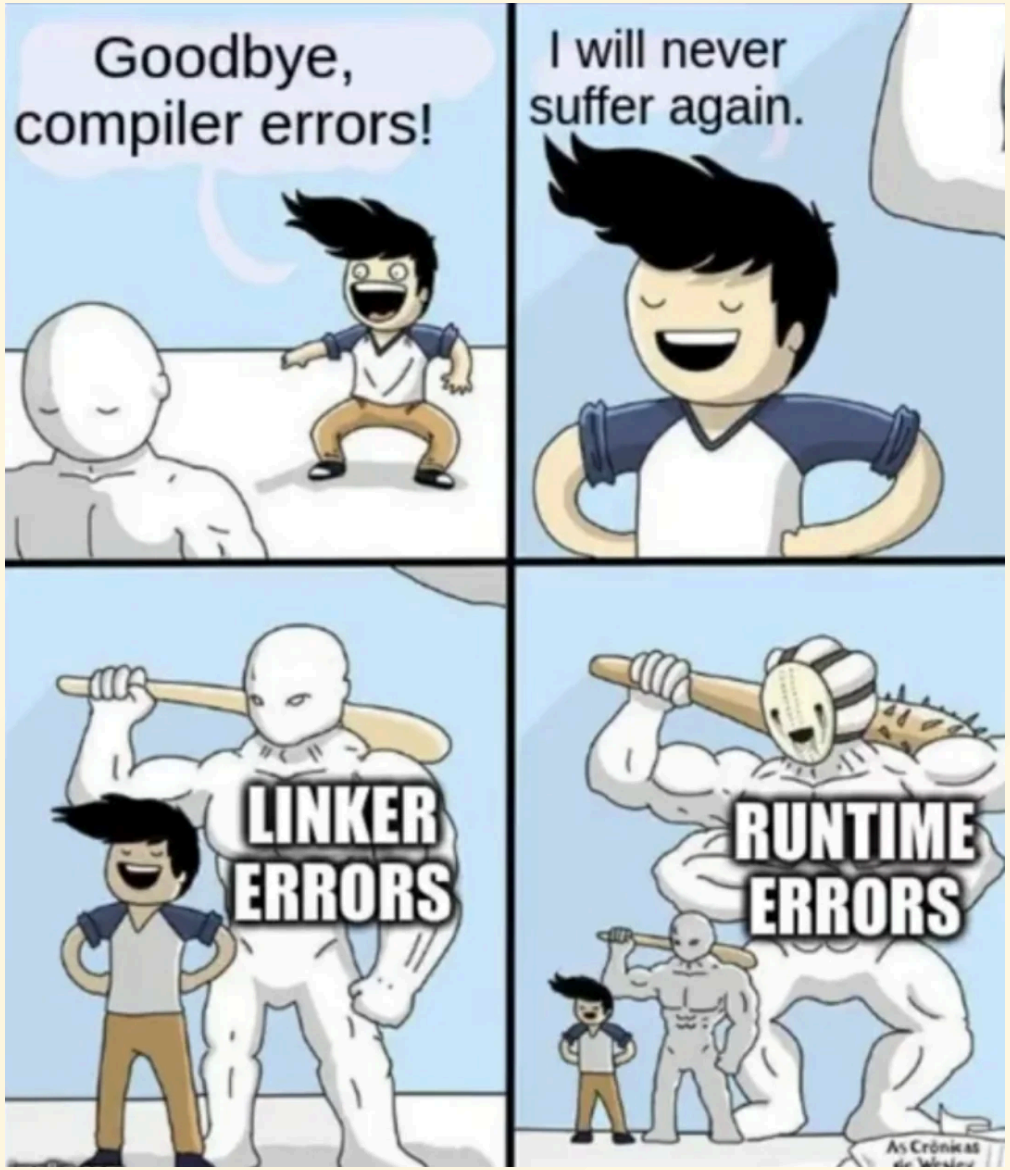
Goodbye, compiler errors!
I will never suffer again.
LINKER ERRORS
RUNTIME ERRORS

The Linker

The Linker

- **Links object files into an executable or library**
- Resolves **symbols** from different sources
- Fixes **addresses** for functions & variables
- Produces **executables, DLLs (Windows), SOs (Linux)**

Static Libraries

- +
- *
- ^

🔗 Full Source Compilation

🧮 `add → mul → pow → main`

```
g++ add.cpp mul.cpp pow.cpp main.cpp
```

- 🛠️ All `.cpp` files compiled and linked in one step
- ✅ No dependency order needed

🧱 Object File Linking

📦 Build objects first, then link

```
g++ add.cpp -c
...
g++ add.o mul.o pow.o main.cpp
```

- ⚙️ Compile each source separately
- ✅ Still no link order issues

📚 Static Library Linking

📁 Libraries need correct order!

```
g++ add.cpp -c
ar rcs libadd.a add.o
...

g++ main.cpp libpow.a libmul.a libadd.a
```

- ⚠️ Linker reads **left to right**
- ❗ Put libraries **after** the code that needs them

📚 Inspecting with `nm`

```
nm libmul.a -C
```

```
mul.o:
                 U _GLOBAL_OFFSET_TABLE_
                 U add(int, int)
0000000000000000 T mul(int, int)
```

Link Stages: Windows

1. Object files (`.obj`) compiled
2. Linker resolves **all symbols**
3. Statically links `.lib` files
4. Produces `.exe` or `.dll`

Link Stages: Linux

1. Object files (`.o`) compiled
2. Linker resolves symbols **in order**
3. Links `.a` and `.so` files
4. Produces `executable`

⚠️ **Order matters in Linux!**

```
g++ main.o -ladd -lmul -lpow    # ❌ Undefined reference
g++ main.o -lpow -lmul -ladd    # ✅ Correct order
```

⚠️ Undefined Reference

- 🔍 Use `nm`, `grep` to find where it's defined
- 🧱 Check link command (`link.txt` in CMake)
- ➕ Ensure dependency is **after** its user
- 🛠️ Fix link order or use `LINK_GROUP`

```
find . -type f -name '*.a' -exec nm -C {} + 2>/dev/null
```

🔍 Undefined Symbol	📚 Missing Library or Flag
`pthread_create`, `pthread_join`	`-lpthread`
`boost::filesystem::path`	`-lboost_filesystem`
`cv::Mat`	`-lopencv_core`
`dlopen`, `dlsym`, `dlclose`	`-ldl`
`std::thread`, `std::mutex`	`-pthread`
`zlibVersion`, `inflate`, `deflate`	`-lz`
`curl_easy_init`, `curl_easy_setopt`	`-lcurl`
`glBegin`, `glVertex3f`, `glEnd`	`-lGL` (OpenGL)
`glutInit`, `glutCreateWindow`	`-lglut`

🧱 Which libraries will be linked ?

🐧 Linux Flags

- `-L<dir>` — Add search path
- `-l<name>` — Link `lib<name>.so` / `.a`
- `-l:filename.a` — Exact archive

🪟 Windows (MSVC)

- Linker -> Input -> Additional Dependencies
- `#pragma comment(lib, "mylib.lib")` in code

📦 CMake: Adding Libraries

- `target_link_libraries()` connects targets

```
add_library(mylib STATIC file.cpp)
target_link_libraries(app PRIVATE mylib)
```

Dynamic Libraries

- +
- *
- ^

	🧱 **Static (`.a`, `.lib`)**	🔗 **Dynamic (`.so`, `.dll`)**
Linked	At **compile/link** time	At **runtime**
Included in	Final executable	External file
Size	Larger executable	Smaller executable
Flexibility	Less (update=rebuild)	More (swap `.dll`/`.so`)
Dependencies	None at runtime	`.so`/`.dll` must be present

💡 Use **static** if you can.

🪟 Create a DLL (Windows)

- 🛠️ Change `CMakeLists.txt`:

```
add_library(add SHARED add.cpp)
```

- 🧩 Mark functions for export:

```
__declspec(dllexport)
int add(int a, int b) {
  return a + b;
}
```

📦 Portable Export Macro

```
#ifdef MYLIB_EXPORTS
  #define MYLIB_API __declspec(dllexport)
#else
  #define MYLIB_API __declspec(dllimport)
#endif


MYLIB_API int add(int a, int b);
```

```
target_compile_definitions(mylib PRIVATE MYLIB_EXPORTS)
```

📚 Linking with a DLL (Windows)

- 🧱 Link against `.lib` (import library)
- 🔗 Loads `.dll` at runtime

🔍 DLL Search Order:

1. Executable directory
2. Current directory
3. System `PATH`

🐧 Create a `.so` (Linux)

- 🛠️ Change `CMakeLists.txt`:

```
add_library(add SHARED add.cpp)
```

- 🧩 No need for `__declspec(dllexport)`

```
int add(int a, int b) {
  return a + b;
}
```

📚 Linking with a `.so` (Linux)

- 🧱 Link with `-ladd`
- Provide `.so` at runtime

🔍 `.so` Search Order:

1. `DT_RPATH` (Deprecated)
2. `LD_LIBRARY_PATH`
3. `DT_RUNPATH`
4. `/etc/ld.so.cache`

Link Stages: Linux

✅ `.so` files can have **undefined symbols**
⚠️ They're checked **only when linking the final executable**

The `--no-undefined` flag changes that:

```
add_library(pow SHARED pow.cpp)

set_target_properties(pow PROPERTIES
  LINK_FLAGS "-Wl,--no-undefined"
)
target_link_libraries(pow PRIVATE mul add)
```

CMake

By default :

- CMake **adds** run path to the executable
- CMake **removes** the run path when installing

Recommended :

```
set(CMAKE_INSTALL_PREFIX ${CMAKE_BINARY_DIR}/install)

set_target_properties(main PROPERTIES
  INSTALL_RPATH "$ORIGIN"
  SKIP_BUILD_RPATH ON
)
```

🔧 CMake Global vs Project

- Global

```
set(CMAKE_SKIP_BUILD_RPATH TRUE)
```

- More fine-grained control

```
set_target_properties(main
  PROPERTIES
  SKIP_BUILD_RPATH TRUE
)
```

🪟🔄🐧 Shared Library Differences

Feature	Linux (`.so`)	Windows (`.dll`)
Link at build	❌ (lazy)	✅ (must resolve)
Search order	`LD_LIBRARY_PATH`, `rpath`, `ld.so.cache`	Current dir, PATH, system dirs
Default visibility	Public	Hidden (needs `__declspec(dllexport)`)
Extension	`.so`	`.dll` + `.lib` (import)

Tool

Dependency walker

- https://www.dependencywalker.com/
- https://github.com/lucasg/Dependencies

🔍 Runtime Linker Debugging Tools

✅ **Linux:** `LD_DEBUG`, `LD_PRELOAD`
✅ **Windows:** `procmon`

🛠️ LD_DEBUG

- Debug **dynamic linker activity** 🏗️
- Show **symbol resolution, library loading** 🕵️

```
LD_DEBUG=all ./my_program  # Show everything 👀
LD_DEBUG=libs ./my_program  # Library loading 🔍
LD_DEBUG=symbols ./my_program  # Symbol lookup 🔡
```

DON'T

⚠️ Exporting All Symbols

- By default, all symbols may be exported 📤
- **Problem:** Exposing unnecessary functions may cause **symbol conflicts** 🛑
- **Example:** Exporting 3rd-party libraries like **Boost, IPP, OpenCV**

🚨 What Can Go Wrong?

❌ **Symbol Conflicts** – Runtime crashes, wrong ABI ⚡
❌ **Linking Issues** – Different version might be used across libraries 📌
❌ **Unintended ABI Exposure** – Internal functions can accidentally be used 🛠️

🔍 How to Verify Symbol Exports

✅ Linux: Use `nm` or `objdump`

```
nm -D myLib.so  # Lists exported symbols
objdump -T myLib.so  # Shows dynamic symbols
```

- **Check exports before releasing shared libraries! 🚀**

Function Visibility

✅ **Windows (DLLs)** → Uses `__declspec(dllexport)` & `__declspec(dllimport)`.
✅ **Linux (SOs)** → Uses `__attribute__((visibility("default")))`.

Notes :

- `__declspec(dllimport)` is optional, for functions.
- Linux - default is visible, Windows - default is invisible.
- Which functions are exported: `nm -D` , `objdump` / `dependencies`

🔹 Function Visibility in Windows

```
#ifdef BUILD_DLL
#define API_EXPORT __declspec(dllexport)
#else
#define API_EXPORT __declspec(dllimport)
#endif

API_EXPORT void myFunction(); // Exported function
```

✅ **Importing (from EXE or another DLL)** → Use `__declspec(dllimport)`

📦 Function Visibility in Linux I

```
#define MYLIB_API __attribute__((visibility("default")))

MYLIB_API int add(int a, int b);
```

```
target_compile_options(mylib PRIVATE -fvisibility=hidden)
```

- ✅ Hides symbols by default
- ✅ Exports only tagged symbols (like `MYLIB_API`)

📦 Function Visibility in Linux II

- 🛠️ Create a version script (`exports.map`):

```
{
  global:
    add;
  local:
    *;
};
```

- 📦 Use it in CMake:

```
set_target_properties(add PROPERTIES
  LINK_FLAGS "-Wl,--version-script=${CMAKE_SOURCE_DIR}/exports.map"
)
```

Advanced

- Manual loading
- Name mangling
- Versioning (`so.1.2.3`)

🧩 Manually Loading DLLs

- ✅ Load at runtime (plugins, late binding)
- 🧠 Use `dlsym` / `GetProcAddress` / `boost::dll`
- 🔄 Flexible, optional dependencies
- ❗ Must know symbol names
- 🪧 `extern "C"` avoids name mangling
- 🕒 Manage global/static init

⚠️ Manual Loading: Pitfalls

- 🔍 Hard to debug missing or broken symbols
- 🚫 Symbol names must match exactly
- 🔗 No linker error → runtime crash
- 🧪 Test for load failure (`nullptr`, try/catch)

Boost::dll Example

```
// Load the shared library
#ifdef _WIN32
boost::dll::shared_library lib("mylib.dll");
#else
boost::dll::shared_library lib("libmylib.so");
#endif
// Import the function dynamically
auto hello_func = lib.get<void()>("hello");
// Call the function
hello_func();
```

Name Mangling

- C++ compilers modify function names
- Compiler specific
- C does NOT mangle names

```
void foo();    // Regular function
void foo(int);  // Overloaded function
namespace A {
    void foo(); // Namespaced function
}
```

Name Mangling in GCC/Clang (`nm`)

```
_Z3foov          # void foo()
_Z3fooRi         # void foo(int)
_N1A3fooEv       # void A::foo()
```

Name Mangling in MSVC (`dumpbin`)

```
?foo@@YAXXZ        # void foo()
?foo@@YAXH@Z       # void foo(int)
?foo@A@@YAXXZ      # void A::foo()
```

extern "C"

```
extern "C" void foo();  // No name mangling applied
extern "C" {
    void bar();
    int baz(int);
}
```

✅ Use `extern "C"` for

- C++ code that needs C-compatible APIs
- Dynamic libraries that must be callable from C
- Avoiding cross-compiler mangling issues

DLL Hell (Windows)

⚠️ Version conflicts

- Different apps require different versions of the same DLL
- **Example:** `gdal.dll`
- **Fix:**
  - Use local DLLs
  - Rename
  - [**SxS** (Side-by-Side assemblies)]

📌 SO Versioning in Linux

🔹 **Structure:**

```
libmylib.so → libmylib.so.1 → libmylib.so.1.2.3
```

- **1** = Major (Breaking changes 🚨)
- **2** = Minor (New features 🛠️)
- **3** = Patch (Bug fixes 🐞)

📌 SO Version in CMake

```
add_library(mylib SHARED mylib.cpp)

set_target_properties(mylib PROPERTIES
    VERSION 1.2.3       # Full version
    SOVERSION 1         # Major version
)
```

- **Creates:**

```
libmylib.so → libmylib.so.1 → libmylib.so.1.2.3
```

LD_PRELOAD

- Inject shared libraries 📌
- Override functions without rebuilding 🔄

```
LD_PRELOAD=/path/to/mylib.so ./my_program
```

Common linker problems, its cause, how to debug, and solutions.

❌ `-fPIC`

Problem

```
/usr/bin/ld: mylib.a(myfile.o): relocation R_X86_64_32S against `.text'
can not be used when making a shared object; recompile with -fPIC
```

Solution

✅ Compile with `-fPIC`

🛠️ MSVC `#pragma detect_mismatch`

Key	Used For	Example Value
`_MSC_VER`	Compiler version	`"1934"`
`RuntimeLibrary`	Static vs. dynamic CRT	`"MD_DynamicDebug"`
`_ITERATOR_DEBUG_LEVEL`	STL iterator checks level	`"0"` / `"2"`

⚠️ Mismatch = link error (LNK2038)

❌ `_ITERATOR_DEBUG_LEVEL` Mismatch (MSVC)

Problem

```
error LNK2038: mismatch detected for '_ITERATOR_DEBUG_LEVEL':
value '0' doesn't match value '2'
```

Solution

✅ Ensure **consistent build types**

✕ `MDd_DynamicDebug` Mismatch (MSVC)

Problem

```
error LNK2038: mismatch detected for 'RuntimeLibrary':
value 'MTd_StaticDebug' doesn't match value 'MDd_DynamicDebug' in program.obj
    C:\Data\ip_core.lib(ip_core.obj)
```

❌ Cannot find `mydll.lib` (MSVC)

Problem

```
LINK : fatal error LNK1104: cannot open file 'Debug\add.lib'
```

Solution

✅ In windows, a `.lib` file is created **Only if there are exported symbols**. Check `__declspec(dllexport)`

❌ Incorrect Shared Object (`.so`) Link Order

Problem

```
/usr/bin/ld: undefined reference to `myFunction`
```

Solution

✅ **Correct order**

✅ Use **grouping**:

```
g++ -Wl,--start-group -lfoo -lbar -Wl,--end-group -o myapp
```

❌ Missing `.so` File at Runtime

Problem

```
./myapp: error while loading shared libraries: libmylib.so: cannot open shared object file
```

Solution

- ✅ Check if the library is found
- ✅ Fix missing paths `LD_LIBRARY_PATH`

❌ Wrong shared object loaded at Runtime

Solution

✅ Check which DLL is loaded

- Windows : Debug -> Windows -> Modules
- Linux : gdb -> `info shared`


Quiz!

❓ Q1: What does the linker do?

1. Runs your code
2. Edits the source files
3. Resolves symbols and produces an executable
4. Downloads libraries

❓ Q2: What happens if you link a static library in the wrong order?

1. Nothing, order doesn’t matter
2. The build will be faster
3. You get undefined reference errors
4. All functions get linked anyway

❓ Q3: What does `-lfoo` link against?

1. foo.c
2. libfoo.so or libfoo.a
3. foo.cpp
4. foo.o

❓ Q4: What does `-L` do?

1. Links a library
2. Specifies a source directory
3. Adds a library search path
4. Loads a DLL

❓ Q5: Which tool shows exported symbols in a Linux `.so`?

1. nm -D
2. make
3. grep
4. g++

❓ Q6: What pragma is used in MSVC to catch mismatched settings?

1. `#pragma mismatch`
2. `#pragma error`
3. `#pragma detect_mismatch`
4. `#pragma validate`

❓ Q7: Which visibility is default on Windows for DLLs?

1. Visible
2. Hidden
3. Public
4. External

❓ Q8: What is the purpose of `__declspec(dllimport)`?

1. Export symbols
2. Load a DLL
3. Import symbols from DLL
4. Hide internal functions

❓ Q9: What happens when `LD_PRELOAD` is used?

1. Loads libraries after program starts
2. Overrides linked symbols
3. Speeds up linking
4. Changes runtime paths

❓ Q10: Which variable controls runtime search path for `.so`?

1. `PATH`
2. `LD_RUN_PATH`
3. `LD_LIBRARY_PATH`
4. `LD_BIN`

❓ Q11: What does `target_link_libraries()` do in CMake?

1. Adds include path
2. Creates shared library
3. Links targets to dependencies
4. Starts the build process

❓ Q12: What does `--no-undefined` flag do?

1. Skips symbols
2. Allows missing functions
3. Requires all symbols to be resolved
4. Only works in debug

❓ Q13: What does `add_library(foo SHARED foo.cpp)` do?

1. Creates a static library
2. Creates a header file
3. Creates a shared object (.so/.dll)
4. Runs `foo.cpp`

❓ Q14: How to hide symbols by default in Linux?

1. Use `-fPIC`
2. Use `-O3`
3. Use `-fvisibility=hidden`
4. Use `strip`

❓ Q15: Which tool can inspect `.lib` or `.obj` files on Windows?

1. ldd
2. dumpbin
3. objdump
4. strings