Modern C++ Programming

28. Binary Size

Federico Busato

2026-01-09

Table of Contents

3 Coding Aspects

Table of Contents

Binary Size Introduction

Binary Size


r/fidelityinvestments • 3 hr. ago
by JustLiving2021
Join
Latest iOS App size >500Mb?!
I don't have much to add beyond the fact that Corporations seem to put 0% effort into optimizing App size these days. I could put an entire day of music on my phone for offline use for the "cost" of a Fidelity App (with no real functionality beyond the Mobile Website)

Binary Size - Examples

- `Android Games`: $\leq$ 1 GB
- `Safari Browser`: 1.4 GB on Mac
- `The CUDA Tookit`: 3 GB+
- `Microsoft Edge`: 3.6 GB+
- `Latex`: 7 GB+ (TeX Live)
- `Windows 11/macOS 14`: 20 GB+
- `Matlab`: 23 GB+ full
- `Microsoft Office`: 100 GB+ full

Impact of Binary Size

Software distribution:

- Download time.
- Data transfer cost.
- Storage cost.
- The effects are also exaggerated by update distributions or third-party integration.

Performance/Resources

- Compile and linking time.
- Run-time performance: Large binaries can lead to poor memory locality (instruction-cache, L1/L2/L3 cache misses, page faults).
- Startup time: loading the binary from disk.
- Disk and memory usage, very important on embedded systems.

Techniques to Reduce the Binary Size

- *Control compiler optimizations, hardware targets (vectorization), and exported symbols*.
- *Minimize C++ code generation*, e.g. templates, inlining, exceptions, etc.
- *Functions and classes organization*, e.g. external linkage.
- *Dependencies management*, e.g. headers, shared libraries, raw data, etc.
- *Just-in-time compilation* (advanced).
- *Compression* (advanced).

Compiler and Linker Techniques

Overview of Compiler and Linker Techniques

One of the most effective ways to reduce binary size with little effort is to instruct the compiler to **optimize for size** instead of performance.

Runtime exceptions require the compiler to introduce additional code to handle them, `example`. The user can control how the compiler treats exceptions.

Link-Time Optimization (LTO) can help reduce binary size, analyzing the *entire program* to remove unused code (*dead code elimination*), function *inlining* across different modules to optimize code usage, and *devirtualization* to replace virtual methods (*vtable*) with direct calls when possible.

Optimization Flags

`-Os` , `/O1` *Prioritizes reducing binary size over speed improvements.*
Enable all `-O2` optimizations that do not increase code size, e.g. dead code elimination, constant propagation, expression simplification, etc. and exclude techniques such as loop unrolling, strong function inlining, code alignment, etc.

`-Oz` *Aggressive size optimization, omitting performance optimizations.*
Supported by Arm and proprietary compilers but not by GCC/Clang. Might result in slower code, loop unrolling and loop vectorization are disabled, loops are generated as `while` loops instead of `do-while` loops.

Optimization Flags

`-Omin` *Smaller size than* `-Oz` *by exploiting a subset of LTO functionalities.* Relying on LTO to remove unused code and data and try to eliminate virtual functions. Not supported by GCC/Clang.

`-fipa-icf` **I**nter**P**rocedural **A**nalysis - **I**dentical **C**ode **F**olding detects and unifies variables with identical values and functions with the exact machine code reducing code size without changing observable behavior

---

Safe ICF: Pointer Safe and Unwinding Aware Identical Code Folding in Gold GCC Optimization Options

Debugging and Runtime Information Flags

`-g`, `-g<N>`, `/DEBUG` Generate debugging information. Ensure the flag is not present. Debugging information could significantly increase the binary size up to 60-80%. `-g3` up to 120-160%.

`-DNDEBUG` `/DNDEBUG` *Remove assertions* which contribute to the binary size.

`-fno-rtti`, `/GR-` *Disable Run-Time Type Information*, such as `typeid` and `dynamic_cast`. The flag has negligible impact.

Symbol Visibility

Exported symbols are stored in the *symbol table*, which includes their fully *mangled* names. They are used for debugging purposes. The associated names can become very long due to templates and namespaces, leading to large symbol tables and bloating the binary size. This aspect can be controlled by function attributes and compiler options.

- `-fvisibility=hidden`
  Sets the default visibility of symbols to hidden (not exported).
  The flag can reduce the binary size of dynamic libraries by 5-20%, up to 80% in extreme cases. The exported functions need to be explicitly marked with `[[gnu::visibility("default")]]` / `__declspec(dllexport)`.

- `-fvisibility-inlines-hidden`
  Sets the visibility of inlined functions to hidden.

`Why is the new C++ visibility support so useful?`

Exceptions Flags

`-fno-exceptions`, `/EHsc` *Remove exception handling code*. The flag has negligible impact in general, especially with optimizations. Result: exceptions are replaced by `std::abort`.

`-fno-unwind-tables` *Remove unwind tables for exception handling*. `Unwind tables` are metadata introduced in the code used to reverse the effects of function calls when an exception occurs. Removing them could reduce the binary size about ∼10-15%.

`-D_HAS_EXCEPTIONS=0`
`/D_HAS_EXCEPTIONS=0` *Disable exceptions in the standard library*.

- `Binary size and exceptions`
- `The true cost of C++ exceptions`

`-s`, `-Wl,-s` [GCC/Clang] *Remove all symbol tables and relocation information*. `Relocation information` is needed by shared libraries and for security purposes.

Alternatively, the programs `strip -s` / `strip.exe` can be used after the binary is compiled. For *shared libraries* the tool provides the flag `--strip-unneeded` to remove all symbols that are not needed for relocation in addition to debugging symbols.

~~`-fpic/-fPIC`~~ Don't use **P**osition-**I**ndependent **C**ode if the target is not a share library.

~~`-fpie/-fPIE`~~ Don't use **P**osition-**I**ndependent **E**xecutable if security features like Address Space Layout Randomization (ASLR) are not needed.

`-Wl,--gc-sections` *Perform garbage collection on sections* (functions or data) that are not referenced anywhere in the whole program. To be effective, the linker flag should be also combined with the compiler flags `-ffunction-sections` and `-fdata-sections`. The flags place global/static variables and functions into its own section of the object file to help the linker.

`-Wl,--exclude-libs,ALL` *Don't export symbols of static libraries that are linked when creating shared libraries.*

`-Wl,-nmagic` *Turn off page memory alignment of sections.*

`-Wl,-pack-dyn-relocs=relr` Dynamic relocations are information stored in shared library used by the runtime loader resolves to map the code into memory. *The flag packs dynamic relocations to reduce their size* in the final ELF by using more compact encodings.

Link-Time Optimization (LTO)

`-flto` *Enable Link Time Optimizations.*

The flag must be used in both compile and link stages. It could reduce the binary size by 30%, especially if combined with optimization flags that doesn't increase the binary size. In other cases, the flag could have the opposite result, increasing the binary size.

- Link-Time Optimizations: New Way to Do Compiler Optimizations
- Linktime optimization in GCC, part 3 - LibreOffice

CMake Support

```
set_target_properties(my_program PROPERTIES
    C_VISIBILITY_PRESET       hidden
    CXX_VISIBILITY_PRESET     hidden
    VISIBILITY_INLINES_HIDDEN YES
)
```

```
set_property(TARGET my_program PROPERTY
    INTERPROCEDURAL_OPTIMIZATION ON  # LTO
)
```

References

- Trying to minimize C/C++ binary size as much as possible
- How to make smaller C and C++ binaries
- How to minimize Rust binary size
- State of the art for reducing executable size with heavily optimized program

Coding Aspects

Function Inlining

Function inlining doesn't automatically translate in larger binary size:

- *Inlining* can reduce code size if the function body is smaller than the overhead of a function call (parameters, returning values, function call/jump).
- *Inlining* increases code size for non-trivial functions due to duplicating the function body at each call site.
- *Inlining* is controlled by
  - Compiler optimization flags, e.g. `-O3`, `-finline-functions`.
  - Function size and inlining depth.
  - Function decorators: `inline` (increase inlining heuristic), `__forceinline`, `[[gnu::always_inline]]`.

Function Visibility

All functions that are not part of the *public interface* should be declared with *hidden visibility* `[[gnu::visibility("hidden")]]` to avoid exporting the associated symbols.

```
[[gnu::visibility("hidden")]]
void private_function() { ... };

void public_function() { private_function(); }
```

```
// -fvisibility=hidden
void private_function() { ... };

[[gnu::visibility("default")]]
void public_function() { private_function(); }
```

Templates Introduction

Templates allow creating *generalized* versions of functions, classes, and variables that work with different data types or compile-time constants.

Even minimal template instantiation, like type traits, costs 1KB of binary size.

Template code can have a large impact on binary size because the compiler generates code for *each instantiation*.

Because the generation process is *automatic and implicit*, it is often difficult to keep track of all instantiations, causing the size of the binary to grow rapidly. The problem is exacerbated by multiple template entities, nested calls, and metaprogramming.

- `2024 LLVM Dev Mtg - Generic implementation strategies in Carbon and Clang`
- `Factoid: Each class template instantiation costs 1KiB`

A `template` **class/function** is **instantiated** in the following cases:

Class/Function Full specialization.

Class/Function Explicit instantiation.

Class/Function The `template` is defined/called within the body of a fully specialized/non-template function or class.

Class A function has a specialized `template` class *return type* and the function is defined (body) or called.

A `template` is **not instantiated** in the following cases:

- A *pointer* to a specialized `template` class.
- A *type declaration*.
- The `template` class/function is defined within the body of a non-fully specialized function or class, and depends on their template parameters.

The compiler doesn't generate the code of a `template` **class/functions** if it has already been *instantiated* before in the same translation unit.

```
void f() { std::array<int, 2> array; }  // first instantiation

struct A {
    std::array<float, 2> array; // second instantiation
void f(std::array<int, 2>* array); // not instantiated
void f(std::array<int, 2>& array); // instantiated

struct A {
    std::array<float, 2>* array1; // not instantiated
    std::array<float, 2>& array2; // instantiated
std::array<int, 2> f(); // not instantiated
f();                    // std::array<int, 2> instantiated
```

```cpp
template<typename T, int N>
void f() { std::array<T, N> array; }  // not instantiated

f<int, 4>();                          // std::array<int, 4> instantiated

template<typename T, int N>
struct A {
    std::array<T, N> array;           // not instantiated
};
template<typename T>
struct A<T, 3> {
    std::array<T, 3> array;           // not instantiated
};

template struct A<int, 3>;            // std::array<int, 3> instantiated
```

Template - Type Erasure

Templates are often overutilized even for simple functionalities. **Type erasure** can be useful for reducing the number of template instantiations.

```
template<typename T>
T* align1(T* ptr, size_t align_size) {
    return ptr + (align_size - reiterpret_cast<uintptr_t>(ptr) % align_size);
} // one instantiation for each type

const void* align2(const void* ptr, size_t align_size) {
    return ptr + (align_size - reiterpret_cast<uintptr_t>(ptr) % align_size);
}
```

Static Storage Duration

Global variables and **local scope** `static` **variables** have *static storage duration* and persist for the *program lifetime*. They directly contribute to the binary size, requiring space in the *data segment* (initialized) and *BSS* (Block Started by Symbol, uninitialized).

If *multiple translation units* need the same global variable, it should be:

- declared `extern` in each translation unit and defined in a single source file.
- defined `inline` in a single header to include in each translation unit.

Linkage

- Entities with **internal linkage** (`static` , *anonymous namespace*) should never be declared in headers to avoid duplication in any translation units that include them.

- Entities that are intended to be used only within a single translation unit should have **internal linkage** to avoid exporting the symbol and save space.
  In addition, *internal linkage* often enables optimizations, such as dead code elimination and function removal, which can further reduce the binary size when combined with the `--gc-sections` flag.

- **Global** `const` / `constexpr` **variables** have **internal linkage**, as `static` variables. When declared in headers, they should be marked `inline` to allow the linker to remove duplicate copies.

Polymorphic classes

A **polymorphic class** is any class that declares or inherits at least one `virtual` method. The run-time dispatch mechanism relies on the **virtual table** (`vTable`).

A *polymorphic class* increases binary size significantly, up to 10 times, compared to a non-virtual class due to the `vtable` and associated metadata.
Most of the overhead comes from the introduction of the first `virtual` method, while the next ones have a negligible impact.

Exceptions

`noexcept` informs the compiler that a function will not throw exceptions, which could potentially reduce the binary size:

- The compiler avoids exception handling code and metadata generation.
- `noexcept` is particularly useful for cross-translation-unit (LTO) optimizations, because the linker can make strong assumptions about function behavior.
- Functions without exceptions are more likely to be inlined.

Merely including headers in source files, even large ones, doesn't directly increase the size of the binary. On the other hand, they do affect binary size, depending on their content and how they are used.

Contribution to the binary size:

- **Symbols with internal linkage**: each translation unit gets its own copy.
  The compiler is able to remove unused symbols (*dead code elimitation*).
- **Inline functions and variables**: code generation, inlining, and symbol table.
  The linker is able to remove duplicate symbols.
- **Templates**: contribute in the same way of regular classes, functions, and variables, but only if they are instantiated.
  The linker is able to remove identical template instantiations.

When a **polymorphic class** is defined in a header, the compiler maintains a copy of the `vTable` and `RTTI` data for each translation unit.

Modern compilers emit the `vTables` in a special section called `COMDAT`. Linkers that support this feature are able to remove duplicate copies in the final program.

Another solution to reduce binary size is to move the implementation of `virtual` methods into a dedicated source file, since the `vTables` and metadata are generated only once, not per translation unit.

- `GCC ld: Vague Linkage`
- `LLVM ldd: Missing Key Function`

It is important to note that the program is often distributed as a *header library* or as a *static library*, namely a collection of object files. In these cases, the behavior of the linker is beyond the control of the developer, and the final program cannot rely on linker techniques to reduce binary size. Aspects such as *internal linkage*, *exported symbols*, and *polymorphic classes* must be addressed directly.

Secondly, even if the linker is involved in program generation, the size of object files can still affect intermediate compilation steps, affecting their load time, disk size, and memory footprint.

Tools like C++ Compile Health Watchdog and STL Explorer show the impact of including the standard library headers, in terms of binary size and dependencies, respectively.

<filesystem>	263 .. 341 ms	30.4 .. 31.1 kLoC	0 .. 363 kB
<future>	179 .. 292 ms	20.5 .. 23.5 kLoC	0 .. 278 kB
<regex>	238 .. 365 ms	38.9 .. 43.7 kLoC	0 .. 188 kB
<iomanip>	115 .. 221 ms	18.8 .. 24.7 kLoC	0 .. 180 kB
<locale>	113 .. 196 ms	18.6 .. 22.1 kLoC	0 .. 178 kB
<thread>	110 .. 189 ms	17.5 .. 20.3 kLoC	0 .. 153 kB
<condition_variable>	112 .. 192 ms	16.5 .. 19.4 kLoC	0 .. 153 kB

Binary Size Tools

nm and objdump

nm and **objdump** are standard tools available on Linux systems for analyzing binary size.

nm can list non-stripped symbols in a binary and their associated sizes:

```
nm --print-size --size-sort <binary>
```

objdump can provide information related to each section of the binary which is useful for understanding how much space code, data, and other sections occupy.

```
objdump --headers <binary>
```

Bloaty

Bloaty is an advanced tool for performing deep analysis of a binary. The tool allows to analyze multiple object files in a simple way, filter binary information depending on its sections or debugging symbols, and even track binary growth over time for CI testing.

```
$ ./bloaty bloaty -d compileunits
    FILE SIZE        VM SIZE
 --------------  --------------
  34.8%  10.2Mi  43.4%  2.91Mi    [163 Others]
  17.2%  5.08Mi   4.3%   295Ki    third_party/protobuf/src/google/protobuf/descriptor.cc
   7.3%  2.14Mi   2.6%   179Ki    third_party/protobuf/src/google/protobuf/descriptor.pb.cc
   4.6%  1.36Mi   1.1%  78.4Ki    third_party/protobuf/src/google/protobuf/text_format.cc
   3.7%  1.10Mi   4.5%   311Ki    third_party/capstone/arch/ARM/ARMDisassembler.c
   1.3%   399Ki  15.9%  1.07Mi    third_party/capstone/arch/M68K/M68KDisassembler.c
```

Executable Packer

An **executable Packer** is a tool for *compressing* executable binaries and shared libraries to reduce their size without changing their functionality. The binary is compressed offline, then the embedded decompression routine rebuilds the original code at runtime before actual execution. Binary compression aims at reducing distribution and storage costs.

- `UPX` is the most popular executable packer. UPX typically reduces the file size of programs and DLLs by around 50%-70%. It is open-source, actively maintained, and offer fast decompression.
- `MPRESS` is an alternative to UPX. The tool is based on the LZMA compression algorithm and could provide a better compression ratio. On the other hand, it is less popular than UPX and no more maintained.