

class diagram reverse engineering c

Published: September 27, 2026 | Index ID: #-

Class Diagram Reverse Engineering in C: A Practical Guide

When you inherit legacy C code or work on a large codebase without any accompanying documentation, the first step to understanding the system is to uncover its structure. One of the most effective ways to do this is through reverse engineering of class diagrams, even though C is not an object oriented language in the same sense as C++ or Java. By mapping structs, function pointers, and module boundaries into a UML style class diagram, you gain a visual representation of relationships, dependencies, and responsibilities that would otherwise remain hidden behind hundreds of lines of code.

In this article we will walk through the entire process of reverse engineering class diagrams from C code. We'll cover the motivations, the tools, the methodology, and how to keep the output maintainable and useful for both developers and stakeholders. The content is written with SEO best practices in mind, so you'll find relevant keywords naturally integrated without compromising readability.

Why Reverse Engineer Class Diagrams for C?

Bridging the Gap Between Code and Architecture

Class diagrams provide a high level view of a system's architecture. They illustrate how different parts of the code interact, what responsibilities each component holds, and where potential bottlenecks or coupling issues exist. For C projects, this is especially valuable because:

- Structs and function pointers can be used to model objects and methods.
- Header files often define interfaces that are not obvious from source files alone.
- Legacy code may have evolved organically, leading to hidden dependencies.

Improving Maintenance and Onboarding

New developers can quickly grasp the overall design without diving into every function. Maintenance tasks, such as refactoring or adding features, become more systematic when you have a diagram that outlines responsibilities and interactions.

Supporting Compliance and Auditing

Many industries require documentation that demonstrates how data flows through the system. Reverse engineered diagrams can serve as evidence during audits or when proving adherence to coding standards.

Key Concepts of Class Diagrams in C

Structs as Classes

In C, struct is the primary way to group related data. By treating a struct as a class, you can attach methods (functions that accept a pointer to the struct) and represent inheritance through composition or embedding.

Function Pointers and Polymorphism

Polymorphic behavior in C is achieved by embedding function pointers inside structs. Each instance can point to different implementations, mimicking virtual functions.

Modules and Namespaces

Header files (.h) act as interfaces, while source files (.c) contain implementations. Grouping related structs and functions into modules is analogous to namespaces in C++.

Preparing Your Environment

Choose the Right Tool

There are several tools capable of parsing C code and generating UML diagrams. Some popular options include:

- Doxygen – With the PlantUML extension, it can produce class diagrams from annotated source.
- Graphviz – Works well when combined with custom scripts to convert code into DOT language.
- SourceTrail – Offers visual navigation and can export diagrams.
- Understand – Provides a comprehensive reverse engineering suite, including class diagram export.

For this guide we'll focus on Doxygen because it's free, widely adopted, and integrates easily into most build systems.

Set Up Doxygen

1. Install Doxygen from the official website or via your package manager.
2. Create a configuration file: `doxygen -g`.
3. Edit Doxyfile to enable diagram generation:

```
EXTRACT_ALL = YES
```

```
4. HAVE_DOT = YES
```

```
5. DOT_GRAPH_MAX_NODES = 200
```

```
6. GENERATE_TREEVIEW = YES
```

Annotate Your Code

While Doxygen can infer many relationships, explicit annotations improve clarity. Use tags such as `/// @class`, `/// @struct`, `/// @memberof`, and `/// @brief` to describe structs, functions, and their associations.

Extracting Structural Information

Parsing Structs and Typedefs

Start by scanning all header files for struct and typedef struct declarations. Record:

- Struct name.
- Member variables and types.
- Nested structs or arrays.

These will become the nodes in your class diagram.

Identifying Functions and Their Signatures

Next, parse all .c files to find function definitions. For each function, note:

- Return type.
- Parameter list.
- Whether the first parameter is a pointer to a struct (common convention for methods).

Functions that accept a struct pointer as the first argument are candidates for being methods of that struct.

Mapping Function Pointers

Look for struct members that are function pointers. These represent virtual methods. Document the pointer's signature and the struct it belongs to.

Detecting Module Boundaries

Group related structs and functions by the header file that declares them. This grouping will become a package or namespace in the diagram.

Constructing the Diagram

Using Doxygen's PlantUML Integration

Once you have all structural data, you can feed it into PlantUML via Doxygen. Add the following to your Doxyfile:

- PLANTUML_JAR_PATH = /path/to/plantuml.jar
- PLANTUML_CFG_FILE = plantuml.cfg

Create a plantuml.cfg file with:

```
@startuml
!define      AWSPUML      https://raw.githubusercontent.com/aws-labs/aws-plantuml/v14.0/Advanced/
AWSCommon.puml
skinparam backgroundColor #FFFFFF
skinparam nodeMargin 30
skinparam nodePadding 15
skinparam roundCorner 20
@enduml
```

Now, when Doxygen processes the code, it will generate PlantUML files that can be rendered into PNG or SVG images.

Manually Refining the Diagram

Automated tools often produce cluttered diagrams for complex codebases. To improve readability:

- Use package blocks to group related classes.
- Show only public interfaces (omit private members).
- Represent inheritance or composition with appropriate arrows.
- Use stereotypes (<<interface>>, <<abstract>>) to clarify roles.
- Apply color coding to differentiate modules.

```
@startuml
package "Networking" {
class Socket {
+int fd
+connect()
+send()
+recv()
}
```

```

}

class TCP : Socket {
    +int timeout
}
}

package "Storage" {
    class FileHandle {
        +char* path
        +open()
        +close()
    }
}

Socket ..&gt; TCP : inherits
Socket ..&gt; FileHandle : uses
@enduml

```

Common Challenges and How to Overcome Them

1. Handling Complex Macro Definitions

Macros can obfuscate function prototypes or struct definitions. Use preprocessor output (gcc -E) to view expanded code, or configure Doxygen to preprocess files.

2. Dealing with Conditional Compilation

Code guarded by `#ifdef` or `#if defined` may lead to incomplete diagrams. Consider creating multiple diagrams for different build configurations or documenting the conditions explicitly.

3. Managing Large Codebases

For very large systems, generate modular diagrams and provide an index page. Use `HAVE_DOT` and `DOT_MAX_NODES` to control diagram size. Split the diagram into layers: high-level architecture, subsystem, and module views.

4. Capturing Runtime Polymorphism

Some polymorphic behavior is only evident at runtime (e.g., dynamic dispatch via function pointers set in initialization functions). Add notes or annotations indicating that certain relationships are dynamic.

Best Practices for Maintaining Diagram Accuracy

Automate Diagram Generation

Integrate Doxygen runs into your CI pipeline so that diagrams are regenerated whenever code changes. This ensures the documentation stays in sync.

Document Assumptions

When reverse engineering, you may need to make assumptions (e.g., which struct a function belongs to). Record

these assumptions in a dedicated section of the documentation.

Use Version Control for Diagrams

Store generated diagram files in a dedicated directory within your repository. This allows tracking changes over time and facilitates review.

Keep Diagrams Focused

Don't overwhelm readers with every detail. Use abstraction levels: start with a high level overview and provide drill down links to more detailed views.

Extending Beyond Class Diagrams

Sequence Diagrams for Function Flow

While class diagrams show structure, sequence diagrams reveal runtime interactions. Tools like SequenceDiagram can generate these from C code, especially when combined with logging or instrumentation.

Component Diagrams for Deployment

When the C application interacts with external services or hardware, a component diagram can illustrate how components are packaged and deployed.

Case Study: Reverse Engineering a Legacy Sensor Driver

Let's walk through a practical example: a legacy sensor driver written in C that interfaces with multiple hardware modules.

Step 1: Identify Core Structs

The header sensor.h defines:

```
typedef struct {  
    int id;  
    char name[32];  
    void (*init)(struct sensor_t*);  
    void (*read)(struct sensor_t*, float*);  
    void (*shutdown)(struct sensor_t*);  
} sensor_t;
```

Here, sensor_t acts as an abstract class with virtual methods.

Step 2: Map Implementations

In temperature.c, we find:

```
static void temp_init(sensor_t* s) { /* ... */ }  
static void temp_read(sensor_t* s, float* val) { /* ... */ }  
static void temp_shutdown(sensor_t* s) { /* ... */ }
```

```
void create_temperature_sensor(sensor_t* s) {
    s->id = 1;
    strcpy(s->name, "TempSensor");
    s->init = temp_init;
    s->read = temp_read;
    s->shutdown = temp_shutdown;
}
```

Similarly, humidity.c provides a different set of functions.

Step 3: Generate Diagram

Using Doxygen + PlantUML, we create a diagram that shows sensor_t as an interface, with two concrete implementations: TemperatureSensor and HumiditySensor. The diagram also indicates that both implementations share a common init sequence.

Step 4: Validate with Runtime Tests

We write unit tests that instantiate each sensor and verify that the correct function pointers are called. This confirms the accuracy of our diagram.

Conclusion

Reverse engineering class diagrams from C code is a powerful technique for unlocking the hidden architecture of legacy systems. By treating structs as classes, mapping function pointers to methods, and grouping code into logical modules,

Baca Juga (Artikel Terkait)

- [chrysler 62te problems pdf](http://www.bbglobalsolutions.com/chrysler-62te-problems-pdf.pdf)

URL: <http://www.bbglobalsolutions.com/chrysler-62te-problems-pdf.pdf>

- [chrysler 300c repair engine](http://www.bbglobalsolutions.com/chrysler-300c-repair-engine.pdf)

URL: <http://www.bbglobalsolutions.com/chrysler-300c-repair-engine.pdf>

- [chrysler 300 service](http://www.bbglobalsolutions.com/chrysler-300-service.pdf)

URL: <http://www.bbglobalsolutions.com/chrysler-300-service.pdf>

- [chrysler 251 industrial engine parts](http://www.bbglobalsolutions.com/chrysler-251-industrial-engine-parts.pdf)

URL: <http://www.bbglobalsolutions.com/chrysler-251-industrial-engine-parts.pdf>

Tags: [#class](#) [#diagram](#) [#reverse](#) [#engineering](#)

