

Optimizer Developer Guide

Windows and Linux compilation, APIs and language integration

Patodia Infotech Private Limited

16 October 2026

Contents

1. Architecture, products and integration choices	3
2. Development directory and release checklist	5
3. Windows compilation: GCC/MinGW-w64	6
4. Linux compilation, ABI baseline and loaders	8
5. Reuse prebuilt HiGHS without compiling it	10
6. CMake builds and build integration	11
7. The legacy C ABI: arguments, ownership and status	12
8. C callers: linked DLL/SO and runtime loading	14
9. C++ callers and RAI	16
10. Python: standalone ctypes and typed Core	17
11. C# / .NET: UTF-8 P/Invoke on both systems	19
12. Java: JNA direct native calls	22
13. Julia: ccall, pointer owners and returned strings	24
14. PHP: FFI and subprocess workflows	26
15. Typed core: configuration, callbacks and mathematical solves	29
16. ABI layouts and type mapping	32
17. Direct resource placement arrays	34
18. Multiobjective, imports and high-precision result handling	36
19. Checkpoints, deadlines and saved progress	37
20. Hosting, concurrency and production lifecycle	39
21. Troubleshooting by failure layer	40
22. Tests, validation status and release practice	41
23. Public headers and export reference	43
24. Official interoperability references and maintenance notes	48

Edition: 16 October 2026

Patodia Infotech Private Limited

Windows DLL and Linux shared-library integration, compilation and deployment.

Copyright © 2026 Patodia Infotech Private Limited. All rights reserved.

This is the implementation and embedding companion to the Optimizer User Guide. It explains the current modular source, executable, C ABI, typed contexts and direct placement interface. It supplies complete calling examples for C, C++, C#, Python, Java, Julia and PHP, with Windows and Linux commands. The companion examples ZIP contains exactly the source listings used here, public-header snapshots, the matching Python core adapter, small models and validation records. It does not contain a solver binary, the full optimizer engine or third-party libraries. Use your matching optimizer source/binary release to build or run it.

The available Linux GCC/Python paths were executed; Windows, CMake, C#/.NET, Java/JNA, Julia and PHP examples are supplied for local validation and are not claimed as executed here. A Java runtime alone is not a JDK/compiler. Full details and captured outputs appear in the validation chapter and companion ZIP.

The guide's compile commands build our optimizer or its callers. **HiGHS is optional and reused as an existing prebuilt dependency. No command here builds HiGHS.** You do not rebuild the optimizer for every optimization request or every language call.

Long code and shell commands wrap in the PDF. Enter each shell command as one logical line; copy from Markdown or the companion source files where possible. The PDF has a linked contents section and bookmarks. The Markdown preserves complete source listings.

1. Architecture, products and integration choices

What you are embedding

Optimizer is a C++17 numerical engine with several dispatch paths. Mathematical models use native equations/declarations/objectives or supported LP/MPS/NL imports. Dedicated routing, packing, scheduling, network and placement services use their own formats. A single binary does not imply one common grammar or that every option applies everywhere.

The general equation/root search and local nonlinear searches are heuristic. Eligible affine/exact-small/native-bounding paths can make their separately qualified numerical conclusions. A verified feasible candidate, local stationarity, a valid bound and completed global/hierarchy/Pareto work are distinct facts. Embedding software must not translate every successful call into the label optimal.

The interfaces are:

Interface	Public entry	Best use	Concurrency / precision
Executable	optimizer.exe / optimizer	Language-neutral subprocesses; strongest separation from the host	Separate processes; all supported CLI precision/modes
Legacy C ABI	optimizer_run in optimizer_dll.h	CLI-compatible arguments, extended printed output and specialized services	General calls serialize in this implementation; captured output
Typed core ABI v1	optimizer_core.h	Mathematical LP/MILP/NLP/MINLP, objective controllers, JSON and typed values	Independent contexts may solve concurrently; doubles
Direct placement	optimizer_place	Borrowed input arrays and typed assignment output	Reentrant independent calls; double capacities/costs

A DLL does not require a separate engine rewrite for each language. Each runtime can call the same C functions. The host language owns its buffers and result handling; the optimizer owns its result/context allocations.

Source layout

File / directory	Responsibility
optimizer.cpp	Thin executable entry point
optimizer_dll.cpp	Legacy C exports, captured-output results and placement bridge
optimizer_dll.h	Legacy and placement declarations, plus a compatibility subset of core declarations
optimizer_core.h	Full current typed-core declarations and versioned structures
optimizer_core.py	ctypes adapter for typed contexts and additive accessors
src/core_api.cpp	Typed-context implementation
src/nonlinear.* / work_search.tpp	Nonlinear search/model and bounded work/checkpoint scheduler
src/linear_milp.*	Affine engine, presolve, branching, hierarchy and incumbent work
src/native_nonlinear.* / global.*	Qualified bounded nonlinear searches
src/multiobjective.*	Blending, epsilon sampling, archive and polishing
src/equations.* / common.hpp	Expressions, root search, arithmetic and shared control
src/import*	LP/MPS/NL parsing and normalization
src/command.cpp / dispatch.cpp / seeds.cpp	CLI dispatch, reporting and multi-seed selection
src/tsp, cvrp, services, placement, network*	Dedicated services

File / directory	Responsibility
build.py / build_windows*.cmd / CMakeLists.txt	Modular compiler/linker orchestration
tests/	Production CLI/DLL and isolated white-box checks

The .hpp declarations and heavy .tpp implementations are separated. Owning .cpp files instantiate double and supported binary128 templates. Shared PIC core objects are reused by executable and library. Compiling optimizer.cpp alone does not build the current engine. Do not include optimizer.cpp into an application as a substitute for the public API.

White-box tests have a separate testing umbrella and may alter internal layouts. Never mix their test-only object files with production objects. There is no requirement to invent an extra runtime_state.cpp; seed/control ownership is already defined in the matching source.

2. Development directory and release checklist

Keep engine source, native outputs and caller examples distinct. Suggested layout:

C:/Optimizer/	or /opt/patodia/optimizer/
source/	matching full optimizer source release
native/	deployed optimizer.exe + optimizer.dll
	or optimizer + liboptimizer.so
examples/	extracted developer companion
include/	
c/ cpp/ csharp/ java/ julia/ php/ python/	
models/ cmake/ validation/	

The companion's example commands assume your working directory is its extracted examples root. Engine build commands instead run from the full engine source directory. Relative library/model paths are resolved against the application's working directory, not automatically against the guide location. Prefer absolute paths in services and batch jobs.

Before integration, record platform, CPU architecture, compiler/version, precision configuration, native binary hashes, enabled external dependencies and public-header hashes. Check exports. A filename such as optimizer.dll does not identify its release; the ABI version remains 1 across additive exports. Deploy headers, adapters, executable/DLL and runtime dependencies from one compatible build.

Use a private working directory for each job. Avoid sharing checkpoint filenames and output paths across independent users or processes. Preserve original models, name sidecars, arguments, stderr and stdout. A mathematical model's variable names may be mapped during import; read the returned names rather than assuming row or variable order.

3. Windows compilation: GCC/MinGW-w64

Supported build route

Use a 64-bit MinGW-w64 GCC toolchain with C++17 support, Python 3.9+ and matching quadmath when binary128 is enabled. Put its g++ and runtime DLL directory on PATH. These instructions target GCC/MinGW; the current engine is not represented as a tested MSVC-native build. MSVC applications can still consume the plain C ABI, as shown later.

From Command Prompt inside the optimizer source directory:

```
where g++
g++ --version
python --version
python build.py --jobs 4 --opt-level 1
```

Outputs are optimizer.exe, optimizer.dll and its MinGW import library, normally liboptimizer.dll.a. The tested source candidate used -O1 on Linux; Windows execution is not claimed. The script's normal default optimization is -O3. Choose the level deliberately when reproducing numerical runs.

A native wrapper is supplied:

```
build_windows_native.cmd --opt-level 1
```

The general wrapper can automatically use an existing dependency configuration:

```
build_windows.cmd --opt-level 1
```

Both wrappers call build.py. They do not execute all tests described in old historical release reports. Run the relevant tests explicitly. Do not pass historical wrapper options such as --independent unless the current script actually accepts them; the current build.py does not.

Build.py reference

Option / environment	Contract
--jobs N	Positive parallel compile jobs; default 2
--cxx PATH	C++ compiler; default CXX environment value or g++
--opt-level 0 1 2 3	Optimization level; default 3
--debug	Uses -O0; this flag alone does not add debug symbols or a sanitizer
--clean	Force rebuilding objects and outputs
--double-only	Defines SOLVER_NO_QUAD and omits quadmath; no --extend support
--with-highs	Link an existing prebuilt HiGHS configuration
--highs-root PATH	Existing SDK/build root; can also be OPTIMIZER_HIGHS_ROOT
--highs-include PATH	Existing header include directory; repeatable
--highs-library FILE	Existing library archive/shared import file; repeatable, in link order
OPTIMIZER_PYTHON	Python executable used by Windows wrappers
OPTIMIZER_BUILD_JOBS	Windows-wrapper compile job count; default 2

Examples:

```
python build.py --jobs 4 --opt-level 1 --double-only
python build.py --jobs 4 --opt-level 1 --clean
python build.py --cxx "C:\msys64\mingw64\bin\g++.exe" --jobs 4 --opt-level 1
```

The builder tracks compiler/flag fingerprints and compiler-generated dependency files. Compatible no-change builds can do nothing; implementation changes rebuild affected owning objects and relink. Do not copy cached objects between unrelated compilers, source revisions or

precision configurations. Rebuilding can invalidate work-mode checkpoints; keep the original executable for a checkpoint's lifetime.

Runtime DLL deployment

Inspect the actual binary rather than guessing which runtimes it needs:

```
objdump -p optimizer.dll  
objdump -p optimizer.exe
```

Depending on toolchain/link flags, dependencies can include libstdc++-6.dll, libgcc_s_seh-1.dll, libwinpthread-1.dll, libquadmath-0.dll and optional external-solver DLLs. Names vary. Use matching x64 files from the same compatible toolchain. Put dependencies in a controlled application-native directory and configure the loader. Do not mix an x86 host with x64 optimizer or use arbitrary runtime DLLs with the same basename.

Windows system error 126 often means a dependency was not found, not that optimizer.dll itself is absent. Error 193 commonly indicates architecture/format mismatch. A file with the .dll extension can still be a Linux symlink or an incompatible artifact; inspect its format before deployment.

4. Linux compilation, ABI baseline and loaders

Native build

Use a Linux GCC/g++ C++17 toolchain, pthread support and Python 3.9+. Quad-enabled builds require libquadmath development/link support and runtime availability. Install prerequisites through your supported operating-system process; exact distribution package names are not assumed here.

From the full optimizer source directory:

```
g++ --version
python3 --version
python3 build.py --jobs 4 --opt-level 1
./optimizer --help
```

The outputs are optimizer and liboptimizer.so. The script compiles PIC core objects and links the shared library. A Linux compatibility symlink named optimizer.dll may also be present in a development tree; it is not a Windows binary. The source builder uses .o and dependency caches under build/objects.

Double-only alternative:

```
python3 build.py --jobs 4 --opt-level 1 --double-only
```

You do not compile a .so each time Python, PHP or Julia makes a request. Load and reuse a compatible deployed library. Rebuild when source/toolchain/options change, not when the model data changes.

Distribution and CentOS 8 compatibility

A binary built on a modern Linux distribution may require a newer glibc or libstdc++ than the target server. An older-compatible release needs to be built and tested against that intended baseline. Simply naming a library CentOS-compatible or copying it to CentOS does not change its symbol requirements. This guide's live examples used the available Linux GCC 13.3 environment; they do not newly certify a CentOS 8 binary.

Inspect files and dynamic requirements:

```
file optimizer liboptimizer.so
ldd ./liboptimizer.so
readelf -d liboptimizer.so
readelf --version-info liboptimizer.so
nm -D --defined-only liboptimizer.so
```

Use ldd only on trusted binaries. Inspect both the optimizer and enabled external-library requirements. A prebuilt external dependency must also fit the target baseline. Test loading and real solves on the destination system.

Library search paths

For a development shell:

```
export LD_LIBRARY_PATH="/opt/patodia/optimizer/native${LD_LIBRARY_PATH:+:$LD_LIBRARY_PATH}"
python3 python/legacy.py /opt/patodia/optimizer/native/liboptimizer.so
```

An absolute path identifies the optimizer library, but its dependencies still need resolution.

Configure `LD_LIBRARY_PATH` before starting the application when it is needed; modifying it after the process loader has started is not a reliable dependency-discovery strategy. For deployments, prefer a controlled application-relative `RUNPATH` or administrator-managed loader configuration instead of globally replacing unrelated libraries.

For a relocatable C/C++ caller placed next to its library:

```
gcc -std=c11 -Iinclude c/legacy.c -L/opt/patodia/optimizer/native -loptimizer -Wl,-rpath,'$ORIGIN'  
-o app
```

The single quotes preserve literal `$ORIGIN` for the ELF loader. This `RUNPATH` points at the app's directory, not the build directory. If the shared library lives in a sibling `lib` directory, use `'$ORIGIN/../lib'`. A main application's `RUNPATH` does not automatically solve every transitive dependency's own search path; inspect and test the dependency chain.

5. Reuse prebuilt HiGHS without compiling it

HiGHS is optional. Native examples in this guide explicitly select the native affine engine where applicable, so they do not require a HiGHS build. To use an already installed compatible SDK:

```
set "OPTIMIZER_HIGHS_ROOT=C:\Dependencies\highs-install"
python build.py --jobs 4 --opt-level 1 --with-highs
```

Linux:

```
python3 build.py --jobs 4 --opt-level 1 --with-highs --highs-root /opt/highs-install
```

For explicit files:

```
python3 build.py --jobs 4 --opt-level 1 --with-highs --highs-include /opt/highs/include/highs
--highs-library /opt/highs/lib/libhighs.so
```

Supply additional existing include paths/archives when your SDK needs them. Static archives must be compatible with the target architecture and shared-library PIC requirements; a static library that can link an executable may still fail when linked into a .so. Link transitive dependencies in the required order. Runtime shared libraries must be deployed/discoverable if dynamically linked. This guide does not fetch, configure or build the external solver.

At solve time:

```
optimizer.exe --file production.mps --lp-engine native --seconds 20
optimizer.exe --file production.mps --lp-engine highs --seconds 20
optimizer.exe --file production.mps --lp-engine auto --seconds 20
```

OPTIMIZER_LP_ENGINE can override auto. Inspect the backend actually used. An explicit unavailable highs request fails. Native-specific cuts/branching/cache controls do not become identical external-engine settings just because arguments parse.

Affine --extend in an enabled build can use double discovery followed by native binary128 verification/refinement of the original model. A fixed-integer refined MILP candidate is not a global mixed-integer certificate. Typed-core v1 remains double-based and does not become a binary128 interface through this option. Use the legacy argument path for supported extended output. See the User Guide for detailed numerical qualification.

6. CMake builds and build integration

The Python builder needs no CMake or Make. CMake is an optional alternative for IDE/build-system integration. Use a recent CMake (3.20+ recommended for the supplied dependency-discovery commands), a generator and the GCC toolchain you intend to deploy.

From the optimizer source directory, Linux or an appropriately configured MinGW shell:

```
cmake -S . -B build-cmake -G Ninja -DCMAKE_C_COMPILER=gcc -DCMAKE_CXX_COMPILER=g++
-DCMAKE_BUILD_TYPE=Release -DOPTIMIZER_BUILD_TESTS=OFF
cmake --build build-cmake --target optimizer optimizer_dll --parallel 4
```

On Windows, make sure the generator/compiler selection does not silently choose MSVC. If Ninja is unavailable but MinGW Makefiles and mingw32-make are installed, select that generator instead. Do not mix objects in one CMake build directory after changing compiler families; use a fresh build directory.

CMake setting	Meaning
OPTIMIZER_DOUBLE_ONLY=ON	Disable quadmath/binary128
OPTIMIZER_BUILD_TESTS=OFF	Build production targets without heavy white-box test targets
OPTIMIZER_WITH_HIGHS=ON	Link existing dependency files only
OPTIMIZER_HIGHS_ROOT	Existing SDK root
OPTIMIZER_HIGHS_INCLUDE_DIRS	Existing include paths; CMake list
OPTIMIZER_HIGHS_LIBRARIES	Existing library files in link order; CMake list

For tests:

```
cmake -S . -B build-cmake-tests -G Ninja -DCMAKE_C_COMPILER=gcc -DCMAKE_CXX_COMPILER=g++
-DCMAKE_BUILD_TYPE=Release -DOPTIMIZER_BUILD_TESTS=ON
cmake --build build-cmake-tests --parallel 4
ctest --test-dir build-cmake-tests --output-on-failure
```

These CMake instructions were checked against the supplied configuration but were not executed in the available environment. The configuration has no general install/package target promised by this guide; stage the required output files explicitly. The production object target is `optimizer_core` and the DLL/shared-library target is `optimizer_dll`; its output name is `optimizer`, producing `optimizer.dll` or `liboptimizer.so` as appropriate.

The companion also contains `cmake/CMakeLists.txt` for building only the C/C++ CALLERS against an existing library. That project never builds the engine or external solver:

```
cmake -S cmake -B caller-build -G Ninja -DOPTIMIZER_LIBRARY=/absolute/path/liboptimizer.so
cmake --build caller-build
```

Windows additionally needs `OPTIMIZER_IMPORT_LIBRARY` pointing to the matching caller-toolchain import library. That is separate from the runtime DLL file.

7. The legacy C ABI: arguments, ownership and status

Exact call model

The core legacy signature is:

```
int optimizer_run(int argc, const char *const *argv, OptimizerResult **result);
```

Arguments EXCLUDE the executable name. A quoted shell expression becomes one argv entry, but there is no shell when calling the DLL directly. Pass '--eq' and the complete model as separate entries. Do not insert extra quote characters around the model/path inside those entries. Native strings must remain valid and NUL-terminated until the synchronous call returns. Reject embedded NUL bytes. UTF-8 is the example encoding; target-platform non-ASCII filesystem path behavior still needs validation.

argc=0 with a valid result pointer requests help. Do not pass a null result-pointer address. Result is an opaque owned handle; never allocate its guessed structure or dereference its layout from another language.

Two status layers

API return	Meaning
0	API call completed; inspect solver exit code and report
-1	Invalid argument
-2	Allocation failure
-3	Unexpected API failure

On an API failure, the result pointer is null. On API success, a returned result can still contain solver exit 1 or a limited/no-candidate outcome. Use `optimizer_exit_code` separately.

Solver exit	Typical interpretation
0	Mode-specific success; optimization may be feasible/limited, not proven optimal; root mode can finish with zero roots
1	Invalid/unsupported input or build capability error
2	Mode-dependent: e.g. no accepted NLP incumbent, numerical affine infeasibility or supported service infeasibility
3	Mode-dependent unresolved/unbounded/limited result without a candidate

Always read the mode's status and stopping reason. An API success is not a feasible-solution certificate, and an exit-2 work run may still publish a valid checkpoint.

Borrowed and owned objects

Function	Returned information / ownership
<code>optimizer_stdout(result)</code>	Borrowed NUL-terminated output string
<code>optimizer_stderr(result)</code>	Borrowed NUL-terminated diagnostic string
<code>optimizer_exit_code(result)</code>	Solver integer status
<code>optimizer_elapsed_seconds(result)</code>	Native call elapsed double, measured within the serialized call scope
<code>optimizer_free(result)</code>	Frees the owned result exactly once through the creating library
<code>optimizer_abi_version()</code>	Legacy ABI version; currently 1
<code>optimizer_has_binary128()</code>	Build capability, not a result-accuracy statement

Function	Returned information / ownership
<code>optimizer_has_model_import(form at)</code>	Supported import subset query for lp/mps/nl/nl-ascii/nl-binary

Copy borrowed strings/arrays before freeing the result. Do not call C free, delete, Marshal.FreeHGlobal, Julia Libc.free or FFI::free on optimizer-owned result memory. Use `optimizer_free` from the SAME loaded library. A result cannot survive unloading its library. Keep library handles alive while any result/context/callback/function pointer can use them. The examples use process-lifetime library loading rather than hot-unloading.

Current `optimizer_run` calls hold a library-wide mutex while executing and capturing `stdout/stderr`. They are serialized. Waiting to acquire that mutex is not the same as the native elapsed field or an exactly controlled request budget. For parallel legacy jobs, use separate processes. Avoid racing unrelated application `std::cout/std::cerr` writes against legacy capture. Typed contexts and direct placement have different concurrency contracts.

8. C callers: linked DLL/SO and runtime loading

Complete linked example

File c/legacy.c:

```
#include <stdio.h>
#include "optimizer_dll.h"
int main(void) {
    const char *args[] = {"--linear", "--eq",
        "binary x,y; minimize 2*x+y; x+y=1",
        "--lp-engine", "native", "--seconds", "5", "--tol", "1e-9"};
    OptimizerResult *r = NULL;
    if (optimizer_abi_version() != 1) return 1;
    int api = optimizer_run((int)(sizeof args / sizeof args[0]), args, &r);
    if (api != 0 || r == NULL) {
        fprintf(stderr, "API failure: %d\n", api);
        return 1;
    }
    int solver = optimizer_exit_code(r);
    fputs(optimizer_stdout(r), stdout);
    fputs(optimizer_stderr(r), stderr);
    printf("API=%d solver=%d elapsed=%.6f\n", api, solver,
        optimizer_elapsed_seconds(r));
    optimizer_free(r); /* All borrowed pointers are invalid now. */
    return solver;
}
```

The sample's integer choices have the known optimum $x=0$, $y=1$, objective 1. It intentionally requests native linear solving with a five-second allowance. The API may finish earlier. Its result is not a speed benchmark.

Windows MinGW caller

From the companion examples root, using an already built optimizer:

```
set "PATH=C:\Optimizer\native;C:\msys64\mingw64\bin;%PATH%"
gcc -std=c11 -Wall -Wextra -Iinclude c\legacy.c "C:\Optimizer\native\liboptimizer.dll.a" -o
c_legacy.exe
c_legacy.exe
```

The .dll.a is a link-time import library; optimizer.dll and its runtime dependencies are required at run time. Linking the import library does not embed the full engine into the caller.

Linux caller

```
gcc -std=c11 -Wall -Wextra -Iinclude c/legacy.c -L/absolute/optimizer/native -loptimizer
-Wl,-rpath,/absolute/optimizer/native -o c_legacy
./c_legacy
```

The shared library carries its own C++/quadmath dynamic dependencies. A C caller does not need to include STL headers or compile the engine. If you choose a relocatable RUNPATH, stage the library in the location it actually names.

MSVC hosts and no import-library dependency

A MinGW-generated .dll.a is not an MSVC .lib. A host can use runtime lookup of stable C exports without either import format. The following Windows example uses an ASCII DLL path for simplicity; a production Unicode loader can convert its path and use LoadLibraryW. Configure dependent runtime search paths before launching it.

File c/dynamic_windows.c:

```
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <stdio.h>
typedef struct OptimizerResult OptimizerResult;
typedef int (__cdecl *run_fn)(int, const char *const *, OptimizerResult **);
typedef const char *(__cdecl *text_fn)(const OptimizerResult *);
typedef int (__cdecl *exit_fn)(const OptimizerResult *);
typedef void (__cdecl *free_fn)(OptimizerResult *);
int main(int argc, char **argv) {
    if (argc != 2) { fputs("Pass an absolute optimizer.dll path\n", stderr); return 1; }
    HMODULE dll = LoadLibraryA(argv[1]); /* This demo expects an ASCII DLL path. */
    if (!dll) { fprintf(stderr, "LoadLibrary error %lu\n", GetLastError()); return 1; }
    run_fn run = (run_fn)GetProcAddress(dll, "optimizer_run");
    text_fn out = (text_fn)GetProcAddress(dll, "optimizer_stdout");
    text_fn err = (text_fn)GetProcAddress(dll, "optimizer_stderr");
    exit_fn status = (exit_fn)GetProcAddress(dll, "optimizer_exit_code");
    free_fn release = (free_fn)GetProcAddress(dll, "optimizer_free");
    if (!run || !out || !err || !status || !release) return 1;
    const char *args[] = {"--linear", "--eq",
        "binary x,y; minimize 2*x+y; x+y=1", "--lp-engine", "native", "--seconds", "5"};
    OptimizerResult *r = NULL;
    int api = run((int)(sizeof args / sizeof args[0]), args, &r);
    if (api != 0 || !r) return 1;
    fputs(out(r), stdout); fputs(err(r), stderr);
    int code = status(r); release(r);
    /* Keep the DLL loaded for process lifetime; do not hot-unload solver code. */
    return code;
}
```

Compile in an x64 Visual Studio developer command prompt:

```
cl /nologo /W4 c\dynamic_windows.c /Fe:dynamic_demo.exe
dynamic_demo.exe "C:\Optimizer\native\optimizer.dll"
```

Or compile the same caller with MinGW gcc. This demonstrates consuming the C ABI, not rebuilding the engine with MSVC. A reviewed but unexecuted Windows example is not a Windows validation claim.

If you prefer MSVC link-time imports, the companion optimizer.def lists the 51 observed C exports. Verify it against YOUR DLL before using:

```
dumpbin /exports "C:\Optimizer\native\optimizer.dll"
lib /def:optimizer.def /machine:x64 /out:optimizer.lib
cl /nologo /W4 /Iinclude c\legacy.c optimizer.lib /Fe:c_legacy.exe
```

Do not generate an import library from functions the actual DLL does not export. Probe optional additive functions; ABI version 1 alone does not imply every later function exists. The observed count is specific to the tested library.

9. C++ callers and RAI

File `cpp/legacy.cpp`:

```
#include <iostream>
#include <memory>
#include <stdexcept>
#include "optimizer_dll.h"
struct Release {
    void operator()(OptimizerResult *p) const noexcept { optimizer_free(p); }
};
int main() {
    try {
        const char *args[] = {"--linear", "--eq",
                               "binary x,y; minimize 2*x+y; x+y=1",
                               "--lp-engine", "native", "--seconds", "5"};
        OptimizerResult *raw = nullptr;
        const int api = optimizer_run(int(sizeof args / sizeof args[0]), args, &raw);
        std::unique_ptr<OptimizerResult, Release> result(raw);
        if (api != 0 || !result) throw std::runtime_error("optimizer_run failed");
        const int code = optimizer_exit_code(result.get());
        std::cout << optimizer_stdout(result.get());
        std::cerr << optimizer_stderr(result.get());
        std::cout << "solver=" << code << '\n';
        return code;
    } catch (const std::exception &e) {
        std::cerr << e.what() << '\n';
        return 1;
    }
}
```

Windows MinGW:

```
g++ -std=c++17 -Wall -Wextra -Iinclude cpp\legacy.cpp "C:\Optimizer\native\liboptimizer.dll.a" -o
cpp_legacy.exe
cpp_legacy.exe
```

Linux:

```
g++ -std=c++17 -Wall -Wextra -Iinclude cpp/legacy.cpp -L/absolute/optimizer/native -loptimizer
-Wl,-rpath,/absolute/optimizer/native -o cpp_legacy
./cpp_legacy
```

The custom deleter guarantees result release when an exception unwinds. It does not own application-created argv strings. No `std::string`, `std::vector`, C++ exception or optimizer internal class crosses the public ABI. This makes caller/engine separation clearer, but architecture, calling convention, scalar layout and native dependencies must still match.

The typed core API is also callable from C++. The complete C example in Chapter 15 compiles as C or can be adapted with RAI context ownership. Finish/join all worker calls before destroying a context. A destructor must not force destruction of a busy solver context.

10. Python: standalone ctypes and typed Core

Runtime loading and pointer declarations

ctypes avoids a Python-version-specific compiled extension. Use Python 3.9+ with a matching architecture. C pointers must use `c_void_p` (or an appropriate `POINTER` type), not the default `c_int`. A missing restype for a pointer-returning function can truncate an x64 pointer and crash the process. Set both `argtypes` and `restype`.

On Windows Python, keep `os.add_dll_directory` handles alive for the solve lifetime. Include the native library directory and required runtime directories. An absolute `optimizer.dll` path does not guarantee its dependencies will be found. On Linux, configure dependency search paths before starting Python where required.

File `python/legacy.py`:

```
"""python legacy.py ABS_LIBRARY [WINDOWS_RUNTIME_DIRECTORY ...]"""
import ctypes as C
import os
import sys
from pathlib import Path
from contextlib import ExitStack

def main():
    if len(sys.argv) < 2:
        raise SystemExit('Pass an absolute optimizer.dll/liboptimizer.so path')
    library = Path(sys.argv[1]).resolve()
    with ExitStack() as stack:
        if os.name == 'nt':
            for directory in [library.parent, *map(Path, sys.argv[2:]):]:
                stack.enter_context(os.add_dll_directory(str(directory.resolve())))
        lib = C.CDLL(str(library))
        lib.optimizer_abi_version.restype = C.c_int
        if lib.optimizer_abi_version() != 1:
            raise RuntimeError('Unsupported legacy ABI')
        lib.optimizer_run.argtypes = [C.c_int, C.POINTER(C.c_char_p), C.POINTER(C.c_void_p)]
        lib.optimizer_run.restype = C.c_int
        for name in ('optimizer_stdout', 'optimizer_stderr'):
            fn = getattr(lib, name); fn.argtypes = [C.c_void_p]; fn.restype = C.c_char_p
        lib.optimizer_exit_code.argtypes = [C.c_void_p]
        lib.optimizer_exit_code.restype = C.c_int
        lib.optimizer_elapsed_seconds.argtypes = [C.c_void_p]
        lib.optimizer_elapsed_seconds.restype = C.c_double
        lib.optimizer_free.argtypes = [C.c_void_p]; lib.optimizer_free.restype = None
        args = ['--linear', '--eq', 'binary x,y; minimize 2*x+y; x+y=1',
                '--lp-engine', 'native', '--seconds', '5', '--tol', '1e-9']
        data = [a.encode('utf-8') for a in args]
        argv = (C.c_char_p * len(data))(*data)
        result = C.c_void_p()
        api = lib.optimizer_run(len(data), argv, C.byref(result))
        if api != 0 or not result.value:
            raise RuntimeError(f'API failure: {api}')
        try:
            output = lib.optimizer_stdout(result).decode('utf-8')
            errors = lib.optimizer_stderr(result).decode('utf-8')
            code = lib.optimizer_exit_code(result)
            elapsed = lib.optimizer_elapsed_seconds(result)
        finally:
            lib.optimizer_free(result)
        print(output, end=''); print(errors, end='', file=sys.stderr)
        print(f'solver={code} elapsed={elapsed:.6f}')
        return code
    if __name__ == '__main__':
        raise SystemExit(main())
```

Windows:

```
python python\legacy.py "C:\Optimizer\native\optimizer.dll" "C:\msys64\mingw64\bin"
```

Linux:

```
python3 python/legacy.py /absolute/optimizer/native/liboptimizer.so
```

The `c_char_p` return converts a borrowed NUL-terminated native string into Python bytes. Decode/copy while the result is alive. Only the opaque result is passed to `optimizer_free`. Preserve extended numeric strings if ordinary float rounding would discard their precision.

Complete typed-core and concurrency example

Use the companion's matching `optimizer_core.py` next to `core_demo.py`. This adapter targets the public typed ABI and parses core JSON, not legacy stdout. Its complete source is included in the ZIP.

```
"""python core_demo.py ABS_LIBRARY [WINDOWS_RUNTIME_DIRECTORY ...]"""
import json
import os
import sys
from pathlib import Path
from contextlib import ExitStack
from concurrent.futures import ThreadPoolExecutor
from optimizer_core import Core
MODEL = 'binary x,y; minimize 2*x+y; x+y=1'

def main():
    library = Path(sys.argv[1]).resolve()
    with ExitStack() as stack:
        if os.name == 'nt':
            for directory in [library.parent, *map(Path, sys.argv[2:]):
                stack.enter_context(os.add_dll_directory(str(directory.resolve())))
            with Core(str(library)) as solver:
                solver.configure(seconds=5, seed=38, constraint_tolerance=1e-9)
                solver.linear_engine('native')
                solver.configure_milp(neighborhood_nodes=128, portfolio_passes=24)
                solver.primal_start({'x':0.0, 'y':1.0})
                def progress(event):
                    if event['seconds'] >= 2:
                        solver.cancel() # Request at an event; not a 2-second guarantee.
                solver.progress(progress)
                result = solver.solve(MODEL)
                print(json.dumps(result, indent=2))
                if not result['feasible']:
                    raise RuntimeError('No feasible candidate; inspect status')
            def job(_):
                with Core(str(library)) as solver:
                    solver.configure(seconds=5)
                    solver.linear_engine('native')
                    return solver.solve(MODEL)['objective']
            with ThreadPoolExecutor(max_workers=2) as pool:
                print('independent contexts:', list(pool.map(job, range(4))))
    if __name__ == '__main__': main()
```

```
python python\core_demo.py "C:\Optimizer\native\optimizer.dll" "C:\msys64\mingw64\bin"
```

```
python3 python/core_demo.py /absolute/optimizer/native/liboptimizer.so
```

Each worker owns a separate context. They may solve concurrently; reusing one context concurrently is not permitted. The callback object and context remain alive through solve. Cancellation is requested at callback opportunities; the tiny sample usually completes before two seconds and is not a cancellation-latency test.

`Core.configure` validates known typed fields; it does not accept arbitrary CLI flags. For example, `configure(work_limit=...)` is not a supported addition. Use the legacy CLI-compatible or subprocess path for work checkpointing and extended precision.

11. C# / .NET: UTF-8 P/Invoke on both systems

Project and implementation

The example targets .NET 8 and uses opaque pointers rather than automatic string ownership marshalling. It declares Cdecl explicitly. The runtime resolver loads the exact absolute native library path rather than assuming the same basename search behavior on every platform.

File csharp/OptimizerDemo.csproj:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>
</Project>
```

File csharp/Program.cs:

```
using System.Runtime.InteropServices;
using System.Reflection;

internal static class Native {
    [DllImport("optimizer", CallingConvention=CallingConvention.Cdecl)]
    internal static extern int optimizer_abi_version();
    [DllImport("optimizer", CallingConvention=CallingConvention.Cdecl)]
    internal static extern int optimizer_run(int argc, IntPtr argv, out IntPtr result);
    [DllImport("optimizer", CallingConvention=CallingConvention.Cdecl)]
    internal static extern IntPtr optimizer_stdout(IntPtr result);
    [DllImport("optimizer", CallingConvention=CallingConvention.Cdecl)]
    internal static extern IntPtr optimizer_stderr(IntPtr result);
    [DllImport("optimizer", CallingConvention=CallingConvention.Cdecl)]
    internal static extern int optimizer_exit_code(IntPtr result);
    [DllImport("optimizer", CallingConvention=CallingConvention.Cdecl)]
    internal static extern double optimizer_elapsed_seconds(IntPtr result);
    [DllImport("optimizer", CallingConvention=CallingConvention.Cdecl)]
    internal static extern void optimizer_free(IntPtr result);
    [DllImport("optimizer", CallingConvention=CallingConvention.Cdecl)]
    internal static extern IntPtr optimizer_core_create();
    [DllImport("optimizer", CallingConvention=CallingConvention.Cdecl)]
    internal static extern int optimizer_core_solve_text_v1(IntPtr ctx, IntPtr utf8);
    [DllImport("optimizer", CallingConvention=CallingConvention.Cdecl)]
    internal static extern IntPtr optimizer_core_result_json_v1(IntPtr ctx);
    [DllImport("optimizer", CallingConvention=CallingConvention.Cdecl)]
    internal static extern int optimizer_core_destroy(IntPtr ctx);
}

internal static class Program {
    private static string Text(IntPtr p) => p==IntPtr.Zero ? "" : Marshal.PtrToStringUTF8(p) ?? "";
    public static int Main(string[] args) {
        if (args.Length<1) { Console.Error.WriteLine("Pass an absolute native library path [--core]");return 1; }
        // One process-lifetime handle. Do not unload while P/Invoke can use it.
        IntPtr library=NativeLibrary.Load(Path.GetFullPath(args[0]));
        NativeLibrary.SetDllImportResolver(typeof(Native).Assembly,
            (name, assembly, searchPath) => name=="optimizer" ? library : IntPtr.Zero);
        if (Native.optimizer_abi_version()!=1) throw new InvalidOperationException("ABI mismatch");
        string model="binary x,y; minimize 2*x+y; x+y=1";
        if (args.Length>1 && args[1]=="--core") return Core(model);
        string[]
            options={"--linear", "--eq", model, "--lp-engine", "native", "--seconds", "5", "--tol", "1e-9"};
        IntPtr[] strings=new IntPtr[options.Length];
        IntPtr argv=IntPtr.Zero, result=IntPtr.Zero;
        try {
```

```

    argv=Marshal.AllocHGlobal((checked(options.Length*IntPtr.Size));
    for (int i=0;i<options.Length;i++) {
        if (options[i].Contains('\0')) throw new ArgumentException("NUL argument");
        strings[i]=Marshal.StringToCoTaskMemUTF8(options[i]);
        Marshal.WriteIntPtr(argv,i*IntPtr.Size,strings[i]);
    }
    int api=Native.optimizer_run(options.Length,argv,out result);
    if (api!=0 || result==IntPtr.Zero) throw new InvalidOperationException($"API failure {api}");
    Console.WriteLine(Text(Native.optimizer_stdout(result)));
    Console.WriteLine(Text(Native.optimizer_stderr(result)));
    int solver=Native.optimizer_exit_code(result);
    Console.WriteLine($"solver={solver}");
    elapsed={Native.optimizer_elapsed_seconds(result):F6}";
    return solver;
} finally {
    if (result!=IntPtr.Zero) Native.optimizer_free(result);
    foreach (IntPtr s in strings) if (s!=IntPtr.Zero) Marshal.FreeCoTaskMem(s);
    if (argv!=IntPtr.Zero) Marshal.FreeHGlobal(argv);
}
}
private static int Core(string model) {
    IntPtr ctx=Native.optimizer_core_create();
    if (ctx==IntPtr.Zero) throw new OutOfMemoryException();
    IntPtr text=IntPtr.Zero;
    try {
        text=Marshal.StringToCoTaskMemUTF8(model);
        int api=Native.optimizer_core_solve_text_v1(ctx,text); // Core defaults, including 20
        seconds.
        if (api!=0) throw new InvalidOperationException($"Core API failure {api}");
        Console.WriteLine(Text(Native.optimizer_core_result_json_v1(ctx)));
        return 0; // Caller must inspect JSON feasibility/status, not just this API success.
    } finally {
        if (text!=IntPtr.Zero) Marshal.FreeCoTaskMem(text);
        if (Native.optimizer_core_destroy(ctx)!=0) throw new
        InvalidOperationException("Context is busy");
    }
}
}
}

```

Arguments and the argv pointer array are caller allocations; the result is native-owned. Strings use `StringToCoTaskMemUTF8` and `FreeCoTaskMem` as a matched pair. The argv block uses `AllocHGlobal/FreeHGlobal`. The optimizer result uses `optimizer_free`. Returning native `char*` as a managed auto-freed string would violate this ownership contract; the example returns `IntPtr` and copies with `PtrToStringUTF8`.

Build and run on Windows

```

set "PATH=C:\Optimizer\native;C:\msys64\mingw64\bin;%PATH%"
dotnet build csharp\OptimizerDemo.csproj -c Release
dotnet run --project csharp\OptimizerDemo.csproj -c Release -- "C:\Optimizer\native\optimizer.dll"

```

Build and run on Linux

```

dotnet build csharp\OptimizerDemo.csproj -c Release
dotnet run --project csharp\OptimizerDemo.csproj -c Release --
/absolute/optimizer/native/liboptimizer.so

```

The .NET runtime architecture must match the native library. A managed AnyCPU build does not magically make an x86 process load x64 code. Inspect the actual deployment process. Configure Linux runtime dependencies as in Chapter 4.

The optional `--core` argument calls the minimal typed context and prints structured JSON:

```

dotnet run --project csharp\OptimizerDemo.csproj -c Release --
/absolute/optimizer/native/liboptimizer.so --core

```

That minimal typed variant uses the core defaults, including 20 seconds; the legacy variant explicitly requests five. Core API completion alone is not feasibility: parse status/feasible before using the decision. To configure typed options or callbacks from C#, map the full matching structures, initialize through the native default function and verify sizes/offsets with Chapter 16. Do not append a field or use Pack=1.

For .NET Framework versions without these UTF-8/NativeLibrary helpers, adapt loading/encoding deliberately or use a subprocess. The supplied project is not presented as source-compatible with every historical .NET release.

12. Java: JNA direct native calls

Requirements

Use a JDK (javac and java), a compatible JNA core JAR and matching native architecture. The example uses Java 11+ language/library features and JNA 5-style APIs. Obtain the appropriate JNA JAR from its official distribution for your chosen JDK; it is not bundled here. No JNI C++ shim or optimizer recompilation is needed for this example. JNA itself has its own runtime native-loading requirements.

File java/OptimizerDemo.java:

```
import com.sun.jna.Library;
import com.sun.jna.Native;
import com.sun.jna.Pointer;
import com.sun.jna.StringArray;
import com.sun.jna.ptr.PointerByReference;
import java.nio.file.Path;
import java.lang.ref.Reference;
import java.util.Map;

public class OptimizerDemo {
    public interface Api extends Library {
        int optimizer_abi_version();
        int optimizer_run(int argc, Pointer argv, PointerByReference result);
        Pointer optimizer_stdout(Pointer result);
        Pointer optimizer_stderr(Pointer result);
        int optimizer_exit_code(Pointer result);
        double optimizer_elapsed_seconds(Pointer result);
        void optimizer_free(Pointer result);
        Pointer optimizer_core_create();
        int optimizer_core_solve_text_v1(Pointer ctx, String model);
        Pointer optimizer_core_result_json_v1(Pointer ctx);
        int optimizer_core_destroy(Pointer ctx);
    }

    private static String text(Pointer p) { return p==null ? "" : p.getString(0,"UTF-8"); }
    public static void main(String[] args) {
        if (args.length<1) throw new IllegalArgumentException("Pass native library path [--core]");
        Api api=Native.load(Path.of(args[0]).toAbsolutePath().toString(),Api.class,
            Map.of(Library.OPTION_STRING_ENCODING,"UTF-8"));
        if (api.optimizer_abi_version()!=1) throw new IllegalStateException("ABI mismatch");
        String model="binary x,y; minimize 2*x+y; x+y=1";
        if (args.length>1 && args[1].equals("--core")) {
            Pointer ctx=api.optimizer_core_create();
            if (ctx==null) throw new OutOfMemoryError();
            try {
                int rc=api.optimizer_core_solve_text_v1(ctx,model); // Core defaults, including 20
                    seconds.
                if (rc!=0) throw new IllegalStateException("Core API failure "+rc);
                System.out.println(text(api.optimizer_core_result_json_v1(ctx)));
            } finally {
                if (api.optimizer_core_destroy(ctx)!=0) throw new IllegalStateException("Context
                    busy");
            }
            return;
        }
        String[]
        options={"--linear","--eq",model,"--lp-engine","native","--seconds","5","--tol","1e-9"};
        StringArray nativeArgs=new StringArray(options,"UTF-8");
        PointerByReference out=new PointerByReference();
        Pointer result=null;
        int solver=1;
        try {
            int rc=api.optimizer_run(options.length,nativeArgs,out);
            result=out.getValue();
            if (rc!=0 || result==null) throw new IllegalStateException("API failure "+rc);
            System.out.print(text(api.optimizer_stdout(result)));
        }
```

```

        System.err.print(text(api.optimizer_stderr(result)));
        solver=api.optimizer_exit_code(result);
        System.out.println("solver="+solver+" elapsed="+api.optimizer_elapsed_seconds(result));
    } finally {
        if (result!=null) api.optimizer_free(result);
        Reference.reachabilityFence(nativeArgs);
    }
    if (solver!=0) throw new IllegalStateException("Solver exit "+solver);
}
}

```

Library selects the ordinary C calling convention; do not replace it with StdCallLibrary. StringArray creates UTF-8 native argv storage and PointerByReference models OptimizerResult**. Borrowed char* is returned as Pointer and copied before freeing the result. The reachability fence preserves the argument owner through the call/cleanup scope.

Windows commands

```

set "PATH=C:\Optimizer\native;C:\msys64\mingw64\bin;%PATH%"
javac -cp "C:\Dependencies\jna.jar" java\OptimizerDemo.java
java -cp "java;C:\Dependencies\jna.jar" OptimizerDemo "C:\Optimizer\native\optimizer.dll"

```

Linux commands

```

javac -cp /opt/java-libs/jna.jar java/OptimizerDemo.java
java -cp java:/opt/java-libs/jna.jar OptimizerDemo /absolute/optimizer/native/liboptimizer.so

```

Classpath separators are semicolon on Windows and colon on Linux. Quote the entire classpath when it contains spaces. The optional final --core runs the typed default-context JSON variant. Use -Djna.debug_load=true before the main class for loader diagnostics. The presence of java without javac is insufficient to compile the sample.

For advanced typed structure maps, use Structure.FieldOrder with exact native field order, 64-bit integer fields for uint64_t, a pointer-sized mapping for size_t and ordinary native alignment. Retain callback instances strongly until native code is finished with them. This guide supplies a minimal typed JSON route without a hand-guessed options layout; use legacy flags or the fully mapped C/Python core examples when you need immediate complete option control.

13. Julia: ccall, pointer owners and returned strings

Julia calls the C ABI without a generated C++ wrapper. Libdl opens the explicit library, dlsym obtains function pointers, and ccall uses literal C argument types. Use Cint for a native int, Cdouble for double, Csize_t for size_t and Ptr/Ref for pointers. Julia's Int is not universally the C int type.

File julia/legacy.jl:

```
# julia legacy.jl ABS_LIBRARY [--core]
using Libdl
length(ARGS)>=1 || error("Pass an absolute native library path [--core]")
# Keep one library handle for this process; do not hot-unload function pointers.
const library = Libdl.dlopen(abspath(ARGS[1]))
const model = "binary x,y; minimize 2*x+y; x+y=1"
function copytext(p::Ptr{UInt8})
    p==C_NULL ? "" : unsafe_string(p)
end
function legacy()
    runfn=Libdl.dlsym(library,:optimizer_run)
    stdoutfn=Libdl.dlsym(library,:optimizer_stdout)
    stderrfn=Libdl.dlsym(library,:optimizer_stderr)
    exitfn=Libdl.dlsym(library,:optimizer_exit_code)
    freefn=Libdl.dlsym(library,:optimizer_free)
    args=["--linear","--eq",model,"--lp-engine","native","--seconds","5","--tol","1e-9"]
    buffers=[Vector{UInt8}(codeunits(s*\0)) for s in args]
    pointers=Ptr{UInt8}[pointer(b) for b in buffers]
    result=Ref{Ptr{Cvoid}}(C_NULL)
    api=GC.@preserve buffers pointers begin
        ccall(runfn,Cint,(Cint,Ptr{Ptr{UInt8}},Ref{Ptr{Cvoid}}),
            length(pointers),pointers,result)
    end
    api==0 && result[]!=C_NULL || error("API failure $api")
    try
        output=copytext(ccall(stdoutfn,Ptr{UInt8},(Ptr{Cvoid},),result[]))
        errors=copytext(ccall(stderrfn,Ptr{UInt8},(Ptr{Cvoid},),result[]))
        status=ccall(exitfn,Cint,(Ptr{Cvoid},),result[])
        print(output);print(stderr,errors)
        println("solver=",status)
        return status
    finally
        ccall(freefn,Cvoid,(Ptr{Cvoid},),result[])
    end
end
function core()
    createfn=Libdl.dlsym(library,:optimizer_core_create)
    solvefn=Libdl.dlsym(library,:optimizer_core_solve_text_v1)
    jsonfn=Libdl.dlsym(library,:optimizer_core_result_json_v1)
    destroyfn=Libdl.dlsym(library,:optimizer_core_destroy)
    ctx=ccall(createfn,Ptr{Cvoid},())
    ctx!=C_NULL || error("Cannot create context")
    try
        api=ccall(solvefn,Cint,(Ptr{Cvoid},Cstring),ctx,model) # Core default 20 seconds.
        api==0 || error("Core API failure $api")
        println(copytext(ccall(jsonfn,Ptr{UInt8},(Ptr{Cvoid},),ctx)))
        return 0
    finally
        ccall(destroyfn,Cint,(Ptr{Cvoid},),ctx)==0 || error("Context busy")
    end
end
end
exit(length(ARGS)>1 && ARGS[2]=="--core" ? core() : legacy())
```

Windows:

```
set "PATH=C:\Optimizer\native;C:\msys64\mingw64\bin;%PATH%"
julia julia\legacy.jl "C:\Optimizer\native\optimizer.dll"
```

Linux:

```
julia julia/legacy.jl /absolute/optimizer/native/liboptimizer.so
```

Add `--core` as the final argument for the typed default-context JSON variant. No additional Julia package is needed for basic `ccall`/Libdl; JSON parsing would require your chosen JSON library or application parser.

The buffers contain explicit terminating NUL bytes. `GC.@preserve` roots the owning vectors while raw pointers are passed. `unsafe_string` copies a native NUL-terminated output before `optimizer_free` invalidates it. Never use `Libc.free` for an optimizer result, and never keep using the borrowed pointer after result release. Julia-level garbage collection cannot release opaque native results for you without an explicit ownership wrapper.

Load once and retain the handle while callbacks, function pointers or native thread state can refer to it. Repeated `dlopen`/`dlclose` per optimization request is not the intended integration life-cycle. A production wrapper can cache its function pointers and expose a try/finally-managed result helper.

14. PHP: FFI and subprocess workflows

Choose a deployment route

PHP can call the DLL/SO directly through its FFI extension or launch the executable as a child process. FFI must be compiled/enabled and permitted by the chosen SAPI/configuration. It is not guaranteed available on shared hosting. The subprocess route avoids native ABI mapping in PHP and is often easier to isolate in a web-job worker.

The supplied examples are CLI scripts. Integrating them into a web front end requires controlled job paths, request/job lifetime handling and the intended server configuration. Do not assume new checkpoint options have automatically appeared in an old optimizer.php form.

Complete FFI example

File php/ffi_demo.php:

```
<?php
// php -d ffi.enable=true ffi_demo.php ABS_LIBRARY [--core]
if (PHP_SAPI !== 'cli') { throw new RuntimeException('This example is a CLI script'); }
if (!extension_loaded('FFI')) { throw new RuntimeException('PHP FFI extension is unavailable'); }
$path = isset($argv[1]) ? realpath($argv[1]) : false;
if ($path === false) { throw new RuntimeException('Pass a native library path'); }
$lib = FFI::cdef(<<<'CDEF'
typedef struct OptimizerResult OptimizerResult;
int optimizer_abi_version(void);
int optimizer_run(int, const char *const *, OptimizerResult **);
const char *optimizer_stdout(const OptimizerResult *);
const char *optimizer_stderr(const OptimizerResult *);
int optimizer_exit_code(const OptimizerResult *);
double optimizer_elapsed_seconds(const OptimizerResult *);
void optimizer_free(OptimizerResult *);
typedef struct optimizer_core_context optimizer_core_context;
optimizer_core_context *optimizer_core_create(void);
int optimizer_core_solve_text_v1(optimizer_core_context *, const char *);
const char *optimizer_core_result_json_v1(const optimizer_core_context *);
int optimizer_core_destroy(optimizer_core_context *);
CDEF, $path);
function copyNativeText($p): string {
    if (is_string($p)) return $p;
    if ($p === null || FFI::isNull($p)) return '';
    return FFI::string($p);
}
if ($lib->optimizer_abi_version() !== 1) throw new RuntimeException('ABI mismatch');
$model = 'binary x,y; minimize 2*x+y; x+y=1';
if (($argv[2] ?? '') === '--core') {
    $ctx = $lib->optimizer_core_create();
    if ($ctx === null || FFI::isNull($ctx)) throw new RuntimeException('Context allocation
failed');
    try {
        $api = $lib->optimizer_core_solve_text_v1($ctx, $model); // Default core time: 20s.
        if ($api !== 0) throw new RuntimeException("Core API failure $api");
        $json = copyNativeText($lib->optimizer_core_result_json_v1($ctx));
        $result = json_decode($json, true, 512, JSON_THROW_ON_ERROR);
        echo json_encode($result, JSON_PRETTY_PRINT | JSON_THROW_ON_ERROR), PHP_EOL;
    } finally {
        if ($lib->optimizer_core_destroy($ctx) !== 0) throw new RuntimeException('Context busy');
    }
    exit(0); // Inspect JSON feasibility/status, not this API completion alone.
}
$args = ['--linear', '--eq', $model, '--lp-engine', 'native', '--seconds', '5', '--tol', '1e-9'];
$n = count($args);
$nativeArgs = $lib->new("char *[$n]");
$buffers = [];
```

```

foreach ($args as $i => $arg) {
    if (strpos($arg, "\0") !== false) throw new RuntimeException('NUL argument');
    $b = $lib->new('char[' . (strlen($arg) + 1) . ']');
    FFI::memcpy($b, $arg, strlen($arg));
    $b[strlen($arg)] = "\0";
    $buffers[] = $b; // Preserve the buffers until the synchronous call returns.
    $nativeArgs[$i] = FFI::addr($b[0]);
}
$argptr = $lib->cast('const char *const *', FFI::addr($nativeArgs[0]));
$result = $lib->new('OptimizerResult *');
$sapi = $lib->optimizer_run($n, $argptr, FFI::addr($result));
if ($sapi !== 0 || $result === null || FFI::isNull($result)) throw new RuntimeException("API
failure $sapi");
try {
    $output = copyNativeText($lib->optimizer_stdout($result));
    $errors = copyNativeText($lib->optimizer_stderr($result));
    $code = $lib->optimizer_exit_code($result);
    echo $output;
    fwrite(STDERR, $errors);
    printf("solver=%d elapsed=%.6f\n", $code, $lib->optimizer_elapsed_seconds($result));
} finally {
    $lib->optimizer_free($result);
}
exit($code);

```

Windows:

```

set "PATH=C:\Optimizer\native;C:\msys64\mingw64\bin;%PATH%"
php -m
php -d ffi.enable=true php\ffi_demo.php "C:\Optimizer\native\optimizer.dll"

```

Linux:

```

php -m
php -d ffi.enable=true php/ffi_demo.php /absolute/optimizer/native/liboptimizer.so

```

The FFI extension must already be present; a php.ini flag cannot install a missing extension. The exact loader/INI setup depends on your PHP distribution. Add --core for the typed default-context JSON route. The legacy route explicitly requests a five-second solve; the minimal core route uses default core options.

The example uses a minimal C declaration string rather than sending an entire C++/macro-heavy header to FFI::cdef. It preserves CData buffer owners through the synchronous call. copyNativeText accepts either an already converted PHP string or a returned C pointer, then copies the native text before result release. Owned FFI allocations remain PHP-managed; optimizer-owned objects use the optimizer's release function.

Complete no-FFI subprocess example

File php/process_demo.php:

```

<?php
// php process_demo.php ABS_OPTIMIZER_EXE
if (PHP_SAPI !== 'cli') throw new RuntimeException('This example is a CLI script');
$exe = isset($argv[1]) ? realpath($argv[1]) : false;
if ($exe === false) throw new RuntimeException('Pass the executable path');
$command = [$exe, '--linear', '--eq', 'binary x,y; minimize 2*x+y; x+y=1',
    '--lp-engine', 'native', '--seconds', '5', '--tol', '1e-9'];
$out = tempnam(sys_get_temp_dir(), 'opt_out_');
$err = tempnam(sys_get_temp_dir(), 'opt_err_');
if ($out === false || $err === false) {
    if ($out !== false) unlink($out);
    if ($err !== false) unlink($err);
    throw new RuntimeException('Cannot allocate log files');
}
try {
    // Files avoid stdout/stderr pipe-buffer deadlock. Argument arrays avoid a shell.
    $process = proc_open($command,
        [0=>['pipe', 'r'], 1=>['file', $out, 'w'], 2=>['file', $err, 'w']],
        $pipes, null, null, ['bypass_shell'=>true]);

```

```
if (!is_resource($process)) throw new RuntimeException('Cannot launch optimizer');
fclose($pipes[0]);
$code = proc_close($process); // Synchronous; not an external hard timeout.
$output = file_get_contents($out);
$errors = file_get_contents($err);
if ($output === false || $errors === false) throw new RuntimeException('Cannot read logs');
echo $output;
fwrite(STDERR, $errors);
echo "process solver exit=$code", PHP_EOL;
} finally {
    unlink($out); unlink($err);
}
exit($code);
```

Run with PHP 8+:

```
php php\process_demo.php "C:\Optimizer\native\optimizer.exe"
```

```
php php/process_demo.php /absolute/optimizer/native/optimizer
```

The argument array avoids constructing a shell command with user-controlled quoting. std-out/stderr go to separate temporary files, so one full pipe does not deadlock the other while the parent waits. A production job manager should implement its chosen timeout/termination policy explicitly. `proc_close` is blocking and the solver's `--seconds` remains cooperative, not an external hard watchdog.

Use dedicated private working directories for model/checkpoint/export files, enforce application job ownership and preserve logs. Do not let two PHP requests write the same checkpoint. In web deployments, queueing/IPC can separate request lifetime from solver lifetime. Killing an in-process C++ thread is not a substitute for supervised worker-process control.

15. Typed core: configuration, callbacks and mathematical solves

Complete native example

File c/core.c:

```
#include <stdio.h>
#include <string.h>
#include "optimizer_core.h"
struct CallbackState { optimizer_core_context *ctx; double stop_after; };
static void progress(void *user, const optimizer_core_progress_v1 *event) {
    struct CallbackState *s = (struct CallbackState *)user;
    /* Only cancellation is reentrant on this same context. */
    if (event->has_objective)
        fprintf(stderr, "event=%u time=%.6f objective=%.17g\n",
            event->event, event->seconds, event->objective);
    if (event->seconds >= s->stop_after) optimizer_core_cancel(s->ctx);
}
int main(void) {
    optimizer_core_context *ctx = optimizer_core_create();
    if (!ctx) return 1;
    optimizer_core_options_v1 options;
    optimizer_core_default_options_v1(&options);
    options.seconds = 5;
    options.seed = 38;
    struct CallbackState state = {ctx, 2.0};
    int api = optimizer_core_set_options_v1(ctx, &options);
    if (api == OPT_CORE_OK)
        api = optimizer_core_set_linear_engine_v1(ctx, OPT_CORE_ENGINE_NATIVE);
    if (api == OPT_CORE_OK)
        api = optimizer_core_set_progress_v1(ctx, progress, &state);
    if (api == OPT_CORE_OK)
        api = optimizer_core_solve_text_v1(ctx,
            "binary x,y; minimize 2*x+y; x+y=1");
    if (api == OPT_CORE_OK) {
        const char *json = optimizer_core_result_json_v1(ctx);
        if (json) puts(json); /* Borrowed, not malloc-owned. */
    } else fprintf(stderr, "Core API error: %d\n", api);
    int closed = optimizer_core_destroy(ctx);
    return api == OPT_CORE_OK && closed == OPT_CORE_OK ? 0 : 1;
}
```

Compile just like c/legacy.c, using the same deployed library and include directory. It demonstrates default initialization, explicit native affine selection and a short callback. The known model normally finishes before the two-second callback threshold, so the example validates callback wiring rather than worst-case cancellation latency.

The full typed interface is declared in optimizer_core.h. The legacy header includes a compatibility core subset guarded by the same macro, so include optimizer_core.h first if an application also needs optimizer_dll.h. Do not assume the compatibility subset contains every later export. The ABI layout checker uses that include order.

Context and status contract

One solve may run per context. Independent contexts may execute concurrently. API statuses are OPT_CORE_OK=0, OPT_CORE_ERROR=1 and OPT_CORE_BUSY=2. They are not the legacy solver exit codes. After API completion, inspect structured status and feasible; bound/objective can be absent. A busy return means the context cannot accept that operation now, not that its model is infeasible.

Options/info structures must use the matching header and normal ABI alignment. Initialize

defaults via `optimizer_core_default_options_v1`; it sets `struct_size` and `abi_version=1`. Accessor info structures require their own size/version initialization. Do not assume a field can be appended to the old options structure merely because the public ABI number remains 1.

Returned strings are borrowed context storage until its next solve or destruction. Query only while idle. Copy JSON, names, CSV and status text before starting the next solve if you need to retain them. Protect context lifetimes in the application; getters and a busy return do not make arbitrary read/destroy races safe.

Options and ranges

Field	Current default / accepted values	Meaning
<code>seconds</code>	20; finite ≥ 0	Cooperative time; zero disables deadline
<code>constraint_tolerance</code>	1e-9; positive finite	Original constraint acceptance
<code>gradient_tolerance</code>	1e-6; positive finite	Local nonlinear stationarity
<code>relative_gap / absolute_gap</code>	0 / 0; nonnegative finite	Applicable numerical objective-gap targets
<code>seed</code>	42; uint64_t	Pseudorandom seed
<code>node_limit / pivot_limit</code>	100000 / 1000000; 0..2147483647	Applicable aggregate work limits
<code>frontier_memory_mb / basis_cache_mb</code>	256 / 64; max 1048576; frontier min 1	Estimated native frontier/cache budgets; cache 0 disables
<code>backend</code>	AUTO=0, LINEAR=1, NLP=2	Mathematical dispatch or forced affine/local nonlinear
<code>starts / iterations</code>	128 / 300; 1..1000000	Local restart/iteration allowances
<code>lbfgs_memory</code>	12; 1..100	Limited-memory history
<code>sparse_nlp</code>	0; nonzero requests sparse path	Nonlinear algebra selection
<code>strong_candidates / strong_pivots</code>	4 / 64; candidates 0..32; pivots ≤ 2147483647	Native branching probes

The affine engine is an additive selector: `optimizer_core_set_linear_engine_v1(ctx,AUTO/NATIVE/HIGHS)`, values 0/1/2. It is distinct from backend AUTO/LINEAR/NLP. Unsupported explicit backend combinations return errors rather than silently becoming arbitrary operational services.

Extra MILP portfolio options use their own versioned structure and default/set functions. Defaults: `enabled=1`, `neighborhood_nodes=128` (0..10000), `portfolio_passes=24` (0..1000), `neighborhood_seconds=5` (positive finite). The current Python adapter exposes `configure_milp` and `milp_info`. These values do not become arbitrary HiGHS parameter names.

Starts, files and result values

Named primal starts use `optimizer_core_set_primal_start_v1`. Names/values are borrowed for the call and copied; starts are reverified guidance, not proof or basis data. Partial starts are different from a complete work-mode incumbent file. The `solve_file` API accepts native, lp, mps and nl. It does not expose every CLI import/conversion/AMPL-driver option; choose the matching entry point.

Basic accessors return variable count/name/value, objective and bound. A successful objective/bound query requires that value to exist. Never replace an absent bound with zero. JSON includes schema/version, solver status, error, feasible, objective, bound, variables, original maximum violation, time, nodes/pivots and supported audit fields. For nonlinear results, `first_order_convergence` and `kkt_measure` are local facts, not a global certificate. Discovery-time fields can be null when not populated.

Typed core v1 uses doubles. There is no typed `--extend`, root-list, checkpoint cursor, arbitrary service-record or universal CLI-argument configuration mechanism. Use the legacy argument or executable route for those supported capabilities.

Callback event table and cancellation

Event	Value	Interpretation
STARTED	1	Solve entered
INCUMBENT	2	Verified feasible incumbent event
BOUND	3	Bound/work event; inspect has_bound
FINISHED	4	Solve finished
OBJECTIVE_STARTED	5	A priority objective pass begins
OBJECTIVE_FINISHED	6	That pass finishes

Callbacks run on the solving thread. The only reentrant same-context operation is `optimizer_core_cancel`. Do not query, solve, configure or destroy the same context from its callback. Keep user-data and callback function/delegate owners alive until the solve is finished. Avoid blocking logging/UI/network operations in callbacks. Catch language exceptions at the boundary; do not unwind across C.

Cancellation is a request checked cooperatively. An event-time threshold does not ensure events arrive every millisecond. A separate controlling thread can request cancellation while another thread solves, but must not destroy that context until the worker returns. No code here promises exact 2.0-second interruption or a 0.1-second result-return bound. OS scheduling, preparation, indivisible algebra, verification, callbacks and output/checkpoint work affect total latency.

16. ABI layouts and type mapping

Fundamental mappings

Native	C#	Python ctypes	Java/JNA	Julia	PHP FFI
int uint32_t	int uint	c_int c_uint32	int int bit pattern	Cint UInt32	int uint32_t declaration
uint64_t	ulong	c_uint64	long bit pattern	UInt64	uint64_t declaration
double void* / opaque pointer size_t	double IntPtr UIntPtr/nuint	c_double c_void_p c_size_t	double Pointer pointer-width unsigned wrapper	Cdouble Ptr{Cvoid} Csize_t	double typed C pointer size_t declaration
char* borrowed text T** out handle	IntPtr, then UTF-8 copy out IntPtr	c_char_p copy or pointer POINTER(c_void_p)	Pointer then getString PointerByReferen ce	Ptr{UInt8}, unsafe_string Ref{Ptr{Cvoi d}}	CData/string conversion FFI::addr(pointer variable)

C long is not a portable substitute for pointers or size_t: Windows x64 uses a different long width than Linux x64. uint64_t is unsigned even when a host runtime uses a signed long to hold its raw bits. Restrict application values to validated ranges before conversion. Do not marshal a C pointer using a 32-bit Java/C#/PHP integer.

Measured Linux x64 structure sizes

Structure	sizeof on tested Linux x64
optimizer_core_options_v1	120 bytes
optimizer_core_progress_v1	64 bytes
optimizer_core_objective_info_v1	104 bytes
optimizer_core_pareto_info_v1	104 bytes
optimizer_core_pareto_summary_v1	72 bytes
optimizer_core_polish_info_v1	80 bytes
optimizer_core_milp_options_v1	32 bytes
optimizer_core_milp_info_v1	72 bytes
optimizer_core_native_info_v1	80 bytes
OptimizerPlacementModel	152 bytes
OptimizerPlacementOptions	48 bytes

Measured important offsets are options.seconds=8, options.seed=48, options.backend=88 and progress.seconds=24. These are evidence for the tested Linux x64 ABI, not permission to bypass matching headers on another platform. The provided c/abi_layout.c reports actual sizes and offsets on the target without solving a model:

```
#include <stdio.h>
#include <stddef.h>
/* Core first avoids the legacy header's compatibility-core guard subset. */
#include "optimizer_core.h"
#include "optimizer_dll.h"
#define SHOW(T) printf(#T " %zu\n", sizeof(T))
int main(void) {
    SHOW(optimizer_core_options_v1); SHOW(optimizer_core_progress_v1);
    SHOW(optimizer_core_objective_info_v1); SHOW(optimizer_core_pareto_info_v1);
    SHOW(optimizer_core_pareto_summary_v1); SHOW(optimizer_core_polish_info_v1);
    SHOW(optimizer_core_milp_options_v1); SHOW(optimizer_core_milp_info_v1);
    SHOW(optimizer_core_native_info_v1);
    SHOW(OptimizerPlacementModel); SHOW(OptimizerPlacementOptions);
}
```

```
printf("options.seconds offset %zu\n", offsetof(optimizer_core_options_v1,seconds));  
printf("options.seed offset %zu\n", offsetof(optimizer_core_options_v1,seed));  
printf("options.backend offset %zu\n", offsetof(optimizer_core_options_v1,backend));  
printf("progress.seconds offset %zu\n", offsetof(optimizer_core_progress_v1,seconds));  
return 0;  
}
```

Compile with your target C compiler and compare to the host-runtime Structure/ctypes/StructLayout mapping before advanced typed calls. Use normal native alignment, not Pack=1. Structures containing pointers differ on a 32-bit process. The guide's intended deployment examples are x64.

17. Direct resource placement arrays

The placement API avoids converting arrays to model text. It is separate from the mathematical typed core and from serialized legacy CLI capture. File `c/placement.c`:

```
#include <stdio.h>
#include "optimizer_dll.h"
int main(void) {
    const double demand[] = {2, 3, 2};
    const double capacity[] = {5, 5};
    /* Dense cost table: object-major, then bin. */
    const double cost[] = {1, 4, 1, 4, 4, 1};
    OptimizerPlacementModel model = {0};
    model.struct_size = sizeof model;
    model.object_count = 3; model.bin_count = 2; model.resource_count = 1;
    model.demands = demand; model.demand_count = 3;
    model.capacities = capacity; model.capacity_count = 2;
    model.dense_costs = cost; model.dense_cost_count = 6;
    OptimizerPlacementOptions options = {0};
    options.struct_size = sizeof options;
    options.seconds = 5; options.tolerance = 1e-9; options.seed = 38;
    options.starts = 1000; options.moves = 2000000; options.neighbors = 8;
    OptimizerResult *result = NULL;
    int api = optimizer_place(&model, &options, &result);
    if (api != 0 || !result) return 1;
    int code = optimizer_exit_code(result);
    int count = optimizer_placement_count(result);
    if (code == 0 && count == 3) {
        const int *bins = optimizer_placement_bins(result);
        printf("objective=%.17g violation=%.17g\n",
            optimizer_placement_objective(result),
            optimizer_placement_violation(result));
        for (int i=0; i<count; ++i) printf("object %d -> bin %d\n", i, bins[i]);
    } else fputs(optimizer_stderr(result), stderr);
    optimizer_free(result);
    return code;
}
```

The example places demand 2 and 3 in bin 0, demand 2 in bin 1, giving objective 3 without exceeding either capacity 5. Bin indices in returned arrays are zero-based; a Julia application may convert to one-based indices for its UI while retaining the original ABI representation.

Input field	Layout / meaning
object_count, bin_count, resource_count	Dimensions n, b, r
demands / demand_count	Object-major n*r doubles
capacities / capacity_count	Bin-major b*r doubles
dense_costs / dense_cost_count	Optional n*b costs, object-major then bin
cost_objects, cost_bins, cost_values / cost_count	Optional sparse triples; do not combine with dense costs
default_cost	Optional n per-object fallback costs; missing sparse entries do not forbid placements
initial	Optional n initial zero-based bins; -1 unspecified
migration	Optional n move costs; interpreted with initial assignment
groups	Optional n group IDs; -1 none; same nonnegative group must occupy distinct bins
eligible_offsets, eligible_bins, eligible_count	Optional CSR whitelist, offsets n+1; each present row nonempty, unique and in range

Null optional arrays use documented defaults. Null `eligible_offsets` means all bins eligible; a present empty row is an error, not all eligible. Input arrays remain caller-owned and valid for the duration of the call. The implementation copies what it needs. Counts must match layouts; do not rely on an array's spare capacity to satisfy a wrong length.

Set `struct_size=sizeof(model/options)`. Null options select default 5 seconds, tolerance 1e-9, seed 42, 1000 starts, 2000000 moves and 8-object neighborhoods. A non-null options struct

must supply valid settings, not just a size field and zero tolerance. The example initializes every required field.

`optimizer_placement_count` and `optimizer_placement_bins` expose a borrowed assignment only when present. Objective/violation scalars for absent placement are not a zero-cost feasible solution. `optimizer_placement_complete=1` means finite enumeration completed under its numerical semantics, not exact-arithmetic certification. Input errors can return an owned result with solver exit 1; still free that result. Multiple independent `optimizer_place` calls are reentrant and do not redirect shared `stdout/stderr`.

18. Multiobjective, imports and high-precision result handling

Named objective controllers

Legacy calls pass the native model through `--eq/--file` and the matching arguments. Typed contexts call `solve_text/solve_file` with the same native mathematical declarations. Weighted blend and two-objective Pareto sampling support their documented LP/MILP/NLP/MINLP scope. Priority objectives require every objective/constraint to be affine. They are not extra controls for TSP or scheduling record grammars.

The companion models/pareto.txt and models/priority.txt are ready-to-run examples. Blending returns a signed scaled aggregate plus original objective values. A scalar Candidate 1 alone does not represent every returned Pareto trade-off. A completed epsilon grid is not a complete Pareto-front certificate. A feasible priority incumbent is not necessarily a completed hierarchy.

Typed JSON and additive objective/Pareto accessors preserve the full archive and per-pass qualification. Query `objective_mode` first. Pass value, final value after permitted degradation, bound and gap describe different quantities. Do not collapse them into an unexplained combined score.

The CLI `--result-json` flag is blend/Pareto-specific; `--pareto-csv` additionally requires Pareto. It is not a universal single-objective exporter. Typed-core JSON is a different interface and supports its own scalar/hierarchy results. Application-level JSON you construct yourself is independent of those CLI flags.

Extended precision

Check `optimizer_has_binary128` before sending `--extend` to the legacy path. Supported modes may print binary128 values; many operational modes reject it, while TSP/CVRP do not gain binary128 refinement simply because a flag is accepted. Preserve decimal strings or use a decimal-capable parser. Converting a long decimal token into a normal float/double discards information.

Typed-core v1 and its C#, Python, Java, Julia or PHP double mappings do not carry binary128. Use legacy text/structured exports where supported for extended witnesses. A tighter original feasibility tolerance is not a general objective-quality control and does not establish global optimality.

Imported models and names

Legacy supports matching CLI imports, conversion and AMPL SOL workflows. Core `solve_file` accepts its format parameter and supported model subsets. It does not import a complete modelling language runtime or arbitrary callbacks. AMPL/GAMS/Pyomo front ends need their own configured runtimes/adapters to export supported instantiated input.

NL primal starts can guide nonlinear initialization; dual starts are not used as dual certificates. An imported seed is reverified, not checkpoint continuation. Keep `.col/.row` naming sidecars with NL data when used. Reported variable order/names may differ from original business labels. Check model notices, unsupported constructs and original constraints before trusting the objective.

19. Checkpoints, deadlines and saved progress

Work mode is a separate algorithm

Supported scalar NLP/MINLP checkpointing is exposed through executable/legacy CLI-compatible calls, not typed-core option fields. A checkpoint preserves the verified incumbent, committed warm/search guidance, random state, assignment tracking and completed work cursor. It is not merely a primal starting point.

One unit is a bounded attempt, not one second. `--seconds 0` removes the clock ceiling but independent work limits still end the run. Work mode requires `--work-limit` whenever checkpoint/resume/work-phase flags are used. New local caps are `--work-iterations` (24), `--work-pivots` (2000) and `--work-qp-passes` (32), with their validated ranges in the User Guide.

Complete process example, `python/checkpoint_demo.py`:

```
"""python checkpoint_demo.py ABS_OPTIMIZER_EXE PRIVATE_WORK_DIRECTORY"""
import subprocess
import sys
from pathlib import Path

def main():
    exe=Path(sys.argv[1]).resolve()
    folder=Path(sys.argv[2]).resolve(); folder.mkdir(parents=True,exist_ok=True)
    model=folder/'work_demo.txt'; checkpoint=folder/'work_demo.owc'
    if model.exists() or checkpoint.exists():
        raise RuntimeError('Use a fresh private directory for this demonstration')
    model.write_text('minimize (x-3)^2+20*z;int z;bound z=[0,1];bound
x=[0,10];x>=2*z',encoding='utf-8')
    base=[str(exe),'--file',str(model),'--seconds','0','--seed','38','--tol','1e-9']
    for tag,units,mode in [('initial',8,'--checkpoint'),('resumed',16,'--resume')]:
        command=base+['--work-limit',str(units),mode,str(checkpoint)]
        run=subprocess.run(command,capture_output=True,text=True,encoding='utf-8')
        (folder/f'{tag}.stdout.txt').write_text(run.stdout,encoding='utf-8')
        (folder/f'{tag}.stderr.txt').write_text(run.stderr,encoding='utf-8')
        print(run.stdout,end='')
        if run.returncode not in (0,2):
            raise RuntimeError(run.stderr or f'Solver error {run.returncode}')
        if not checkpoint.is_file():raise RuntimeError('No confirmed checkpoint')
        # Exit 2 may still save state. Do not prevent its continuation with &&.
if __name__=='__main__':main()

python python\checkpoint_demo.py "C:\Optimizer\native\optimizer.exe" "C:\Optimizer\jobs\demo-new"

python3 python/checkpoint_demo.py /absolute/optimizer/native/optimizer /tmp/optimizer-demo-new
```

The demo insists on a fresh private directory, saves after eight completed units and adds 16 on resume. It accepts exit 2 as a potentially resumable no-incumbent state. Treat exit 1 or a failed checkpoint save as an error. The same argument lists can be passed to `optimizer_run` without the executable element, preserving the legacy two-layer status handling.

Compatibility and durability

Keep the same executable, parsed model, sidecars, precision, seed, tolerances and relevant algorithm settings. Compilation timestamp/platform/compiler and model settings are included in checkpoint identity. Rebuilding may invalidate the file. Additional unit count, time ceiling and output path can change; caps/tolerances/precision require a fresh state. To transfer a known point to a fresh configuration, import its complete coordinates and reverify it; an objective scalar is insufficient.

One snapshot is published at graceful invocation end, including cooperative stop. Hard process termination can lose unpublished progress. An incomplete unit replays its committed guidance later while any independently verified better partial point is retained. Atomic replacement/read-back/checksum protect confirmed publication; the checksum is not authenticity, and POSIX power-loss durability is not promised.

.running/.writing directories protect checkpoint ownership/publication. Do not share files between writers. A crash may leave a stale directory: verify process ownership before removing it. Use smaller work chunks for more frequent recovery points. Checkpoint writes and final verification are part of application latency; a search timer alone is not the total RPC deadline.

Known quality limitation

The user reported Specialized heat exchange at 166509.57334840705 after 32+128 work units, versus a reference-matching 154997.33486584452 from ordinary 30-second search. The timed command used the default seed rather than explicit work seed 38, so it is not a controlled same-seed experiment. It nevertheless demonstrates that checkpoint mode is not a universal quality replacement. Its schedule/caps differ. Earlier good-incumbent preservation tests did not establish cold-start discovery quality. Use the successful timed path for that case pending a tested correction; this developer guide does not claim that correction is implemented.

20. Hosting, concurrency and production lifecycle

Choose an integration architecture using required fault isolation, throughput, observability and deadline behavior. In-process loading reduces process-launch work but shares address space, native dependencies and crash consequences. A subprocess has launch/IPC costs but offers clearer job isolation and supervised termination. Neither design creates a universal optimality or deadline guarantee.

Pattern	Useful property	Engineering responsibility
Serialized legacy calls in one process	Simple argument-compatible integration	Account for mutex queueing and shared-stream capture
Independent typed contexts	Parallel mathematical jobs and callbacks	One active solve per context; protect lifecycle and callback ownership
Independent direct placement calls	Array-based concurrent placement	Correct dimensions, owners and result release
Worker process pool	Isolated jobs and controlled restarts	Stage files, supervise workers, retain verified fallback results

Initialize native loading once per worker. Keep immutable input buffers valid through call return. Copy results before releasing native ownership. Finish/join workers before destroying contexts or unloading code. Do not manipulate native function pointers after a reload or share handles created by different loaded copies.

Keep callbacks short and nonthrowing at the ABI. Marshal notifications to the application/UI asynchronously instead of blocking the solving thread. In long-lived services, limit application queues and output retention as well as native work. Estimated MILP frontier/cache settings are not whole-process RSS limits. The guide does not claim that bounded work implies no leaks or no crashes; measure repeated-call behavior and actual peak memory on target workloads.

Cancellation is cooperative. A strict external budget needs a hosting policy that accounts for parsing, scheduling, loader work, callbacks, verification, serialization and checkpointing. A request arriving after a feasible point exists can return it only if it has been safely published/copied under the chosen architecture. Do not kill arbitrary in-process C++ threads to enforce deadlines.

For web APIs, associate each model/checkpoint/result with a job ID and owned directory. Separate model-validation errors, search-limited no-candidate results, verified feasible candidates and numerical proof statuses. Do not treat stdout text as trusted HTML. Escape it before browser presentation; keep raw diagnostic files separately for reproducibility.

21. Troubleshooting by failure layer

Symptom	Likely layer and checks
Windows LoadLibrary error 126	Inspect dependent DLLs and controlled search paths, not only optimizer.dll existence
Windows error 193 / BadImageFormatException	Match x64/x86 and native file format; Windows DLL is not Linux SO
EntryPointNotFoundException / undefined symbol	Verify actual exports, header/adaptor version and calling convention
Linux GLIBC_* / GLIBCXX_* not found	Rebuild/stage against intended baseline; inspect transitive dependency requirements
Python crashes on returned pointer	Declare restype/argtypes; never truncate pointers into c_int
C# heap corruption	Check ownership, UTF-8 allocation pairs and pointer/struct alignment; never auto-free optimizer-owned char*
JNA UnsatisfiedLinkError	Check library path, native deps, JDK/JNA architecture and actual symbol spelling
Julia intermittent GC-related crash	Preserve pointer owners; copy borrowed data before optimizer_free
PHP FFI class absent / forbidden	Extension/SAPI policy; use the subprocess route where appropriate
CLI succeeds but DLL differs	Same binary release/model/argv/precision? Native arguments exclude executable and contain no shell quoting
PHP and terminal objectives differ	Compare complete argv, actual executable/DLL path, seeds, precision, starts/caps, input/sidecars and timed machine load
API=0 but no feasible point	Inspect solver code/status/feasible and mode semantics; API success only means call completed
Core API_BUSY	Another solve or protected operation owns that context; do not destroy/race it
Memory growth across requests	Verify every native result/context release, callback owners, retained JSON/logs, process queues and scoped cache behavior
Checkpoint mismatch	Restore the original binary/model/settings or start fresh from complete reverified coordinates
--seconds 0 ends early	Independent starts/iterations/nodes/work or a completed schedule can still stop search
Requested --extend ignored/rejected	Check build capability and service-specific precision contract

For diagnostics, reduce to the companion model with one native call and explicit library path. Confirm architecture/exports before changing numerical settings. Preserve exact errors and stderr; do not suppress them into an empty zero-cost result. A failed local search or no incumbent under a limit is not a generic infeasibility proof.

22. Tests, validation status and release practice

Executed checks for this companion

Path	Status
C legacy/core/placement/layout callers	GCC C11 compile and execution passed
C++ RAI caller	GCC C++17 compile and execution passed
Python ctypes legacy	Executed successfully
Python typed	Executed successfully
Core/callback/independent contexts	
Python subprocess checkpoint/resume	Executed successfully; 8+16 gives 24 completed units
Existing production DLL suite	48 integration checks passed
Windows / MinGW / MSVC caller	Not executed here
CMake, .NET, Java/JNA, Julia, PHP	Reviewed examples; required toolchains/runtimes unavailable for execution
Real external HiGHS-linked path	Not executed here

The validation runner performed 14 compile/run/suite steps on g++ (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0.

The companion validation/linux_checks.json retains command arguments, compiler identification, tested library SHA256 and raw outputs. abi_layout_linux_x64.txt records the measured layout. These validate the available paths on this build; they do not prove every language runtime, Windows portability, real-time deadlines or general memory safety.

Useful existing source tests

Run applicable tests from the engine source directory after building:

```
optimizer.exe --help
optimizer.exe --solver-self-test
optimizer.exe --lp-self-test
optimizer.exe --milp-self-test
optimizer.exe --lex-self-test
optimizer.exe --multiobjective-self-test
python tests\dll_smoke_test.py --library optimizer.dll
```

Linux:

```
./optimizer --self-test
./optimizer --solver-self-test
./optimizer --lp-self-test
./optimizer --milp-self-test
python3 tests/dll_smoke_test.py --library ./liboptimizer.so
python3 tests/nonlinear_multiobjective.py --library ./liboptimizer.so --executable ./optimizer
```

The current DLL smoke suite expects a quad-enabled build for its extended checks. A double-only product needs an appropriate subset/expectation adjustment; do not claim its expected binary128 absence is a solver regression. Some independent scripts need NumPy/SciPy or other runtimes. Inspect each current script's CLI/dependencies rather than passing obsolete wrapper options.

Suggested embedding release gate

Verify deterministic small answers, original feasibility, bad inputs, missing dependencies, architecture errors, repeated allocation/free, callback exception handling, cooperative cancellation, independent-context concurrency and no shared-output collisions. Exercise both cold-start and seeded searches, precision boundaries, imported names, multiobjective qualification and checkpoint incompatibility/corruption. Measure wall time, first feasible point, objective progression and memory on representative workloads.

Record unsuccessful cases and objective-quality regressions, including the Specialized work-mode limitation. A checkpoint byte-equivalence check validates state continuity, not cold-start quality. Test the deployed Windows/Linux artifacts on their actual target machines. Do not infer broad industrial superiority from small sample callers.

23. Public headers and export reference

Complete typed-core header snapshot

The following is the matching optimizer_core.h included in the companion. It is the authoritative list of typed structures and functions for these examples. Use a matching release header rather than manually copying a guessed subset into a production build.

```
#ifndef OPTIMIZER_CORE_V1_H
#define OPTIMIZER_CORE_V1_H
#include <stddef.h>
#include <stdint.h>
#if defined(_WIN32) && defined(OPTIMIZER_BUILD_CORE_API)
#define OPT_CORE_API __declspec(dllexport)
#else
#define OPT_CORE_API
#endif
#ifdef __cplusplus
extern "C" {
#endif
typedef struct optimizer_core_context optimizer_core_context;
enum { OPT_CORE_OK=0, OPT_CORE_ERROR=1, OPT_CORE_BUSY=2 };
enum { OPT_CORE_AUTO=0, OPT_CORE_LINEAR=1, OPT_CORE_NLP=2 };
enum { OPT_CORE_ENGINE_AUTO=0, OPT_CORE_ENGINE_NATIVE=1, OPT_CORE_ENGINE_HIGHS=2 };
enum { OPT_CORE_STARTED=1, OPT_CORE_INCUMBENT=2, OPT_CORE_BOUND=3, OPT_CORE_FINISHED=4,
OPT_CORE_OBJECTIVE_STARTED=5, OPT_CORE_OBJECTIVE_FINISHED=6 };
typedef struct optimizer_core_options_v1 {
    uint32_t struct_size, abi_version;
    double seconds, constraint_tolerance, gradient_tolerance, relative_gap, absolute_gap;
    uint64_t seed, node_limit, pivot_limit, frontier_memory_mb, basis_cache_mb;
    uint32_t backend, starts, iterations, lbfgs_memory, sparse_nlp, strong_candidates,
    strong_pivots;
} optimizer_core_options_v1;
typedef struct optimizer_core_progress_v1 {
    uint32_t struct_size, abi_version, event, has_objective, has_bound;
    double seconds, objective, bound;
    uint64_t nodes, pivots;
} optimizer_core_progress_v1;
typedef void (*optimizer_core_progress_callback_v1)(void*,const optimizer_core_progress_v1*);
/* Independent contexts may solve concurrently. One solve per context. Callback
runs on the solving thread; only cancel is reentrant on the same context.
Destroy/read/change a context only while idle. Returned strings live until
its next solve. C ABI uses standard doubles; no CLI output is parsed. Lexicographic progress
events 5/6 delimit each pass in descending-priority order; event objective
and bound values always refer to the active pass. Existing options layout
and events 1..4 are unchanged. */
OPT_CORE_API uint32_t optimizer_core_abi_version(void);
OPT_CORE_API void optimizer_core_default_options_v1(optimizer_core_options_v1*);
OPT_CORE_API optimizer_core_context* optimizer_core_create(void);
OPT_CORE_API int optimizer_core_destroy(optimizer_core_context*);
OPT_CORE_API int optimizer_core_set_options_v1(optimizer_core_context*,const
optimizer_core_options_v1*);
/* Additive engine selector; existing ABI-v1 option layouts are unchanged. */
OPT_CORE_API int optimizer_core_set_linear_engine_v1(optimizer_core_context*,uint32_t);
OPT_CORE_API int
optimizer_core_set_progress_v1(optimizer_core_context*,optimizer_core_progress_callback_v1,void*);
/* Named primal seeds are reverified; they are not proof or basis data. */
OPT_CORE_API int optimizer_core_set_primal_start_v1(optimizer_core_context*,size_t,const char*
const*,const double*);
OPT_CORE_API void optimizer_core_cancel(optimizer_core_context*);
OPT_CORE_API int optimizer_core_solve_text_v1(optimizer_core_context*,const char*);
OPT_CORE_API int optimizer_core_solve_file_v1(optimizer_core_context*,const char*,const char*);
OPT_CORE_API const char* optimizer_core_result_json_v1(const optimizer_core_context*);
OPT_CORE_API const char* optimizer_core_status_v1(const optimizer_core_context*);
OPT_CORE_API const char* optimizer_core_error_v1(const optimizer_core_context*);
OPT_CORE_API size_t optimizer_core_variable_count_v1(const optimizer_core_context*);
OPT_CORE_API const char* optimizer_core_variable_name_v1(const optimizer_core_context*,size_t);
OPT_CORE_API int optimizer_core_primal_value_v1(const optimizer_core_context*,size_t,double*);
```

```

OPT_CORE_API int optimizer_core_objective_v1(const optimizer_core_context*,double*);
OPT_CORE_API int optimizer_core_bound_v1(const optimizer_core_context*,double*);
/* Additive ABI v1 lexicographic accessors. Names/status strings are borrowed.
   These accessors require an idle context. Count is zero for a traditional
   single-objective solve. Bound and gap describe the pass, not final degradation.
   Initialize struct_size and abi_version=1 before get_objective_v1. */
typedef struct optimizer_core_objective_info_v1 {
    uint32_t struct_size, abi_version, index, priority, maximize;
    uint32_t has_pass_value, has_final_value, has_bound;
    double pass_value, final_value, bound, gap, absolute_tolerance, relative_tolerance, seconds;
    uint64_t nodes, pivots;
} optimizer_core_objective_info_v1;
OPT_CORE_API size_t optimizer_core_objective_count_v1(const optimizer_core_context*);
OPT_CORE_API const char* optimizer_core_objective_name_v1(const optimizer_core_context*,size_t);
OPT_CORE_API const char* optimizer_core_objective_status_v1(const optimizer_core_context*,size_t);
OPT_CORE_API int optimizer_core_get_objective_v1(const
optimizer_core_context*,size_t,optimizer_core_objective_info_v1*);
/* Additive multiobjective ABI v1. Point accessors require an idle context.
   Objective order is declaration order in blend/Pareto mode. Point indices
   are zero-based. quality=1 means both supported scalar/tie solves established
   numerical optimality; quality=0 means nondominated among returned samples.
   Bound/gap belong to bound_objective_index. Front completeness is never claimed.
   Borrowed mode/CSV/status strings survive until the context's next solve. */
typedef struct optimizer_core_pareto_info_v1 {
    uint32_t
    struct_size,abi_version,index,bound_objective_index,quality,stationary,has_epsilon,has_bound;
    double epsilon,objective_1,objective_2,bound,gap,maximum_violation,seconds;
    uint64_t nodes,pivots;
} optimizer_core_pareto_info_v1;
OPT_CORE_API const char* optimizer_core_objective_mode_v1(const optimizer_core_context*);
OPT_CORE_API int optimizer_core_objective_weight_scale_v1(const
optimizer_core_context*,size_t,double*,double*);
OPT_CORE_API size_t optimizer_core_pareto_count_v1(const optimizer_core_context*);
OPT_CORE_API int optimizer_core_get_pareto_v1(const
optimizer_core_context*,size_t,optimizer_core_pareto_info_v1*);
OPT_CORE_API int optimizer_core_pareto_primal_v1(const
optimizer_core_context*,size_t,size_t,double*);
OPT_CORE_API const char* optimizer_core_pareto_status_v1(const optimizer_core_context*,size_t);
OPT_CORE_API const char* optimizer_core_pareto_csv_v1(const optimizer_core_context*);
/* Sampling coverage is separate from global front completeness (always zero).
   Initialize struct_size and abi_version=1; requires an idle Pareto context. */
typedef struct optimizer_core_pareto_summary_v1 {
    uint32_t struct_size,abi_version,schedule_complete,all_thresholds_feasible,
    all_thresholds_optimal,front_complete;
    uint64_t thresholds,attempted_thresholds,feasible_thresholds,optimal_thresholds,
    anchor_retries,rejected_epsilon_points;
} optimizer_core_pareto_summary_v1;
OPT_CORE_API int optimizer_core_get_pareto_summary_v1(const
optimizer_core_context*,optimizer_core_pareto_summary_v1*);
/* Additive nonlinear multiobjective polishing audit; all counters describe
   verified search work, not optimality certification. Idle contexts only. */
typedef struct optimizer_core_polish_info_v1 {
    uint32_t struct_size,abi_version,enabled,reserved;
    uint64_t direct_tests,paired_tests,repair_trials,local_runs,accepted_improvements,
    reused_seeds,final_points;
    double seconds;
} optimizer_core_polish_info_v1;
OPT_CORE_API int optimizer_core_get_polish_v1(const
optimizer_core_context*,optimizer_core_polish_info_v1*);

/* Additive MILP portfolio controls and work audit. Existing ABI-v1 layouts
   are unchanged. Initialize struct_size and abi_version=1. Idle contexts only.
   Neighborhood/auxiliary LP bounds never enter the global objective bound. */
typedef struct optimizer_core_milp_options_v1 {
    uint32_t struct_size,abi_version,enabled,neighborhood_nodes,portfolio_passes;
    double neighborhood_seconds;
} optimizer_core_milp_options_v1;
typedef struct optimizer_core_milp_info_v1 {
    uint32_t struct_size,abi_version,supported,reserved;
    uint64_t absorbed_bound_rows,aggregation_cuts,auxiliary_lp_solves,neighborhood_calls,
    neighborhood_nodes,block_candidates,warm_starts;
} optimizer_core_milp_info_v1;
OPT_CORE_API void optimizer_core_default_milp_options_v1(optimizer_core_milp_options_v1*);

```

```

OPT_CORE_API int optimizer_core_set_milp_options_v1(optimizer_core_context*, const
optimizer_core_milp_options_v1*);
OPT_CORE_API int optimizer_core_get_milp_v1(const
optimizer_core_context*, optimizer_core_milp_info_v1*);

/* Additive native nonlinear search audit. No ABI-v1 structure was enlarged.
Counters are work performed, not exact-arithmetic certificates. */
typedef struct optimizer_core_native_info_v1 {
    uint32_t struct_size, abi_version, supported, reserved;
    uint64_t nodes, probes, pruned, auxiliary_substitutions, nlp_solves, ratio_rewrites, lp_relaxations,
    bound_tightenings;
} optimizer_core_native_info_v1;
OPT_CORE_API int optimizer_core_get_native_v1(const
optimizer_core_context*, optimizer_core_native_info_v1*);

#ifdef __cplusplus
}
#endif
#endif

```

Legacy and placement declarations

The complete optimizer_dll.h is in the ZIP. Its legacy/placement section is reproduced below; the compatibility typed-core subset that follows it in the file is intentionally not duplicated here. When using advanced typed accessors, include optimizer_core.h as described earlier.

```

#ifndef OPTIMIZER_DLL_H
#define OPTIMIZER_DLL_H

/* ABI version 1. Plain C arguments; no Python or C++ types cross this API. */
#ifdef WIN32
# if defined(OPTIMIZER_DLL_EXPORTS)
#  define OPTIMIZER_API __declspec(dllexport)
# else
#  define OPTIMIZER_API __declspec(dllimport)
# endif
# define OPTIMIZER_CALL __cdecl
#else
# define OPTIMIZER_API __attribute__((visibility("default")))
# define OPTIMIZER_CALL
#endif

#ifdef __cplusplus
extern "C" {
#endif

typedef struct OptimizerResult OptimizerResult;

/* Arguments EXCLUDE the executable name. argc=0 requests help.
API status: 0 = call completed (inspect optimizer_exit_code separately),
-1 = invalid argument, -2 = allocation failure, -3 = unexpected failure.
On failure, *result is NULL. Strings must be NUL-terminated, and remain
valid during the call. No shell interpretation is performed. */
OPTIMIZER_API int OPTIMIZER_CALL optimizer_run(
    int argc, const char* const* argv, OptimizerResult** result);

/* Borrowed strings belong to the result, valid until optimizer_free.
They contain captured solver output, including extended-precision digits.
A NULL result returns empty output, exit code -1 and elapsed time zero. */
OPTIMIZER_API const char* OPTIMIZER_CALL optimizer_stdout(const OptimizerResult* result);
OPTIMIZER_API const char* OPTIMIZER_CALL optimizer_stderr(const OptimizerResult* result);
OPTIMIZER_API int OPTIMIZER_CALL optimizer_exit_code(const OptimizerResult* result);
OPTIMIZER_API double OPTIMIZER_CALL optimizer_elapsed_seconds(const OptimizerResult* result);
/* Free exactly once using THIS library; optimizer_free(NULL) is safe. */
OPTIMIZER_API void OPTIMIZER_CALL optimizer_free(OptimizerResult* result);
OPTIMIZER_API int OPTIMIZER_CALL optimizer_abi_version(void);
OPTIMIZER_API int OPTIMIZER_CALL optimizer_has_binary128(void);
/* Native import capability, not a full-language parser. 1: documented subset

```

```

    of "lp", "mps", "nl-ascii", or "nl-binary" ("nl" includes both); 0 otherwise/NULL. */
OPTIMIZER_API int OPTIMIZER_CALL optimizer_has_model_import(const char* format);

/* Optional resource-placement API, added without changing ABI version 1.
All arrays are borrowed for the duration of optimizer_place; they are copied
by the solver. Caller must supply valid pointers with the stated lengths.
demands: object-major n*r; capacities: bin-major b*r.
Nullable n-element arrays: initial (-1 means unspecified), migration,
groups (-1 means none), default_cost. Group IDs are nonnegative integers;
members of one group must occupy different bins. One group per object.
Costs: dense n*b OR sparse triples, never both; defaults are zero unless
default_cost is supplied. Missing sparse costs do not forbid assignments.
Eligibility: NULL offsets = all bins eligible; otherwise CSR offsets n+1,
monotone from 0 to eligible_count. Each row must be nonempty and unique.
Null optional arrays use defaults. Empty rows are errors, not 'all bins'.
struct_size must equal sizeof the structure in this header. No ownership
crosses the ABI except OptimizerResult, released with optimizer_free. */
typedef struct OptimizerPlacementModel {
    unsigned int struct_size;
    int object_count, bin_count, resource_count;
    unsigned long long demand_count, capacity_count, dense_cost_count;
    const double *demands, *capacities, *dense_costs, *default_cost, *migration;
    const int *initial, *groups;
    const unsigned long long *eligible_offsets;
    unsigned long long eligible_count;
    const int *eligible_bins;
    unsigned long long cost_count;
    const int *cost_objects, *cost_bins;
    const double *cost_values;
} OptimizerPlacementModel;
typedef struct OptimizerPlacementOptions {
    unsigned int struct_size;
    double seconds, tolerance;
    unsigned long long seed;
    int starts, moves, neighbors;
} OptimizerPlacementOptions;
/* NULL options use 5 seconds, 1e-9 capacity tolerance, seed 42,
1000 starts, 2000000 moves, 8-object repair neighborhoods.
Same API status convention as optimizer_run. Input errors return an owned
result with exit code 1 and a clear stderr message. Exit 2 means no verified
feasible assignment under the search limits. Successful exit is 0.
This function is reentrant and does not redirect shared stdout/stderr.
Concurrent calls own independent models, PRNGs and results. */
OPTIMIZER_API int OPTIMIZER_CALL optimizer_place(
    const OptimizerPlacementModel* model, const OptimizerPlacementOptions* options,
    OptimizerResult** result);
/* Borrowed object-order, zero-based bin indices; NULL if no placement result.
Count 0 / objective 0 / violation 0 for absent placement. These scalar values
describe a candidate only when optimizer_placement_count is positive.
complete=1 means finite enumeration completed, NOT an exact arithmetic proof.
Otherwise optimality has not been established. */
OPTIMIZER_API int OPTIMIZER_CALL optimizer_placement_count(const OptimizerResult* result);
OPTIMIZER_API const int* OPTIMIZER_CALL optimizer_placement_bins(const OptimizerResult* result);
OPTIMIZER_API double OPTIMIZER_CALL optimizer_placement_objective(const OptimizerResult* result);
OPTIMIZER_API double OPTIMIZER_CALL optimizer_placement_violation(const OptimizerResult* result);
OPTIMIZER_API int OPTIMIZER_CALL optimizer_placement_complete(const OptimizerResult* result);

#ifdef __cplusplus
}
#endif

```

Observed exports

```

optimizer_abi_version
optimizer_core_abi_version
optimizer_core_bound_v1
optimizer_core_cancel
optimizer_core_create
optimizer_core_default_milp_options_v1
optimizer_core_default_options_v1
optimizer_core_destroy

```

```
optimizer_core_error_v1
optimizer_core_get_milp_v1
optimizer_core_get_native_v1
optimizer_core_get_objective_v1
optimizer_core_get_pareto_summary_v1
optimizer_core_get_pareto_v1
optimizer_core_get_polish_v1
optimizer_core_objective_count_v1
optimizer_core_objective_mode_v1
optimizer_core_objective_name_v1
optimizer_core_objective_status_v1
optimizer_core_objective_v1
optimizer_core_objective_weight_scale_v1
optimizer_core_pareto_count_v1
optimizer_core_pareto_csv_v1
optimizer_core_pareto_primal_v1
optimizer_core_pareto_status_v1
optimizer_core_primal_value_v1
optimizer_core_result_json_v1
optimizer_core_set_linear_engine_v1
optimizer_core_set_milp_options_v1
optimizer_core_set_options_v1
optimizer_core_set_primal_start_v1
optimizer_core_set_progress_v1
optimizer_core_solve_file_v1
optimizer_core_solve_text_v1
optimizer_core_status_v1
optimizer_core_variable_count_v1
optimizer_core_variable_name_v1
optimizer_elapsed_seconds
optimizer_exit_code
optimizer_free
optimizer_has_binary128
optimizer_has_model_import
optimizer_place
optimizer_placement_bins
optimizer_placement_complete
optimizer_placement_count
optimizer_placement_objective
optimizer_placement_violation
optimizer_run
optimizer_stderr
optimizer_stdout
```

All listed functions use C linkage. The tested native shared library exposed 51 optimizer-prefixed C symbols. This count is a release observation, not a substitute for capability detection. The ABI version stays 1 across additive features; check for optional function presence when supporting older binaries.

24. Official interoperability references and maintenance notes

Primary references consulted for the examples:

- .NET native library loading: <https://learn.microsoft.com/en-us/dotnet/standard/native-interop/native-library-loading>
- .NET native interop ownership guidance: <https://learn.microsoft.com/en-us/dotnet/standard/native-interop/best-practices>
- JNA Native API: <https://java-native-access.github.io/jna/5.19.0/javadoc/com/sun/jna/Native.html>
- JNA StringArray API: <https://java-native-access.github.io/jna/5.19.0/javadoc/com/sun/jna/StringArray.html>
- Julia calling C and GC preservation: <https://docs.julialang.org/en/v1/manual/calling-c-and-fortran-code/>
- PHP FFI class and cdef: <https://www.php.net/manual/en/class.ffi.php> and <https://www.php.net/manual/en/ffi.cdef.php>

These links explain the language mechanisms, not the optimizer's numerical guarantees. The optimizer source headers, implementations and matching build scripts define its interface. API examples here were independently written for this library and do not copy another optimizer's implementation.

Maintain public C ABI layouts additively/versionedly. Do not export STL containers or change an existing struct size/order without an intentional compatibility design. Add regression tests for a new accessor, ownership path or callback. Preserve numerical acceptance checks and do not silently label heuristic points as proven optima.

The current guide is based on the available source candidate dated 16 October 2026. No optimizer source was modified to produce it. The examples package intentionally excludes engine binaries, third-party libraries and external solver source/builds. Feature availability, deployment rights, support and distribution terms belong to the matching product agreement/release, not to assumptions made by a language wrapper.