:

```
int ival = 42;
double dval = 3.14;

ival % 12; // ok: result is 6
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 `bools`, the result of chaining these operators together is likely to be surprising:

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

The compiler converts `val` to `bool`:

```
if (val) { /* ... */} // true if val is any nonzero value
if (!val) { /* ... */}// true if val is zero
if (val == true) { /* ... */}// trueonly if valis equal to 1!
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:

```
int i = 0, j = 0, k = 0;     // initializations, not assignment
const int ci = i;            // initialization, not assignment
1024 = k;                    // error: literals are rvalues
i + j = k;                       // error: arithmetic expressions are rvalues
ci = k;                      // error: ci is a const(nonmodifiable) lvalue
k = 0;                       // result: type int, value 0
k = 3.14159;                 // result: type int, value 3
k = {3.14};                  // error: narrowing conversion
vector<int> vi;              // initially empty
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:

```
int ival, jval;
ival = jval = 0; // ok: each assigned 0
int ival, *pval; // ival is an int; pval is a pointer to int
ival = pval = 0; // error: cannot assign the value of a pointer to an int
string s1, s2;
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 occur in conditions. Because assignment has relatively low precedence, we usually must parenthesize the assignment for the condition to work properly:

```
// a verbose and therefore more error-prone way to write this loop
int i = get_value(); // get the first value
while (i != 42) {
    // do something . . .
    i = get_value();
    // get remaining values

}

int i;
// a better way to write our loop---what the condition does is now clearer
while ((i = get_value()) != 42) {
    // do something . . .
}
```

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)*:

```
string s1 = "a␣string", *p = &s1;
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	expr
«	left shift	expr1 « expr2
»	right shift	expr1 » expr2
&	bitwise AND	expr1 & expr2
^	bitwise XOR	expr1 êxpr2
|	bitwise OR	expr1 | expr2

Table 2.7: Bitwise Operators

The built-in menaing 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 («) or right (»). Bits that are shifted off the end are discarded:

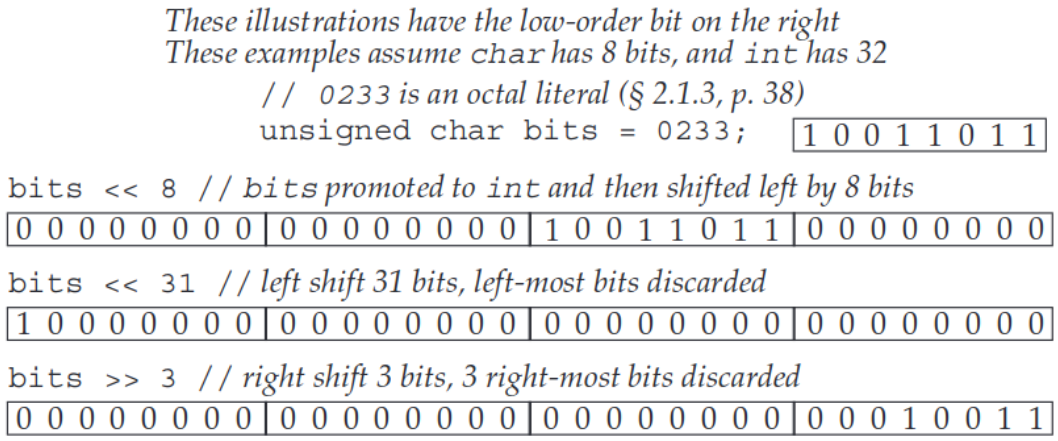

Figure 2.1: Bitwise Shift Operation

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:

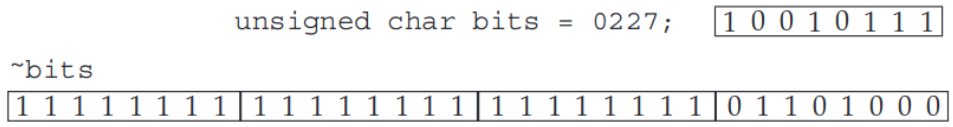

Figure 2.2: Bitwise NOT Operation

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

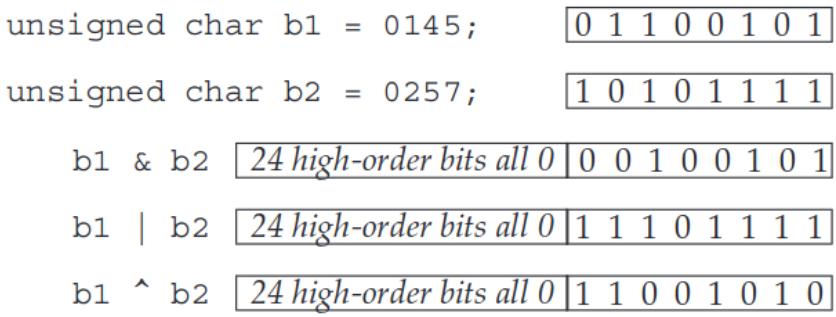

Figure 2.3: Bitwise AND, OR, and XOR Operation

`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 evaulated. Most common use for the comma operator is in a `for` loop:

```
vector<int>::size_type cnt = ivec.size();
// assign values from size...1 to the elements in ivec
for(vector<int>::size_type ix = 0;
        ix != ivec.size(); ++ix, --cnt)
    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, non`bool` 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:

```
bool flag;      char cval;
short sval;     unsigned short usval;
int ival;       unsigned int uival;
long lval;      unsigned long ulval;
float fval;     double dval;

3.14159L + 'a'; // 'a' promoted to int, then that intconverted to long double
dval + ival; // ival converted to double
dval + fval; // fval converted to double
ival = dval; // dval converted (by truncation) to int
flag = dval; // if dval is 0,then flag is false, otherwise true
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) /* ... */// trueif the pointer cp is not zero
3 while (*cp) /* ... */// trueif *cpis 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:

```
if (condition)
    statement
else
    statement2
```

We use a block to enclose multiple statements:

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

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

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

```
// WRONG: execution does NOT match indentation; the else goes with the inner if
if (grade % 10 >= 3)
    if (grade % 10 > 7)
        lettergrade += '+'; // grades ending in 8 or 9 get a +
else
    lettergrade += '-'; // grades ending in 3, 4, 5, 6, or 7 get a minus!

// add a plus for grades that end in 8 or 9 and a minus for those ending in 0, 1,
    or 2
if (grade % 10 >= 3) {
    if (grade % 10 > 7)
        lettergrade += '+'; // grades ending in 8 or 9 get a +
} else // curlies force the elseto go with the outer if
    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:

```
// for (initializer; condition; expression)
//     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:

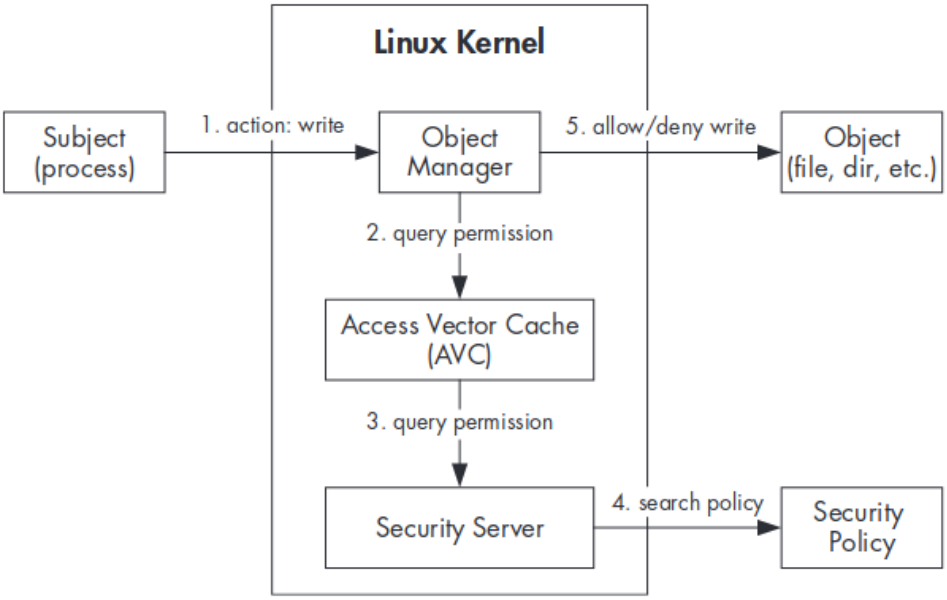

Figure 2.4: SELinux Components

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 assoicated 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 fileds delimited with colons: username, role, type, and an optional MLS security range.

An SELinux username is typically asosciated 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:

```
# ps -Z
u:r:su:s0   root    1847    1834    10842820    3384    sigsuspe+
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* secuirty context, they would normally inherit this context, but Android's SELinux poly 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:

```
# ls -Z
lrwxr-xr-x 1 root shell u:object_r:system_file:s0                        6
    2023-11-19 00:50 uuidgen -> toybox
-rwxr-xr-x 1 root shell u:object_r:vdc_exec:s0                      101920
    2023-11-19 00:50 vdc
-rwxr-xr-x 1 root shell u:object_r:viewcompiler_exec:s0             277472
    2023-11-19 00:50 viewcompiler
lrwxr-xr-x 1 root shell u:object_r:system_file:s0                        6
    2023-11-19 00:50 vmstat -> toybox
-rwxr-xr-x 1 root shell u:object_r:vold_exec:s0                     994368
    2023-11-19 00:50 vold
-rwxr-xr-x 1 root shell u:object_r:vold_prepare_subdirs_exec:s0      38576
    2023-11-19 00:50 vold_prepare_subdirs
-rwxr-xr-x 1 root shell u:object_r:system_file:s0                      169
    2023-11-19 01:02 vr
lrwxr-xr-x 1 root shell u:object_r:system_file:s0                        6
    2023-11-19 00:50 watch -> toybox
-rwxr-xr-x 1 root shell u:object_r:watchdogd_exec:s0                10760
    2023-11-19 00:50 watchdogd
lrwxr-xr-x 1 root shell u:object_r:system_file:s0                        6
    2023-11-19 00:50 wc -> toybox
lrwxr-xr-x 1 root shell u:object_r:system_file:s0                        6
    2023-11-19 00:50 which -> toybox
lrwxr-xr-x 1 root shell u:object_r:system_file:s0                        6
    2023-11-19 00:50 whoami -> toybox
-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). [5]

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[6]:

```
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
```

[5] Type, attribute and permission statements make up the bulk of a security policy.

[6] Most domains in Android's current base policy are permissive.

```
class chr_file
class blk_file
class sock_file
class fifo_file
--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):

```
--snip--
common file
{
    ioctl
    read
    write
    create
    getattr
    setattr
    lock
    --snip--
}
--snip--
class dir
inherits file
{
    add_name
    remove_name
    reparent
    search
    rmdir
    --snip--
}
--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:

```
# from wpa_supplicant.te

# wpa - wpa supplicant or equivalent
type wpa, domain;
permissive wpa;
type wpa_exec, exec_type, file_type;

init_daemon_domain(wpa)
unconfined_domain(wpa)

```

```
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 identifer 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 explicityly listed should be included, respectively:

```
type vold, domain;
type vold_exec, exec_type, file_type;
init_daemon_domain(vold)

# allows daemons running in vold domain to mount, unmount, and remount filesystems
     of sdcard_type
allow vold sdcard_type:filesystem { mount remount unmount };

# allows daemons running in vold domain to use the CAP_SYS_PTRACE and CAP_KILL
    Linux capabilities
# self means that target domain is same as source (vold in this case)
allow vold self:capability { sys_ptrace kill };

type installd, domain;

# no audit log will be created if the installd daemon is denied the CAP_SYS_ADMIN
    capability
dontaudit installd self:capability sys_admin;

# forbids all domains but the init domain to load the SELinux policy
neverallow { domain -init } kernel:security load_policy;
```