UNIVERSITÀ DI PISA

Department of Computer Science

Internal Memory Management in GCC During Compilation

An Architectural Analysis of Obstacks, GGC, and Memory Pools

Course:
Advanced Programming

Student:
Andrea Filippi

Academic Year
2025/2026

Index

Internal Memory Management in GCC

Architectural Overview of Memory Management in the GNU Compiler Collection (GCC)

The GNU Compiler Collection (v13+) employs a specialized, hybrid memory management architecture designed to sustain high compilation throughput while managing the complex lifecycles of Intermediate Representation (IR) artifacts. Modern GCC has transitioned significantly to C++, utilizing templates and strongly typed allocators to optimize the legacy C-style management mechanisms.

1. Management of Long-Lived Objects: The GGC Subsystem

The **GGC (GCC Garbage Collector)** serves as the persistence layer for the compiler, managing objects with indeterminate lifetimes or those that must survive across multiple compilation passes.

- **Target Objects:** GGC manages the core IR structures: `tree` nodes (high-level IR), `gimple` statements (middle-end IR), `rtx` (Register Transfer Language), and global symbol table entries.
- **Mechanism:** GGC utilizes a precise mark-and-sweep garbage collection strategy. It eschews reference counting in favor of "roots"—global or static pointers explicitly registered with the collector.
- **GTY and Metaprogramming:** To handle root registration and traversal automatically, GCC uses a custom generator, `gengtype`. Source structures requiring collection are annotated with `GTY(())` markers. During the build, `gengtype` generates the `gt-*.h` files containing the typed traversal functions used by the collector.
- **Allocation Strategy (Modern C++):**
    - **Frontend Interface:** Allocation is primarily handled via strongly typed templates defined in `ggc.h`, such as `ggc_alloc<T>()`. These are implemented as inline functions to reduce call overhead.
    - **Backend Implementation:** The inline templates delegate to the physical allocator, principally implemented in `ggc-page.cc`. This module organizes the heap into "pages" of specific order sizes (powers of two). Large allocations bypass the page logic and are handled via standard OS primitives (e.g., `mmap`).

2. Management of Short-Lived Data: Obstacks and Object Allocators

For data with deterministic scopes (e.g., pass-local analysis), GCC bypasses the GGC to reduce GC pressure, favoring region and pool allocation.

A. Obstacks (Object Stacks)

Despite the C++ transition, GCC retains the efficient "Obstack" mechanism (via `libiberty`) for variable-sized, stack-like allocations.

- **Use Case:** Obstacks are the standard backing store for `bitmap` structures (via `bitmap_obstack`), which are ubiquitous in data-flow analysis and SSA renaming. They are also used extensively during the parsing phase for temporary string storage.
- **Lifecycle:** Freeing is region-based; releasing an obstack or a specific point within it (`obstack_free`) invalidates all subsequently allocated objects, making it ideal for phase-based cleanup.
- **Source:** Defined in `include/obstack.h` and heavily utilized in `bitmap.cc`.

B. Object Allocators (Pools)

Modern GCC wraps the legacy `alloc_pool` logic in C++ templates to provide type-safe, fixed-size allocation.

- **Use Case:** High-frequency, fixed-size objects required for specific graph algorithms, such as `et_node` in dominance calculations or specific SSA update records.
- **Mechanism:** The `object_allocator<T>` template class (defined in `alloc-pool.h`) maintains a specialized free list for type `T`. This ensures $O(1)$ allocation/deallocation and maximizes cache locality by grouping identical objects.
- **Source:** `alloc-pool.h` and `alloc-pool.cc`.

3. Memory Ownership Transition: FE → ME → BE

The ownership model shifts as the compilation pipeline progresses, dictating which allocator is dominant.

Front End (FE) Phase

- **Activity:** Lexing and Parsing (C++, Fortran, Rust, etc.).
- **Memory Behavior:** The Front End constructs the `tree` IR (GGC-managed). However, the parsing process relies heavily on **Obstacks** for scope tracking and token buffering.

- **Transition:** Upon parse completion, temporary obstacks are released. The resulting `tree` root is preserved in GGC memory, serving as the interface to the Middle End.

Middle End (ME) Phase

- **Activity:** GIMPLE Lowering, SSA Construction, Optimization Passes.
- **Memory Behavior:** The `tree` IR is lowered to `gimple` (GGC-managed). The *Pass Manager* orchestrates the lifetime of analysis data. Passes typically instantiate local `object_allocator` pools or `bitmap_obstack`s for dominance frontiers, liveness sets, or loop structures.
- **Transition:** At the end of a pass (e.g., `pass_impl::execute`), local pools are destructed. `ggc_collect()` may be triggered by the `TODO_ggc_collect` flag to reclaim unreachable `gimple` statements (e.g., after Dead Code Elimination).

Back End (BE) Phase

- **Activity:** RTL Expansion, Register Allocation, Code Gen.
- **Memory Behavior:** GIMPLE transforms into `rtx` (GGC-managed). The Register Allocator (IRA/LRA) creates significant memory pressure, utilizing specialized allocators within `ira-build.cc` and `lra.cc` (often custom pools) to manage interference graphs.
- **Finalization:** Post-assembly generation, GGC roots are cleared. In JIT contexts (`libgccjit`), the context is torn down, releasing all associated GGC pages and pools.

Deep Analysis: Obstack Memory Management in GCC (v13+)

The **Obstack** (Object Stack) is a foundational memory management mechanism in GCC provided by `libiberty`. Despite the codebase's transition to C++, Obstacks remain the critical infrastructure for high-performance, region-based memory allocation where object lifetimes are strictly nested or phase-bound.

1. Internal Mechanism: The Bump Pointer Strategy

The Obstack is designed to eliminate the overhead of `malloc`/`free` for individual small objects. It implements a "fast path" allocation strategy often referred to as **Bump Pointer Allocation** or **Linear Allocation**.

- **The Chunk Structure:** An Obstack maintains a linked list of large memory blocks called "chunks" (typically 4KB or larger). It tracks three critical pointers within the current chunk:
    1. `chunk_limit`: The end of the currently allocated physical memory.
    2. `object_base`: The start of the object currently being constructed.

3. `next_free`: The "bump pointer" indicating the next available byte.

- **Allocation ($O(1)$):** When a new object of size N is requested:
  1. **Check:** The allocator calculates `next_free + N`.
  2. **Fast Path:** If the result is ≤ `chunk_limit`, the allocation is a simple pointer increment. The old `next_free` is returned as the object address, and `next_free` is updated. This involves zero search overhead, zero fragmentation checks, and is strictly $O(1)$.
  3. **Slow Path:** If the request exceeds the remaining space, the `libiberty` backend allocates a new chunk (via `xmalloc`), links it to the previous chunk, and satisfies the request from the new block.

2. Modern Usage & API in the C++ Codebase

While `obstack` is a legacy C structure, it is deeply integrated into the modern C++ files (`.cc`).

- **Direct Macro Usage:** The standard macros defined in `include/obstack.h`—specifically `obstack_alloc`, `obstack_grow`, and `obstack_finish`—are heavily used directly within C++ source files.
  - *Modern C++ Context:* You will frequently see these macros in `cp/parser.cc` (the C++ Front End parser) and `c-family/c-common.cc`. The macros operate on the raw `struct obstack`.
- **The `bitmap_obstack` Wrapper:** One of the most critical modern applications is within the Data Flow Analysis (DFA) subsystem.
  - **Definition:** In `bitmap.h` and `bitmap.cc`, GCC defines `struct bitmap_obstack`. This structure wraps a standard `obstack` specifically for allocating `bitmap_element` nodes (linked list nodes representing bits).
  - **Efficiency:** Instead of allocating a `bitmap_element` (approx 24-32 bytes) via `new` or `malloc`, the bitmapping subsystem calls `bitmap_alloc`. This function fetches raw memory from the wrapped obstack. This drastically improves cache locality, as sequential bitmap elements are packed contiguously in memory, which is vital for traversing liveness sets or dominance frontiers.

3. Use Case: The Front End (Lexing and Parsing)

The Front End (FE) utilizes Obstacks to solve the "Unknown Size" problem during Lexing.

- **String Interning & Tokenization:** When the lexer encounters a string literal or identifier, it does not initially know the length. Using `std::string` would imply repeated reallocations and copies.

- **Mechanism:** The FE uses `obstack_1grow` (adds 1 byte) to push characters onto the "current object" as they are read. The `next_free` pointer simply advances.
- **Finalization:** Once the closing quote or delimiter is found, `obstack_finish` is called. This function "seals" the object, returns the pointer to `object_base`, and updates `object_base` to equal `next_free`.
- **Performance:** This allows constructing strings in-place with zero intermediate copying.

4. The "Freeing" Logic: Scope-Based Reclaiming

Obstacks enforce a strict LIFO (Last-In, First-Out) reclamation policy, which aligns perfectly with compiler scoping rules.

- **Mechanism (`obstack_free`):** The function `obstack_free(obstack_ptr, object_ptr)` does not strictly "free" the object pointed to by `object_ptr`. Instead, it resets the `next_free` pointer of the obstack *back* to `object_ptr`.
- **Implication:** This effectively frees `object_ptr` **and every object allocated after it**.
- **Compilation Scope Example:**
    1. **Enter Scope:** The compiler records the current state of the obstack (e.g., `void *mark = obstack_alloc(obs, 0)`).
    2. **Process Scope:** It allocates dozens of temporary nodes, strings, and lists on the obstack.
    3. **Exit Scope:** It calls `obstack_free(obs, mark)`.
    - **Result:** The "bump pointer" snaps back to the `mark`. All temporary memory is instantly invalidated without iterating over the objects. This is significantly faster than destructing a `std::vector` of pointers.

The GGC Subsystem: Managing Persistent IR in Modern GCC

The **GGC (GCC Garbage Collector)** is the dedicated memory manager for the compiler's long-lived internal state. It is specifically engineered to handle the complex, graph-heavy data structures of the Intermediate Representation (IR), such as `tree` nodes and `rtx` objects, which persist across multiple optimization passes.

1. The "Roots" Problem and `gengtype`

A fundamental challenge in implementing garbage collection in C++ is the lack of native reflection; the runtime system does not automatically know which fields in a struct are pointers that must be traced.

- **The Solution (`GTY` Markers):** GCC solves this via a custom markup system. Developers annotate source code structures with `GTY(())` ("Garbage Type") markers.
    - *Example:* `struct GTY(()) tree_base { ... }`
- **The Generator (`gengtype`):** During the build process, a specialized tool called `gengtype` parses the source code. It identifies all `GTY`-marked types and global variables (roots).
- **Generated Reflection:** For every source file using GGC (e.g., `tree.cc`), `gengtype` generates a corresponding header file (e.g., `gt-tree.h`). These files contain generated functions (typically named `gt_ggc_mx_*`) that know exactly the layout of the structures and how to recursively "mark" every pointer contained within them. This provides the precise object graph traversal required for the collector.

2. Trade-offs: Why a Custom GC?

GCC avoids standard memory management paradigms in favor of GGC for two primary reasons related to compiler workloads.

- **vs. Smart Pointers (`std::shared_ptr`):**
    - **Cycles:** The compiler's IR (Control Flow Graphs, SSA use-def chains) is inherently cyclic. Standard reference counting (`shared_ptr`) leaks memory in cyclic graphs unless complex `weak_ptr` strategies are strictly enforced, which is impractical for a codebase of this scale.
    - **Performance:** The atomic increments/decrements required by thread-safe smart pointers impose overhead. Even non-atomic ref-counting is expensive given the massive number of pointer mutations during optimization.
- **vs. Conservative GC (Boehm-GC):**
    - **Precision:** Conservative collectors scan the stack/heap and guess what looks like a pointer. This can lead to "false retention" (integers looking like addresses). GGC is a **precise** collector; thanks to `gengtype`, it visits *only* valid pointers.
    - **PCH Support:** GGC is tightly coupled with Precompiled Headers (PCH). Because GGC manages the pages, it can serialize the entire memory state (the heap) to disk and `mmap` it back in for fast compiler startup, a feat difficult to achieve with generic allocators.

3. Allocation Mechanics

- **Allocation (`make_node`):** The creation of IR nodes is centralized. For example, in `tree.cc`, the function `make_node` (and its variations) is the primary factory.
    - It determines the size of the tree node based on its code (e.g., `PLUS_EXPR`).

    - It calls `ggc_alloc<tree_node>(size)`, delegating to the inline template in `ggc.h`.

- **Collection (`ggc_collect`):** This function triggers the Mark-and-Sweep cycle.

    1. **Mark:** It iterates over all registered global roots (annotated with `GTY` and collected by `gengtype`). It recursively calls the generated `gt_ggc_mx` functions to set a "marked" bit on reachable objects.
    2. **Sweep:** It scans the object pages. Any object not marked is returned to the free list.

4. Paging Strategy: `ggc-page.cc`

To combat fragmentation—a significantly high risk when allocating millions of small nodes—GGC employs a segregated free list allocator organized by "Orders."

- **Orders (Size Classes):** `ggc-page.cc` divides the heap into pages (typically 4KB or 64KB). Each page is exclusively dedicated to objects of a specific size class (an "order").
- **Mechanism:**
    - If the compiler needs a 32-byte object, it asks for memory from a "32-byte order" page.
    - Because all objects on that page are the same size, there is **no external fragmentation** within the page.
    - Freed objects are simply added to a singly linked free list maintained within the page descriptor.
- **Benefit:** This is highly optimized for compilers, where millions of identical structures (like `tree_common` or `gimple` statements) are allocated.

Allocation Strategies for GIMPLE and RTL

While `tree` nodes represent the high-level semantic structure of the program, the Middle End and Back End of GCC rely on more specialized Intermediate Representations (IR): **GIMPLE** and **RTL**. Both utilize the GGC subsystem, but they employ distinct allocation strategies optimized for their specific structural requirements.

1. The Middle End: GIMPLE Tuples

In modern GCC, the Middle End operates on "GIMPLE Tuples". GIMPLE (GNU SIMPLE) is a simplified subset of the C language derived from the generic tree structure. Unlike the tree representation, GIMPLE statements are "flat" (mostly 3-address code) and are stored as variable-sized structures designed to be more memory-efficient and cache-friendly than the pointer-heavy trees they replace.

- **Structure:** GIMPLE statements inherit from the base `struct gimple`. They are not uniform; a `GIMPLE_ASSIGN` has a different memory layout and size than a `GIMPLE_COND`.
- **Source File:** `gcc/gimple.cc`
- **Allocation Mechanism (`gimple_alloc`):** The core allocation routine is `gimple_alloc`. It calculates the precise size required for the statement, including the "op" (operand) slots which are allocated contiguously with the statement header to reduce cache misses.

```
/* Pseudo-code logic from gcc/gimple.cc */
gimple *stmt = ggc_alloc_cleared_gimple_statement_stat (size,
pass_defined_mem_stat);
```

- **GGC Integration:** GIMPLE relies strictly on GGC. The `gimple_alloc` function delegates to the standard GGC template allocators. This ensures that when a GIMPLE statement becomes dead (e.g., removed by Dead Code Elimination), the Garbage Collector can reclaim the memory during the next `ggc_collect()` cycle without manual reference counting.

2. The Back End: RTL (Register Transfer Language)

As the compiler transitions to the Back End, it lowers GIMPLE to RTL. RTL nodes (`rtx`) represent the target machine instructions and are extremely low-level.

- **Structure:** RTL nodes are defined by `struct rtx_def`. Similar to GIMPLE, they are polymorphic; an `rtx` representing a register (`REG`) is smaller than one representing a parallel instruction set (`PARALLEL`).
- **Source File:** `gcc/rtl.cc`
- **Allocation Mechanism (`rtx_alloc`):** The fundamental allocator is `rtx_alloc` (often wrapped by `gen_rtx_*` functions in `emit-rtl.cc`).
    - It determines the size of the RTL node based on its `code` (e.g., `PLUS`, `MEM`, `SET`) using the `rtx_code_size` table.
    - It calls `ggc_alloc_rtx_def_stat` to acquire memory.
- **GGC Integration:** RTL generation creates immense memory pressure. Because `rtx` nodes are often temporary (created and discarded during instruction combination or peephole optimization), relying on GGC is crucial. The `rtx_def` structure is marked with `GTY(())`, allowing the collector to trace pointers from instruction chains into the RTL graph.

Per-Pass Memory Management (Pools & Bitmaps)

While the GGC subsystem manages the persistent "spine" of the compilation (the AST/IR), the bulk of the compiler's work occurs during optimization passes (e.g., SSA construction, Loop Invariant Motion). These passes generate massive quantities of auxiliary data—interference graphs, dominance frontiers, and constraint sets—that are extremely short-lived.

1. The Problem: Transient High-Volume Data

Optimization algorithms frequently exhibit a "bursty" memory profile. A pass like **Tree-SSA** might allocate hundreds of thousands of small graph nodes or bitmaps to calculate data flow.

- **Inefficiency of GGC:** If these transient objects were allocated via GGC, they would trigger frequent, expensive garbage collection cycles. Furthermore, because these objects die at the end of the pass, the "Mark" phase of the GC would waste significant time tracing a graph that is about to be destroyed entirely.
- **The Requirement:** A mechanism for rapid allocation and bulk deallocation that bypasses the global garbage collector.

2. The Solution: Specialized Allocators

To address this, modern GCC (v13+) employs specialized C++ allocators tailored for pass-local storage.

A. Allocation Pools (`object_allocator`)

For fixed-size structures used heavily within a specific pass, GCC uses the `object_allocator` template.

- **Source:** `gcc/alloc-pool.h`
- **Mechanism:** This template wraps the legacy `alloc_pool` logic. It maintains a linked list of free blocks of size `sizeof(T)`.
- **Performance:** Allocation is strictly $O(1)$ (popping from the free list). When the pool is destroyed, it releases all underlying pages to the system immediately, without a sweep phase. It provides excellent cache locality because objects of type `T` are packed contiguously in memory pages.

B. Bitmap Obstacks (`bitmap_obstack`)

Data flow analysis relies heavily on bitmaps to represent sets (e.g., "registers live at instruction X").

- **Source:** `gcc/bitmap.h`
- **Mechanism:** A `bitmap` in GCC is a linked list of elements. To prevent fragmentation, passes initialize a `bitmap_obstack`. All linked list nodes for bitmaps created in that context are allocated from this linear memory region.
- **Scope:** When the pass finishes, releasing the single `bitmap_obstack` instantly frees every node of every bitmap used during the analysis.

3. Case Study: The Tree-SSA Pass

The transition into Static Single Assignment (SSA) form demonstrates this hybrid model perfectly. The logic for renaming variables into SSA versions uses transient data structures that are scoped strictly to the renaming phase.

- **Source File:** `gcc/tree-into-ssa.cc`
- **Lifecycle Management:** The memory management is explicitly tied to the SSA builder pass (specifically within `pass_build_ssa::execute`).
- **Initialization:** Before the renaming logic begins, the function `init_ssa_renamer` is called. It establishes the local allocation context:

```
/* From tree-into-ssa.cc, inside init_ssa_renamer */
bitmap_obstack_initialize (&update_ssa_obstack);
```

This prepares update_ssa_obstack, a region of memory dedicated entirely to bitmaps required during the renaming (such as tracking old vs. new variable versions).

- **Usage:** Throughout the renaming process, temporary bitmaps are allocated. These bitmaps act as sets to track variable definitions across basic blocks. Because they are backed by the update_ssa_obstack, these allocations are fast pointer bumps.
- **Termination:** Once the SSA form is built, the function `fini_ssa_renamer` is invoked. This performs the critical bulk deallocation

```
/* From tree-into-ssa.cc, inside fini_ssa_renamer */
bitmap_obstack_release (&update_ssa_obstack);
```

With this single call, every temporary bitmap node created during the entire SSA construction pass is invalidated. The memory is reclaimed instantly without requiring the global Garbage Collector to scan it.

4. Trade-off Analysis: Pools vs. GGC

GCC's "Hybrid" memory model is a deliberate architectural choice driven by specific trade-offs.

Feature	Allocation Pools / Obstacks	GGC (Garbage Collector)
Lifetime Model	**Explicit/Scoped:** Data lives only as long as the pass.	**Indeterminate:** Data lives until it is no longer reachable.
Allocation Cost	**Ultra-Low ($O(1)$):** Simple pointer bump or free-list pop.	**Medium:** Requires size-class lookup and potential page management.
Deallocation Cost	**Zero (Amortized):** Bulk release of entire pages/regions.	**High:** Requires scanning the entire object graph (Mark & Sweep).
Cache Locality	**Excellent:** Related objects (e.g., constraints) are packed tightly.	**Variable:** Objects may be scattered across pages over time.
Safety	**Low:** Dangling pointers are possible if references escape the pass.	**High:** Safe against dangling pointers (auto-reclaimed).

Design Rationale:
GCC uses GGC for the **IR (Intermediate Representation)** because IR nodes form a complex, cyclic graph where ownership is shared across many passes. It uses **Pools/Obstacks** for **Analysis Data** because this data is strictly hierarchical and local to the algorithm computing it. Mixing these models prevents the "GC Pause" problem from dominating compilation time.

Conclusion: The Hybrid Memory Architecture of GCC

Our analysis of the GNU Compiler Collection (v13+) reveals that GCC does not rely on a singular memory management strategy. Instead, it employs a sophisticated, three-tiered hybrid architecture tailored to the distinct distinct phases of the compilation pipeline. This design is not accidental; it is a necessary response to the conflicting requirements of compilation: the need for persistent, cyclic graph structures versus the need for high-frequency, ephemeral analysis data.

1. Summary of the Architecture

Through our source code inspection, we have identified three distinct layers of memory management:

1. **Front End (Lexing/Parsing): Region-Based Allocation.**
   During the initial translation of source code, GCC leverages **Obstacks** (via `libiberty`). This aligns with the stack-like nature of parsing. As scopes are entered and exited, memory is allocated linearly and freed in bulk, minimizing overhead for temporary strings and tokens.
2. **Persistent Storage (IR Spine): The GGC Subsystem.**
   The core Intermediate Representations—**Tree** (High-Level), **GIMPLE** (Middle-End), and **RTL** (Back-End)—are managed by the **GGC (Garbage Collector)**. This layer uses `gengtype` for reflection and Mark-and-Sweep for reclamation, ensuring that complex, cyclic structures (like Control Flow Graphs) are managed safely without manual intervention.
3. **Optimization Passes (Transient Data): Pools and Bitmaps.**
   The computational heavy lifting (SSA, Liveness, Scheduling) generates massive amounts of temporary data. Here, GCC bypasses the GGC in favor of **Pools** (`object_allocator`) and **Bitmap Obstacks**. This isolates high-frequency allocation churn from the long-lived heap, preventing fragmentation and GC pauses.

2. Rationale: Why Not Standard Allocators?

A key finding of this research is why GCC rejects standard C++ paradigms (like `new`/`delete` or `std::shared_ptr`) in favor of this complex custom implementation.

- **The "Graph" Problem vs. `std::shared_ptr`:**
  Compiler IRs are inherently cyclic (e.g., a Loop header pointing to a latch block, which points back to the header). Standard reference counting (`std::shared_ptr`) fails in cyclic graphs, leading to memory leaks unless complex `weak_ptr` schemes are strictly enforced. The GGC Mark-and-Sweep algorithm naturally handles these cycles, guaranteeing correctness.
- **The "Fragment" Problem vs. `malloc`:**
  Compilers allocate millions of tiny objects (16-40 bytes). Using the system `malloc` for these would incur significant per-object overhead (headers) and severe fragmentation. GCC's **Pool Allocators** and **Page Orders** eliminate this overhead by packing identical objects tightly, achieving near-perfect memory density.

3. Final Verdict: Performance and Correctness

The efficacy of GCC's memory management lies in its **specialization**.

- **Correctness:** By delegating the lifetime management of the complex IR graph to GGC, GCC minimizes the class of bugs related to dangling pointers and memory leaks in the persistent state.
- **Performance:** By utilizing `object_allocator` and `bitmap_obstack` for pass-local data, GCC achieves $O(1)$ allocation and deallocation performance. This ensures that

the heavy memory churn of optimization passes (allocating millions of temporary constraints) does not degrade the performance of the global heap or trigger unnecessary garbage collection cycles.

In conclusion, GCC's memory strategy is a highly tuned system where **GGC provides the safety mechanism** for the compiler's long-term memory, while **Pools and Obstacks provide the speed** required for its computational algorithms.