# Beyond Debug Information: Improving Program Reconstruction in LLDB using C++ Modules

Master's Thesis in Computer Science and Engineering

RAPHAEL ISEMANN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2019

# Beyond Debug Information: Improving Program Reconstruction in LLDB using C++ Modules

Raphael Isemann

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Beyond Debug Information: Improving Program Reconstruction in LLDB using C++ Modules

Raphael Isemann



Supervisor: Thomas Sewell, Department of Computer Science and Engineering
Examiner: Magnus Myreen, Department of Computer Science and Engineering

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: The LLVM logo, owned by and royality-free licensed from Apple Inc.

Typeset in LaTeX
Gothenburg, Sweden 2019

Beyond Debug Information: Improving Program Reconstruction in LLDB using C++ Modules

Raphael Isemann
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

# Abstract

Expression evaluation is a core feature of every modern C++ debugger. Still, no C++ debugger currently features an expression evaluator that consistently supports advanced language features such as meta-programming with templates. The underlying problem is that the debugger can often only partially reconstruct the debugged program from the debug information. This thesis presents a solution to this problem by using C++ modules as an additional source of program information. We developed a prototype based on the LLDB debugger that is loading missing program components from the C++ modules used by the program. With this approach, our prototype is able to reliably reconstruct more components than other widely used C++ debuggers such as GDB, Microsoft Visual Studio Debugger and LLDB itself. However, our prototype was slower than LLDB and could only improve program reconstruction for components which are defined in a C++ module.

Keywords: compilers, debuggers, C++ , LLVM, C++ modules, Clang, LLDB

# Acknowledgements

I would like to thank Magnus Myreen and my supervisor Thomas Sewell for giving me a chance to work on this topic. I would also like to thank the LLVM community, especially Frédéric Riss, Adrian Prantl, Shafik Yaghmour, Jim Ingham and Davide Italiano, for all the feedback, explanations, code reviews and support. Finally, I would like to thank my fellow students and my friends for their support during my time at Chalmers and this thesis.

Raphael Isemann, Gothenburg, June 2019

# Contents

# List of Figures

# 1 Introduction

Debuggers are an important tool when developing software. They allow stopping a running program and inspecting its internal state, which can give important insights to developers on why the program behaves the way it does. This makes it easier and faster to resolve bugs and to improve software.

One aspect of debugging is to evaluate expressions inside the target program. This means that a developer can provide a piece of code to the debugger, which will then be executed as if it was part of the debugged program. Expression evaluation allows a user to easily inspect and change the state of the target program.

Even though it is an important feature, debuggers do not always feature a fully functional expression evaluator. One reason for this is that some programming languages make it difficult to implement expression evaluation. For example, in the C++ programming language, expression evaluation is usually complicated by the fact that the running program has already been translated and compiled into machine code. The original source code is sometimes no longer available at this point and needs to be reconstructed by the debugger. This reconstruction process is often error-prone and sometimes even impossible if the compiler has not emitted the necessary debug information into the executable. Even if the original source code is available, parsing and loading it would be too slow to be viable for interactive debugging.

With the following example we try to illustrate the current state of expression evaluation in C++ debuggers. The given program is creating an empty vector and then returning the last element of the vector. As the vector is empty, the program will crash when trying to retrieve the last element which does not exist. To fix this, we try to evaluate the simple expression `v.push_back(1)` when the execution reaches line 7:

```
1 #include <vector>
2
3 int main() {
4   // Creates an empty vector.
5   std::vector<int> v;
6   // We set a break point here and evaluate `v.push_back(1)`.
7   return v.back();
8 }
```

The expected result of our expression is that our vector now has one element and the program will now behave correctly. However, all three major C++ debuggers,

Microsoft's Visual Studio debugger, GDB and LLDB, fail to evaluate the example expression and return an error instead. There are several possible reasons for why the evaluation of the expression fails. The most likely reason in our example is that the function `push_back()` was unused in the original program and therefore not emitted into the executable by the compiler. Because of this, the debugger can not call the compiled function when we later debug the program. As the function body is not part of the debug information, the function can not be reconstructed and compiled by the debugger.

To reconstruct the missing parts of the program that are not in the debug information, we need to find another source of information about the original program. One such source are C++ modules, which are an upcoming technology for efficiently loading C++ program contents. In this thesis we use C++ modules to provide this missing information to the debugger and implement a C++ expression evaluator that can evaluate expressions like the one in our example.

## 1.1 Aim

The goal of the thesis is to improve the C++ expression evaluation in the LLDB debugger by integrating C++ modules into the expression evaluator. After the project is done, LLDB should be able to evaluate new kinds of expressions that were previously rejected with an error.

The following is a list of problematic expressions alongside examples for them in Figure 1.1.

1. Expressions with calls to functions that are inlined by the compiler, either because the compiler optimized it by inlining or because it was explicitly requested by the programmer to inline the function.
2. Expressions with calls to functions that are unused in the program and therefore not compiled and emitted into the executable.
3. Expressions with calls to templated functions that are not called with the same templated parameters as in the program.
4. Expressions with calls to member functions in class templates that are not instantiated at all or not with the same templated parameters as in the program.
5. Expressions referencing alias templates.
6. Expressions referencing macros.

## 1.2 Problem formulation

This thesis aims to answer the following questions:

- Is it feasible to import C++ modules into the expression evaluator of a C++ debugger?

```
1  // 1. Function manually inlined by programmer.
2  // Example expression: f1()
3  inline __attribute__((always_inline)) int f1() { return 1; }
4
5  // 2. Unused function.
6  // Example expression: f2()
7  int f2() { return 1; }
8
9  // 3. Templated function. We call it with type 'int', but we use it
10 // with type 'float' in the expression evaluator.
11 // Example expression: twice(1.0f)
12 template<typename T>
13 T twice(T v) { return v + v; }
14
15 // 4. Member function in a templated class
16 // Example expression: c.f()
17 template<typename T>
18 struct class C {
19   int f() { return 1; }
20 };
21
22 // 5. Alias template.
23 // Example expression: IdAlias<int>(0)
24 template<class T>
25 using IdAlias = T;
26
27 // 6. Macro.
28 // Example expression: TWO
29 #define TWO 2
```

**Figure 1.1:** Examples of declarations and the problematic expressions referencing them.

- What are potential problems when trying to add C++ module support to a C++ debugger?
- Is the debug information commonly emitted by C++ compilers enough to correctly configure the debugger for importing modules?
- What kind of declarations and expressions are better supported with C++ modules?
- How do C++ modules effect the performance of the expression evaluation?

## 1.3 Limitations

The work presented in this thesis is subject to the following limitations:

- Our project has the goal to make major improvements over the current state of the art regarding expression evaluation. However, the project is not about completely solving program reconstruction in C++ programs as this might not be possible.
- There exists a multitude of C++ compilers and therefore also a multitude of C++ module implementations. This project is limited to the C++ modules implementation in Clang and LLDB.
- There also exists several kinds of different debugging formats which are supported by LLDB and Clang. This project is limited to the DWARF debugging format.

## 1.4 Thesis outline

This thesis is separated into 7 chapters. The first chapter gives a brief introduction into the problem of expression evaluation and the motivation for improving it with C++ modules. The second chapter provides background information about the relevant technologies discussed in this thesis. The third chapter describes our approach for integrating C++ into LLDB to improve expression evaluation. The fourth chapter describes the reliability and performance results we got from our finished implementation. The fifth chapter discusses the results and the alternative approaches used by other debuggers. The sixth chapter reviews related work about C++ modules and compares it to our work. The last chapter provides a conclusion of this thesis.

# 2
# Background

This chapter provides an overview of the technologies that are referenced in this thesis: C++ , C++ modules, debuggers, DWARF debug information, and the LLVM-based debugger LLDB which forms the foundation for this project.

## 2.1 The ISO C++ programming language

C++ is a programming language standardized by the International Organization for Standardization (ISO). C++ is a general-purpose and supports multiple programming paradigms such as generic, functional, procedural and object-oriented programming. Originally C++ was an extension of C that added support for object-oriented programming, but has since then developed into its own independent programming language.

The C++ standard describes the language semantics that C++ implementations have to follow. However, some parts of the language semantics are left up to the implementations to decide and some parts of the language semantics are left undefined by the standard. These undefined semantics are usually referred to as *undefined behavior* and C++ programs are in general supposed to avoid making use of these kinds of semantics.

The most recent C++ language standard released by the ISO is ISO/IEC 14882:2017[5]. For simplicity reasons, the different C++ standard versions are also informally named after their release year (i.e., ISO/IEC 14882:2017 would be referred to as *C++ 17* because it was released in 2017).

## 2.2 Templates in C++

A core feature of C++ is its template system which allows expressing generic data structures and algorithms. As several parts of this thesis discuss template semantics, we dedicate this section to explain some fundamental template semantics.

A template declaration in C++ usually consists of two parts. First, a number of template parameters which are usually types or integers. Second, a normal function or struct/class declaration that makes use of these template parameters in some form. A template declaration itself will never be compiled into some form of executable code but only serves as a recipe for the compiler to create an actual declaration.

```
// Generic add function.
template<typename T>
T add(T a, T b) { return a + b; }

// Generic implementation of an 'Optional' value.
template<typename T>
struct Optional {
  T value;
  bool has_value;
};
```

**Figure 2.1:** Two templates describing a generic function for adding two values and a generic container holding a value and a flag if the value is set.

The process of creating an actual declaration from a template is called *template instantiating*[5, 17.8]. This is usually done automatically by the compiler once a template is used with a specific type or integer parameter. During instantiation, all occurrences of the template parameters in the template declaration are replaced by the actual template parameter values. When instantiating the templates in Figure 2.1 with the type `int`, the compiler would generate the code seen in Figure 2.2. Of course these generated declarations are never visible by the user but are inserted into the AST by the compiler while instantiating the template.

```
int add(int a, int b) { return a + b; }

struct Optional {
  int value;
  bool has_value;
};
```

**Figure 2.2:** The templates from the example in Figure 2.1 expanded for the type `int`.

Besides letting the compiler instantiate a template, there is also the possibility to *specialize* a template in the source code. Template specializations are manually written template instantiations for certain types. These are useful in cases where a template instantiation itself does not produce correct code. For example, in Figure 2.3 we implement an `isThree` function that would incorrectly directly compare IEEE 754 floating point numbers to 3 when being used with the `float` type. This can be corrected by introducing a template specialization that correctly handles the precision problems with floating point numbers.

```cpp
// Generic check if value is equal to 3.
template<typename T>
bool isThree(T v) { return v == 3; }

// Specialized check for floating point precision.
template<>
bool isThree<float>(float v) { return v < 3.01 && v > 2.99; }

int main() {
  // Correct check because it uses template specialization.
  return isThree(3.0f);
}
```

**Figure 2.3:** Template and a specialization for the `float` type that ensures that the zero check is still correct even for floating point types.

## 2.3 C++ modules

C++ modules are a modern way of making library interfaces. The C++ module system replaces the preprocessor-based system of textually including header files that was inherited from C.

C++ modules have several advantages compared to the currently used header files, most notably faster compilation times and better isolation of code.

### 2.3.1 A brief history of C++ modules

C++ inherited C's header files approach for making code reusable between files. Header files rely on the preprocessor of a compiler to insert code from one file into another. The preprocessor step in a C or C++ compiler is the first compilation stage before the actual parsing, where simple preprocessor directives are executed and macros are expanded.

Header and source files make use of the `#include` directive which commands the preprocessor to replace the directive with the contents of the specified file. For example, in the source file in Figure 2.4 we have an `#include` directive that tells the preprocessor to include the `header.h` file contents.

header.h

```
1 struct Foo { void doSomething(); int i; };
```

main.cpp

```
1 #include "header.h"
2
3 void doSomethingWithFoo(Foo &F);
```

**Figure 2.4:** A C++ source file including a header file.

After the preprocessor has processed `main.cpp` it will output the source code seen in Figure 2.5. The output is equal to the original file contents besides the fact that the `#include` directive has been replaced with the file contents of `header.h`. This preprocessor output is then sent to the actual parser of the compiler.

```
1 struct Foo { void doSomething(); int i; };
2
3 void doSomethingWithFoo(Foo &F);
```

**Figure 2.5:** Preprocessor output of `main.cpp` from Figure 2.4.

In Figure 2.5 the file contents of `header.h` contained the declaration of the `Foo` struct that is referenced in `main.cpp`. The purpose of header files like `header.h` is to share their declarations with different source files. However, implementation details like the definition of the `doSomething` function are not contained in the header file but are written in a separate source file. Each actual source file is compiled on its own by the C++ compiler in a separate invocation. All code that is parsed in a single compiler invocation is usually referred to as *translation unit*.

This header system has a few advantages, such as being straightforward to implement and allowing a highly parallelized build. However, it is also causing redundant parsing of header files and does not isolate source files well.

The inefficiency of header files comes from the fact that each translation unit has to re-parse the contents of all included header files, therefore leading to redundant parsing work when building a software project. This became a bigger and bigger problem over time, as the code inside header files became bigger and more complicated. One reason for this growth is the increased popularity of templated code, which requires that all templated declarations are fully defined within the header.

The lack of code isolation with header files is owed to the way the `#include` directive works. It replaces the directive with the file contents and then expands any macros in the inserted code. As these macros can be defined in the including source file, an identical `#include` directive can lead to different inserted code depending on the previously defined macros.

The example in Figure 2.6 demonstrates this by defining a macro before the inclusion of our header. As we defined `Foo` to expand to `Bar`, all following `Foo` tokens were replaced.

content of `main_macro.cpp`

```
1 // Here we define a macro replacing the 'Foo' token.
2 #define Foo Bar
3 #include "header.h"
4
5 void doSomethingWithFoo(Foo &F);
```

preprocessor output of `main_macro.cpp`

```
1 struct Bar { void doSomething(); int i; };
2
3 void doSomethingWithFoo(Bar &F);
```

**Figure 2.6:** Demonstrating the context-sensitive nature of the `include` directive.

The lack of code isolation and the inefficiency due to redundant reparsing of headers are connected issues. The redundant parsing could be solved by caching the part of the AST that is generated by the include, but the fact that includes are context-sensitive makes this impossible without changing the language semantics.

This necessary change to the language was done in the upcoming C++ 20 standard, where a replacement for the header file system was announced, namely C++ modules. The `#include` directive still has the same semantics as in previous C++ standards, but there is now a new `import` statement that does not have the aforementioned problems.

The `import` statement allows specifying a *module* that should be imported into the current translation unit. The semantics of `import` state that the imported code is no longer dependent on any previously defined macros in the importing code. The references module also will be generated from a standalone parsable set of source files which contain declarations. This allows C++ compilers now to cache the AST of these source files and just attach their AST as a sub tree of the currently parsed AST.

As C++ modules are not officially standardized yet, they are currently not officially supported in any C++ compiler. The adoption of C++ modules is therefore currently limited to projects that are used to test C++ modules implementations. When C++ modules are finally released as part of the C++ standard, the first libraries that will adopt it are most likely the standard libraries that are developed alongside certain compilers. One example for this is LLVM's libc++ implementation of the C++ standard library, which we will use later in this thesis as a basis for our testing in this area.

## 2.4 Debuggers

A debugger is software that assists developers with understanding why their programs behave the way they do. The reason why developers want to gain this understanding is usally to eliminate some form of bug in their program.

Debuggers usually support a set of basic operations. Richard Stallmann describes in his overview of GDB[9] these four basic operations:

- Start the program with specified arguments.
- Stop the program on specified conditions.
- Examine the internal program state when it has stopped.
- Change the internal program state when it has stopped.

There exists a multitude of debuggers for various programming languages, but in this thesis we only consider debuggers for the C++ programming language, such as the GDB debugger or LLDB debugger. When debugging a C++ program, a debugger is usually given a program in the form of a compiled executable. As most of the program structure such as types or statements are lost during compilation, C++ debuggers are usually tasked with reconstructing the original program.

The reconstructed program is used to translate the internal state of the program back to the structure in the source code. For example, a debugger will usually allow the user to display what value a variable currently has, even though the compiled program has no notion of variable names and variable values. The debugger will translate the relevant internal state for the user, which is for example just the contents of a register, back to a variable with the correct name and the bytes inside the register interpreted as the data type.

As mentioned, the compiled program lost most of the information about the original program, which leaves the question where the debugger gets the necessary information to reconstruct the program. The answer to this lies in the additional data that is generated by the debugger when it compiled a program. This additional data is usually referred to as debug information and contains the necessary data for reconstruction. Debug information is optimized to be read by other programs and usually not readable by humans. This optimization is necessary because debuggers often have to quickly reconstruct large parts of the original program to understand the current internal state.

As debug information is generated during compilation, the compiler can describe in the debug information how it decided to compile certain parts of the program. For example, it can describe how it decided to store a certain data structure in memory. This information is necessary for the debugger to correctly read the internal state of the program and can not be generated by the debugger itself.

## 2.5 The DWARF debugging file format

There are several formats for expressing debug information which are in use by modern compilers. One of these formats is DWARF[2][4], a debug informating format mostly used on UNIX-based systems. DWARF is a format which is standardized

by the DWARF Standards Committee and is supported in most major C and C++ debuggers.

The DWARF format uses in part a tree structure with every node representing a Debugging Information Entry or *DIE*. Every DIE represents a certain language construct in the original program such as a type, a function declaration or a class declaration. Every DIE is describing what language construct it is representing by having a certain *tag*. A tag can have values such as `DW_TAG_variable` which means it describes a variable in the original program.

Figure 2.7 shows an example for the DWARF generated by the Clang C++ compiler for a given program. The debug information is visualized in its textual form, where each line starting with `DW_TAG` represents a DIE and the indentation of each DIE represents the tree structure. The attributes of each DIE are represented by the lines starting with `DW_AT`. DIEs that represent a certain language construct in the source code also have attributes designating the file and line number where the declaration they represent is written. These attributes (i.e., file, line number and column number) are usually referred to as *declaration coordinates*. The example also illustrates how the DWARF debug information only contains necessary information to understand the program but not everything. The class declaration contained in the actual source code for example can not be found in the debug information as the class is never actually used by any other code.

DWARF itself is no specific to any programming language but designed to be used for expressing the debug information for a wide range of programming languages. Because of its language-independent nature, DWARF supports expressing a wide range of language-specific constructs that might occur in a compiled program. This problem is approached by DWARF by abstracting language constructs into general concepts. For example the act of importing declarations from other files into the current file (which is a concept in many programming languages), expressed in DWARF by the single tag value `DW_TAG_imported_declaration`.

While this abstraction allows DWARF to support many different programming languges, it also limits how precisely it can describe source programs. For example, C++ 's template parameter packs can't be expressed in DWARF without vendor-specific extensions. These imprecisions in DWARF can degrade the quality of the reconstructed program which in turn degrades the functionality of the debugger itself.

## 2.6 LLVM, Clang and LLDB

LLVM is an open source compiler infrastructure that is widely used in the industry as a compiler backend. The LLVM umbrella project hosts the LLVM project itself and a wide variety of related projects, such as the Clang C++ compiler, the libc++ standard library, the LLD linker and the LLDB debugger. These related projects build upon LLVM as their compiler backend. They also make use of the data structures and algorithms provided by LLVM.

The input program in the file `dwarf.cpp`

```
class A {};
int main() {
  int i = 0;
}
```

Generated DWARF for `dwarf.cpp` in its textual representation

```
DW_TAG_compile_unit
  DW_AT_producer    ("clang version 8.0.0")
  DW_AT_language    (DW_LANG_C_plus_plus)
  DW_AT_name        ("dwarf.cpp")
  DW_AT_comp_dir    ("/home/")

  DW_TAG_subprogram
    DW_AT_frame_base        (DW_OP_reg6 RBP)
    DW_AT_name      ("main")
    DW_AT_decl_file ("/home/dwarf.cpp")
    DW_AT_decl_line (2)
    DW_AT_type      (0x00000052 "int")
    DW_AT_external  (true)

    DW_TAG_variable
      DW_AT_location        (DW_OP_fbreg -4)
      DW_AT_name    ("i")
      DW_AT_decl_file       ("/home/dwarf.cpp")
      DW_AT_decl_line       (3)
      DW_AT_type    (0x00000052 "int")

  DW_TAG_base_type
    DW_AT_name      ("int")
    DW_AT_encoding  (DW_ATE_signed)
    DW_AT_byte_size (0x04)
```

**Figure 2.7:** A C++ program and the corresponding DWARF debug information.

### 2.6.1 The Clang compiler

Clang is a compiler for C-languages, which refers to languages that are either based on C or closely resemble C's syntax and semantics. Examples for this are C++ , Objective-C or OpenGL's shader language. Clang was originally developed by Apple when the use of GNU's C and Objective-C compiler GCC become too impractical for the use in Apple's developer tools.

A fundamental design idea of Clang is that it is unlike GCC supposed to be used as a library for compiling and analysing C-languages. This is made possible by two implementation strategies.

First, Clang's parsing and compiling pipeline is extensible. All parts of Clang contains functionality to register callbacks that will be called for certain events. In the parsing logic the callbacks are used to provide additional AST nodes that can not be found in a source file (referred to in Clang as `ExternalASTSource` or `ExternalSemaSource`). In the compilation logic the callbacks forward all parsed AST nodes to a consumer. The default consumer for Clang is usually the code generator which is creating LLVM IR (the LLVM intermediate representation).

Second, Clang's Abstract Syntax Tree (AST) needs to be expressive and straightforward to use by different clients. As an AST in its traditional form is usually not very easy to understand by a client (e.g., because they do not directly see in the AST what type an expression returns), the AST in Clang looks very different to the AST in other compilers. The AST is actually a graph because some nodes have a link to their parent node or link to related declarations (e.g., an expression links to the type that it evaluates to). This makes the AST simple to traverse and analyze for external tools. As resolving types is obviously in contrast to the idea of keeping the AST only focused on syntax and not semantics, Clang's AST is also in this regard not conforming to the normal approach to ASTs. In fact, Clang's AST is an intermediate representation of the source code rather than an actual abstract syntax tree.

For the rest of the thesis we keep referring to Clang's AST as just AST as it is the only compiler we discuss in detail. We will also refer to Clang's concept of an *ASTContext* or just *context* which is essentially an AST bundled with its associated information such as types or language options.

Since its release Clang has been used for several tools that support C-languages. These Clang-based tools range from source code formatters like `clang-format` to whole debuggers like `LLDB`.

### 2.6.2 C++ modules in Clang

In Section 2.3 we described C++ modules in general. In this chapter we want to revisit the topic from the point of view of a C++ compiler. As there exists several C++ compilers that are used in practice, there are also several C++ modules implementations in development. In this paper we only consider Clang's C++ module implementation as Clang is the compiler used in LLDB for C++ functionality.

Clang already shipped some form of module functionality since 2013. This was mostly to support Objective-C's modules feature, which had a similar goal and approach as the upcoming C++ modules.

This functionality in Clang was then extended to support the C++ language. As no language changes were standardized yet, Clang made its module feature available to non-modular C++ code by automatically translating `#include` directives into `import` statements. The decision whether an `#include` directive can be safely rewritten into an `import` statement was based on external configuration files that were placed alongside the header files (so-called *modulemaps*). The feature was released under the name *implicit modules* and is currently the most common way to use C++ modules (e.g., Clang/LLVM use this mode to build themselves).

With implicit modules every encountered module is compiled by Clang when it is needed for the first time. All following uses of the module just load the compiled module from disk. Due to this the build system does not need to know which modules are needed to compile certain files, as any missing modules are build implicitly by Clang. However, this also means that the build system can not properly schedule the compiler invocations and would cause a congestion. For example, if three source files all need module A but the build system is not aware of this, the build system could schedule the compilation of all three source files at the same time on a multithreaded system. The first compiler invocation would then start building the module A while the other two would just wait on the first compiler invocation to finish building the module so that they can use it.

To fix this problem with having a build system inside the compiler, Clang introduced *explicit modules* feature that actually uses the build system to build C++ modules before they are used[7]. With this the build system can properly schedule the Clang invocations while building the program and prevent congestion by scheduling Clang jobs so that they do not depend on the same module.

Finally, Clang also supports the standardized C++ modules as proposed in C++ 20. However, as C++ 20 is not released yet, we will only mention it here for completeness.

To summarize, there are four different module implementations in Clang:

- Objective-C modules, which implement Objective-C's module system.
- Implicit modules, where Clang automatically translates `#include` directives into `import` statements.
- Explicit modules, where Clang requires that all used modules are passed to the compiler before compilation.
- The standardized C++ 20 modules.

All module implementations in Clang share the same binary data format on disk for storing the AST nodes. The data format is designed to be memory-mapped by the reader and already contains all data fields in a format that Clang also uses internally in its AST (e.g., an 4-byte integer is also stored as a 4-byte integer with the same byte order on disk). Since the data is stored in the same way as Clang's internal AST data structures, the data format is also inherently unstable and not compatible between different Clang versions. This means that a C++ module compiled by Clang version 4 can not by read by Clang version 5 and vice versa. This makes Clang modules not a suitable format to store AST nodes for other tools or other devices to use.

Using Clang modules is very efficient. All data from a module is loaded on demand when it is needed by Clang. When initially importing a C++ module into Clang, it will barely modify the AST and only place hooks in the AST which trigger the loading of their direct child nodes. Once these triggers have been activated, Clang will load more AST nodes which in turn can trigger the loading of their children. This implementation is done in a way that the AST automatically grows when it is explored by Clang or a tool using Clang. Because of this design it is usually not necessary to add any module-specific logic to most software that is using Clang for parsing C++ code.

## 2.7 The LLDB debugger

LLDB is the debugger of the LLVM project. Like other LLVM umbrella projects, LLDB makes use of LLVM's libraries and uses LLVM as a compiler backend. LLDB supports debugging programs written in C, C++ and Objective-C.

LLDB offers similar functionality to the GDB debugger from the GNU project by offering an interactive command line frontend (see Figure 2.8). However, the primary focus of LLDB is to be used as a debugging backend from an IDE or interactive debugger. At the time of writing, LLDB is most prominently used as the debugging backend in Apple's Xcode IDE and Google's Android Studio IDE.

```
(lldb) target create "helloworld"
Current executable set to "helloworld" (x86_64).
(lldb) b main
Breakpoint 1: where = helloworld main + 22 at helloworld.cpp:4:13
(lldb) r
Process 1469 stopped
* thread 1, name = "helloworld", stop reason = breakpoint 1.1
   1    #include <iostream>
   2
   3    int main(int argc, char **argv) {
-> 4      std::cout << "Hello, World!\n";
   5    }
(lldb) expr argc
(int) $0 = 1
(lldb)
```

**Figure 2.8:** Example debugging session in the interactive LLDB command line frontend.

### 2.7.1 Expression evaluation in LLDB

LLDB comes with an expression evaluator which allows the user to run code within the target program. An example for how this feature would be used from the

command line interface can be seen in Figure 2.9. Usually the expression that needs to be evaluated is passed to LLDB in textual form (e.g., `1 + 2`). This means LLDB is required to first parse the input. Afterwards the parsed AST of the expression is compiled and executed within the target program. Figure 2.11 provides an overview of this process in LLDB.

```
(lldb) r
Process 1469 stopped
* thread 1, name = "helloworld", stop reason = breakpoint 1.1
   1    #include <iostream>
   2
   3    int main(int argc, char **argv) {
-> 4      std::cout << "Hello, World!\n";
   5    }
(lldb) expr 3 * 3
(int) $0 = 9
(lldb) expr argv[0]
(char *) $1 = "/helloworld"
```

**Figure 2.9:** Example usage of LLDB's expression command.

Parsing and compiling C++ in LLDB is handled by an embedded instance of the Clang compiler. For each expression that needs to be evaluated, LLDB creates a new Clang instance which is used to parse the expression. Before passing the expression to Clang, LLDB wraps the expression inside a function to make it valid C++ (see Figure 2.10), as the language does not support free-standing expressions in the global scope.

```
void $__lldb_expr(void *$__lldb_arg)
{
    /*LLDB_BODY_START*/
    argc + 1;
    /*LLDB_BODY_END*/
}
```

**Figure 2.10:** Wrapper code generated for the expression `argc + 1`.

The embedded Clang instance can now start parsing the wrapped expression as its legal C++ syntax. However, the expression still has semantic errors as all references to variables and types from the target program are not contained in the wrapped expression. In the example above, the `argc` variable is not defined and Clang does not know how to resolve the variable to the target program.

To solve this, LLDB attaches itself to Clang in the form of an `ExternalASTSource`, which is a Clang API that allows external code to help resolve unknown references

that are encountered while parsing. This API receives requests from Clang for every unknown reference. LLDB tries to answer every one of these requests by searching the debug information of the target program for the appropriate declaration and then constructing the respective declaration in the AST of our expression. Afterwards Clang can use this constructed declaration to continue parsing the program.

After Clang has successfully constructed an AST from the expression, LLDB will send the AST to Clang's code generator which will generate executable code that can be injected into the target. This injected code will be executed within the target process. Afterwards, the result of the evaluated expression is read back and displayed by LLDB to the user.

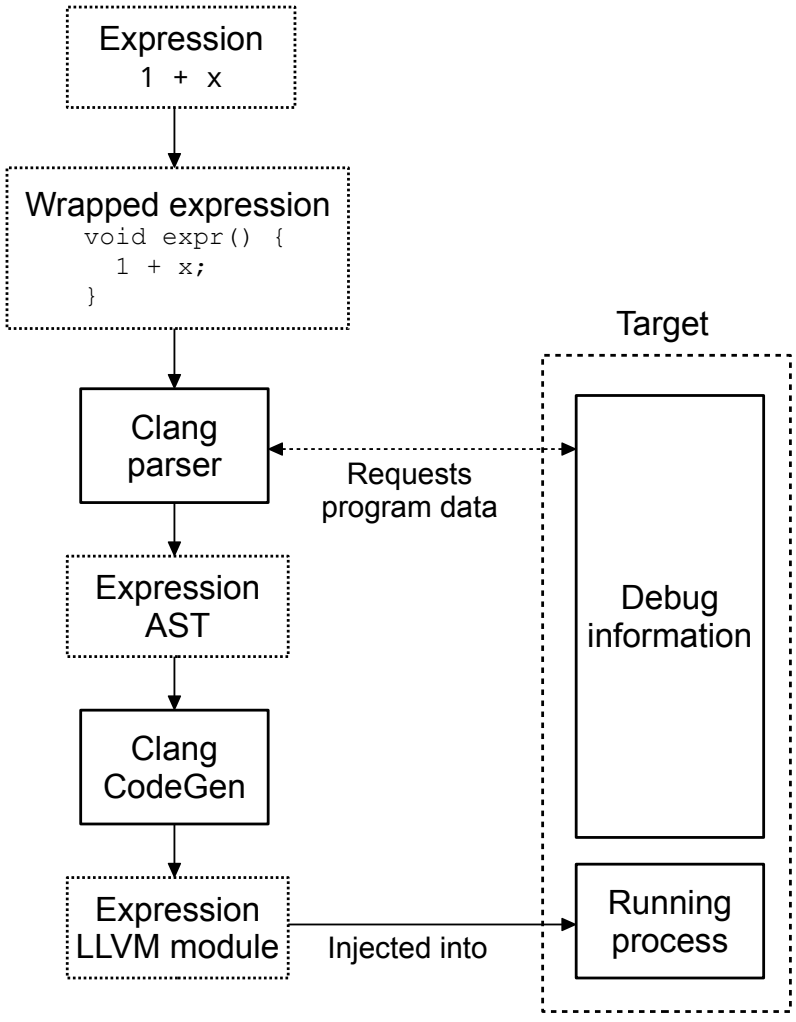


**Figure 2.11:** Overview of the expression evaluation process in LLDB.

# 3

# Integrating C++ modules into LLDB

This chapter describes our integration of C++ into the LLDB expression evaluator which is the centerpiece of this thesis.

This chapter is divided into two parts describing the two prototypes that were created as part of the project. The first part describes the implementation of a prototype that focuses on integrating the standard library C++ module. The second part describes a more generic prototype that is capable of handling all modules that are used within a target program.

## 3.1 The standard library module prototype

C++ comes with a standard library that provides a basic set of functionality and is always available on every C++ implementation. The standard library is currently provided by a set of header files, but will most likely be standardized as a C++ module named `std`. A central part of the C++ standard library is the *standard template library* (**STL**), a set of templates for commonly used algorithms (e.g., sorting) and data structures (e.g., lists and maps).

Most of the functionality of the STL is unsupported in LLDB's expression evaluators due to the way the STL is implemented. Many declarations in the STL are templates, which means they are not actually compiled unless they are instantiated by the user. Also, some implementations of the STL force the compiler to actually inline many templated member functions such as *std::vector::size*. In both cases there will be no compiled function inside the debugged executable that LLDB could call from the expression evaluator.

This limits the possible interactions with STL container classes from within LLDB. For example, developers currently can not reliably inspect the last element of an STL list from within the expression command, as the necessary member functions for this such as `std::list::back` are not callable from LLDB.

The current workaround for this in LLDB is to have custom code that supports printing the whole contents of a STL container. This custom code is known as *data formatters* and is provided by LLDB developers alongside LLDB. Data formatters are activated by writing an expression that returns an STL container as a result, which causes LLDB to display the textual representation of the container contents to the user. This workaround however only solves part of the problem. For example, looking up the n-th element of a list requires the developer to manually search the

list for this element.

```
(lldb) p v
(std::vector<int, std::allocator<int> >) $0 = size=3 {
  [0] = 1
  [1] = 2
  [2] = 3
}
```

**Figure 3.1:** LLDB displaying the contents of vector 'v' to the user.

As the STL is widely used in C++ programs and using the STL declarations in the expression evaluator is an unsolved problem, we decided to focus our project first on integrating only the `std` C++ module. Most of the `std` module's contents are stable as they are standardized by the language standard itself, so focusing on this code base would also allow us to avoid most of the complexity of handling arbitrary user code or user-defined build configurations.

### 3.1.1 Approach

The expression evaluation problems when interacting with the STL are mostly caused by two specific problems:

1. Certain functions are not compiled into the executable and therefore not callable from the debugger.
2. Some types are only partially reconstructed from the debug information and cause unexpected errors when they interact with the compiler.

We can solve both problems with the help of C++ modules as follows. Problem 1 requires us to reconstruct the function body of a given function from a C++ module, generate executable code for it and inject the generated code alongside the expression into the process. Problem 2 means that we have to replace partially reconstructed types from the debug information with their respective intact counterpart from a C++ module. Also, before we can load any C++ module from LLDB, we first need to identify all used C++ modules for a given executable and then build these C++ modules. The following sections describe these steps in more detail.

### 3.1.2 Module discovery

Before LLDB can load any C++ modules, it first needs to identify which modules are available and used by the target program. This information is usually not contained within the debug information of a program as C++ modules are not widely used or implemented in compilers yet. The only way to have this information available in the debug information is by compiling the program with Clang and allowing it to emit LLDB specific debug information via the `-glldb` flag.

With this flag Clang emits `DW_TAG_module` DWARF tags into the debug information for every used module in a compilation unit (see Figure 3.2). These tags contain information such as the module name, its child modules and custom attributes for include paths and configuration macros that were used for building.

```
DW_TAG_module
     DW_AT_name      ("std")
     DW_AT_LLVM_config_macros        ("\"-DFOOBAR=1\"")
     DW_AT_LLVM_include_path ("/usr/include/c++/v1")
     DW_AT_LLVM_isysroot     ("/")

     DW_TAG_module
          DW_AT_name    ("vector")
          DW_AT_LLVM_config_macros      ("\"-DFOOBAR=1\"")
          DW_AT_LLVM_include_path       ("/usr/include/c++/v1")
          DW_AT_LLVM_isysroot   ("/")
```

**Figure 3.2:** DWARF tags and attributes for the C++ module `std.vector`.

As this information is in the debug information for each translation unit, LLDB can create a list of modules by finding the translation unit in which the current stop location is and then iterating over all `DW_TAG_module` tags.

### 3.1.3 Module building

Once LLDB has identified the modules that are used in the current translation unit, LLDB still needs the respective compiled module files.

In theory, these files could already exist on the system, as all the module files should have been generated when the executable was built. In practice, however, it is not uncommon for users to delete modules in the module cache directory. Clang itself even offers a *prune* functionality that automatically removes module files after a certain period of time.

Loading the compiled module files that were used to build the executable is also problematic for another reason. Clang's module files have no stable layout and do not support being loaded by any Clang version other than the one which generated a module file. Because of this, LLDB can only load these module files if its internal Clang version exactly matches the Clang version used to compile the used modules. As this strict restriction on the supported build configurations would severely limit the usefulness of module-enabled debugging, we decided not to use this approach when implementing module loading.

The alternative that we use in our prototype is to rebuild all C++ modules that we need for a certain expression. The compiled modules can be reused between expressions and therefore should only be built once. If the referenced source files or the build configuration changes, we rebuild the used module files.

Our implementation of this feature relies upon Clang's implicit module build mode. In this mode, Clang automatically builds module files when they are needed

by the parsed source code. Clang's implicit module build mode also has further advantages that made it a good candidate for being the backend mechanism in LLDB's module building process. For example, Clang checks the modify time of the referenced header files and rebuilds modules if necessary in this mode. Clang also splits the module cache into different subdirectories for each used build configuration, so that we can have the same module with multiple incompatible build modes in the cache. Finally, Clang automatically takes care of building any missing dependencies of any requested C++ modules.

To use this mode, we modified the internal Clang instance used by the expression evaluator to be capable of parsing C++ files. For this we had to inject the include directories extracted from the debug information into the Clang configuration. These include directories are necessary as Clang will use them to find modulemap files and resolve any textual includes from within the modules. We also had to resolve several differences between the configuration of the internal Clang instance and the normal configuration of Clang. Most of the differences were related to language extensions that were used by system headers and therefore needed to be activated before Clang could parse them.

Our solution for triggering the build of the module files is to inject the respective `import` directives into the wrapped source code of the expression (see Figure 3.3). As Clang is configured to build any missing modules when they are needed, these `import` directive will cause Clang to implicitly build all modules that have not been compiled yet.

```
1 @import std;
2 @import hoge;
3
4 void $__lldb_expr(void *$__lldb_arg)
5 {
6     /*LLDB_BODY_START*/
7     argc + 1;
8     /*LLDB_BODY_END*/
9 }
```

**Figure 3.3:** Wrapper code generated for an expression that requests the build of the `std` and `hoge` modules.

### 3.1.4 Embedding modules in the expression evaluator

Once LLDB has compiled the required C++ modules, we still need to load the necessary AST nodes into the expression context. As described in Section 3.1.1, we need to substitute partially reconstructed types and provide all missing functions to Clang to improve the expression evaluator for the STL. However, our prototype has to consider two constraints that limit the possible ways of loading these nodes into our context.

The first constraint is that we must never have two nodes with different contents describing the same declaration. This would directly violate Clang's internal assumption that every declaration is only defined once as required by the *One Definition Rule*[5, 6.2-1]: "No translation unit shall contain more than one definition of any variable, function, class type, enumeration type or template."

For example, there can be at most one AST node describing the `std::vector` template in our context. This means that we must never load for any given declaration both the AST node from the C++ module and the AST node from the debug information.

Because of this constraint we can not just add both the C++ module and the debug information from LLDB to Clang's lookup. This would either lead to Clang behaving in some undefined way or trigger a diagnostic by Clang which would abort the expression evaluation.

The second constraint is that we can not just load all AST nodes in advance from the C++ module and look into the debug information AST for any missing declarations. The reason for this is that deserializing all modules can severely slow down the expression evaluator. This performance impact is negligible in the current prototype as we only load the `std` C++ module (which contains only about 16 megabytes of AST nodes), but it would certainly impact the second prototype which potentially would need to preload Gigabytes of AST nodes.

Because of these constraints we designed our prototype so that the loading of AST nodes from a C++ module only occurs in these two scenarios:

1. Clang is requesting additional information about an unknown declaration via the `ExternalASTSource` interface.
2. LLDB is copying requested AST nodes into the expression evaluation context.

#### 3.1.4.1 Clang is requesting additional information

In the first scenario we receive a request from Clang that a certain identifier was encountered and we need to load any related AST nodes into the expression evaluation context. LLDB would normally just forward this request to its internal AST context that was created from the debug information as shown in Figure 3.4.

In our LLDB prototype we intercept this request and first try to answer it by loading potentially related AST nodes from the attached C++ modules as shown in Figure 3.5. If we did find any node within the C++ module that could fulfill the request, we consider the request as fulfilled and will not forward it to the debug information context. However, as C++ modules might not contain all possible declarations a program uses, we still have to forward the request to the debug information if our C++ module can not provide any relevant AST nodes. In all possible cases either the C++ module or the debug information will answer a request, but they will never both answer the same request. This ensures that we never load any conflicting nodes from the debug information into our expression evaluation context.

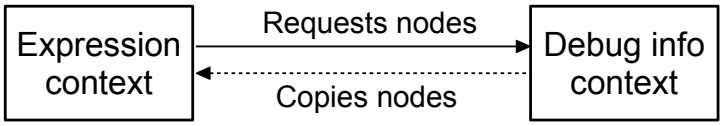


**Figure 3.4:** Loading of external AST nodes in LLDB.

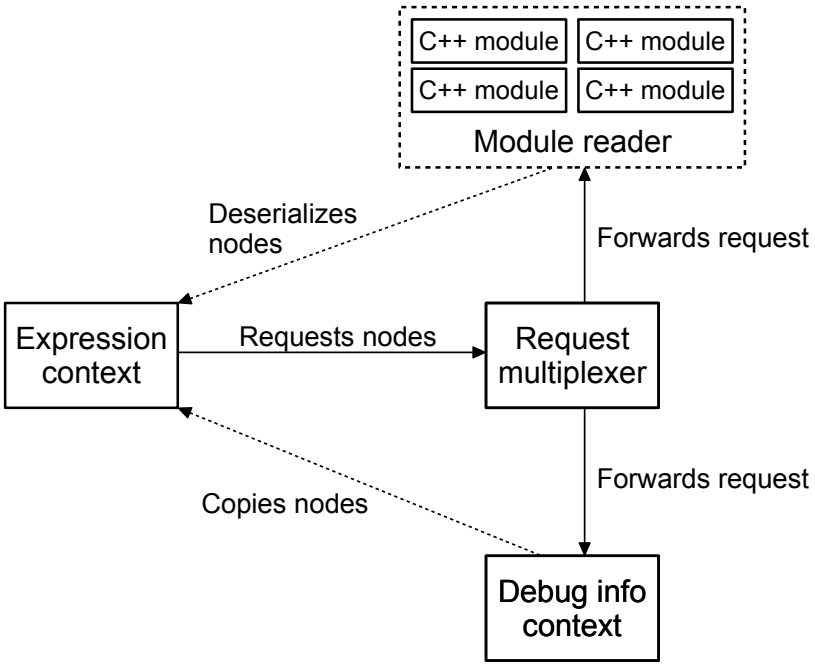


**Figure 3.5:** Loading of external AST nodes in LLDB with C++ modules.

#### 3.1.4.2 LLDB is copying requested AST nodes

In the second scenario where we load AST nodes from a C++ module is when LLDB is copying AST nodes from the debug information context to the expression context. For example, this happens when Clang requests a local variable of the current function the debugger has stopped in (see Figure 3.6). The local variable will be found in the debug information and the related AST nodes will be copied over into the expression context. The problem is that the related AST nodes might contain an AST node which is also available in the C++ module.

Our prototype has to intercept this copying process as the AST nodes from the debug information context are expected to be incomplete. When an AST node is intercepted, our prototype loads instead the corresponding AST node from the C++ module and then stops the recursive AST copying.

```
(lldb) r
Process 1469 stopped
* thread 1, name = "helloworld", stop reason = breakpoint 1.1
   1    #include <Foo.h>
   2    int main(int argc, char **argv) {
   3      Foo f;
-> 4      return f.bar();
   5    }
(lldb) expr f
```

**Figure 3.6:** Example scenario where LLDB would request more information about the variable 'f'.

### 3.1.5 Manual reconstruction of templates instantiations

Even though the standard library module contains the declarations for all STL containers, it actually does not contain any declarations for the container instantiations used in user programs. The reason for this is that user programs actually instantiate the STL container templates with (potentially user-defined) types, while the C++ module only contains the templates itself. For example, the user might use the `std::vector<double>` type in the program, but the C++ module only contains the `std::vector` itself.

To reconstruct these types we have to manually instantiate the container templates in our prototype, e.g., instantiate `std::vector` with the type `double`. Usually instantiating a template with a type does not guarantee that the resulting template instantiation is actually identical to the one used in the program. The C++ template system allows the custom specialization of templates for certain types and these specializations do not have to be declared inside the C++ module. Figure 3.7 shows an example where manually instantiating a templated type from the original template would lead to a wrong result. In this example, the `S<char>` specialization is hidden

inside the `main.cpp` file which is not included in the C++ module. A manual reconstruction of `S<char>` from the original template would create a struct with one `char` member, not one `int` member as defined by the specialization.

module.h

```
1  template<typename T>
2  struct S {
3    T i;
4  };
```

main.cpp

```
1  #include "module.h"
2
3  template<>
4  struct S<char> {
5    int a;
6  };
7
8  int main() {
9    S<char> c;
10 }
```

**Figure 3.7:** A template specialization that is not in the C++ module. Manually instantiating `S` with the type `char` to reconstruct the type `S<char>` would lead to a wrongly reconstructed declaration.

For standard library templates it is more viable to reconstruct these declarations by instantiating them from their original templates. This is because the C++ standard is disencouraging STL container template specialization by the user. In general the C++ standard disallows adding any declarations to the standardized `std` namespace, but it allows standard library templates to be specialized if the specialization depends on a user-defined type [5, 20.5.4.2.1-1]. Also, if the container is specialized for a user-defined type, it must still meet the same standard library requirements as the original container[5, 20.5.4.2.1-3]. To summarize, it is allowed for users to specialize the STL templates we instantiate, which would lead to incorrectly evaluted expressions in our prototype. However, because of the limited scope in which it is allowed, it is unlikely that any user actually specializes the STL containers. In fact, we could not find any such specialization in the source code of the LLVM umbrella projects.

Because it is so unlikely that users specialize STL containers, we decided that template instantiation is a viable way to construct STL types. In the unlikely case that a user decided to specialize a STL container template, our prototype would just fall back to the normal LLDB reconstruction process using debug information. This will prevent that we incorrectly reconstruct templates as was illustrated in Figure

3.7.

An open question we had to answer was how to determine if a template instantiation inside the debug information was created by instantiation or by user specialization. DWARF actually can not express how a templated declaration was created, so from the perspective of LLDB all templated declarations are treated as user-defined specializations. The detection mechanism we developed for our prototype relies on the following specification of declaration coordinates in templates[1, pp. 94].

> If the class type entry representing the template instantiation or any of its child entries contains declaration coordinate attributes, those attributes should refer to the source for the template definition, not to any source generated artificially by the compiler.

This means that the declaration coordinate in DWARF should never point to compiler-generated declarations. As template instantiations are compiler-generated, the declaration coordinate for them have to point to their original template definition. Template specializations however are written by the user in the source code and will get assigned unique declaration coordinates. With this information we can define that a DWARF declaration was created by instantiation if and only if its declaration coordinates match with its original template definition. There is still the problem that DWARF does not emit any information at all about the original template definition, so we do not actually know its declaration coordinates. However, the C++ module contains the declaration coordinates of the original template, which means we can compare DWARF's declaration coordinates against the ones generated from the C++ module.

We only had to change our prototype slightly to implement instantiation of STL containers. Clang is already correctly instantiating STL containers due to the way we implemented our lookup (see Section 3.1.4.1). For example, if we parse an expression containing `std::vector<Foo>` (where `Foo` is a user-defined type), Clang would first find both the `std` namespace and the `vector` namespace in our C++ module and therefore ignores the debug information for this type. The `Foo` type is not found in our C++ module, which means the debug information will supply this type and Clang will instantiate it by itself.

When LLDB is copying nodes (see Section 3.1.4.2), it also checks if any node we copy is an instantiation of an STL container and then manually instantiate the template in our expression AST context.

## 3.2 The generic module prototype

The second prototype we implemented as part of this thesis is the generic module prototype. This prototype expands the loading mechanism of our standard library prototype by loading all C++ modules used to compile a C++ program. The generic module prototype was built upon the standard library prototype as the implementation of this prototype can be done by extending the first prototype. In the rest of this thesis we will therefore test this unified prototype and refer to it as *LLDB with*

*C++ modules.*

### 3.2.1 Module discovery process

The module discovery process for the generic module prototype was supposed to be identical to the discovery process in the first prototype. However, this discovery process turned out to be only viable for finding the standard library module. The reason for this is that Clang only emits debug information for C++ modules that are actually used in the program. This is consistent with how debug information is in general not generated for unused declarations. However, in this case it prevented the generic module prototype from building the modules in a certain build setup.

The specific problem is related to the build configuration we generate from the debug information. For every imported module we find in the debug information we expand the include directories in our C++ module build environment. This is necessary to resolve all `#include` directives we encounter when compiling a C++ module for the expression evaluator. The problem we faced was now that if a C++ module is imported in a source file but never used, then we also do not get its include directory. This becomes problematic when we actually compile the module and we are unable to resolve this specific module to any specific file location. This causes the module compilation to fail which renders the prototype useless in these build setups.

As unused imports are not uncommon, we either had to change this behavior in Clang or find another way to force the emission of all imported modules in the debug information. As changing Clang's behavior in this case would have actually increases the debug information size (due to the additional module descriptions that would have been emitted), we instead decided to change the way we required supported programs to be compiled.

The best solution we found to emit all modules into the debug information was to make use of Clang's module debugging[8]. We already describe this feature in the related work section of this thesis (see Section 6.2), so we will not further explain it here. However, the important change with modules debugging is that Clang is forced to declare all modules inside the debug information.

### 3.2.2 Module building

Module building in our general module prototype is more complex than with our standard library prototype. The standard library is usually only slightly influenced by most compiler settings while a user-defined module can potentially be fully dependent on a specific build parameter to even build at all. The precise reconstruction of the used build parameters is therefore of utmost importance in our generic module prototype.

While DWARF itself does not have any way to express the used build configuration (i.e. the used compiler flags), there are several different sources from where we can retrieve information about if and how a certain compiler flag has been used in the original program. The defines given to the compiler are emitted, similar to the include directories, by Clang into the `DW_AT_module` DWARF tags of the specific

module. The specific version of the C++ standard used to compile the program is also found in DWARF.

However, other flags are not emitted into the debug information which can pose a problem for correctly compiling user-defined C++ modules. In Figure 3.8 we can see an example for where the content of a module would depend on the build configuration, specifically whether Streaming SIMD Extensions (SSE) support is enabled or not. With SSE enabled during compilation of the original program, the `Data` struct would contain an aligned vector member. With SSE disabled during compilation, the `Data` struct would instead contain a normal `std::vector`. To correctly reconstruct the `Data` struct in our C++ module we need to determine when building the C++ module whether SSE was enabled during the compilation of the original program. This information however is not emitted into the debug information in any way which means our prototype can not guarantee that it correctly compiled this C++ module. Not only SSE support is causing this problem but in general all compiler flags that can somehow alter the AST of a header file. This includes the used C++ dialect (e.g., GNU, Borland or Microsoft extensions), SSE-like features such as AVX2, whether run-time type identification (RTTI) was enabled and POSIX thread support.

```
1  #include <aligned_vector.h>
2
3  struct Data {
4  // Feature check if SSE is supported or not.
5  #ifdef __SSE__
6    aligned_vector<int> v;
7  #else
8    std::vector<int> v;
9  #endif
10 };
```

**Figure 3.8:** A header that compiles differently depending on whether it was compiled with Streaming SIMD Extensions (SSE) support enabled or not.

A possible workaround for this is to let the user manually specify all the original compiler flags used in the program for the current expression. This however would make the debugger more complicated to use for the user and is prone to errors due to wrongly entered compiler flags.

As we could not find a good solution to this problem as part of this thesis, we only reconstruct the build configuration for the language version and the user-defines in our current generic module prototype.

### 3.2.3 Integration into the expression evaluator

After finding and building the C++ modules of the project with our prototype, we sill have to integrate them into our expression evaluator. We could reuse most of the infrastructure we already developed inside LLDB for integrating the standard

library module. The lookup mechanism described in Section 3.1.4.1 stayed identical as it was designed to work independently from the actual declarations that are being requested by Clang. The interception mechanism in Section 3.1.4.2 has been extended to cover a wider range of possible data structures that need to be copied. With the standard library prototype we only needed to support class templates, but in the generic prototype we needed support for intercepting and replacing generic class and struct declarations.

# 4
# Evaluation

This chapter presents our evaluation of the reliability and performance of our LLDB prototypes. To measure these properties, we performed several experiments that test our prototype when evaluating expressions. To put our test results into context, we repeat the experiments with the following three debuggers: the Microsoft Visual Studio Debugger, the GDB debugger and the normal LLDB debugger without any of our patches.

## 4.1 Evaluation setup

The hardware setup we use in this thesis for gathering performance data consists of an AMD Threadripper 2990WX with 48 GB of memory. All parsed source code, debug information and executables are stored on a Samsung 970 EVO during testing. The software setup for our measurements consists of three different debuggers running on Linux: GDB, LLDB and our own prototype based on LLDB. The Microsoft Visual Studio Debugger is running on Windows 10.

#### Microsoft Visual Studio Debugger (MVSD)

The Microsoft Visual Studio Debugger (or *MVSD* for the rest of this paper) is the default debugger for Microsoft's Visual Studio IDE. We use the default debug build configuration of Visual Studio to compile the test programs. The used version of Visual Studio is 2019 16.0.4.

#### GNU Debugger (GDB)

The GNU Debugger is the default debugger on all major Linux systems and is widely used by developers. Due to its large userbase and prominence we will treat its expression evaluator as the state-of-the-art debugging experience on these platforms.

The GDB version we use in this evaluation is 8.2.1, which is the latest version of GDB at the time of writing. The programs we debug with GDB are compiled the GNU Compiler Collection (GCC) version 8.3, which is also the latest version at the time of writing. All programs debugged with GDB are compiled with GCC's default flags on Linux, with the exception of the `-g` flag which tells GDB to emit debug information for the compiled program.

#### LLDB

LLDB is the default debugger on several BSD-based distributions and on macOS, so we consider it as the state-of-the-art debugging experience on these platforms. The

version of LLDB we test is 8.0.0 which is the latest release at the time of writing. We compile all executables for this debugger with Clang version 8.0.0. We keep Clang's default flags for Linux when compiling the executables, but enable module support and tell Clang to emit debug information specifially for LLDB via the flags `-fmodules -fcxx-modules -glldb`.

**LLDB with C++ modules**

This represents our LLDB with C++ modules prototype. Our patches are applied to LLDB version 8.0.0.

### 4.1.1 Standard libraries

As we make use of the standard library in the following sections, we also need to specify which standard library implementation we use in our test setup. We decided two use three different standard libraries in our test setup. All programs compiled by Clang are using LLVM's own `libc++` implementation of the C++ standard library. This is necessary as the GCC standard library is not configured by default to support Clang's module implementation, which prevents our LLDB prototype from loading the standard library module. As `libc++` is by default configured to work with Clang's modules and is the default standard library on macOS and some BSD-based distributions, we decided to compile our LLDB and prototype tests against `libc++`.

When compiling for GDB with GCC, we use the `libstdc++` implementation that is included with GCC. We decided to let GDB/GCC use their own standard library implementation as the debugging support for their own standard library is more mature than for other standard library implementations.

When compiling for MVSD we use Microsoft's MSVC compiler that is the default in Microsoft Visual Studio. MSVC is also using its own standard library implementation for these tests, as neither `libstdc++` or `libc++` seem to be officially supported by MSVC.

## 4.2 Reliability of expression evaluation

The main goal of this project was to provide a reliable expression evaluator in LLDB. We define reliability as the chance to successfully evaluate expressions. An evaluated expression is successful if it was correctly parsed, its side effects were correctly applied to the target program and the expected evaluation result was returned to the debugger. If any of these steps failed, we consider the evaluation unsuccessful.

### 4.2.1 Selection of the benchmark data

To measure the reliability of an expression evaluator we can evaluate a list of expressions. The actual reliability can then be defined as the percentage of successful evaluations within this list. As the actual composition of an expression influences whether a debugger can successfully evaluate it, therfore the composition of our list

of expressions also influences our reliability measurement. For example, if our set of expressions would only contain templated member function calls, we would probably measure a high reliability for our C++ modules prototype. This is because our prototype was designed to handle these kind of expressions and has a higher chance to successfully evaluate them. However, this measurement is not very meaningful as expressions in the real world are not just templated member function calls.

Creating a fair and representative list of expressions is a challenging task since it would first require us to create statistics about what kind of expressions are usually evaluated in debuggers. In this section we instead decided to use an exhaustive list of expressions as a basis for our reliability measurement. *Exhaustive* means in this context that our expression list covers all possible kinds of expressions, even if that means it is potentially not representative of expressions used in the real world. This keeps the measurements free of bias, but also makes the actual reliability measurements harder to relate to the actual end user experience. We leave this interpretation of the measurements for the next chapter which discusses the results.

When creating our list of expressions we try to let the debugger construct declarations with as many different language features and other properties as possible. However, we try to avoid certain language features in the expressions themselves, as we do not want to test the expression parser capabilities of the debuggers. This mostly means that we do not use constructs like `new` to allocate function arguments and always try to use simple data types for arguments (e.g., integers instead of custom classes) as these are both language constructs which seem to be mostly unsupported by GDB and MVSD. We also limit our expressions to C++ 14 features as we do not want our tests to be influenced by potential immature support in debuggers for the most recent C++ 17.

#### 4.2.1.1 Allowed workarounds in LLDB

Besides the workarounds and limitations above, we also allow one workaround for expressions evaluated in LLDB. This workaround is only necessary when the expression we evaluate returns a non-trivial type. Expressions that return these types can be successfully evaluated in our prototype, but after we evaluate such an expression, LLDB attempts to copy the expression results to a persistent AST. This copying process is likely to fail as it was not designed to handle the complicated AST nodes we reconstruct with our prototype. However, the copying process itself is not relevant for the actual evaluation of the expression but instead only used for the persistent expression result feature in LLDB (which is supposed to store expression results). For this reason we decided that if an expression fails due to the copying process and we can workaround the issue, then we would still treat the expression evaluation as successful. For the sake of making our results easier to reproduce, we marked all expressions where we used a workaround in the appendix.

#### 4.2.1.2 Testing unused and used declarations

Not only the expression itself determines how likely it is that a debugger can successfully evaluate it, but also the surrounding code in the program. Especially if the program contains a declaration that is only references by the expression but not

in the original program, the chance that the expression can be evaluated decreases. Declarations in the program that are never used are often not emitted by the compiler into the executable in the form of executable code or debug information, which makes it impossible for the debugger to reconstruct it.

When we generate our test data, we have to make the decision whether the expressions we want to test should reference declarations that are used or unused in the original program. In practice, expressions are usually using both used and unused declarations, so we decided to run all our tests with two kinds of programs. First, programs that use all declarations that the expressions are going to reference. Second, programs that use nearly no declaration the expression is going to reference. While these two extremes are not very realistic, they still allow us to define an upper and lower bound for the reliability of expression evaluation. We expect the program where all declarations are used to be the upper bound of the reliability and the program with only unused declarations to be the lower bound.

### 4.2.2 Evaluating the standard library prototype

This section explores the reliability of the standard module prototype. As this prototype is concerned with only the limited code of the standard library, we can create an exhaustive list of expressions by just creating an expression for every declaration in the standard library. Each expression should use the referenced declaration in some logical way, e.g., a function should be called and a variable read.

The list of expressions, the results and the program in which they are used can be found in the appendix of this thesis.

#### 4.2.2.1 A short introduction to GDB's Xmethods

In the following sections we refer to some expressions as being evaluated with the help of *Xmethods*. This refers to a feature of GDB that provides a Python function, the *Xmethod*, as an alternative implementation of a C++ function in the target program[3]. When GDB tries to evaluate a call to a C++ function where such an Xmethod was provided, GDB will call the Xmethod instead of the C++ function in the target program. The Xmethod usually runs in the GDB process and reads values from the target program if necessary to calculate the same result that the C++ function would return.

Xmethods try to solve a similar problem as we do in this thesis by allowing the user to evaluate expressions using functions that are inlined or otherwise unavailable in the target program. GDB allows users to provide their own Xmethods, but also provides a set of Xmethods for the C++ standard library which allows GDB in the following chapters to evaluate expressions involving inlined functions.

We assume that the Microsoft Visual Studio Debugger is providing a similar functionality as it also able to call some simple functions that were inlined or not emitted into the executable. However we can not confirm this theory as we do not have access to the source code and could not find any mention of this in the MVSD documentation. As we do not have an official name for this feature in MVSD, we will also refer to it as Xmethods.

#### 4.2.2.2 Sequence containers

The C++ standard library offers several *sequence containers*, which are containers that preserve the insertion order of the elements that are stored in them[5, 26.3]. The sequence containers defined by C++ 17 are: `array`, `vector`, `stack`, `queue`, `deque`, `list` and `forward_list`. All these containers have different approaches for storing their elements and therefore different implementations. However, they have in common that they usually do not inspect the elements they store in any way, i.e., they do not try to compare different elements to establish an order or generate hash values for them. Because these containers barely interact with the API of the stored elements we will use normal integers as the stored data type in the related evaluation tests. The test results were identical to other simple data types like a user-defined struct, so this decision does not influence the following test results.

First we try to evaluate the member methods of each type of sequential container. No member method is used in the original program in this test, so most of the relevant symbols and debug information will not be emitted into the executable. The reliability measurements for this test can be seen in Figure 4.1.

The unmodified LLDB debugger was not able to evaluate a call to any member method of the tested sequential containers. The GDB debugger was able to call a few commonly used member functions like `size()` or `empty()` by emulating the call with its Xmethod feature. However, for the more rarely used containers, like `stack` or `queue`, GDB did not seem to ship with any relevant xmethods and could not evaluate a single member function call. Our standard library prototype was able to evaluate every single member function call and reached the maximum reliability score.

The second test is identical with the first test in that it evaluates member function calls to the sequential containers. However, this time all expressions are also done inside the program to force the emission of debug information and code into the executable. The reliability results for this test can be see in in Figure 4.2.

The reliability of the unmodified LLDB increased slightly compared to the previous test, but LLDB was still unable to evaluate most expressions. The main failure comes from libc++'s methods that were forcibly inlined with function attributes. This made LLDB unable to call them from the expression evaluator.

The GDB debugger became more reliable in this test. It was able to evaluate nearly all expressions and only failed expressions that required more advanced parsing such as support for templated member methods and using iterators as function arguments. This is most likely a shortcoming of the parser GDB uses which does not support advanced C++ language features.

Our C++ modules prototype was, as in the previous test, able to evaluate every single expression in our list.

#### 4.2.2.3 Smart pointers

Smart pointers are a part of the standard library that provide an alternative to manual memory management. They are implemented as templates which, when instantiated for a given type, take care of freeing the memory associated with a given object. There are three kinds of smart pointers in the standard library that

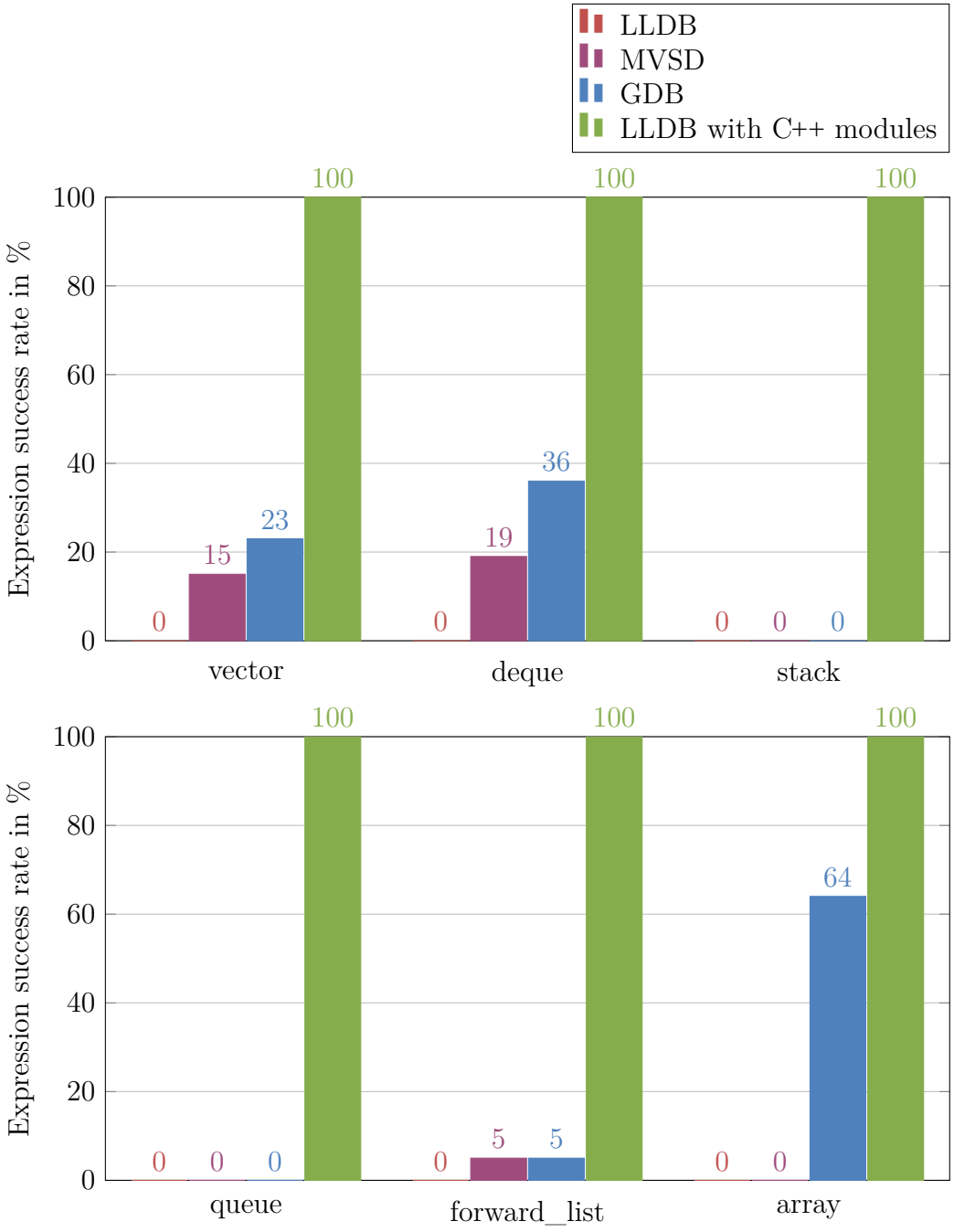


**Figure 4.1:** Expression reliability when evaluating member functions of **unused** sequential containers from the standard library.

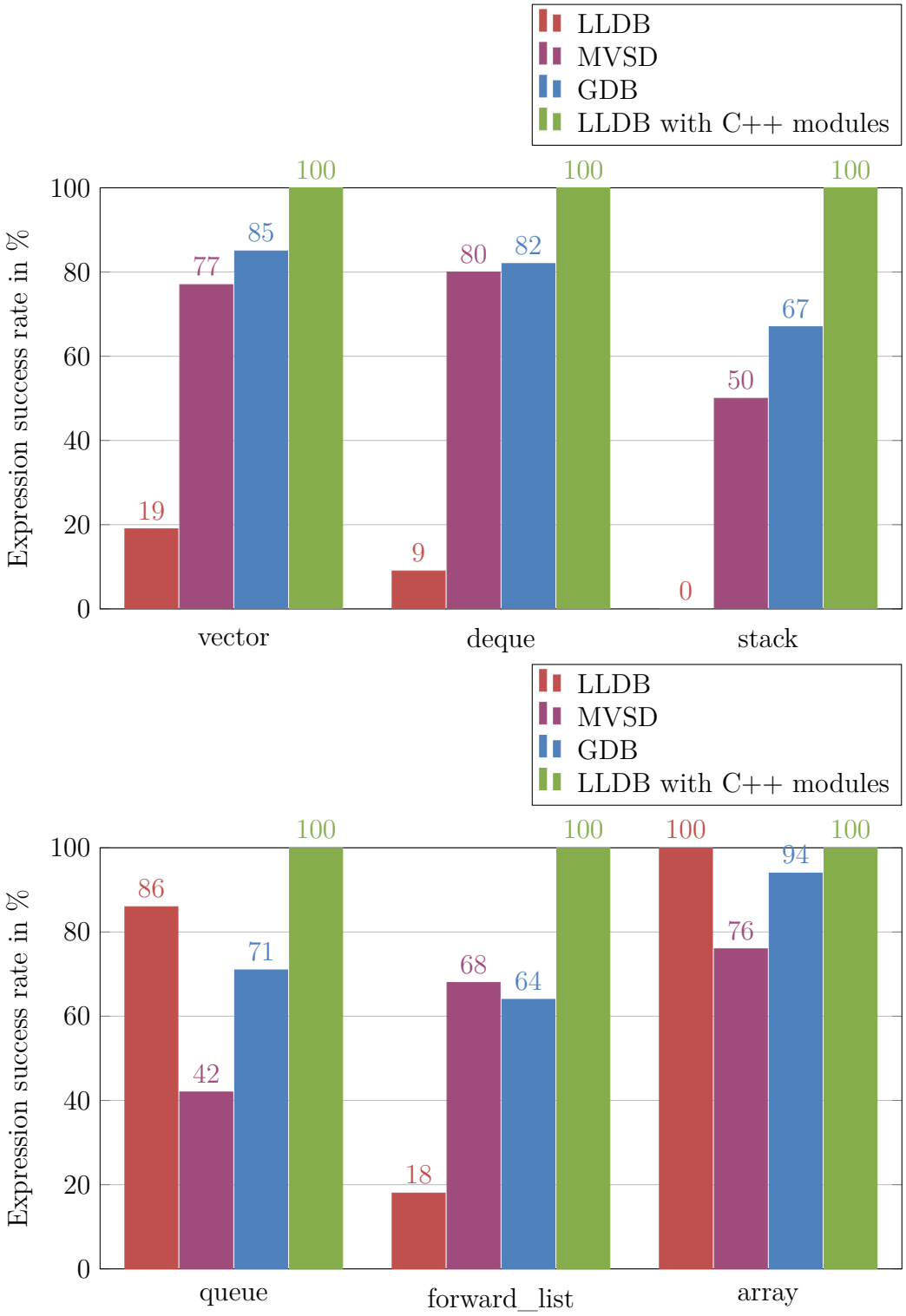


**Figure 4.2:** Expression reliability when evaluating member functions of **used** sequential containers from the standard library.

are being used:

- `std::unique_ptr` which represents unique ownership of the object and will free it when it goes out of scope.
- `std::shared_ptr` which represents a pointer which shares ownership of the object with other shared_ptrs. A `shared_ptr` tracks a reference count for the given object (i.e., how many `shared_ptrs` in total own the object) and deletes it if this reference count reaches zero.
- `std::weak_ptr` which represents temporary ownership of an object and is usually used to prevent reference cycles when using `shared_ptr`.

Our first test tries to evaluate smart pointers when they are otherwise unused in the user program. We can see our reliability measurements for this test in Figure 4.3. Our prototype performs again as intended and can evaluate all expressions involving smart pointers. For other debuggers, the reliability is the highest when evaluating expressions involving `std::unique_ptr` with `std::shared_ptr` and `std::weak_ptr` being less supported. This is reflective of the popularity of the different smart pointers, with `std::unique_ptr` being usually more commonly used than `std::shared_ptr` or `std::weak_ptr` in C++ programs. This correlation is most likely caused by the increased popularity of data structures leading to more type-specific workarounds in debuggers.

Our second test tries to evaluate the same expressions but also uses all related member functions of the smart pointers inside the program. The reliability results of this test can be seen in Figure 4.4. All debuggers could now consistently evaluate the majority of the expressions involving smart pointers. The main problem for the remaining failing expressions were expressions involving the `owner_before` method of `std::shared_ptr` and `std::weak_ptr` which could not be successfully called by any debugger besides our prototype. One possible explanation for this is that `owner_before` takes another smart pointer as an argument which does not seem to be supported by most expression evaluators.

#### 4.2.2.4 Associative containers

The standard library comes with four associative containers which are designed to allow efficient access to their elements by some form of key and optionally store the elements ordered by their keys. These four containers are a `set` (where each element is unique) and a `map` (where each key is unique but with an associated value that does not need to be unique), with both having an alternative implementation in the standard library that does not order its keys (`unordered_set` and `unordered_map`).

The ordered containers compare their keys with the `<` operator implementation for the key types, while the unordered containers hash the keys via `std::hash`. This differentiates them from the other data structures in the standard library which usually only access the destructors and constructors of their containing types, but never related functions and structures like the `<` operator or `std::hash`.

Before we inspect this part of the associative containers, we first perform our two standard reliability tests as in the sections before with integers as keys and values. We can see the reliability measurements for unused associative containers in Figure

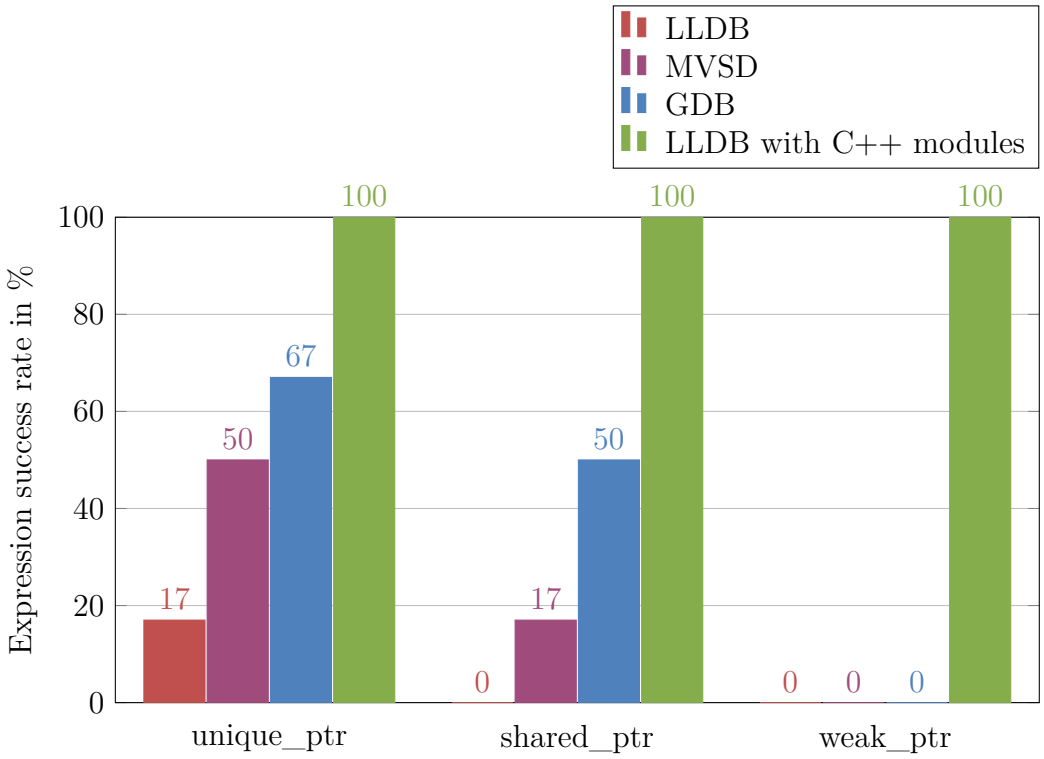


**Figure 4.3:** Expression reliability when evaluating member functions of **unused** smart pointers from the standard library.

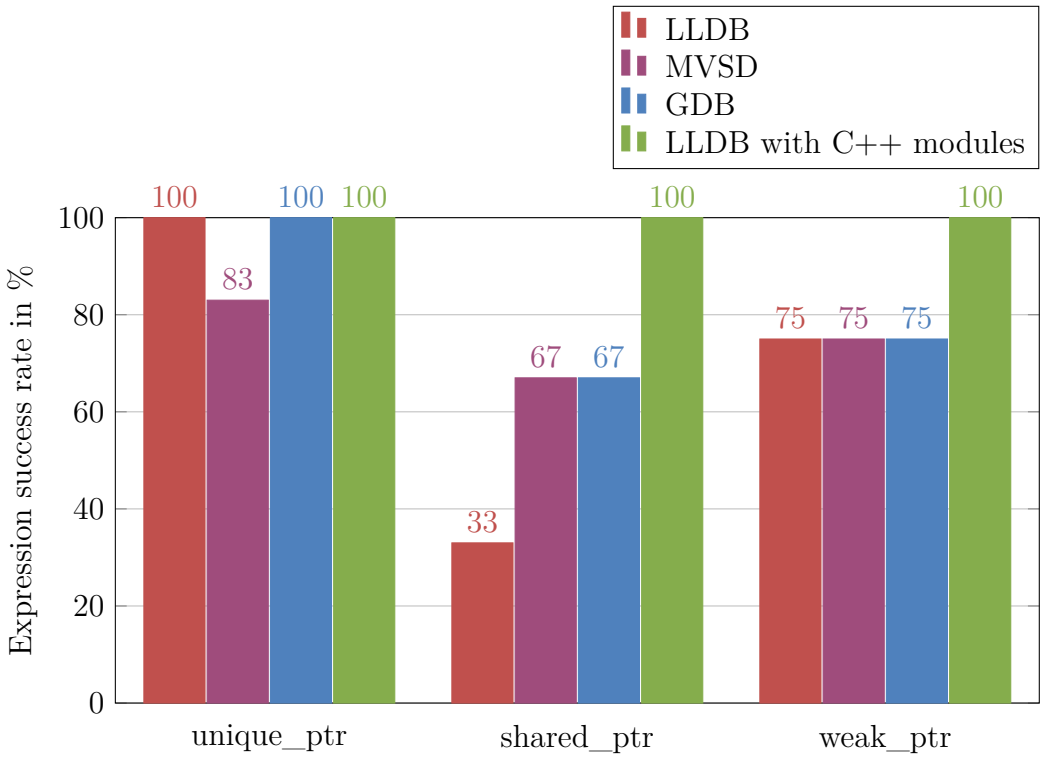


**Figure 4.4:** Expression reliability when evaluating member functions of **used** smart pointers from the standard library.

4.5. Our modules prototype delivered as before full reliability when evaluating member functions of the containers. Similar to earlier results, the other three debuggers could not evaluate most of the expressions with LLDB not being able to evaluate a single member function call.

The reliability results for used associative containers are displayed in Figure 4.6. This time LLDB outperformed every other debugger as both GDB and MVSD struggled with correctly reconstructing and calling the complex function signatures of the containers.

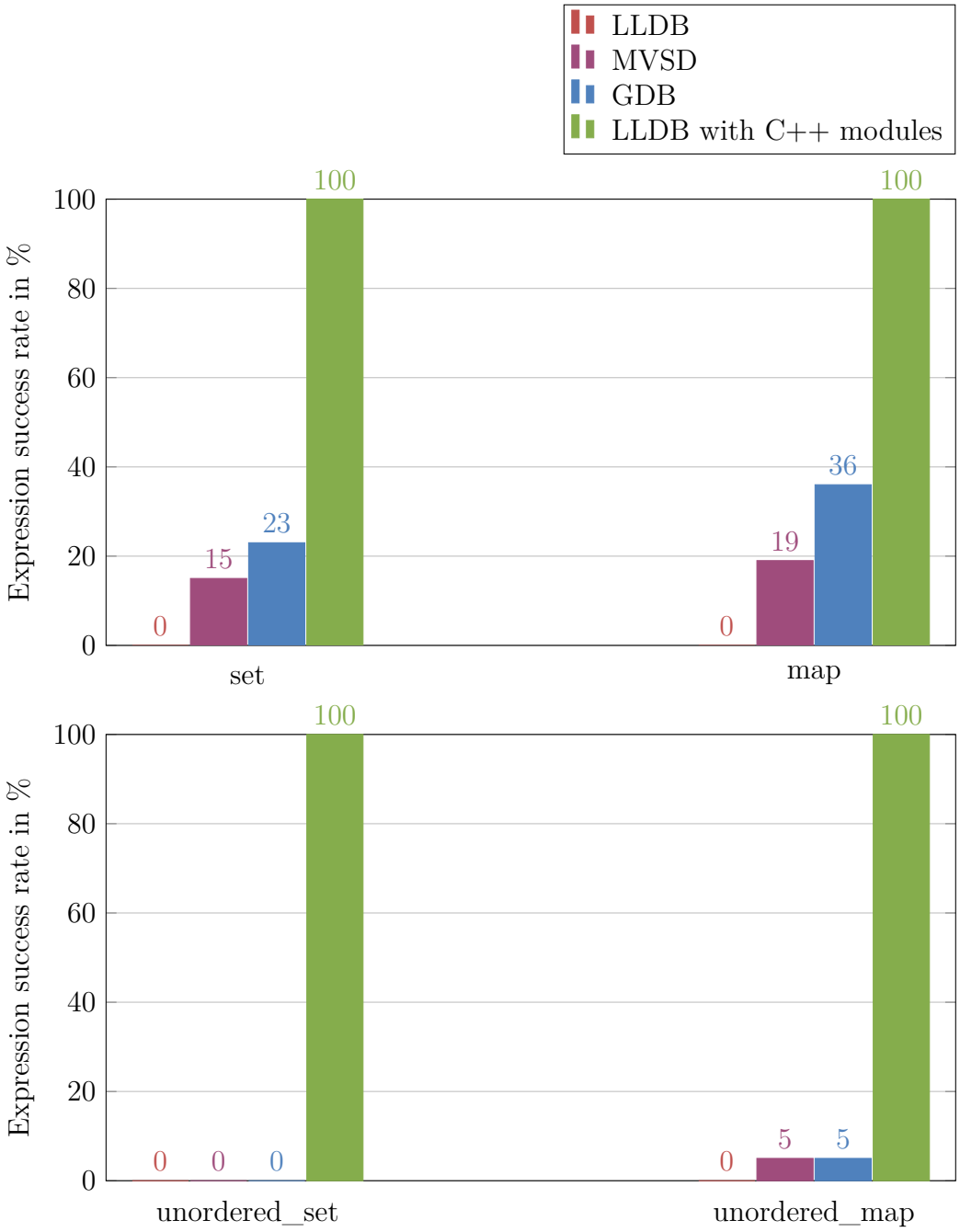


**Figure 4.5:** Expression reliability when evaluating member functions of **unused** associative containers from the standard library.

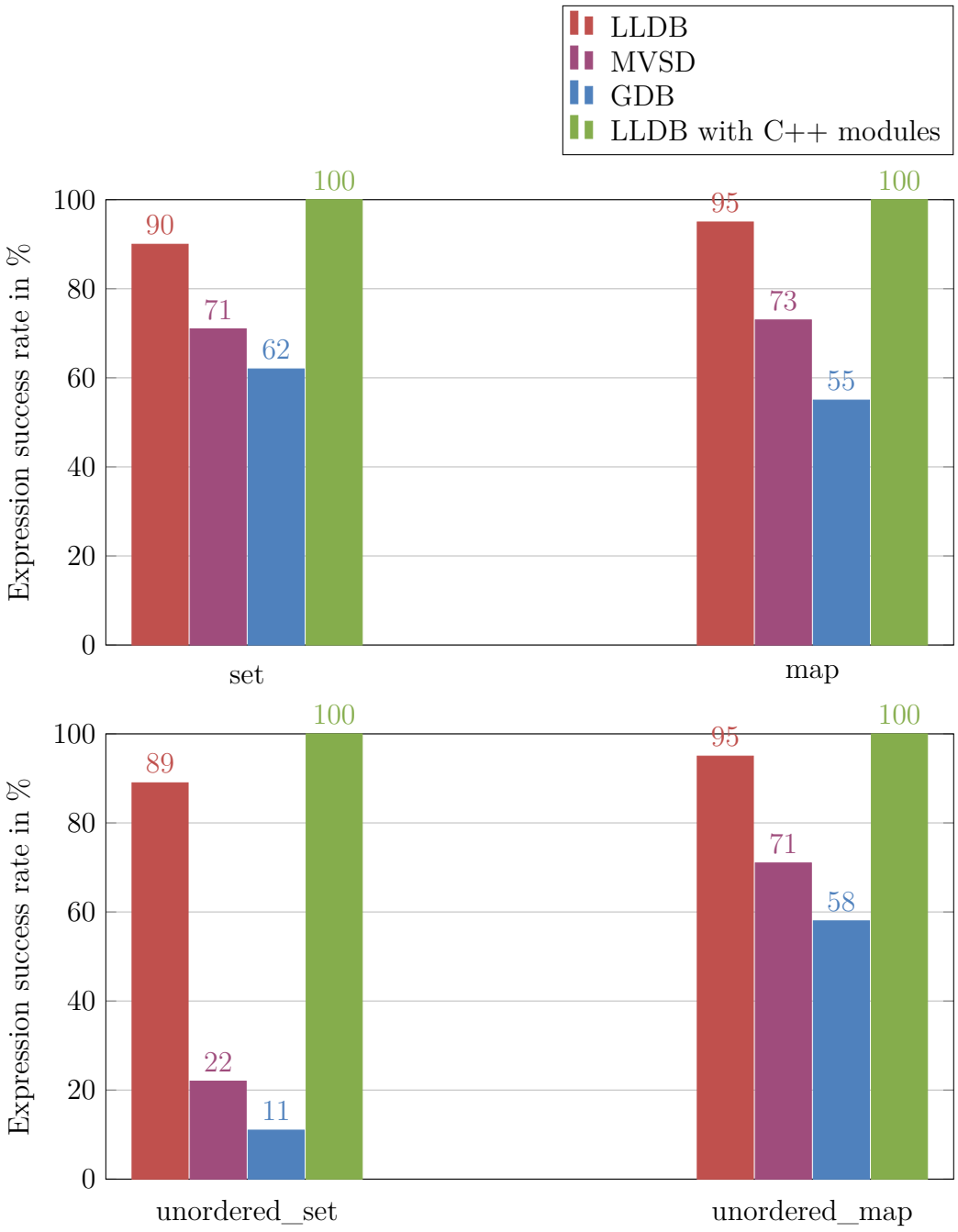


**Figure 4.6:** Expression reliability when evaluating member functions of **used** associative containers from the standard library.

So far we only tested simple integers as keys but as we mentioned above it is also possible to use more complicated types in associative containers. To test this we implemented a custom type (see Figure 4.7) that can be used as a key for both unordered and ordered containers as it provides interfaces for hashing and comparison. We created two derivative test cases from our associative container test above but with the integer keys and values replaced by our custom key type:

1. A test where we used our custom type as the key for an unused and a used

`std::set`, `std::map`, `std::unordered_set` and `std::unordered_map`. The custom type was implemented in the source file and was therefore not contained in any C++ module. This test is supposed to see how having to reconstruct the hash or comparison operator from debug information will affect reliability.

2. A test where we used our custom type as the key for an unused and a used `std::set`, `std::map`, `std::unordered_set` and `std::unordered_map`. The custom type was implemented in a C++ module and we loaded this module with our generic module prototype. This test is supposed to see how custom comparison or hash operators influence the reliability with associative containers.

We repeated the expression tests we used on the original test case with our two new test cases and our prototype. We did not repeat the tests for all debuggers due to time constraints. The reliability results from the new test cases were for the identical to the original test case. Our own prototype kept its full reliability score as it seems reconstructing hash and comparison operators from debug information does not interfer in any way with the rest of our prototype.

```
1  struct A {
2    A() = default;
3    A(int i) : i(i) {}
4    int i;
5  };
6  // Custom comparison operator for A.
7  bool operator<(A a, A b) { return a.i < b.i; }
8
9  // Custom hash implementation for A.
10 namespace std {
11   template<> struct hash<A> {
12     size_t operator()(const A &a) const {
13       return a.i;
14     }
15   };
16 }
```

**Figure 4.7:** Implementation of our custom data type A that can be used as a key in both ordered and unordered associative containers.

#### 4.2.2.5 Standard library algorithms and functions

Besides container templates, the standard library also contains several functions that either provide miscellaneous functionality (e.g., `std::exit` for exiting the process) or implement generic algorithms (e.g., sorting or calculating the absolute value of a number). When we tried calling these functions from our set of tested debuggers,

we found that nearly no function call could actually be evaluated by any debugger successfully. This was true both for unused functions and when we used them (with the same type arguments for templates) in our test program. However, our prototype was able to call every single function inside the standard library independently if it was used or not.

We could not find a precise explanation for why other debuggers were not able to evaluate these functions even when they were used inside our program.

#### 4.2.2.6 Generic module declarations

So far we only tested the reliability of our prototype by using the standard library. This section will present the results when testing the reliability of the generic module prototype with a small testbed that makes use a wide variety of C++ language features. The program and the expressions we evaluate can be found in Appendix B. We divide the expressions we test in the generic prototype into two categories: Expressions that reference templates and expressions that do not. The reason for this is that templates are one of the areas where our prototype sees the biggest difference in reliability. We can see the reliability measurements of our tests in Figure 4.8.

Our own C++ modules prototype performed better than the other three debuggers in this test, but failed to evaluate all expressions. The failures were related to our prototype currently incorrectly importing macros from the module and template specializations hidden inside source files which could not be reconstructed.

The other debuggers performed in general much better when evaluating expressions that do not involve templates. This again is related to them not having the original template in the debug information so they ca not create new instantiations for types that do not occur in the original program. When the templated declaration was already instantiated by the program with the same types, the other debuggers were usually able to correctly evaluate the expression.

When evaluating normal expressions the other debuggers performed better and could evaluate about half of our test expressions. The failed expressions could not be evaluated because they fell into one of two categories:

1. Expressions calling inlined functions that were not emitted into the executable. It should be noted that member functions are implicitly inline which means these failures are also caused by inlining.
2. Expressions involving information that was not emitted into the debug information. This includes the failures related to expanding macros or calling functions with their default arguments.

## 4.3 Expression evaluator performance

In this section we measure the performance of our expression evaluator. We define the performance as the elapsed real time that our expression evaluation needs from receiving the evaluation command to returning the expression result. The measurements are done in real time and not CPU time because usually debuggers are used

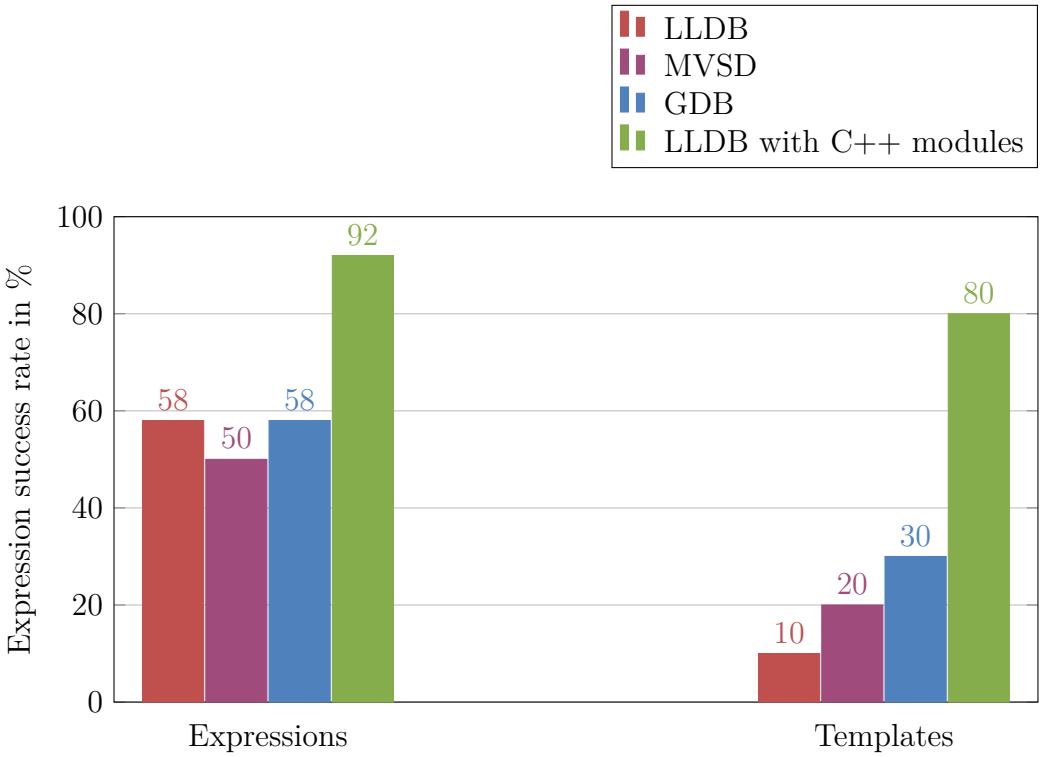


**Figure 4.8:** Expression reliability when evaluating different kinds of expressions. We divide the expressions into expressions using templates and expressions that do not use templates.

as interactive programs. This means that a user is usually waiting on the expression result and minimizing the user's wait time has the highest priority.

The debuggers we test in this section are LLDB and our LLDB with C++ modules. We do not test GDB or MVSD in this section for several reasons. One reason is that they use a very different expression evaluation infrastructure and differently compiled test executables compared to LLDB, so comparing their timings with LLDB's timings is not very meaningful. Another reason is that they could not evaluate all the expressions we test in this section.

### 4.3.1 Performance when calling vector member functions

We start our performance evaluation with a list of expressions that consists of calls to member methods of a `std::vector<int> v`. The first expression is `v.at(v.at(0)/3)`, which is a compound of member function calls which should check the behavior in case LLDB employs any single-function call optimizations. The second expression `v.max_size()` is just calling a simple member function of the vector. The last expression `v.push_back(1)` is calling a more complicated member function of the vector class. We call each expression 100 times and measure the average time it took for each debugger to evaluate them.

The measured time for each expression can be seen in Figure 4.9. We can see that all our expressions are being evaluated at a slower pace in our C++ modules prototype. The first two expressions are around two to three times slower, while our call to `push_back` was around 25 times slower in our C++ modules prototype.

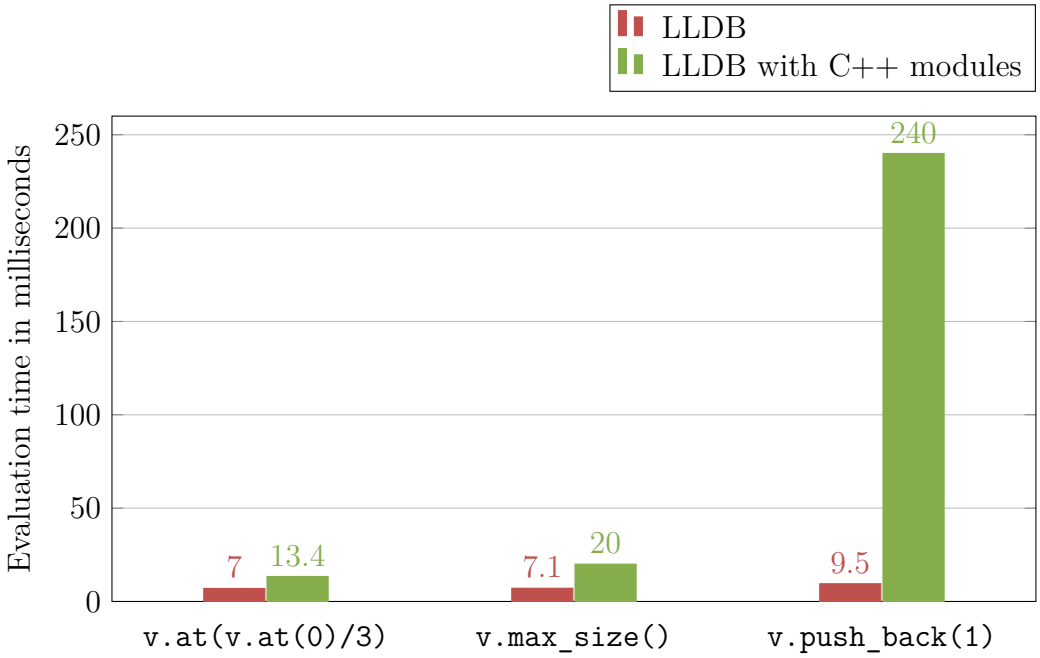


**Figure 4.9:** Expression performance when evaluating expressions. The expression that is evaluated is listed on the x-axis.

To better understand what could cause these slowdowns we can look at the generated LLVM intermediate representation (IR) that was created when the expressions

were evaluated. In Figure 4.10 we can see the number of LLVM IR instructions that each expression generated. All expressions in the LLDB with C++ modules prototype are generating more instructions than when they are evaluated in the unmodified LLDB. This is mostly because in our prototype we compile these methods ourselves from the AST, while in the unmodified LLDB we just generate a function call to the function inside the executable. Generating just a function call is obviously faster but comes with the unreliability we measured in the previous section. On the other hand, compiling the whole function comes with a potentially high slowdown when we call a complicated function such as the `push_back` member function.

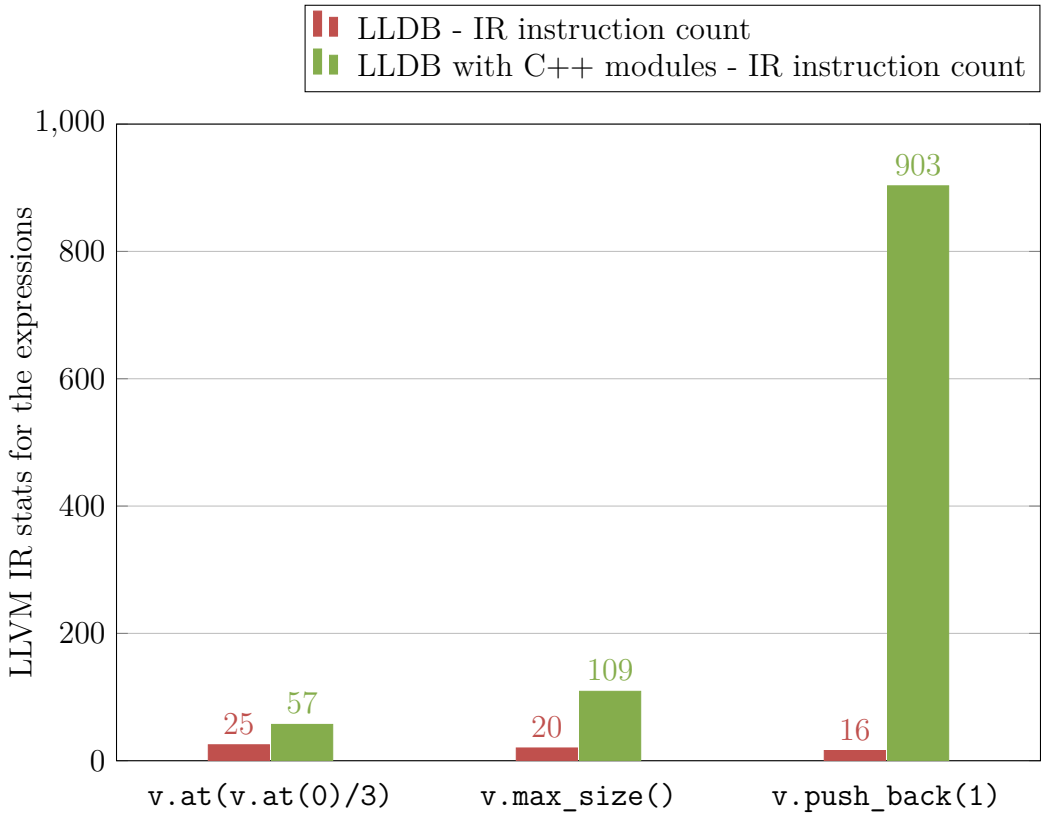


**Figure 4.10:** Number of instructions in the generated LLVM IR when evaluating certain expressions. The expression that is evaluated is listed on the x-axis.

### 4.3.2 Performance for trivial expressions

We now try to evaluate expressions that do not cause the compilation of imported functions. These tests demonstrate the performance impact of using C++ modules themselves, which includes opening and searching the module file for potential declarations and importing them into the AST. We are again evaluating these expressions in a compiled function with a `std::vector<int> v` variable.

The measurements of this test can be seen Figure 4.11. Our first expression just performs a simple calculation without actually using any declaration from the standard library module. The slowdown of around 2.6 milliseconds we see for this expression is therefore the time we need to load the standard library module and

integrate it into the expression evaluator. The second prototype tries to use the vector `v` type but without causing any function from a C++ module to be compiled. As the slowdown is identical to the previous expression, we see that deserializing and instantiating the `std::vector` type itself is efficiently possible within our prototype.

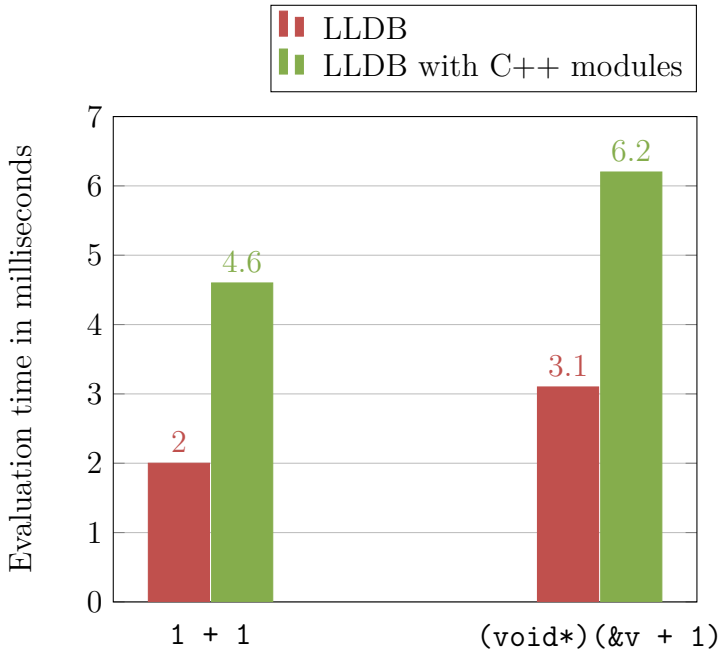


**Figure 4.11:** Expression performance when evaluating expressions. The expression that is evaluated is listed on the x-axis.

Now we also review the LLVM IR generated by the simple expressions. As both expressions do not reference any information from the module, we expect to see the same amount of instructions generated for both expressions. The data we measured can be seen in Figure 4.12 and confirms this theory.

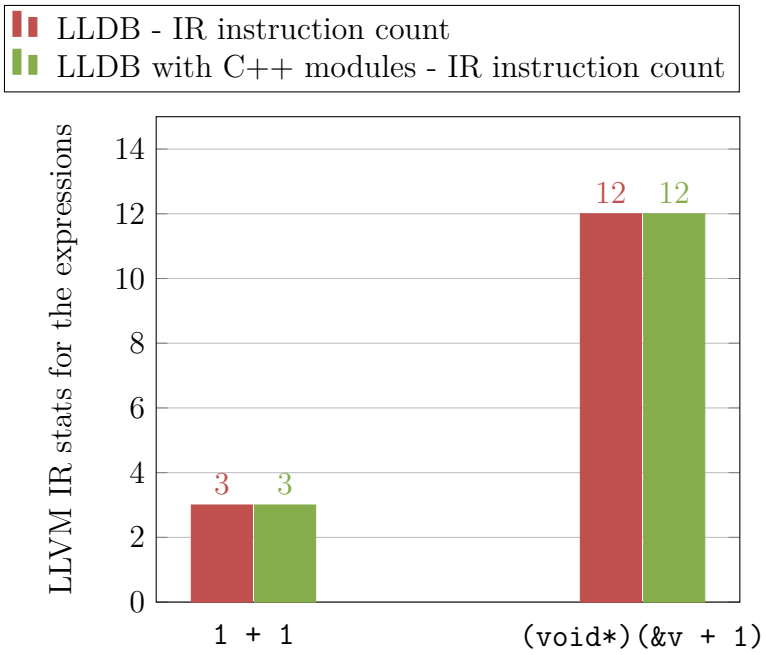


**Figure 4.12:** Number of instructions in the generated LLVM IR when evaluating certain expressions. The expression that is evaluated is listed on the x-axis.

# 5
# Discussion

This section discusses our prototype based on the evaluation we did in the previous chapter.

## 5.1 Review of results

When looking at our results and the theory we set out at the start of the thesis, we can see that C++ modules did bring the improved reliability we expected at the start. In the case of the standard library, using C++ modules in LLDB resulted in a perfect expression reliability. This outcome was possible thanks to the fact that LLDB already contained a full Clang compiler in its expression evaluator. Once we were able to load AST nodes into Clang, we essentially already had a full-featured C++ programming environment.

Our prototype showed weaknesses when certain details of the source code are hidden in the implementation files of the project. The best example of this are template specializations where our prototype will have no positive effect on the expression reliability. The same will be true with projects that do not contain their declarations inside a C++ module the debugger could load.

C++ modules had undoubtedly a negative impact of the performance of our expression evaluator. However, this slowdown was mostly due to recompiling functions that were already compiled in the executable. This problem is therefore rather a bug in our implementation than being related to the C++ module system itself. Also, the slowdown is barely noticeable due to the interactive nature of debugging and the already very short expression evaluation times.

Also, even without solving this performance problem, C++ modules can already be used as a fallback when the normal expression evaluation fails. This would mean that most users will not experience any performance degradation until they reach an expression which would fail without importing C++ modules.

### 5.1.1 Xmethods as an alternative to C++ modules

We saw that GDB and presumably MVSD provided an alternative approach to using C++ modules in the form of Xmethods (see Section 4.2.2.1). In the evaluation section we tested these Xmethods and found that they did also improve the reliability of the expression evaluation in cases where functions were either unused or inlined. However, the actual increase in reliability was often very limited as only the most important functions of STL containers were provided with an Xmethod.

A simple reason for why not all functions are provided with an Xmethod is that Xmethods are not trivial to implement. They have to reimplement the logic of the actual container which is, in simple cases, only reading a certain value (e.g., for `std::size`) or recompute hashes for specific objects as it is the case for most methods in `std::unordered_map`. In comparison to our approach based on C++ modules, Xmethods require more time to add support for an additional function as every new supported function needs to be reimplemented. In our LLDB prototype adding a new function is essentially free as we only add support for specific language constructs and not support for specific methods. However, Xmethods have the advantage that their initial implementation is simpler than integrating C++ modules and they also work in programs that do not support building with C++ modules.

### 5.1.2 Potential problems with synchronizing debug information and modules

There is one problem we have not discussed so far in this thesis: what happens if sources are modified but the target program is not recompiled? The AST information we get from the C++ modules is gathered by compiling and loading these modules when the compiler needs them. However, the sources we compile might not actually match the sources that were used to compile the target program. It is very likely that a developer modifies a program while debugging to fix any bugs that were found. However, when modifying the source code, the developer also changes the source files we use in our prototype when compiling our C++ modules. Due to this, we could compile C++ modules that are actually a different, newer AST than the one we need to reconstruct for the compiled program. In the best case this difference is not affecting the expression evaluation because all data structures and functions stayed compatible. However, in the worst case these data structures change their binary layout without our debugger noticing it, which would lead to incorrectly evaluated expressions.

There is no real solution to this problem besides pointing out to the user that the sources we use to compile our modules have a newer modification time stamp than the target program we are about to reconstruct.

## 5.2 Revisiting the initial problem formulation

At the start of this thesis we formulated the problem we want to solve by asking five questions that we wanted to answer in this thesis. In this section we try to provide a specific answer to each of these questions.

**Is it feasible to import C++ modules in the expression evaluator of a C++ debugger?**

As we shown with our prototype, it is possible to integrate C++ modules into a C++ debugger. However, the actual work that is needed to make this possible depends heavily on the way the expression evaluator in the debugger is implemented. For LLDB this implementation was relatively straightforward as it already utilized a full

C++ compiler to parse user expressions which could build and import the modules. For debuggers that do not have a real C++ compiler in their expression evaluator, integration of C++ modules would most likely be more difficult. They would first have to implement logic for loading and building C++ modules. Then they would need to extend their custom expression parser to be able to handle all the parsing logic required to make use of the C++ module contents (e.g., template instantiating).

**What are potential problems when trying to add C++ module support to a C++ debugger?**

The biggest problem was related to template specializations in source files and supporting non-default build configurations. Template specializations prevented us from doing our most important feature which is implicitly reconstructing templated declarations by instantiating their original templates.

**Is the debug information commonly emitted by C++ compilers enough to correctly configure the debugger for importing modules?**

The answer for this depends on whether the debugger needs to build the C++ modules or can directly import them.

For building the used C++ modules the compiler needs to reconstruct the build configuration (i.e. defines, include directories and language settings like C++ version or the used C++ dialect). Most of this information is currently not emitted in the debug information generated by current compilers. In our prototype we only had this information available because the `DW_TAG_module` tags in DWARF were extended with LLVM-specific attributes. The debugger also needs a list of imported modules which is possible to be expressed in DWARF.

In this thesis we did not test an approach based on loading the C++ modules that were used during compilation. However, we can assume that a debugger needs significantly less information for just loading an already compiled C++ module compared to building it. The only information that would be needed for Clang is a path to the directories containing the module files and the used modules.

**What kind of declarations and expressions are better supported with C++ modules?**

In general all declarations that are contained in a C++ module will be better supported with integrated C++ modules. Any declarations that are hidden inside a source file or a traditional header file will not become more usable inside the expression evaluator.

Templates and other declarations that are usually optimized away or not fully described benefit the most from this. In our tests we could use templates to the same extent as inside the original program, while in other debuggers meta-programming (i.e., using templates) inside the expression evaluators is often barely or not at all supported. Calling inlined functions is now also reliable possible from the debugger even though they are not present in the executable.

**How do C++ modules effect the performance of the expression evaluation?**

In our prototype C++ modules reduced the performance of the expression evaluation. The main reason for this was due to the way we implemented the merging of AST nodes from a C++ module and debug information. As we gave precedence to the C++ module, we often ended up loading function definitions from the module that we needed to compile and evaluate alongside the user expression. This was significantly slower than just calling the compiled function inside the target program. In theory this redundant work could be avoided by deciding ,while merging AST nodes, if we can call the already compiled function or if we have to compile it on our own.

Another but less important reason for the slowdown was the additional overhead of loading the C++ modules into the expression evaluator. This overhead was included in every expression as we create a new Clang instance for every expression. This could be improved by reusing a Clang instance for multiple expressions.

## 5.3 Future work

This section offers a list of possible improvements to our work and potential future work in this area.

The most obvious place for improvements in our prototype is the expression evaluation performance. As using C++ modules causes in our prototype unnecessary recompilation of imported functions, the best possible approach here would be to determine in the debugger when compiling a function is not necessary. This could be done by looking at the executable and verifying that it contains enough information (i.e. debug information, symbols and compiled code) to receive the function call. An alternative solution would be to automatically use C++ modules to re-evaluate expressions that have failed without C++ modules. This would be much simpler to implement and is less prone to unintentional errors, but would also be slower as the expression needs to be parsed twice.

Another possible way of improving the prototype would be to find a generalized strategy for reconstructing the build configuration of certain translation units. In theory, this information is available to the compiler when it is emitting the debug information, so finding a solution for how to emit this information into the debug information in a DWARF-conform way would help the prototype with supporting more build configurations.

Future work in this area could be implementing and comparing C++ module support in other debuggers. However, this might be very difficult for compilers that do not feature a real compiler as their expression evaluator. It would also be interesting to see how well this approach would translate to other debug information formats, such as Microsoft's PDB format.

# 6
# Related work

This chapter reviews related work which is also either concerned with improving C++ interpreters with C++ modules or using C++ modules to improve debug information. We mainly focus on two related projects: The integration of C++ modules into the Cling C++ interpreter[12][10] and Clang's *module debugging* which optimizes debug information size by bundling it alongside compiled module files[8].

## 6.1 C++ modules in ROOT and Cling

CERN's data analysis framework *ROOT* comes with a C++ interpreter named *Cling*[11]. Cling provides ROOT users with the means do high-performance explorative programming by offering a REPL that can directly call optimized C++ libraries. The architecture of Cling is similar to LLDB. Cling takes the user provided expressions, wraps them inside a function body and then uses Clang to parse and compile them.

The Cling version in ROOT offers an additional functionality called *autoloading*. This allows users to omit the needed `#include` directives when using the REPL and directly type in their expressions. The external code that is necessary to parse the expression is then determined by Cling and loaded into the interpreter on demand. Autoloading is useful for making the interpreter easier to use, but also comes with the added complexity of maintaining a lookup that maps missing declarations to the corresponding external code.

ROOT implemented autoloading by injecting forward declarations of all externally available declarations into the interpreter AST. When Clang tries to access these forward declarations, ROOT interrupts the parsing process and starts a new nested parsing process. This nested parsing would start parsing the header files associated with the used forward declarations. After the nested parsing process is complete, the original expression parsing continues and can use the declarations parsed from the header.

While this implementation is straightforward, it is neither fully correct nor very efficient. For example, autoloading of namespaces is not possible with this approach as it is not possible to make a forward declaration for a namespace. Even if the autoloading works as intended, the external code will be loaded by parsing the whole source file containing the external code, even if only a single declaration from this file is needed.

These problems were resolved in Cling in a similar fashion as in this paper. Cling's custom autoloading mechanism was replaced with Clang's C++ module sys-

tem as explained in this paper by Vassilev[12]. With this new system, Cling loads on startup the C++ modules for all header files which it previously loaded manually. As explained in Section 2.6.2, loading a module does not actually load all the module contents but only creates hooks in the AST which pull in the actual AST nodes when they are needed. The autoloading of declarations is now implicitly supported in Cling by the lazy nature of Clang's AST loading.

The results of this project were a more reliable and often faster loading of external code into the Cling interpreter as illustrated in a paper by Yuka Takahashi[10]. Cling now supported autoloading related to previous problematic declarations such as namespaces. External code could now also be loaded faster, which could reduce the CPU time of short-lived Cling invocations by about 25%.

Compared to this thesis, the work done in Cling was only focused on bringing C++ modules into a traditional interpreter which avoided many of the difficulties related to reconstructing compiled programs. For example, Cling does not have to reconstruct any build configurations to build modules but can rely on the user to manually provide it with the correct information. Cling also did not have to handle two different kind of sources for AST information. In LLDB these two sources are C++ modules and the debug information which are both providing different parts of the final AST.

## 6.2 Module debugging in Clang

When generating debug information for larger C++ projects, the size of the DWARF debug information on disk can often take up several gigabytes. One reason for this is the redundant debug information produced by header files. Every header file that is included in a source file means that the compiler has to emit the debug information for the types introduced by the header files into the generated object file. As header files are usually included several times by different source files, their respective debug information is duplicated in several object files as illustrated in Figure 6.1.

As described in Section 2.3.1, a compiler can not cache the resulting AST nodes of a header because `#include` directives are context sensitive. For the same reason it is also not possible to cache the debug information injected by an `#include` directive. The new semantics of the `import` statements opened up the opportunity to cache the debug information for each module in the same way as the AST nodes themselves. Since the AST that a translation unit sees when referencing a module is always identical, the debug information is therefore also identical and only needs to be generated once.

Clang made use of this with its *Module debugging* feature, which was presented by Prantl and Smith at the 2015 LLVM developers' meeting in San Jose[8]. With Module debugging, the DWARF debug information is bundled alongside the AST in a compiled C++ module file. The actual DWARF debug information in each object file is replaced by references to the respective module files as seen in Figure 6.2. When debugging a program compiled with Modules debugging, LLDB opens the module files and reads the debug information from them. It should be noted that LLDB only reads DWARF debug information and does not read the AST nodes generated by Clang. This is because the Clang AST format is as discussed in Section

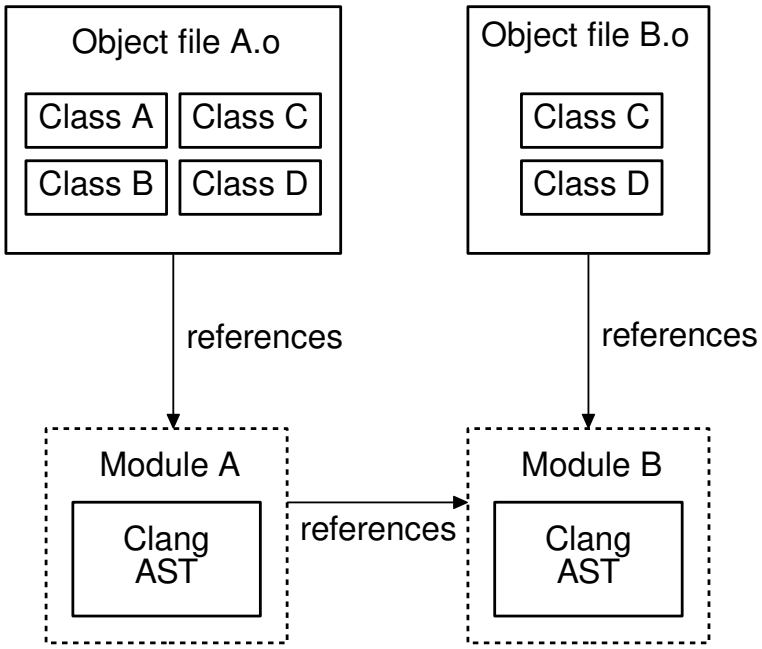


**Figure 6.1:** Debug information organization without Module debugging.

2.6.2 unstable.

This thesis is a logical continuation of the module debugging feature. Both our thesis and module debugging rely on the new semantics introduced with modules to improve the program reconstruction. While module debugging is more focused on reducing redundant debug information, our thesis is focused on improving beyond the debug information by utilizing C++ modules.

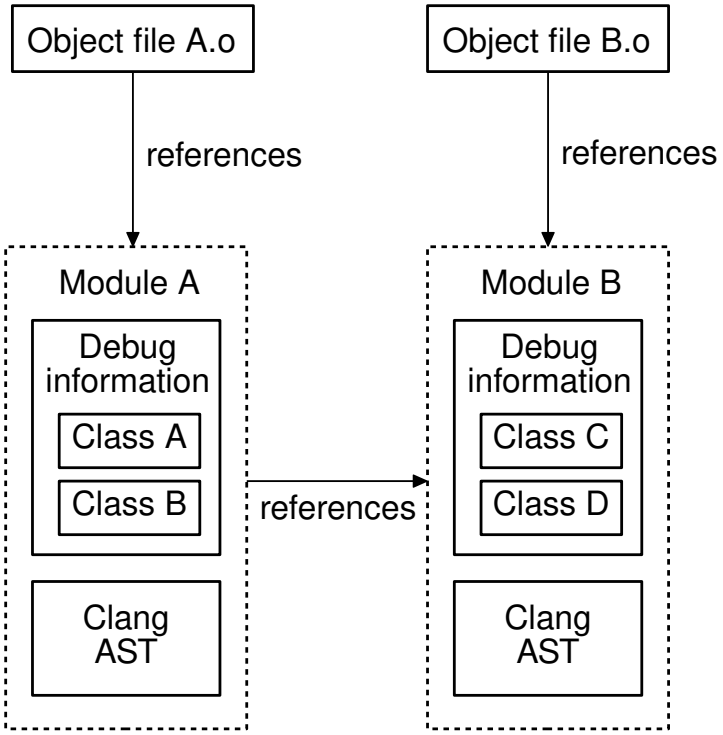


**Figure 6.2:** Debug information organization with Module debugging.

# 7
# Conclusion

Our goal was to explore if, and to what extent, C++ modules can be used to improve program reconstruction in C++ debuggers. We implemented two prototypes based on the LLDB debugger that made use of Clang's C++ module system to reconstruct program information. The first prototype only loaded the C++ standard library module and used it to improve expression evaluation when interacting with the container classes from the standard library. As the standard library module only provided templates, this prototype had to manually instantiate these templates to reconstruct the actual types found in user programs.

During our evaluation we found that our first prototype drastically improved the reliability of LLDB's expression evaluator when debugging code that made use of the standard library. However, using C++ modules instead of debug information turned out to be slower in our implementation.

As the first prototype worked as intended, we implemented a second prototype which loaded all user-defined and system C++ modules used by the target program. Our second prototype was also able to improve the reliability of the expression evaluator, but the actual impact on real world programs depends on the structure of the project. Programs that have the majority of their types and templates inside C++ modules will have a improved expression evaluation reliability, while program that keep their types hidden inside implementation files will not benefit at all. Also correctly reconstructing the original build flags of the modules can be problematic, which means that projects using uncommon build configurations may not be supported by our approach. The performance results were identical to the first prototype. This means that C++ modules may not be a good replacement for debug information-based expression evaluation, but certainly can be used as a fallback in cases where debug information can not reconstruct enough of the program.

In the end, the future of using C++ modules in debuggers mainly depends on the widespread adoption of C++ modules itself. Without them being supported by compilers and libraries, any attempt to utilize them in debuggers will be in vain. With the recent standardization of C++ modules and the current efforts to implement them in the available C++ compilers, the only limiting factor is how quickly users can migrate their source code and build infrastructure.

# Bibliography


[1] DWARF Debugging Information Format Committee et al. *DWARF debugging information format, version 4*. 2010.

[2] DWARF Debugging Information Format Committee et al. *DWARF debugging information format, version 5*. 2017.

[3] *Debugging with GDB : Xmethods in Python*. https://sourceware.org/gdb/current/onlinedocs/gdb/Xmethods-In-Python.html#Xmethods-In-Python. Accessed: 2019-05-15.

[4] Michael J Eager et al. *Introduction to the dwarf debugging format*. 2007.

[5] ISO. *ISO/IEC 14882:2017 Information technology - Programming languages - C++*. Fifth. Dec. 2017, p. 1605. URL: https://www.iso.org/standard/68564.html.

[6] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society. 2004, p. 75.

[7] Alex Lorenz and Michael Spencer. "clang-scan-deps: Fast dependency scanning for explicit modules". 2019 European LLVM Developers Meeting. 2019. URL: https://llvm.org/devmtg/2019-04/slides/TechTalk-Lorenz-clang-scan-deps_Fast_dependency_scanning_for_explicit_modules.pdf.

[8] Adrian Prantl and Duncan Exon Smith. "Debug Information - From Metadata to Modules". LLVM Developers' Meeting 2015, San Jose. 2015. URL: https://llvm.org/devmtg/2015-10/slides/Prantl-ExonSmith-DebugInfoMetadataToModules.pdf.

[9] Richard Stallman, Roland Pesch, Stan Shebs, et al. "Debugging with GDB". In: *Free Software Foundation* 51 (2002).

[10] Yuka Takahashi et al. *Optimizing Frameworks Performance Using C++ Modules Aware ROOT*. 2018.

[11] V Vasilev et al. "Cling – The New Interactive Interpreter for ROOT 6". In: *Journal of Physics: Conference Series* 396.5 (Dec. 2012). DOI: 10.1088/1742-6596/396/5/052071. URL: https://doi.org/10.1088%2F1742-6596%2F396%2F5%2F052071.

[12] Vassil Vassilev. "IOP: Optimizing ROOT's performance using C++ Modules". In: *J. Phys.: Conf. Ser.* Vol. 898. 2017, p. 072023.

# A

# Appendix 1 - std module evaluation

The following source code listings and tables describe the tests for evaluating our standard module prototype. There exists always two tests for each standard library container. One test where the container is unused in the original program and one where all functions called in the expression evaluator are used in the program. The source code listing provides the program we compiled. Each program contains a source line marked with a comment that describes where we stopped the execution and started to evaluate our list of expressions. This list of expressions can be found in the table below each source code listing. Each table contains the result of evaluating the expression in the current row with different compilers. If the evaluation is successful, we mark the test with a ✓ sign. Otherwise we mark the test with a ✗ sign. If the test passes but only with the workarounds described in Section 4.2.1.1, we mark the test with a ✓* sign.

Source code for testing unused std::vector

```
#include <vector>

int main() {
  std::vector<int> v = {3, 2, 1};
  return 0; // We evaluate expressions here.
}
```

| Evaluation success rate for unused std::vector | | | | |
|---|---|---|---|---|
| **Expression** | **MVSD** | **GDB** | **LLDB** | **LLDB with C++ modules** |
| `v.data()` | ✗ | ✗ | ✗ | ✓* |
| `v[0]` | ✓ | ✓ | ✗ | ✓ |
| `v.front()` | ✗ | ✓ | ✗ | ✓ |
| `v.back()` | ✗ | ✓ | ✗ | ✓ |
| `v.push_back(1)` | ✗ | ✗ | ✗ | ✓ |
| `v.emplace_back(1)` | ✗ | ✗ | ✗ | ✓ |
| `v.at(0)` | ✗ | ✓ | ✗ | ✓ |
| `v.begin()` | ✗ | ✗ | ✗ | ✓* |
| `v.cbegin()` | ✗ | ✗ | ✗ | ✓* |
| `v.end()` | ✗ | ✗ | ✗ | ✓* |
| `v.cend()` | ✗ | ✗ | ✗ | ✓* |
| `v.rbegin()` | ✗ | ✗ | ✗ | ✓* |
| `v.crbegin()` | ✗ | ✗ | ✗ | ✓* |
| `v.rend()` | ✗ | ✗ | ✗ | ✓* |
| `v.crend()` | ✗ | ✗ | ✗ | ✓* |
| `v.empty()` | ✗ | ✓ | ✗ | ✓ |
| `v.size()` | ✓ | ✓ | ✗ | ✓ |
| `v.max_size()` | ✓ | ✗ | ✗ | ✓ |
| `v.reserve(20)` | ✗ | ✗ | ✗ | ✓ |
| `v.capacity()` | ✓ | ✗ | ✗ | ✓ |
| `v.shrink_to_fit()` | ✗ | ✗ | ✗ | ✓ |
| `v.clear()` | ✗ | ✗ | ✗ | ✓ |
| `v.insert(v.end(), 4)` | ✗ | ✗ | ✗ | ✓* |
| `v.erase(v.end())` | ✗ | ✗ | ✗ | ✓* |
| `v.pop_back()` | ✗ | ✗ | ✗ | ✓ |
| `v.resize(10)` | ✗ | ✗ | ✗ | ✓ |
| Success rate | 15 % | 23 % | 0 % | 100 % |

Source code for testing fully used std::vector

```
#include <vector>
int main() {
  std::vector<int> v = {3, 2, 1};
  v.front(); // We evaluate expressions here.
  v.at(0); v[0]; v.back(); v.data(); v.resize(20);
  v.begin(); v.end(); v.cbegin(); v.cend(); v.crbegin(); v.crend();
  v.empty(); v.size(); v.max_size(); v.reserve(20);
  v.capacity(); v.shrink_to_fit(); v.clear(); v.insert(v.begin(), 1);
  v.emplace_back(1); v.erase(v.begin()); v.pop_back(); v.push_back(1);
}
```

| **Evaluation success rate for fully used std::vector** | | | | |
|---|---|---|---|---|
| **Expression** | **MVSD** | **GDB** | **LLDB** | **LLDB with C++ modules** |
| `v.data()` | ✗ | ✓ | ✗ | ✓* |
| `v[0]` | ✓ | ✓ | ✗ | ✓ |
| `v.front()` | ✗ | ✓ | ✗ | ✓ |
| `v.back()` | ✓ | ✓ | ✗ | ✓ |
| `v.push_back(1)` | ✗ | ✗ | ✗ | ✓ |
| `v.emplace_back(1)` | ✗ | ✗ | ✗ | ✓ |
| `v.at(0)` | ✓ | ✓ | ✓ | ✓ |
| `v.begin()` | ✗ | ✓ | ✗ | ✓* |
| `v.cbegin()` | ✓ | ✓ | ✗ | ✓* |
| `v.end()` | ✓ | ✓ | ✗ | ✓* |
| `v.cend()` | ✓ | ✓ | ✗ | ✓* |
| `v.rbegin()` | ✓ | ✓ | ✗ | ✓* |
| `v.crbegin()` | ✓ | ✓ | ✗ | ✓* |
| `v.rend()` | ✓ | ✓ | ✗ | ✓* |
| `v.crend()` | ✓ | ✓ | ✗ | ✓* |
| `v.empty()` | ✓ | ✓ | ✗ | ✓ |
| `v.size()` | ✓ | ✓ | ✗ | ✓ |
| `v.max_size()` | ✓ | ✓ | ✓ | ✓ |
| `v.reserve(20)` | ✓ | ✓ | ✓ | ✓ |
| `v.capacity()` | ✓ | ✓ | ✗ | ✓ |
| `v.shrink_to_fit()` | ✓ | ✓ | ✓ | ✓ |
| `v.clear()` | ✗ | ✓ | ✗ | ✓ |
| `v.insert(v.end(), 4)` | ✓ | ✗ | ✗ | ✓* |
| `v.erase(v.end())` | ✓ | ✗ | ✗ | ✓* |
| `v.pop_back()` | ✓ | ✓ | ✗ | ✓ |
| `v.resize(10)` | ✓ | ✓ | ✓ | ✓ |
| Success rate | 77 % | 85 % | 19 % | 100 % |

Source code for testing unused std::deque

```
#include <deque>

int main() {
  std::deque<int> d = {3, 2, 1};
  return 0; // We evaluate expressions here.
}
```

| Evaluation success rate for unused std::deque | | | | |
|---|---|---|---|---|
| **Expression** | **MVSD** | **GDB** | **LLDB** | **LLDB with C++ modules** |
| `d.front()` | ✗ | ✓ | ✗ | ✓ |
| `d.back()` | ✗ | ✓ | ✗ | ✓ |
| `d.push_back(1)` | ✗ | ✗ | ✗ | ✓ |
| `d.emplace_back(1)` | ✗ | ✗ | ✗ | ✓ |
| `d[0]` | ✓ | ✓ | ✗ | ✓ |
| `d.at(0)` | ✗ | ✓ | ✗ | ✓ |
| `d.begin()` | ✗ | ✓ | ✗ | ✓ |
| `d.cbegin()` | ✗ | ✗ | ✗ | ✓* |
| `d.end()` | ✗ | ✓ | ✗ | ✓* |
| `d.cend()` | ✗ | ✗ | ✗ | ✓* |
| `d.rbegin()` | ✗ | ✗ | ✗ | ✓* |
| `d.crbegin()` | ✗ | ✗ | ✗ | ✓* |
| `d.rend()` | ✗ | ✗ | ✗ | ✓* |
| `d.crend()` | ✗ | ✗ | ✗ | ✓* |
| `d.empty()` | ✓ | ✓ | ✗ | ✓ |
| `d.size()` | ✓ | ✓ | ✗ | ✓ |
| `d.max_size()` | ✓ | ✗ | ✗ | ✓ |
| `d.clear()` | ✗ | ✗ | ✗ | ✓ |
| `d.insert(d.end(), 4)` | ✗ | ✗ | ✗ | ✓* |
| `d.erase(d.end())` | ✗ | ✗ | ✗ | ✓* |
| `d.pop_back()` | ✓ | ✗ | ✗ | ✓ |
| `d.resize(10)` | ✗ | ✗ | ✗ | ✓ |
| Success rate | 23 % | 36 % | 0 % | 100 % |

Source code for testing fully used std::deque

```
#include <deque>

int main() {
  std::deque<int> d = {3, 2, 1};
  d.front(); // We evaluate expressions here.
  d.back(); d.push_back(1); d.emplace_back(1); d[0]; d.at(0);
  d.begin(); d.cbegin(); d.end(); d.cend(); d.rbegin(); d.crbegin();
  d.rend(); d.crend(); d.empty(); d.size(); d.max_size(); d.clear();
  d.insert(d.end(), 4); d.erase(d.end()); d.pop_back(); d.resize(10);
}
```

| **Evaluation success rate for fully used std::deque** | | | | |
|---|---|---|---|---|
| **Expression** | **MVSD** | **GDB** | **LLDB** | **LLDB with C++ modules** |
| `d.front()` | ✗ | ✓ | ✗ | ✓ |
| `d.back()` | ✓ | ✓ | ✗ | ✓ |
| `d.push_back(1)` | ✗ | ✗ | ✓ | ✓ |
| `d.emplace_back(1)` | ✗ | ✗ | ✗ | ✓ |
| `d[0]` | ✓ | ✓ | ✗ | ✓ |
| `d.at(0)` | ✓ | ✓ | ✗ | ✓ |
| `d.begin()` | ✓ | ✓ | ✗ | ✓ |
| `d.cbegin()` | ✓ | ✓ | ✗ | ✓* |
| `d.end()` | ✓ | ✓ | ✗ | ✓* |
| `d.cend()` | ✓ | ✓ | ✗ | ✓* |
| `d.rbegin()` | ✓ | ✓ | ✗ | ✓* |
| `d.crbegin()` | ✓ | ✓ | ✗ | ✓* |
| `d.rend()` | ✓ | ✓ | ✗ | ✓* |
| `d.crend()` | ✓ | ✓ | ✗ | ✓* |
| `d.empty()` | ✓ | ✓ | ✗ | ✓ |
| `d.size()` | ✓ | ✓ | ✗ | ✓ |
| `d.max_size()` | ✓ | ✓ | ✗ | ✓ |
| `d.clear()` | ✓ | ✓ | ✗ | ✓ |
| `d.insert(d.end(), 4)` | ✗ | ✗ | ✗ | ✓* |
| `d.erase(d.end())` | ✗ | ✗ | ✗ | ✓* |
| `d.pop_back()` | ✓ | ✓ | ✗ | ✓ |
| `d.resize(10)` | ✓ | ✓ | ✓ | ✓ |
| Success rate | 77 % | 82 % | 9 % | 100 % |

Source code for testing unused std::stack

```
#include <stack>

int main() {
  std::stack<int> s; s.push(1); s.push(2);
  return 0; // We evaluate expressions here.
}
```

| Evaluation success rate for unused std::stack | | | | |
|---|---|---|---|---|
| **Expression** | **MVSD** | **GDB** | **LLDB** | **LLDB with C++ modules** |
| `s.top()` | ✗ | ✗ | ✗ | ✓ |
| `s.empty()` | ✗ | ✗ | ✗ | ✓ |
| `s.size()` | ✗ | ✗ | ✗ | ✓ |
| `s.push(1)` | ✗ | ✗ | ✗ | ✓ |
| `s.emplace(1)` | ✗ | ✗ | ✗ | ✓ |
| `s.pop()` | ✗ | ✗ | ✗ | ✓ |
| Success rate | 0 % | 0 % | 0 % | 100 % |

Source code for testing fully used std::stack

```
#include <stack>

int main() {
  std::stack<int> s; s.push(1); s.push(2);
  s.top(); // We evaluate expressions here.
  s.empty();
  s.size();
  s.push(1);
  s.emplace(1);
  s.pop();
  return 0;
}
```

| **Evaluation success rate for fully used std::stack** | | | | |
|---|---|---|---|---|
| **Expression** | **MVSD** | **GDB** | **LLDB** | **LLDB with C++ modules** |
| `s.top()` | ✓ | ✓ | ✗ | ✓ |
| `s.empty()` | ✓ | ✓ | ✗ | ✓ |
| `s.size()` | ✓ | ✓ | ✗ | ✓ |
| `s.push(1)` | ✗ | ✗ | ✗ | ✓ |
| `s.emplace(1)` | ✗ | ✗ | ✗ | ✓ |
| `s.pop()` | ✗ | ✓ | ✗ | ✓ |
| Success rate | 50 % | 67 % | 0 % | 100 % |

Source code for testing unused std::queue

```
#include <queue>

int main() {
  std::queue<int> q; q.push(1); q.push(2);
  return 0; // We evaluate expressions here.
}
```

| **Evaluation success rate for unused std::queue** | | | | |
|---|---|---|---|---|
| **Expression** | **MVSD** | **GDB** | **LLDB** | **LLDB with C++ modules** |
| `q.front()` | ✗ | ✗ | ✗ | ✓ |
| `q.back()` | ✗ | ✗ | ✗ | ✓ |
| `q.empty()` | ✗ | ✗ | ✗ | ✓ |
| `q.size()` | ✗ | ✗ | ✗ | ✓ |
| `q.push(1)` | ✗ | ✗ | ✗ | ✓ |
| `q.emplace(1)` | ✗ | ✗ | ✗ | ✓ |
| `q.pop()` | ✗ | ✗ | ✗ | ✓ |
| Success rate | 0 % | 0 % | 0 % | 100 % |

Source code for testing fully used std::queue

```
#include <queue>

int main() {
  std::queue<int> q; q.push(1); q.push(2);
  q.front(); // We evaluate expressions here.
  q.back();
  q.empty();
  q.size();
  q.push(1);
  q.emplace(1);
  q.pop();
  return 0;
}
```

| **Evaluation success rate for fully used std::queue** | | | | |
|---|---|---|---|---|
| **Expression** | **MVSD** | **GDB** | **LLDB** | **LLDB with C++ modules** |
| `q.front()` | ✗ | ✓ | ✓ | ✓ |
| `q.back()` | ✓ | ✓ | ✓ | ✓ |
| `q.empty()` | ✓ | ✓ | ✓ | ✓ |
| `q.size()` | ✓ | ✓ | ✓ | ✓ |
| `q.push(1)` | ✗ | ✗ | ✓ | ✓ |
| `q.emplace(1)` | ✗ | ✗ | ✗ | ✓ |
| `q.pop()` | ✗ | ✓ | ✓ | ✓ |
| Success rate | 43 % | 71 % | 86 % | 100 % |

Source code for testing unused std::forward_list

```
#include <forward_list>

int main() {
  std::forward_list<int> f = {1};
  return 0; // We evaluate expressions here.
}
```

| Evaluation success rate for unused std::forward_list | | | | |
|---|---|---|---|---|
| **Expression** | **MVSD** | **GDB** | **LLDB** | **LLDB with C++ modules** |
| `f.front()` | ✗ | ✓ | ✗ | ✓ |
| `f.before_begin()` | ✗ | ✗ | ✗ | ✓* |
| `f.cbefore_begin()` | ✗ | ✗ | ✗ | ✓* |
| `f.begin()` | ✗ | ✗ | ✗ | ✓* |
| `f.cbegin()` | ✗ | ✗ | ✗ | ✓* |
| `f.end()` | ✗ | ✗ | ✗ | ✓* |
| `f.cend()` | ✗ | ✗ | ✗ | ✓* |
| `f.empty()` | ✗ | ✗ | ✗ | ✓ |
| `f.max_size()` | ✗ | ✗ | ✗ | ✓ |
| `f.insert_after(f.begin(), 1)` | ✗ | ✗ | ✗ | ✓* |
| `f.emplace_after(f.begin(), 1)` | ✗ | ✗ | ✗ | ✓* |
| `f.erase_after(f.begin())` | ✗ | ✗ | ✗ | ✓* |
| `f.push_front(1)` | ✗ | ✗ | ✗ | ✓ |
| `f.emplace_front(1)` | ✗ | ✗ | ✗ | ✓ |
| `f.pop_front()` | ✗ | ✗ | ✗ | ✓ |
| `f.resize(20)` | ✗ | ✗ | ✗ | ✓ |
| `f.clear()` | ✗ | ✗ | ✗ | ✓ |
| `f.remove(1)` | ✗ | ✗ | ✗ | ✓ |
| `f.remove_if([](int){return 1;});` | ✗ | ✗ | ✗ | ✓ |
| `f.reverse()` | ✗ | ✗ | ✗ | ✓ |
| `f.unique()` | ✗ | ✗ | ✗ | ✓ |
| `f.sort()` | ✗ | ✗ | ✗ | ✓ |
| Success rate | 0 % | 5 % | 0 % | 100 % |

Source code for testing fully used std::forward_list

```
#include <forward_list>

int main() {
  std::forward_list<int> f = {1};
  f.front(); // We evaluate expressions here.
  f.before_begin(); f.cbefore_begin(); f.begin(); f.cbegin();
  f.end(); f.cend(); f.empty(); f.max_size(); f.insert_after(f.begin(), 1);
  f.emplace_after(f.begin(), 1); f.erase_after(f.begin()); f.push_front(1);
  f.emplace_front(1); f.pop_front(); f.resize(20); f.clear(); f.remove(1);
  f.remove_if([](int n){ return n; }); f.reverse(); f.unique(); f.sort();
  return 0;
}
```

| Evaluation success rate for fully used std::forward_list | | | | |
|---|---|---|---|---|
| **Expression** | **MVSD** | **GDB** | **LLDB** | **LLDB with C++ modules** |
| `f.front()` | ✗ | ✓ | ✗ | ✓ |
| `f.before_begin()` | ✓ | ✓ | ✗ | ✓* |
| `f.cbefore_begin()` | ✓ | ✗ | ✗ | ✓* |
| `f.begin()` | ✓ | ✓ | ✗ | ✓* |
| `f.cbegin()` | ✓ | ✗ | ✗ | ✓* |
| `f.end()` | ✓ | ✓ | ✗ | ✓* |
| `f.cend()` | ✓ | ✓ | ✗ | ✓* |
| `f.empty()` | ✓ | ✓ | ✗ | ✓ |
| `f.max_size()` | ✓ | ✓ | ✗ | ✓ |
| `f.insert_after(f.begin(), 1)` | ✗ | ✗ | ✗ | ✓* |
| `f.emplace_after(f.begin(), 1)` | ✗ | ✗ | ✗ | ✓* |
| `f.erase_after(f.begin())` | ✗ | ✗ | ✗ | ✓* |
| `f.push_front(1)` | ✗ | ✗ | ✓ | ✓ |
| `f.emplace_front(1)` | ✗ | ✗ | ✗ | ✓ |
| `f.pop_front()` | ✗ | ✓ | ✓ | ✓ |
| `f.resize(20)` | ✓ | ✓ | ✗ | ✓ |
| `f.clear()` | ✓ | ✓ | ✗ | ✓ |
| `f.remove(1)` | ✓ | ✗ | ✓ | ✓ |
| `f.remove_if([](int n){return n;});` | ✗ | ✓ | ✗ | ✓ |
| `f.reverse()` | ✓ | ✓ | ✓ | ✓ |
| `f.unique()` | ✓ | ✓ | ✗ | ✓ |
| `f.sort()` | ✓ | ✓ | ✗ | ✓ |
| Success rate | 64 % | 64 % | 18 % | 100 % |

Source code for testing unused std::array

```
#include <array>

int main() {
  std::array<int, 3> a = {3, 2, 1};
  return 0; // We evaluate expressions here.
}
```

| Evaluation success rate for unused std::array | | | | |
|---|---|---|---|---|
| **Expression** | **MVSD** | **GDB** | **LLDB** | **LLDB with C++ modules** |
| `a.at(0)` | ✗ | ✓ | ✗ | ✓ |
| `a[0]` | ✗ | ✓ | ✗ | ✓ |
| `a.front()` | ✗ | ✓ | ✗ | ✓ |
| `a.back()` | ✗ | ✓ | ✗ | ✓ |
| `a.data()` | ✗ | ✗ | ✗ | ✓* |
| `a.begin()` | ✗ | ✗ | ✗ | ✓* |
| `a.end()` | ✗ | ✗ | ✗ | ✓* |
| `a.cbegin()` | ✗ | ✗ | ✗ | ✓* |
| `a.cend()` | ✗ | ✗ | ✗ | ✓* |
| `a.rbegin()` | ✗ | ✗ | ✗ | ✓* |
| `a.rend()` | ✗ | ✗ | ✗ | ✓* |
| `a.crbegin()` | ✗ | ✗ | ✗ | ✓* |
| `a.crend()` | ✗ | ✗ | ✗ | ✓* |
| `a.empty()` | ✗ | ✓ | ✗ | ✓ |
| `a.size()` | ✗ | ✓ | ✗ | ✓ |
| `a.max_size()` | ✗ | ✗ | ✗ | ✓ |
| `a.fill(0)` | ✗ | ✗ | ✗ | ✓ |
| Success rate | 0 % | 35 % | 0 % | 100 % |

Source code for testing fully used std::array

```
#include <array>

int main() {
  std::array<int, 3> a = {3, 2, 1};
  a.at(0); a[0]; a.front(); a.back(); a.begin(); a.end();
  a.cbegin(); a.cend(); a.rbegin(); a.rend(); a.crbegin();
  a.crend(); a.empty(); a.size(); a.max_size(); a.fill(0);
  return 0; // We evaluate expressions here.
}
```

| **Evaluation success rate for fully used std::array** | | | | |
|---|---|---|---|---|
| **Expression** | **MVSD** | **GDB** | **LLDB** | **LLDB with C++ modules** |
| `a.at(0)` | ✗ | ✓ | ✓ | ✓ |
| `a[0]` | ✓ | ✓ | ✓ | ✓ |
| `a.front()` | ✗ | ✓ | ✓ | ✓ |
| `a.back()` | ✗ | ✓ | ✓ | ✓ |
| `(int*)a.data()` | ✗ | ✓ | ✓ | ✓* |
| `a.begin()` | ✓ | ✓ | ✓ | ✓* |
| `a.end()` | ✓ | ✓ | ✓ | ✓* |
| `a.cbegin()` | ✓ | ✓ | ✓ | ✓* |
| `a.cend()` | ✓ | ✓ | ✓ | ✓* |
| `a.rbegin()` | ✓ | ✓ | ✓ | ✓* |
| `a.rend()` | ✓ | ✓ | ✓ | ✓* |
| `a.crbegin()` | ✓ | ✓ | ✓ | ✓* |
| `a.crend()` | ✓ | ✓ | ✓ | ✓* |
| `a.empty()` | ✓ | ✓ | ✓ | ✓ |
| `a.size()` | ✓ | ✓ | ✓ | ✓ |
| `a.max_size()` | ✓ | ✓ | ✓ | ✓ |
| `a.fill(0)` | ✓ | ✗ | ✓ | ✓ |
| Success rate | 76 % | 94 % | 100 % | 100 % |

Source code for testing unused std::unique_ptr

```
#include <memory>

int main() {
  std::unique_ptr<int> u(new int(1));
  return 0; // We evaluate expressions here.
}
```

| **Evaluation success rate for unused std::unique_ptr** | | | | |
|---|---|---|---|---|
| **Expression** | **MVSD** | **GDB** | **LLDB** | **LLDB with C++ modules** |
| `u.get()` | ✓ | ✓ | ✗ | ✓ |
| `u.reset(0)` | ✗ | ✗ | ✓ | ✓ |
| `*u` | ✓ | ✗ | ✗ | ✓ |
| `u.release()` | ✗ | ✓ | ✗ | ✓ |
| `u.get_deleter()` | ✓ | ✓ | ✗ | ✓ |
| `(bool)u` | ✗ | ✓ | ✗ | ✓ |
| Success rate | 50 % | 67 % | 17 % | 100 % |

Source code for testing fully used std::unique_ptr

```
#include <memory>

int main() {
  std::unique_ptr<int> u(new int(1));
  u.get(); // We evaluate expressions here.
  u.reset();
  u.release();
  u.get_deleter();
  (bool)u;
  *u;
  return 0;
}
```

| Evaluation success rate for fully used std::unique_ptr | | | | |
|---|---|---|---|---|
| **Expression** | **MVSD** | **GDB** | **LLDB** | **LLDB with C++ modules** |
| `u.get()` | ✓ | ✓ | ✓ | ✓ |
| `u.reset(0)` | ✓ | ✓ | ✓ | ✓ |
| `*u` | ✓ | ✓ | ✓ | ✓ |
| `u.release()` | ✓ | ✓ | ✓ | ✓ |
| `u.get_deleter()` | ✓ | ✓ | ✓ | ✓ |
| `(bool)u` | ✗ | ✓ | ✓ | ✓ |
| Success rate | 83 % | 100 % | 100 % | 100 % |

Source code for testing unused std::shared_ptr

```
#include <memory>

int main() {
  std::shared_ptr<int> u(new int(1));
  return 0; // We evaluate expressions here.
}
```

| **Evaluation success rate for unused std::shared_ptr** | | | | |
|---|---|---|---|---|
| **Expression** | **MVSD** | **GDB** | **LLDB** | **LLDB with C++ modules** |
| `u.get()` | ✗ | ✓ | ✗ | ✓ |
| `*u` | ✓ | ✓ | ✗ | ✓ |
| `u.reset()` | ✗ | ✗ | ✗ | ✓ |
| `u.use_count()` | ✗ | ✗ | ✗ | ✓ |
| `(bool)u` | ✗ | ✓ | ✗ | ✓ |
| `u.owner_before(u)` | ✗ | ✗ | ✗ | ✓ |
| Success rate | 17 % | 50 % | 0 % | 100 % |

Source code for testing fully used std::shared_ptr

```
#include <memory>

int main() {
  std::shared_ptr<int> u(new int(1));
  u.reset(); // We evaluate expressions here.
  u.get();
  u.use_count();
  (bool)u;
  u.owner_before(u);
  return 0;
}
```

| Evaluation success rate for fully used std::shared_ptr | | | | |
|---|---|---|---|---|
| **Expression** | **MVSD** | **GDB** | **LLDB** | **LLDB with C++ modules** |
| `u.get()` | ✓ | ✓ | ✓ | ✓ |
| `*u` | ✓ | ✓ | ✗ | ✓ |
| `u.reset()` | ✓ | ✓ | ✓ | ✓ |
| `u.use_count()` | ✓ | ✗ | ✓ | ✓ |
| `(bool)u` | ✗ | ✓ | ✓ | ✓ |
| `u.owner_before(u)` | ✗ | ✗ | ✗ | ✓ |
| Success rate | 67 % | 67 % | 67 % | 100 % |

Source code for testing unused std::weak_ptr

```
#include <memory>

int main() {
  auto sp = std::make_shared<int>(42); std::weak_ptr<int> u = sp;
  return 0; // We evaluate expressions here.
}
```

| Evaluation success rate for unused std::weak_ptr | | | | |
|---|---|---|---|---|
| **Expression** | **MVSD** | **GDB** | **LLDB** | **LLDB with C++ modules** |
| `u.expired()` | ✗ | ✗ | ✗ | ✓ |
| `u.owner_before(u)` | ✗ | ✗ | ✗ | ✓ |
| `(bool)u.lock()` | ✗ | ✗ | ✗ | ✓ |
| `u.reset()` | ✗ | ✗ | ✗ | ✓ |
| Success rate | 0 % | 0 % | 0 % | 100 % |

Source code for testing fully used std::weak_ptr

```
#include <memory>

int main() {
  auto sp = std::make_shared<int>(42); std::weak_ptr<int> u = sp;
  u.expired(); // We evaluate expressions here.
  u.owner_before(u);
  (bool)u.lock();
  u.reset();
  return 0;
}
```

| Evaluation success rate for fully used std::weak_ptr | | | | |
|---|---|---|---|---|
| **Expression** | **MVSD** | **GDB** | **LLDB** | **LLDB with C++ modules** |
| `u.expired()` | ✓ | ✓ | ✓ | ✓ |
| `u.owner_before(u)` | ✗ | ✗ | ✗ | ✓ |
| `(bool)u.lock()` | ✓ | ✓ | ✓ | ✓ |
| `u.reset()` | ✓ | ✓ | ✓ | ✓ |
| Success rate | 75 % | 75 % | 75 % | 100 % |

Source code for testing unused std::map

```
#include <map>

int main() {
  std::map<int, int> m = {{1, 2}, {2, 4}};
  return 0; // We evaluate expressions here.
}
```

| Evaluation success rate for unused std::map | | | | |
|---|---|---|---|---|
| **Expression** | **MVSD** | **GDB** | **LLDB** | **LLDB with C++ modules** |
| `m[1]` | ✗ | ✗ | ✗ | ✓ |
| `m.at(1)` | ✗ | ✗ | ✗ | ✓ |
| `m.begin()` | ✓ | ✗ | ✗ | ✓* |
| `m.end()` | ✓ | ✗ | ✗ | ✓* |
| `m.cbegin()` | ✗ | ✗ | ✗ | ✓* |
| `m.cend()` | ✗ | ✗ | ✗ | ✓* |
| `m.rbegin()` | ✗ | ✗ | ✗ | ✓* |
| `m.rend()` | ✗ | ✗ | ✗ | ✓* |
| `m.crbegin()` | ✗ | ✗ | ✗ | ✓* |
| `m.crend()` | ✗ | ✗ | ✗ | ✓* |
| `m.empty()` | ✗ | ✓ | ✗ | ✓ |
| `m.size()` | ✓ | ✓ | ✗ | ✓ |
| `m.max_size()` | ✓ | ✗ | ✗ | ✓ |
| `m.clear()` | ✗ | ✗ | ✗ | ✓ |
| `m.insert(1, 1);` | ✗ | ✗ | ✗ | ✓* |
| `m.emplace(1, 1);` | ✗ | ✗ | ✗ | ✓* |
| `m.erase(m.begin())` | ✗ | ✗ | ✗ | ✓* |
| `m.equal_range(1)` | ✗ | ✗ | ✗ | ✓* |
| `m.lower_bound(1)` | ✗ | ✗ | ✗ | ✓* |
| `m.upper_bound(1)` | ✗ | ✗ | ✗ | ✓* |
| `m.count(1)` | ✗ | ✗ | ✗ | ✓ |
| `m.find(1)` | ✗ | ✗ | ✗ | ✓* |
| Success rate | 18 % | 9 % | 0 % | 100 % |

Source code for testing used std::map

```
#include <map>

int main() {
  std::map<int, int> m = {{1, 2}, {2, 4}};
  // We evaluate expressions here.
  m[1]; m.at(1); m.begin(); m.end(); m.cbegin(); m.cend(); m.rbegin();
  m.rend(); m.crbegin(); m.crend(); m.empty(); m.size(); m.max_size();
  m.clear(); m.insert({1, 1}); m.emplace(1, 1); m.erase(m.begin());
  m.equal_range(1); m.lower_bound(1); m.upper_bound(1);
  m.count(1); m.find(1);
}
```

**Evaluation success rate for used std::map**

| Expression | MVSD | GDB | LLDB | LLDB with C++ modules |
|---|---|---|---|---|
| `m[1]` | ✗ | ✗ | ✓ | ✓ |
| `m.at(1)` | ✗ | ✗ | ✓ | ✓ |
| `m.begin()` | ✓ | ✓ | ✓ | ✓* |
| `m.end()` | ✓ | ✓ | ✓ | ✓* |
| `m.cbegin()` | ✓ | ✓ | ✓ | ✓* |
| `m.cend()` | ✓ | ✓ | ✓ | ✓* |
| `m.rbegin()` | ✓ | ✓ | ✓ | ✓* |
| `m.rend()` | ✓ | ✓ | ✓ | ✓* |
| `m.crbegin()` | ✓ | ✓ | ✓ | ✓* |
| `m.crend()` | ✓ | ✓ | ✓ | ✓* |
| `m.empty()` | ✓ | ✓ | ✓ | ✓ |
| `m.size()` | ✓ | ✓ | ✓ | ✓ |
| `m.max_size()` | ✓ | ✓ | ✓ | ✓ |
| `m.clear()` | ✗ | ✓ | ✓ | ✓ |
| `m.insert(1, 1);` | ✗ | ✗ | ✓ | ✓* |
| `m.emplace(1, 1);` | ✗ | ✗ | ✗ | ✓* |
| `m.erase(m.begin())` | ✗ | ✗ | ✓ | ✓* |
| `m.equal_range(1)` | ✓ | ✗ | ✓ | ✓* |
| `m.lower_bound(1)` | ✓ | ✗ | ✓ | ✓* |
| `m.upper_bound(1)` | ✓ | ✗ | ✓ | ✓* |
| `m.count(1)` | ✓ | ✗ | ✓ | ✓ |
| `m.find(1)` | ✓ | ✗ | ✓ | ✓* |
| Success rate | 73 % | 55 % | 95 % | 100 % |

Source code for testing unused std::unordered_map

```
#include <unordered_map>

int main() {
  std::unordered_map<int, int> m = {{1, 2}, {2, 4}};
  return 0; // We evaluate expressions here.
}
```

| Evaluation success rate for unused std::unordered_map | | | | |
|---|---|---|---|---|
| **Expression** | **MVSD** | **GDB** | **LLDB** | **LLDB with C++ modules** |
| `m[1]` | ✓ | ✗ | ✗ | ✓ |
| `m.at(1)` | ✗ | ✗ | ✗ | ✓ |
| `m.begin()` | ✗ | ✗ | ✗ | ✓* |
| `m.end()` | ✗ | ✗ | ✗ | ✓* |
| `m.cbegin()` | ✗ | ✗ | ✗ | ✓* |
| `m.cend()` | ✗ | ✗ | ✗ | ✓* |
| `m.empty()` | ✗ | ✓ | ✗ | ✓ |
| `m.size()` | ✓ | ✓ | ✗ | ✓ |
| `m.max_size()` | ✗ | ✗ | ✗ | ✓ |
| `m.clear()` | ✗ | ✗ | ✗ | ✓ |
| `m.insert(1, 1)` | ✗ | ✗ | ✗ | ✓* |
| `m.emplace(1, 1)` | ✗ | ✗ | ✗ | ✓* |
| `m.erase(1)` | ✗ | ✗ | ✗ | ✓* |
| `m.equal_range(1)` | ✗ | ✗ | ✗ | ✓ |
| `m.count(1)` | ✗ | ✗ | ✗ | ✓ |
| `m.find(1)` | ✗ | ✗ | ✗ | ✓* |
| `m.load_factor()` | ✗ | ✗ | ✗ | ✓ |
| `m.max_load_factor()` | ✗ | ✗ | ✗ | ✓ |
| `m.reserve(10)` | ✗ | ✗ | ✗ | ✓ |
| Success rate | 11 % | 11 % | 0 % | 100 % |

Source code for testing used std::unordered_map

```
#include <unordered_map>

int main() {
  std::unordered_map<int, int> m = {{1, 2}, {2, 4}};
  // We evaluate expressions here.
  m[1]; m.at(1); m.begin(); m.end(); m.cbegin(); m.cend(); m.empty();
  m.size(); m.max_size(); m.clear(); m.emplace(1, 1); m.erase(m.begin());
  m.equal_range(1); m.count(1); m.find(1); m.load_factor();
  m.max_load_factor(); m.reserve(10);
}
```

| **Evaluation success rate for used std::unordered_map** | | | | |
|---|---|---|---|---|
| **Expression** | **MVSD** | **GDB** | **LLDB** | **LLDB with C++ modules** |
| `m[1]` | ✓ | ✗ | ✓ | ✓ |
| `m.at(1)` | ✗ | ✗ | ✓ | ✓ |
| `m.begin()` | ✓ | ✓ | ✓ | ✓* |
| `m.end()` | ✓ | ✓ | ✓ | ✓* |
| `m.cbegin()` | ✓ | ✓ | ✓ | ✓* |
| `m.cend()` | ✓ | ✓ | ✓ | ✓* |
| `m.empty()` | ✓ | ✓ | ✓ | ✓ |
| `m.size()` | ✓ | ✓ | ✓ | ✓ |
| `m.max_size()` | ✓ | ✓ | ✓ | ✓ |
| `m.clear()` | ✗ | ✓ | ✓ | ✓ |
| `m.insert(1, 1)` | ✗ | ✗ | ✗ | ✓* |
| `m.emplace(1, 1)` | ✗ | ✗ | ✓ | ✓* |
| `m.erase(1)` | ✗ | ✗ | ✓ | ✓* |
| `m.equal_range(1)` | ✓ | ✗ | ✓ | ✓ |
| `m.count(1)` | ✓ | ✗ | ✓ | ✓ |
| `m.find(1)` | ✓ | ✗ | ✓ | ✓* |
| `m.load_factor()` | ✓ | ✓ | ✓ | ✓ |
| `m.max_load_factor()` | ✓ | ✓ | ✓ | ✓ |
| `m.reserve(10)` | ✓ | ✓ | ✓ | ✓ |
| Success rate | 74 % | 58 % | 95 % | 100 % |

Source code for testing unused std::set

```
#include <set>

int main() {
  std::set<int> m = {1, 2, 3};
  return 0; // We evaluate expressions here.
}
```

| Evaluation success rate for unused std::set | | | | |
|---|---|---|---|---|
| **Expression** | **MVSD** | **GDB** | **LLDB** | **LLDB with C++ modules** |
| `m.begin()` | ✓ | ✗ | ✗ | ✓* |
| `m.end()` | ✓ | ✗ | ✗ | ✓* |
| `m.cbegin()` | ✗ | ✗ | ✗ | ✓* |
| `m.cend()` | ✗ | ✗ | ✗ | ✓* |
| `m.rbegin()` | ✗ | ✗ | ✗ | ✓* |
| `m.rend()` | ✗ | ✗ | ✗ | ✓* |
| `m.crbegin()` | ✗ | ✗ | ✗ | ✓* |
| `m.crend()` | ✗ | ✗ | ✗ | ✓* |
| `m.empty()` | ✗ | ✓ | ✗ | ✓ |
| `m.size()` | ✓ | ✓ | ✗ | ✓ |
| `m.max_size()` | ✓ | ✗ | ✗ | ✓ |
| `m.clear()` | ✗ | ✗ | ✗ | ✓ |
| `m.insert(1)` | ✗ | ✗ | ✗ | ✓* |
| `m.emplace(1)` | ✗ | ✗ | ✗ | ✓* |
| `m.emplace_hint(m.begin(), 1)` | ✗ | ✗ | ✗ | ✓* |
| `m.erase(m.begin())` | ✗ | ✗ | ✗ | ✓* |
| `m.equal_range(1)` | ✗ | ✗ | ✗ | ✓* |
| `m.lower_bound(1)` | ✗ | ✗ | ✗ | ✓* |
| `m.upper_bound(1)` | ✗ | ✗ | ✗ | ✓* |
| `m.count(1)` | ✗ | ✗ | ✗ | ✓ |
| `m.find(1)` | ✗ | ✗ | ✗ | ✓* |
| Success rate | 19 % | 10 % | 0 % | 100 % |

Source code for testing used std::set

```
#include <set>

int main() {
  std::set<int> m = {1, 2, 3};
  // We evaluate expressions here.
  m.begin(); m.end(); m.cbegin(); m.cend(); m.rbegin(); m.rend();
  m.crbegin(); m.crend(); m.empty(); m.size(); m.max_size(); m.clear();
  m.insert(1); m.emplace(1); m.emplace_hint(m.begin(), 1);
  m.erase(m.begin()); m.equal_range(1); m.lower_bound(1);
  m.upper_bound(1); m.count(1); m.find(1);
}
```

| **Evaluation success rate for used std::set** | | | | |
|---|---|---|---|---|
| **Expression** | **MVSD** | **GDB** | **LLDB** | **LLDB with C++ modules** |
| `m.begin()` | ✗ | ✓ | ✓ | ✓* |
| `m.end()` | ✗ | ✓ | ✓ | ✓* |
| `m.cbegin()` | ✓ | ✓ | ✓ | ✓* |
| `m.cend()` | ✓ | ✓ | ✓ | ✓* |
| `m.rbegin()` | ✓ | ✓ | ✓ | ✓* |
| `m.rend()` | ✓ | ✓ | ✓ | ✓* |
| `m.crbegin()` | ✓ | ✓ | ✓ | ✓* |
| `m.crend()` | ✓ | ✓ | ✓ | ✓* |
| `m.empty()` | ✓ | ✓ | ✓ | ✓ |
| `m.size()` | ✓ | ✓ | ✓ | ✓ |
| `m.max_size()` | ✓ | ✓ | ✓ | ✓ |
| `m.clear()` | ✗ | ✓ | ✓ | ✓ |
| `m.insert(1)` | ✓ | ✗ | ✓ | ✓* |
| `m.emplace(1)` | ✗ | ✗ | ✗ | ✓* |
| `m.emplace_hint(m.begin(), 1)` | ✗ | ✗ | ✗ | ✓* |
| `m.erase(m.begin())` | ✗ | ✓ | ✓ | ✓* |
| `m.equal_range(1)` | ✓ | ✗ | ✓ | ✓* |
| `m.lower_bound(1)` | ✓ | ✗ | ✓ | ✓* |
| `m.upper_bound(1)` | ✓ | ✗ | ✓ | ✓* |
| `m.count(1)` | ✓ | ✗ | ✓ | ✓ |
| `m.find(1)` | ✓ | ✗ | ✓ | ✓* |
| Success rate | 71 % | 62 % | 90 % | 100 % |

Source code for testing unused std::unordered_set

```
#include <unordered_set>

int main() {
  std::unordered_set<int> m = {1, 2, 3};
  return 0; // We evaluate expressions here.
}
```

| Evaluation success rate for unused std::unordered_set | | | | |
|---|---|---|---|---|
| **Expression** | **MVSD** | **GDB** | **LLDB** | **LLDB with C++ modules** |
| `m.begin()` | ✗ | ✓ | ✗ | ✓* |
| `m.end()` | ✗ | ✓ | ✗ | ✓* |
| `m.cbegin()` | ✗ | ✓ | ✗ | ✓* |
| `m.cend()` | ✗ | ✓ | ✗ | ✓* |
| `m.empty()` | ✗ | ✓ | ✗ | ✓* |
| `m.size()` | ✓ | ✓ | ✗ | ✓ |
| `m.max_size()` | ✓ | ✓ | ✗ | ✓ |
| `m.clear()` | ✗ | ✓ | ✗ | ✓ |
| `m.insert(1)` | ✗ | ✗ | ✗ | ✓* |
| `m.emplace(1)` | ✗ | ✗ | ✗ | ✓* |
| `m.emplace_hint(m.begin(), 1)` | ✗ | ✗ | ✗ | ✓* |
| `m.erase(m.begin())` | ✗ | ✗ | ✗ | ✓* |
| `m.equal_range(1)` | ✗ | ✗ | ✗ | ✓* |
| `m.count(1)` | ✗ | ✗ | ✗ | ✓ |
| `m.find(1)` | ✗ | ✗ | ✗ | ✓* |
| `m.load_factor()` | ✓ | ✓ | ✗ | ✓ |
| `m.max_load_factor()` | ✓ | ✓ | ✗ | ✓ |
| `m.reserve(10)` | ✗ | ✓ | ✗ | ✓ |
| Success rate | 22 % | 61 % | 0 % | 100 % |

Source code for testing used std::unordered_set

```
#include <unordered_set>

int main() {
  std::unordered_set<int> m = {1, 2, 3};
  // We evaluate expressions here.
  m.begin(); m.end(); m.cbegin(); m.cend(); m.empty(); m.size();
  m.max_size(); m.clear(); m.insert(1); m.emplace(1);
  m.emplace_hint(m.begin(), 1); m.erase(m.begin()); m.equal_range(1);
  m.count(1); m.find(1); m.load_factor();
  m.max_load_factor(); m.reserve(10);
}
```

| **Evaluation success rate for used std::unordered_set** | | | | |
|---|---|---|---|---|
| **Expression** | **MVSD** | **GDB** | **LLDB** | **LLDB with C++ modules** |
| `m.begin()` | ✓ | ✗ | ✓ | ✓* |
| `m.end()` | ✓ | ✗ | ✓ | ✓* |
| `m.cbegin()` | ✓ | ✗ | ✓ | ✓* |
| `m.cend()` | ✓ | ✗ | ✓ | ✓* |
| `m.empty()` | ✓ | ✓ | ✓ | ✓ |
| `m.size()` | ✓ | ✓ | ✓ | ✓ |
| `m.max_size()` | ✓ | ✗ | ✓ | ✓ |
| `m.clear()` | ✗ | ✗ | ✓ | ✓ |
| `m.insert(1)` | ✗ | ✗ | ✓ | ✓* |
| `m.emplace(1)` | ✗ | ✗ | ✗ | ✓* |
| `m.emplace_hint(m.begin(), 1)` | ✗ | ✗ | ✗ | ✓* |
| `m.erase(m.begin())` | ✗ | ✗ | ✓ | ✓* |
| `m.equal_range(1)` | ✓ | ✗ | ✓ | ✓* |
| `m.count(1)` | ✓ | ✗ | ✓ | ✓ |
| `m.find(1)` | ✓ | ✗ | ✓ | ✓* |
| `m.load_factor()` | ✓ | ✗ | ✓ | ✓ |
| `m.max_load_factor()` | ✓ | ✗ | ✓ | ✓ |
| `m.reserve(10)` | ✓ | ✗ | ✓ | ✓ |
| Success rate | 72 % | 11 % | 89 % | 100 % |

# B

# Appendix 1 - general evaluation

Source code for the general test - header.h

```
int functionUnused() { return 1; }
int functionUsed() { return 1; }

inline int inlineFunctionUsed() { return 1; }
inline int inlineFunctionUnused() { return 1; }

int functionDefaultArg(int i = 0) { return 1; }

int functionOverloaded(int i) { return 1; }
int functionOverloaded(float f) { return 2; }

namespace N {
  int functionInNamespace() { return 1; }
}

struct S {
  int memberFuncUnused() { return 1; }
  int memberFuncUsed() { return 1; }
};

extern int globalVar;
typedef int int_t;
#define MACRO 1
```

Source code for the general test - templates.h

```
template<typename T> T templatedUnused() { return 1; }

template<typename T> int templatedArgsUnused(T t) { return 1 + t; }

template<typename T> T templatedUsed() { return 1; }

template<typename T> int templatedArgsUsed(T t) { return 1 + t; }

template<typename T> T templatedUnusedWithType() { return 1; }

template<typename T> struct TS1 { T i; };
template<typename T> TS1<T> templatedReturnUnused() { return {1}; }

template<typename T> struct TS2 { T i; };
template<typename T> TS2<T> templatedReturnUnusedType() { return {1}; }

template<typename T> struct TS3 { T i; };
template<typename T> TS3<T> templatedReturnUsed() { return {1}; }

template<class T> constexpr T templatedVarUsed = T(3.14);

template<class T> constexpr T templatedVarUnused = T(-3.14);

template<typename T> T templatedSpecialized() { return 1; }
```

Source code for the general test - main.cpp

```
#include "header.h"
#include "templates.h"

int globalVar = 0;

template<> int templatedSpecialized<int>() { return 2; }

int main() {
  functionUsed();
  functionPartlyUsed(1.0f);
  N::functionInNamespace();
  S s;
  s.memberFuncUsed();

  templatedUsed<int>();
  templatedArgsUsed(1);
  templatedUnusedWithType<float>();
  templatedReturnUnusedType<float>();
  templatedReturnUsed<int>();
  templatedSpecialized<int>();

  float f = templatedVarUsed<float>;
  int_t i = MACRO;
}
```

| Evaluation success rate for expressions not involving templates | | | | |
|---|---|---|---|---|
| **Expression** | **MVSD** | **GDB** | **LLDB** | **LLDB with C++ modules** |
| `functionUnused()` | ✓ | ✓ | ✓ | ✓ |
| `functionUsed()` | ✓ | ✓ | ✓ | ✓ |
| `inlineFunctionUsed()` | ✗ | ✗ | ✗ | ✓ |
| `inlineFunctionUnused()` | ✗ | ✗ | ✗ | ✓ |
| `functionDefaultArg()` | ✗ | ✗ | ✗ | ✓ |
| `functionOverloaded(0.0f)` | ✗ | ✓ | ✓ | ✓ |
| `N::functionInNamespace()` | ✓ | ✓ | ✓ | ✓ |
| `s.memberFuncUnused()` | ✗ | ✗ | ✗ | ✓ |
| `s.memberFuncUsed()` | ✓ | ✓ | ✓ | ✓ |
| `globalVar` | ✓ | ✓ | ✓ | ✓ |
| `(int_t)1` | ✓ | ✓ | ✓ | ✓ |
| `MACRO` | ✗ | ✗ | ✗ | ✗ |
| Success rate | 50 % | 58 % | 58 % | 92 % |

| Evaluation success rate for expressions involving templates | | | | |
|---|---|---|---|---|
| **Expression** | **MVSD** | **GDB** | **LLDB** | **LLDB with C++ modules** |
| `templatedUnused<int>()` | ✗ | ✗ | ✗ | ✓ |
| `templatedArgsUnused(1)` | ✗ | ✗ | ✗ | ✓ |
| `templatedUsed<int>()` | ✓ | ✓ | ✗ | ✓ |
| `templatedArgsUsed(1)` | ✗ | ✗ | ✓ | ✓ |
| `templatedUnusedWithType<int>()` | ✗ | ✗ | ✗ | ✓ |
| `templatedReturnUnused<int>()` | ✗ | ✗ | ✗ | ✓ |
| `templatedReturnUnusedType<int>()` | ✗ | ✗ | ✗ | ✓ |
| `templatedReturnUsed<int>()` | ✓ | ✓ | ✗ | ✓ |
| `templatedVarUsed<float>` | ✗ | ✓ | ✗ | ✗ |
| `templatedVarUnused<float>` | ✗ | ✗ | ✗ | ✗ |
| `templatedSpecialized<int>()` | ✓ | ✓ | ✗ | ✗ |
| `templatedSpecializedHeader<int>()` | ✓ | ✓ | ✗ | ✓ |
| Success rate | 33 % | 42 % | 8 % | 75 % |