# Dev Cheat Sheet — iovec, writev/pwritev, off_t, size_t, uint64_t, and related C++ keywords

> A practical, zero-fluff guide so you know **what each thing is**, **when to use it**, **why**, and the **pros/cons**—with simple, practical examples.

---

## 1) POSIX scatter/gather I/O: `iovec`, `writev`, `pwritev`

### What is `iovec`?

- A tiny struct (`<sys/uio.h>`) that **points to a buffer** in memory: base pointer + length.
- You build an **array of** `iovec` values and pass them to `writev`/`pwritev` to write **multiple buffers in one syscall**.

```
struct iovec { void* iov_base; size_t iov_len; };
```

### `writev` vs `pwritev`

- `writev(fd, iov, iovcnt)` writes many buffers sequentially **at the current file offset**.
- `pwritev(fd, iov, iovcnt, offset)` writes many buffers **at an explicit file offset** (does not change the file pointer; thread-safe for shared fds).

#### When to use

- You have **many small pieces** (e.g., header + payload, or many entries) and want **one syscall** instead of many.
- In practice: batch multiple `{header, data}` pairs for ops/index flushes to minimize open/close/seek/write overhead.

#### Pros

- Fewer syscalls; better throughput and CPU efficiency.
- Kernel can optimize the combined write.

#### Cons / gotchas

- You must ensure each `iov_base` points to **valid, still-alive memory** until the call returns.
- Very large `iovcnt` may hit OS limits (e.g., `IOV_MAX`). Batch sensibly.

---

## 2) File positions & sizes: `off_t` vs `size_t`

### `off_t` (file offset type)

- Signed integer type used by POSIX to represent **byte offsets** in a file.
- Why signed? Historically ties to `lseek` semantics; modern use is just "position in the file".
- **Use `off_t` for: where to read/write** (e.g., `ops_file_position_`, `index_file_position_`).

**Pros:** Standard for file offsets; large-file ready on 64-bit builds. **Cons:** Signed; be careful with negative checks and conversions.

### `size_t` (object size/length type)

- Unsigned type returned by `sizeof` and container `.size()`. Represents **sizes and counts**.
- **Use `size_t` for: how many bytes** (e.g., `ops_file_reserved_size_`, buffer lengths, vector sizes).

**Pros:** Natural result of `sizeof` and STL size APIs; wide on 64-bit. **Cons:** Unsigned—be careful with underflow and signed/unsigned comparisons.

> **Rule of thumb:** - **Where?** Use `off_t` for positions in a file. - **How much?** Use `size_t` for lengths/sizes.

---

## 3) Big integer choices: `uint64_t`, `int64_t`, `uint32_t`, `int`

### `uint64_t`

- Unsigned 64-bit integer (0 … 2^64-1). Great for **IDs, counters, sizes** that should never be negative.
- In practice: `file_id_` is `uint64_t` so we can scale to **huge numbers of files/ops** without overflow.

### Peers & when to use them

- `int64_t` → need negatives and large range (e.g., deltas, signed math over large domain).
- `uint32_t`/`int32_t` → smaller ranges; good for compact structures or network formats.
- `int` → convenient but **width varies** by platform/ABI; avoid for on-disk/network formats.

**Pros of fixed-width types (`uint64_t`, etc.):** deterministic size across platforms. **Cons:** Slightly more verbose; may use more memory than necessary if over-provisioned.

---

## 4) C++ keywords you see in these structs

### `final`

- `struct X final { ... };` → **prevents inheritance** from `X`.
- Rationale: freeze the data model; avoid vtables/ABI surprises for simple POD-like structs.

### `static`

- Member belongs to the **type**, not each instance.

### `constexpr`

- Value is a **compile-time constant**. Enables optimizations and compile-time checks.

### `auto`

- **Type deduction** by the compiler. Here used with `constexpr` for a named flag.

### Example in your code

```
struct ops_index_flush_entry final {
  static constexpr auto Reflect = true; // compile-time flag for reflection
utilities
  ...
};
```

**Why this exists:** your logging/serialization helpers (e.g., `reflect::toString`) can gate behavior on `Reflect`.

---

## 5) Two example "flush entry" structs & their roles

### `OpsFlushEntry`

- **What:** one ops update: which file, which ops file, **where** to write it (`off_t`), **how big** (`size_t`), and the **data buffer**.
- **Used by:** a function that writes ops entries to disk.
- **Goal:** build all `{header, data}` iovecs **first**, then do **batched** `pwritev` per target file via your I/O layer.

### `IndexFlushEntry`

- **What:** the index-level record that points to the ops block (filename + offset + reserved size) for a `file_id`.
- **Used by:** a function that writes index entries to an index file.

- **Goal:** similarly batch writes to the index file.

---

## 6) How to choose the right type quickly (decision guide)

- **Need to mark a byte POSITION in a file?** → `off_t`.
- **Need a BYTES LENGTH or container size?** → `size_t`.
- **Need a huge non-negative ID/counter (won't be negative)?** → `uint64_t`.
- **Need large numbers that may be negative?** → `int64_t`.
- **Writing many buffers at once to a file?** → Build `std::vector<iovec>` and call `pwritev`.
- **Writing at a specific offset without moving the file pointer?** → `pwritev` (not `writev`).

---

## 7) Mini examples

### 7.1 Build iovecs and use `pwritev`

```
#include <sys/uio.h>
#include <unistd.h>

ssize_t write_header_and_data(int fd, off_t at, const std::string& hdr, const
std::string& data) {
  iovec vecs[2] = {
    { const_cast<char*>(hdr.data()), hdr.size() },
    { const_cast<char*>(data.data()), data.size() }
  };
  return ::pwritev(fd, vecs, 2, at);
}
```

### 7.2 `off_t` vs `size_t`

```
off_t position = 4096;    // start 4 KiB into the file
size_t length  = 1024;    // write 1 KiB
```

### 7.3 Big IDs

```
uint64_t file_id = 9876543210123456789ULL; // plenty of room for growth
```

---

## 8) Pros & cons summary table

| Concept | Use for | Pros | Cons |
|---|---|---|---|
| `iovec` + `pwritev` | One syscall for many buffers | Fewer syscalls, better throughput | Must manage lifetimes; `IOV_MAX` cap |
| `off_t` | File offsets | Large-file ready, standard | Signed; watch negative values |
| `size_t` | Sizes/lengths | Matches STL APIs, wide on 64-bit | Unsigned pitfalls in comparisons |
| `uint64_t` | IDs/counters (non-negative) | Huge range, portable width | More memory than smaller ints |
| `final` | Seal a type | Prevents unsafe subclassing | Less flexible for extension |
| `static constexpr` | Compile-time flags/ constants | Zero run-time cost | Requires compile-time known value |

---

## 9) Applying this to a typical flush pipeline

- **Today:** per entry → open, `pwritev(header,data)`, close.
- **Target:** group entries by target file, prebuild all `iovec`s, **one** `pwritev` **per file** through your I/O layer.
- **Benefits:** significant reduction in open/close/syscall churn; better sequential I/O; cleaner integration with your async I/O pipeline.

---

## 10) Quick checklist when implementing batching

- [ ] Group `ops_index_flush_entry` by `ops_file_name_`.
- [ ] For each group, sort by `ops_file_position_`.
- [ ] Build a single `std::vector<iovec>` as `{header1,data1, header2,data2, ...}`.
- [ ] Ensure buffers stay alive until `pwritev` completes.
- [ ] Respect `IOV_MAX` → split into multiple calls if needed.
- [ ] Use `pwritev(fd, iovecs.data(), iovecs.size(), start_offset_of_first)` **or** post via `drive_io` op.
- [ ] Handle partial writes / errors robustly; log and disable bucket on corruption.

---

---

# Diagrams

## A) Scatter/Gather I/O with `iovec`

```
User buffers in RAM

[ header1 ]  [ data1 ]   [ header2 ]  [ data2 ]   ...
     |            |            |            |
     v            v            v            v
  iovec[0]    iovec[1]     iovec[2]    iovec[3]   ...  --> single pwritev(fd,
iovec[], n, offset)
                                                         (kernel writes header1,
then data1, then header2, ...)
```

**Key idea:** many small pieces → one syscall.

---

## B) Offsets vs Sizes

```
File (bytes)
0        4096                 5120
|---------|====================|---->
^         ^
|         |
|         +-- size_t length (e.g., 1024 bytes)
+------------ off_t offset (e.g., 4096)
```

- `off_t` = **where** in the file.
- `size_t` = **how much** to read/write.

---

## C) Batching entries per file

```
Entries (unsorted):
  (file A, off 0,    H1,D1)
  (file A, off 4096, H2,D2)
  (file B, off 0,    H3,D3)

Group by file → sort by offset → build iovecs per file
```

```
File A iovecs: [H1][D1][H2][D2]  --> pwritev(fd_A, vecA, nA, 0)
File B iovecs: [H3][D3]          --> pwritev(fd_B, vecB, nB, 0)
```

Benefits: fewer opens/closes, fewer syscalls, better locality.

---

## D) Type picking quick guide

```
Need a file POSITION (byte offset)?
  └─> off_t

Need a BYTES LENGTH or container size?
  └─> size_t

Need a huge non-negative ID/counter?
  └─> uint64_t
       └─ If negatives possible → int64_t

Writing multiple buffers at once?
  └─> iovec[] + pwritev()
```

---

## E) Error handling flow (batched write)

```
Build iovecs → call pwritev → check return
                               ├─ -1 → log errno, retry/backoff or fail
                               └─ < sum(iov_len) → partial write: adjust iovecs, 
retry remaining
```