

# An Analysis of Library Usage in the C++ Code Base of Fedora Linux 37

Jiachao Deng and Michael D. Adams

2024-10-29



# Outline

## 1 Introduction

- ▶ Introduction
- ▶ Background
- ▶ Proposed Framework
- ▶ Proposed Analysis Tool
- ▶ Results of Applying Framework
- ▶ Results of Library Usage Analysis

# Challenges of Large-Scale C++ Code Analysis

## 1 Introduction

```
1 #include <iostream>
2
3 int main(int argc,
4         char const *argv[]) {
5
6     std::cout << "Hello, World!\n";
7     return std::cout.flush() ? 0 : 1;
8 }
```

Listing 1: A hello world program in C++

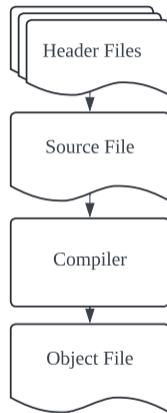


Figure 1: Translating a C++ source file to a binary file.

# Outline

## 2 Background

- ▶ Introduction
- ▶ **Background**
- ▶ Proposed Framework
- ▶ Proposed Analysis Tool
- ▶ Results of Applying Framework
- ▶ Results of Library Usage Analysis

# Software Packaging

## 2 Background

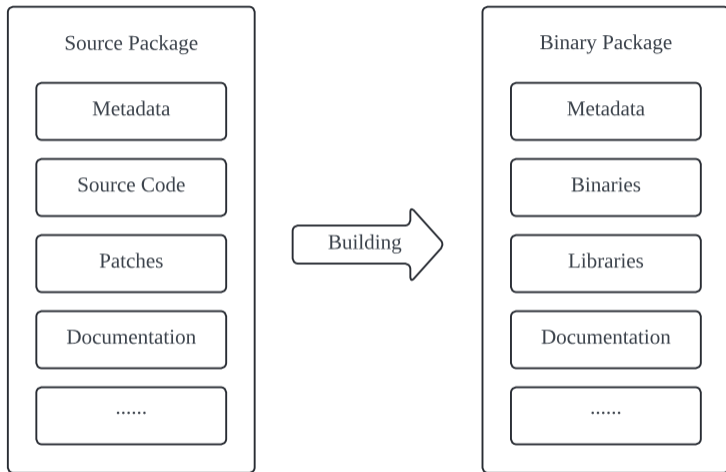


Figure 2: A binary package built from a source package.

```

1 #include <iostream>
2
3 int main(int argc,
4   char const *argv[]) {
5
6   std::cout << "Hello,
7     World!\n";
8   return std::cout.flush() ? 0
9     : 1;
10 }
  
```

Listing 2: A hello world program.

```

Dumping main:
FunctionDecl 0x2954a38 </home/jcdeng/jcdeng-test/lib_usage_analyzer/va
|-ParmVarDecl 0x29547f0 <col:10, col:14> col:14 argc 'int'
|-ParmVarDecl 0x29548e8 <line:4:3, col:20> col:15 argv 'const char **'
`-CompoundStmt 0x2956270 <col:23, line:8:1>
  |-CXXOperatorCallExpr 0x2955e50 <line:6:3, col:16> 'basic_ostream<ch
  | |-ImplicitCastExpr 0x2955e38 <col:13> 'basic_ostream<char, std::ch
  | | `DeclRefExpr 0x2955dc0 <col:13> 'basic_ostream<char, std::char_
rator<<' 'basic_ostream<char, std::char_traits<char>> 8(basic_ostream<
  | |-DeclRefExpr 0x2954b68 <col:3, col:8> 'std::ostream': 'std::basic_
  | `ImplicitCastExpr 0x2955da8 <col:16> 'const char *' <ArrayToPoint
  | `StringLiteral 0x2954b98 <col:16> 'const char[15]' lvalue "Hell
  `ReturnStmt 0x2956260 <line:7:3, col:34>
    `ConditionalOperator 0x2956230 <col:10, col:34> 'int'
      |-ImplicitCastExpr 0x2956218 <col:10, col:26> 'bool' <UserDefine
      | `CXXMemberCallExpr 0x29561f8 <col:10, col:26> 'bool'
      |   `MemberExpr 0x29561c8 <col:10, col:26> '<bound member funct
      |     `ImplicitCastExpr 0x29561a8 <col:10, col:26> 'const std::
      |       `CXXMemberCallExpr 0x29560c0 <col:10, col:26> 'std::bas
      |         `MemberExpr 0x2956090 <col:10, col:20> '<bound member
      |           `DeclRefExpr 0x2956060 <col:10, col:15> 'std::ostre
      `IntegerLiteral 0x2956108 <col:30> 'int' 0
    `IntegerLiteral 0x2956128 <col:34> 'int' 1
  
```

Figure 3: Abstract language tree of the hello world program.

```
1 #include <iostream>
2
3 #ifdef ENABLE_FOO_FEATURE
4 void foo() { std::cout << "Foo feature enabled\n"; }
5 #endif
6
7 int main() {
8     #ifdef ENABLE_FOO_FEATURE
9     foo();
10    #else
11    std::cout << "Foo feature not available.\n";
12    #endif
13    return 0;
14 }
```

Listing 3: Code blocks toggled by a compiler flag.

# Build Systems and Build Wrapper

## 2 Background

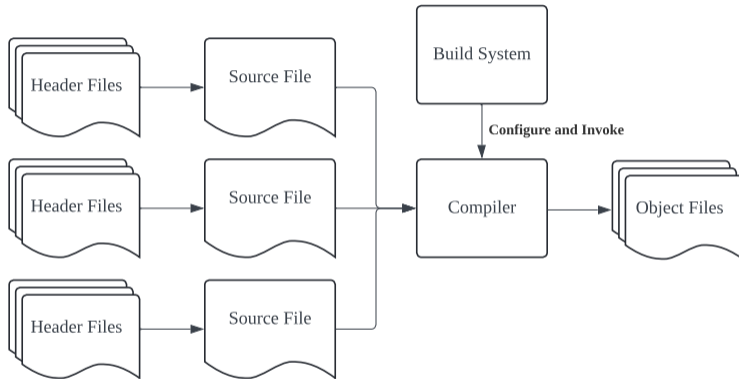


Figure 4: A build system orchestrates the compilation of source files.



# The bear Build Wrapper

## 2 Background

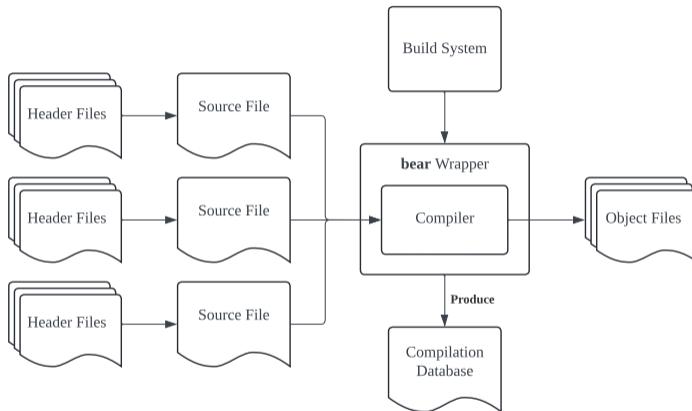


Figure 5: The bear build wrapper captures compiler flags and outputs a compilation database.

# Software Dependency

## 2 Background

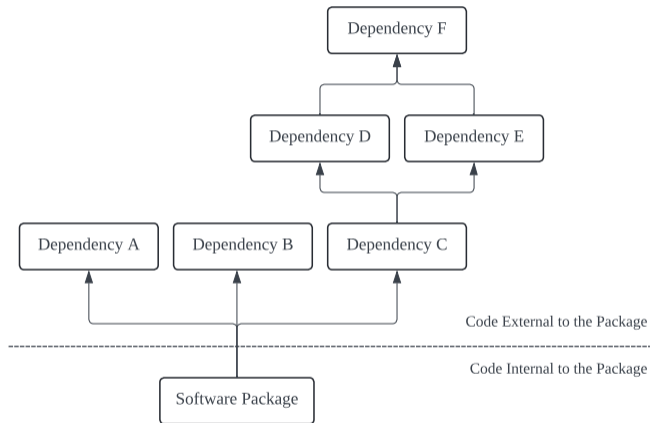


Figure 6: Software dependencies must be installed before building a package.

# Software Repositories and the `dnf` Package Manager

## 2 Background

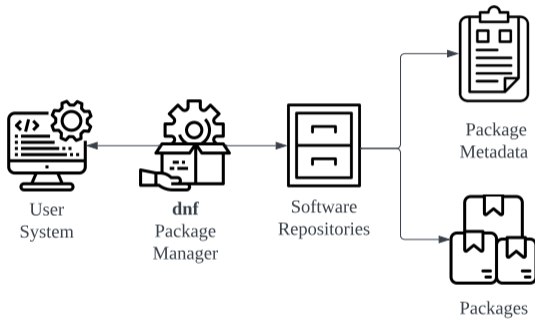


Figure 7: Obtain software packages from software repositories using the `dnf` package manager.

# Outline

## 3 Proposed Framework

- ▶ Introduction
- ▶ Background
- ▶ **Proposed Framework**
- ▶ Proposed Analysis Tool
- ▶ Results of Applying Framework
- ▶ Results of Library Usage Analysis

### Software package selection:

- Source packages that depend on the `gcc-c++` package.
- Source packages that depend on the `clang` package and have source files with C++ file extensions.

Information that must be prepared for each source package:

- Any header files internal/external to the package that are included by the one or more C++ source files in the package.
- All C++ source files in the package that would be compiled during the build process, including generated source files.
- Compiler flags used to compile each C++ source file in the package.

# Package Processing Steps

## 3 Proposed Framework

The framework has four major steps to process a source package:

- Prebuild. Before building a package, install all dependencies of the package and extract package contents.
- Build. Build the package and capture compiler flags using a build wrapper.
- Postbuild. Clean up unnecessary files for source code analysis.
- Analysis. Apply the analysis tool to package source code.

# Outline

## 4 Proposed Analysis Tool

- ▶ Introduction
- ▶ Background
- ▶ Proposed Framework
- ▶ Proposed Analysis Tool**
- ▶ Results of Applying Framework
- ▶ Results of Library Usage Analysis

# Definition of the Library Usage

## 4 Proposed Analysis Tool

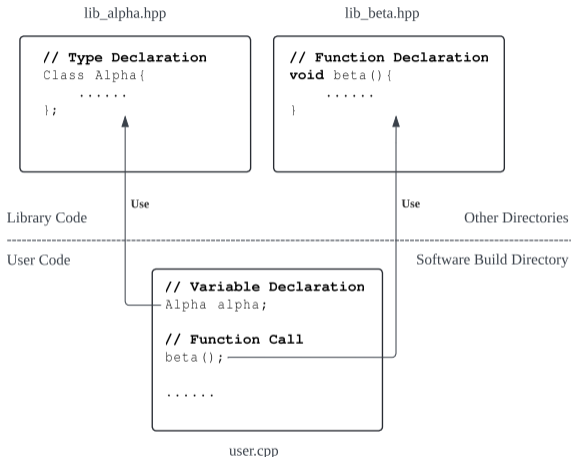


Figure 8: A package source file uses entities declared in library locations.



# Examples of Library Usage

## 4 Proposed Analysis Tool

```
1  #include <iostream>
2
3  int main(int argc,
4  char const *argv[]) {
5
6      std::cout << "Hello, World!\n";
7      return std::cout.flush() ? 0 : 1;
8  }
```

Listing 4: Library usages in the hello world program.

# Outline

## 5 Results of Applying Framework

- ▶ Introduction
- ▶ Background
- ▶ Proposed Framework
- ▶ Proposed Analysis Tool
- ▶ Results of Applying Framework**
- ▶ Results of Library Usage Analysis

# Results of Package Processing

## 5 Results of Applying Framework

Table 1: Package processing outcomes

Outcome	Number of Packages	% of Packages
prebuild failed	17	0.57
build completed but failed	196	6.58
build failed due to hanging	31	1.04
postbuild failed	11	0.37
[0%, 50%) analysis success	84	2.82
[50%, 100%) analysis success	372	12.48
100% analysis success	2269	76.14
total	2980	100.00

# Source Package Sizes

## 5 Results of Applying Framework

**Table 2:** Distribution of processed source code size for packages with at least one successfully analyzed C++ source file

Lines of Code	Number of Packages	% of Packages
[1, 10)	7	0.29
[10, 100)	6	0.25
[100, 1000)	201	8.45
[1000, 10000)	645	27.11
[10000, 100000)	1018	42.79
[100000, 1000000)	448	18.83
[1000000, 10000000)	49	2.06
[10000000, 100000000)	5	0.21
<b>total</b>	<b>2379</b>	<b>100.00</b>

# Largest Source Packages Processed

## 5 Results of Applying Framework

Table 3: 10 packages with the most lines of C++ code

Name of Source Package	Number of Lines
libint2-0:2.6.0	28724722
swig-0:4.0.2	14987941
godot-0:3.4.5	14451968
qt5-qtwebengine-0:5.15.10	12959278
cross-gcc-0:12.1.1	10843595
nextpnr-0:1-12.20220912gitf1349e1	10763584
root-0:6.26.06	7421955
libint-0:1.2.1	7094630
swift-lang-0:5.7	7004290
llvm9.0-0:9.0.1	6662584

# Outline

## 6 Results of Library Usage Analysis

- ▶ Introduction
- ▶ Background
- ▶ Proposed Framework
- ▶ Proposed Analysis Tool
- ▶ Results of Applying Framework
- ▶ **Results of Library Usage Analysis**

Table 4: Most frequently used STD algorithms

Function Name	Appear in % of Packages
<code>min</code>	40.31
<code>max</code>	37.24
<code>sort</code>	34.59
<code>find</code>	29.80
<code>copy</code>	22.36
<code>find_if</code>	20.47
<code>transform</code>	20.39
<code>reverse</code>	13.37
<code>fill</code>	12.90
<code>lower_bound</code>	12.40

Table 5: Least frequently used STD algorithms

Function Name	Appear in % of Packages
<code>remove_copy</code>	0.25
<code>partition_copy</code>	0.21
<code>prev_permutation</code>	0.17
<code>replace_copy_if</code>	0.13
<code>rotate_copy</code>	0.08
<code>replace_copy</code>	0.08
<code>search_n</code>	0.04
<code>is_sorted_until</code>	0.04
<code>is_partitioned</code>	0.04
<code>sample</code>	0.04

# Bounds-Checking When Indexing

## 6 Results of Library Usage Analysis

**Table 6:** Fraction of indexing operations without built-in bounds-checking when using sequential container/view types

Type Name	% of Indexing Operations
<code>array</code>	95.54
<code>basic_string</code>	91.96
<code>vector</code>	91.69
<code>deque</code>	90.91
<code>basic_string_view</code>	87.52

**Table 7:** Fraction of packages using only indexing operations without built-in bounds-checking when using sequential container/view types

Type Name	% of Packages
<code>array</code>	86.80
<code>basic_string_view</code>	85.42
<code>vector</code>	77.20
<code>basic_string</code>	74.57
<code>deque</code>	63.01



# An Analysis of Library Usage in the C++ Code Base of Fedora Linux 37

*Thank you for listening!*