Paradyn Parallel Performance Tools

SymtabAPI Programmer's Guide

12.3 Release
February 2023

Computer Sciences Department
University of Wisconsin–Madison
Madison, WI 53706

Computer Science Department
University of Maryland
College Park, MD 20742

Email dyninst-api@cs.wisc.edu
Web https://github.com/dyninst/dyninst

Contents

1 Introduction

SymtabAPI is a multi-platform library for parsing symbol tables, object file headers and debug information. SymtabAPI currently supports the ELF (IA-32, AMD-64, ARMv8-64, and POWER) and PE (Windows) object file formats. In addition, it also supports the DWARF debugging format.

The main goal of this API is to provide an abstract view of binaries and libraries across multiple platforms. An abstract interface provides two benefits: it simplifies the development of a tool since the complexity of a particular file format is hidden, and it allows tools to be easily ported between platforms. Each binary object file is represented in a canonical platform independent manner by the API. The canonical format consists of four components: a header block that contains general information about the object (e.g., its name and location), a set of symbol lists that index symbols within the object for fast lookup, debug information (type, line number and local variable information) present in the object file and a set of additional data that represents information that may be present in the object (e.g., relocation or exception information). Adding a new format requires no changes to the interface and hence will not affect any of the tools that use the SymtabAPI.

Our other design goal with SymtabAPI is to allow users and tool developers to easily extend or add symbol or debug information to the library through a platform-independent interface. Often times it is impossible to satify all the requirements of a tool that uses SymtabAPI, as those requirements can vary from tool to tool. So by providing extensible structures, SymtabAPI allows tools to modify any structure to fit their own requirements. Also, tools frequently use more sophisticated analyses to augment the information available from the binary directly; it should be possible to make this extra information available to the SymtabAPI library. An example of this is a tool operating on a stripped binary. Although the symbols for the majority of functions in the binary may be missing, many can be determined via more sophisticated analysis. In our model, the tool would then inform the SymtabAPI library of the presence of these functions; this information would be incorporated and available for subsequent analysis. Other examples of such extensions might involve creating and adding new types or adding new local variables to certain functions.

2 Abstractions

SymtabAPI provides a simple set of abstractions over complicated data structures which makes it straight-forward to use. The SymtabAPI consists of five classes of interfaces: the symbol table interface, the type interface, the line map interface, the local variable interface, and the address translation interface.

Figure 1 shows the ownership hierarchy for the SymtabAPI classes. Ownership here is a "contains" relationship; if one class owns another, then instances of the owner class maintain an exclusive instance of the other. For example, each Symtab class instance contains multiple instances of class Symbol and each Symbol class instance belongs to one Symtab class instance. Each of four interfaces and the classes belonging to these interfaces are described in the rest of this section. The API functions in each of the classes are described in detail in Section 6.

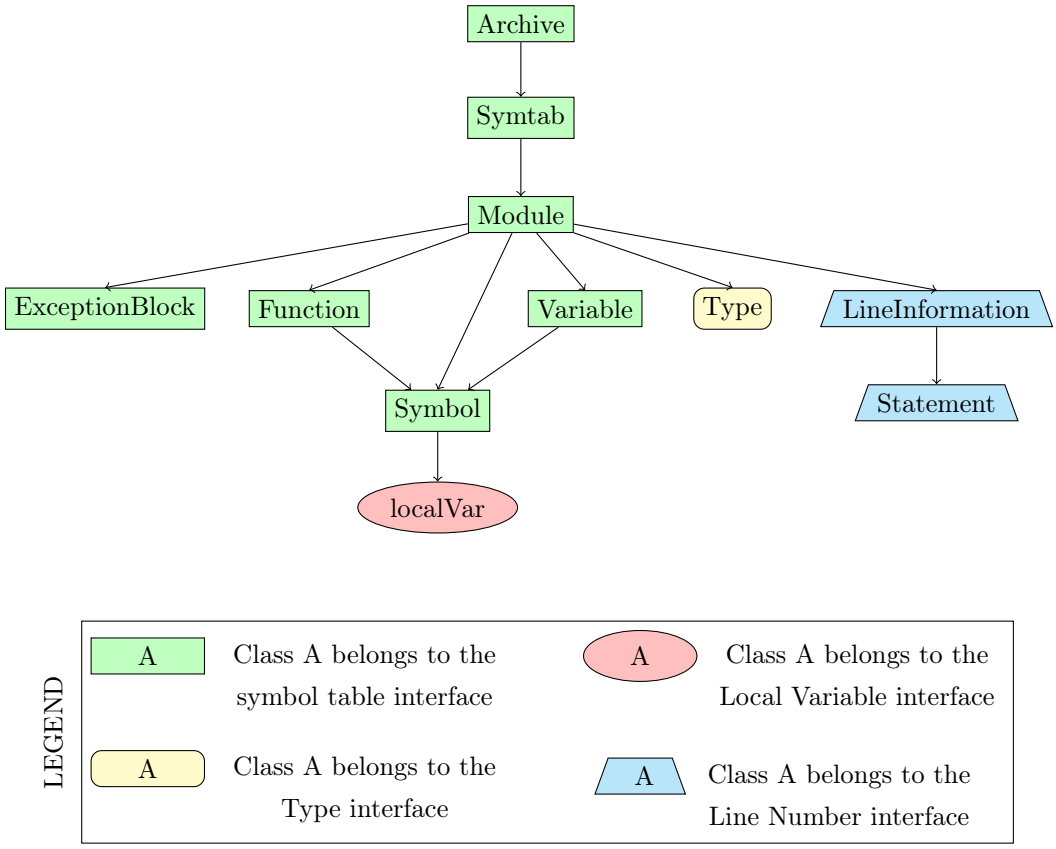

Figure 1: SymtabAPI Object Ownership Diagram

2.1 Symbol Table Interface

The symbol table interface is responsible for parsing the object file and handling the look-up and addition of new symbols. It is also responsible for the emit functionality that SymtabAPI supports. The Symtab and the Module classes inherit from the LookupInterface class, an abstract class, ensuring the same lookup function signatures for both Module and Symtab classes.

Symtab A Symtab class object represents either an object file on-disk or in-memory that the SymtabAPI library operates on.

Symbol A Symbol class object represents an entry in the symbol table.

Module A Module class object represents a particular source file in cases where multiple files were compiled into a single binary object; if this information is not present, we use a single default module.

Archive An Archive class object represents a collection of binary objects stored in a single file (e.g., a static archive).

ExceptionBlock An ExceptionBlock class object represents an exception block which contains the information necessary for run-time exception handling.

In addition, we define two symbol aggregates, Function and Variable. These classes collect multiple symbols with the same address and type but different names; for example, weak and strong symbols for a single function.

2.2 Type Interface

The Type interface is responsible for parsing type information from the object file and handling the look-up and addition of new type information. Figure 2 shows the class inheritance diagram for the type interface. Class Type is the base class for all of the classes that are part of the interface. This class provides the basic common functionality for all the types, such as querying the name and size of a type. The rest of the classes represent specific types and provide more functionality based on the type.

Some of the types inherit from a second level of type classes, each representing a separate category of types.

fieldListType - This category of types represent the container types that contain a list of fields. Examples of this category include structure and the union types.

derivedType - This category of types represent types derived from a base type. Examples of this category include typedef, pointer and reference types.

rangedType - This category represents range types. Examples of this category include the array and the sub-range types.

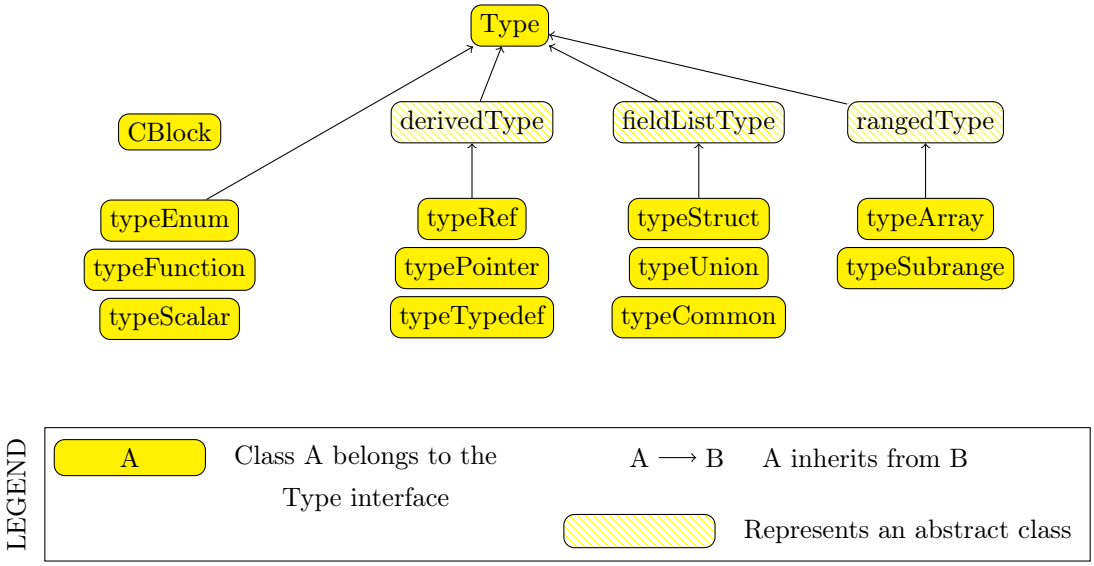

Figure 2: SymtabAPI Type Interface - Class Inheritance Diagram

The enum, function, common block and scalar types do not fall under any of the above category of types. Each of the specific types is derived from Type.

2.3 Line Number Interface

The Line Number interface is responsible for parsing line number information from the object file debug information and handling the look-up and addition of new line information. The main classes for this interface are LineInformation and LineNoTuple.

LineInformation - A LineInformation class object represents a mapping of line numbers to address range within a module (source file).

Statement/LineNoTuple - A Statement class object represents a location in source code with a source file, line number in that source file and start column in that line. For backwards compatibility, Statements may also be referred to as LineNoTuples.

2.4 Local Variable Interface

The Local Variable Interface is responsible for parsing local variable and parameter information of functions from the object file debug information and handling the look-up and addition of new add new local variables. All the local variables within a function are tied to the Symbol class object representing that function.

localVar - A localVar class object represents a local variable or a parameter belonging to a function.

2.5 Dynamic Address Translation

The AddressLookup class is a component for mapping between absolute addresses found in a running process and SymtabAPI objects. This is useful because libraries can load at different addresses in different processes. Each AddressLookup instance is associated with, and provides mapping for, one process.

3 Simple Examples

To illustrate the ideas in the API, this section presents several short examples that demonstrate how the API can be used. SymtabAPI has the ability to parse files that are on-disk or present in memory. The user program starts by requesting SymtabAPI to parse an object file. SymtabAPI returns a handle if the parsing succeeds, whcih can be used for further interactions with the SymtabAPI library. The following example shows how to parse a shared object file on disk.

```
using namespace Dyninst;
using namespace SymtabAPI;

//Name the object file to be parsed:
std::string file = "libfoo.so";

//Declare a pointer to an object of type Symtab; this represents the file.
Symtab *obj = NULL;

// Parse the object file
bool err = Symtab::openFile(obj, file);
```

Once the object file is parsed successfully and the handle is obtained, symbol look up and update operations can be performed in the following way:

```
using namespace Dyninst;
using namespace SymtabAPI;
std::vector <Symbol *> syms;
std::vector <Function *> funcs;

// search for a function with demangled (pretty) name "bar".
if (obj->findFunctionsByName(funcs, "bar")) {
       // Add a new (mangled) primary name to the first function
       funcs[0]->addMangledName("newname", true);
}

// search for symbol of any type with demangled (pretty) name "bar".
if (obj->findSymbol(syms, "bar", Symbol::ST_UNKNOWN)) {

    // change the type of the found symbol to type variable(ST_OBJECT)
    syms[0]->setType(Symbol::ST_OBJECT);

    // These changes are automatically added to symtabAPI; no further
    // actions are required by the user.
}
```

New symbols, functions, and variables can be created and added to the library at any point using the handle returned by successful parsing of the object file. When possible, add a function or variable rather than a symbol directly.

```
using namespace Dyninst;
using namespace SymtabAPI;

//Module for the symbol
Module *mod;

// obj represents a handle to a parsed object file.
// Lookup module handle for "DEFAULT_MODULE"
obj->findModuleByName(mod, "DEFAULT_MODULE");

// Create a new function symbol
Variable *newVar = mod->createVariable("newIntVar",  // Name of new variable
                                       0x12345,      // Offset from data section
                                       sizeof(int)); // Size of symbol
```

SymtabAPI gives the ability to query type information present in the object file. Also, new user defined types can be added to SymtabAPI. The following example shows both how to query type information after an object file is successfully parsed and also add a new structure type.

```
// create a new struct Type
// typedef struct{
//int field1,
//int field2[10]
// } struct1;

using namespace Dyninst;
using namespace SymtabAPI;

// Find a handle to the integer type; obj represents a handle to a parsed object file
Type *lookupType;
obj->findType(lookupType, "int");

// Convert the generic type object to the specific scalar type object
typeScalar *intType = lookupType->getScalarType();

// container to hold names and types of the new structure type
vector<pair<string, Type *> >fields;

//create a new array type(int type2[10])
typeArray *intArray = typeArray::create("intArray",intType,0,9, obj);

//types of the structure fields
fields.push_back(pair<string, Type *>("field1", intType));
fields.push_back(pair<string, Type *>("field2", intArray));

//create the structure type
typeStruct *struct1 = typeStruct::create("struct1", fields, obj);
```

Users can also query line number information present in an object file. The following example shows how to use SymtabAPI to get the address range for a line number within a source file.

```
using namespace Dyninst;
using namespace SymtabAPI;

// obj represents a handle to a parsed object file using symtabAPI
// Container to hold the address range
vector< pair< Offset, Offset > > ranges;

// Get the address range for the line 30 in source file foo.c
obj->getAddressRanges(ranges, "foo.c", 30);
```

Local variable information can be obtained using symtabAPI. You can query for a local variable within the entire object file or just within a function. The following example shows how to find local variable foo within function bar.

```
using namespace Dyninst;
using namespace SymtabAPI;

// Obj represents a handle to a parsed object file using symtabAPI
// Get the Symbol object representing function bar
vector<Symbol *> syms;
obj->findSymbol(syms, "bar", Symbol::ST_FUNCTION);

// Find the local var foo within function bar
vector<localVar *> *vars = syms[0]->findLocalVarible("foo");
```

The rest of this document describes the class hierarchy and the API in detail.

4 Definitions and Basic Types

The following definitions and basic types are referenced throughout the rest of this document.

4.1 Definitions

Offset Offsets represent an address relative to the start address(base) of the object file. For executables, the Offset represents an absolute address. The following definitions deal with the symbol table interface.

Object File An object file is the representation of code that a compiler or assembler generates by processing a source code file. It represents .o's, a.out's and shared libraries.

Region A region represents a contiguous area of the file that contains executable code or readable data; for example, an ELF section.

Symbol A symbol represents an entry in the symbol table, and may identify a function, variable or other structure within the file.

Function A function represents a code object within the file represented by one or more symbols.

Variable A variable represents a data object within the file represented by one or more symbols.

Module A module represents a particular source file in cases where multiple files were compiled into a single binary object; if this information is not present, or if the binary object is a shared library, we use a single default module.

Archive An archive represents a collection of binary objects stored in a single file (e.g., a static archive).

Relocations These provide the necessary information for inter-object references between two object files.

Exception Blocks These contain the information necessary for run-time exception handling The following definitions deal with members of the Symbol class.

Mangled Name A mangled name for a symbol provides a way of encoding additional information about a function, structure, class or another data type in a symbol name. It is a technique used to produce unique names for programming entities in many modern programming languages. For example, the method *foo* of class C with signature *int C::foo(int, int)* has a mangled name *_ZN1C3fooEii* when compiled with gcc. Mangled names may include a sequence of clone suffixes (begins with '.' that indicate a compiler synthesized function), and this may be followed by a version suffix (begins with '@').

Pretty Name A pretty name for a symbol is the demangled user-level symbolic name without type information for the function parameters and return types. For non-mangled names, the pretty name is the symbol name. Any function clone suffixes of the symbol are appended to the result of the demangler. For example, a symbol with a mangled name *_ZN1C3fooEii* for the method *int C::foo(int, int)* has a pretty name *C::foo*. Version suffixes are removed from

the mangled name before conversion to the pretty name. The pretty name can be obtained by running the command line tool `c++filt` as `c++filt -i -p` *`name`*, or using the libiberty library function `cplus_demangle` with options of `DMGL_AUTO | DMGL_ANSI`.

Typed Name A typed name for a symbol is the demangled user-level symbolic name including type information for the function parameters. Typically, but not always, function return type information is not included. Any function clone information is also included. For non-mangled names, the typed name is the symbol name. For example, a symbol with a mangled name *_ZN1C3fooEii* for the method *int C::foo(int, int)* has a typed name *C::foo(int, int)*. Version suffixes are removed from the mangled name before conversion to the typed name. The typed name can be obtained by running the command line tool `c++filt` as `c++filt -i` *`name`*, or using the libiberty library function `cplus_demangle` with options of `DMGL_AUTO | DMGL_ANSI | DMGL_PARAMS`.

Symbol Linkage The symbol linkage for a symbol gives information on the visibility (binding) of this symbol, whether it is visible only in the object file where it is defined (local), if it is visible to all the object files that are being linked (global), or if its a weak alias to a global symbol.

Symbol Type Symbol type for a symbol represents the category of symbols to which it belongs. It can be a function symbol or a variable symbol or a module symbol. The following definitions deal with the type and the local variable interface.

Type A type represents the data type of a variable or a parameter. This can represent language pre-defined types (e.g. int, float), pre-defined types in the object (e.g., structures or unions), or user-defined types.

Local Variable A local variable represents a variable that has been declared within the scope of a sub-routine or a parameter to a sub-routine.

4.2 Basic Types

```
typedef unsigned long Offset
```

An integer value that contains an offset from base address of the object file.

```
typedef int typeId_t
```

A unique handle for identifying a type. Each of types is assigned a globally unique ID. This way it is easier to identify any data type of a variable or a parameter.

```
typedef ... PID
```

A handle for identifying a process that is used by the dynamic components of SymtabAPI. On UNIX platforms PID is a int, on Windows it is a HANDLE that refers to a process.

```
typedef unsigned long Address
```

An integer value that represents an address in a process. This is used by the dynamic components of SymtabAPI.

5 Namespace SymtabAPI

The classes described in the following sections are under the C++ namespace Dyninst::SymtabAPI. To access them a user should refer to them using the Dyninst:: and SymtabAPI:: prefixes, e.g. Dyninst::SymtabAPI::Type. Alternatively, a user can add the C++ using keyword above any references to SymtabAPI objects, e.g, using namespace Dyninst and using namespace SymtabAPI.

6 API Reference - Symbol Table Interface

This section describes the symbol table interface for the SymtabAPI library. Currently this interface has the following capabilities:

- Parsing the symbols in a binary, either on disk or in memory
- Querying for symbols
- Updating existing symbol information
- Adding new symbols
- Exporting symbols in standard formats
- Accessing relocation and exception information
- Accessing and modifying header information

The symbol table information is represented by the Symtab, Symbol, Archive, and Region classes. Module, Function, and Variable provide abstractions that support common use patterns. Finally, LocalVar represents function-local variables and parameters.

6.1 Class Symtab

Defined in: `Symtab.h`

The `Symtab` class represents an object file either on-disk or in-memory. This class is responsible for the parsing of the `Object` file information and holding the data that can be accessed through look up functions.

Method name	Return type	Method description
`file`	std::string	Full path to the opened file or provided name for the memory image.
`name`	std::string	File name without path.
`memberName`	std::string	For archive (.a) files, returns the object file (.o) this Symtab represents.
`getNumberOfRegions`	unsigned	Number of regions.
`getNumberOfSymbols`	unsigned	Total number of symbols in both the static and dynamic tables.
`mem_image`	char *	Pointer to memory image for the Symtab; not valid for disk files.
`imageOffset`	Offset	Offset of the first code segment from the start of the binary.
`dataOffset`	Offset	Offset of the first data segment from the start of the binary.
`imageLength`	Offset	Size of the primary code-containing region, typically .text.
`dataLength`	Offset	Size of the primary data-containing region, typically .data.
`isStaticBinary`	bool	True if the binary was compiled statically.
`isExecutable`	bool	True if the file is an executable.
`isSharedLibrary`	bool	True if the file is a shared library.
`isExec`	bool	True if the file is can only be an executable, false otherwise including files that are both exeutables and shared libraries. Typically files that are both executables and shared libraries are primarily used as libraries, if you need to determine specifics use the methods `isExecutable` and `isSharedLibrary`.
`isStripped`	bool	True if the file was stripped of symbol table information.
`getAddressWidth`	unsigned	Size (in bytes) of a pointer value in the Symtab; 4 for 32-bit binaries and 8 for 64-bit binaries.
`getArchitecture`	Architecture	Representation of the system architecture for the binary.
`getLoadOffset`	Offset	The suggested load offset of the file; typically 0 for shared libraries.
`getEntryOffset`	Offset	The entry point (where execution begins) of the binary.
`getBaseOffset`	Offset	(Windows only) the OS-specified base offset of the file.

```
ObjectType getObjectType() const
```

This method queries information on the type of the object file.

```
bool isExecutable()
bool isSharedLibrary()
bool isExec()
```

These methods respectively return true if the Symtab's object is an executable, a shared library, and an executable is that is not a shared library. An object may be both an executable and a shared library.

An Elf Object that can be loaded into memory to form an executable's image has one of two types: ET_EXEC and ET_DYN. ET_EXEC type objects are executables that are loaded at a fixed address determined at link time. ET_DYN type objects historically were shared libraries that are

loaded at an arbitrary location in memory and are position independent code (PIC). The ET_DYN object type was reused for position independent executables (PIE) that allows the executable to be loaded at an arbitrary location in memory. Although generally not the case an object can be both a PIE executable and a shared library. Examples of these include libc.so and the dynamic linker library (ld.so). These objects are generally used as a shared library so `isExec()` will classify these based on their typical usage. The methods below use heuristics to classify ET_DYN object types correctly based on the properties of the Elf Object, and will correctly classify most objects. Due to the inherent ambiguity of ET_DYN object types, the heuristics may fail to classify some libraries that are also executables as an executable. This can happen in object is a shared library and an executable, and its entry point happens to be at the start of the .text section.

`isExecutable()` is equivalent to elfutils' `elfclassify --program` test with the refinement of the soname value and entry point tests. Pseudocode for the algorithm is shown below:

- **if** (**not** loadable()) **return** *false*
- **if** (object type is ET_EXEC) **return** *true*
- **if** (has an interpreter (PT_INTERP segment exists)) **return** *true*
- **if** (PIE flag is set in FLAGS_1 of the PT_DYNAMIC segment) **return** *true*
- **if** (DT_DEBUG tag exists in PT_DYNAMIC segment) **return** *true*
- **if** (has a soname and its value is "linux-gate.so.1") **return** *false*
- **if** (entry point is in range .text section offset plus 1 to the end of the .text section) **return** *true*
- **if** (has a soname and its value starts with "ld-linux") **return** *true*
- **otherwise return** *false*

`isSharedLibrary()` is equivalent to elfutils' `elfclassify --library`. Pseudocode for the algorithm is shown below:

- **if** (**not** loadable()) **return** *false*
- **if** (object type is ET_EXEC) **return** *false*
- **if** (there is no PT_DYNAMIC segment) **return** *false*
- **if** (PIE flag is set in FLAGS_1 of the PT_DYNAMIC segment) **return** *false*
- **if** (DT_DEBUG tag exists in PT_DYNAMIC segment) **return** *false*
- **otherwise return** *true*

Elf files can also store data that is neither an executable nor a shared library including object files, core files and debug symbol files. To distinguish these cases the `loadable()` function is defined using the pseudocode shown below and returns true is the file can loaded into a process's address space:

- **if** (object type is neither ET_EXEC nor ET_DYN) **return** *false*
- **if** (there is are no program segments with the PT_LOAD flag set) **return** *false*
- **if** (contains no sections) **return** *true*
- **if** (contains a section with the SHF_ALLOC flag set and a section type of neither SHT_NOTE nor SHT_NOBITS) **return** *true*
- **otherwise return** *false*

6.1.1 File opening/parsing

```
static bool openFile(Symtab *&obj,
                     string filename)
```

Creates a new `Symtab` object for an object file on disk. This object serves as a handle to the parsed object file. `filename` represents the name of the `Object` file to be parsed. The `Symtab` object is returned in `obj` if the parsing succeeds. Returns `true` if the file is parsed without an error, else returns `false`. `getLastSymtabError()` and `printError()` should be called to get more error details.

```
static bool openFile(Symtab *&obj,
                     char *mem_image,
                     size_t size,
                     std::string name)
```

This factory method creates a new `Symtab` object for an object file in memory. This object serves as a handle to the parsed object file. `mem_image` represents the pointer to the `Object` file in memory to be parsed. `size` indicates the size of the image. `name` specifies the name we will give to the parsed object. The `Symtab` object is returned in `obj` if the parsing succeeds. Returns `true` if the file is parsed without an error, else returns `false`. `getLastSymtabError()` and `printError()` should be called to get more error details.

```
static Symtab *findOpenSymtab(string name)
```

Find a previously opened `Symtab` that matches the provided name.

6.1.2 Module lookup

```
Module *getDefaultModule()
```

Returns the default module, a collection of all functions, variables, and symbols that do not have an explicit module specified.

```
bool findModuleByName(Module *&ret,
                      const string name)
```

This method searches for a module with name `name`. If the module exists returns `true` with `ret` set to the module handle; otherwise returns `false` with `ret` set to `NULL`.

```
bool findModuleByOffset(Module *&ret,
                        Offset offset)
```

This method searches for a module that starts at offset `offset`. If the module exists returns `true` with `ret` set to the module handle; otherwise returns `false` with `ret` set to `NULL`.

```
bool getAllModules(vector<module *> &ret)
```

This method returns all modules in the object file. Returns `true` on success and `false` if there are no modules. The error value is set to `No_Such_Module`.

6.1.3 Function, Variable, and Symbol lookup

```
bool findFuncByEntryOffset(Function *&ret,
                           const Offset offset)
```

This method returns the `Function` object that begins at `offset`. Returns `true` on success and `false` if there is no matching function. The error value is set to `No_Such_Function`.

```
bool findFunctionsByName(std::vector<Function *> &ret,
                         const std::string name,
                         NameType nameType = anyName,
                         bool isRegex = false,
                         bool checkCase = true)
```

This method finds and returns a vector of `Function`s whose names match the given pattern. The `nameType` parameter determines which names are searched: mangled, pretty, typed, or any. If the `isRegex` flag is set a regular expression match is performed with the symbol names. `checkCase` is applicable only if `isRegex` has been set. This indicates if the case be considered while performing regular expression matching. `ret` contains the list of matching Functions, if any. Returns `true` if it finds functions that match the given name, otherwise returns `false`. The error value is set to `No_Such_Function`.

```
bool getContainingFunction(Offset offset,
                           Function *&ret)
```

This method returns the function, if any, that contains the provided `offset`. Returns `true` on success and `false` on failure. The error value is set to `No_Such_Function`. Note that this method does not parse, and therefore relies on the symbol table for information. As a result it may return incorrect information if the symbol table is wrong or if functions are either non-contiguous or overlapping. For more precision, use the ParseAPI library.

```
bool getAllFunctions(vector<Function *> &ret)
```

This method returns all functions in the object file. Returns `true` on success and `false` if there are no modules. The error value is set to `No_Such_Function`.

```
bool findVariablesByOffset(std::vector<Variable *> &ret,
                           const Offset offset)
```

This method returns a vector of `Variable`s with the specified offset. There may be more than one variable at an offset if they have different sizes. Returns `true` on success and `false` if there is no matching variable. The error value is set to `No_Such_Variable`.

```
bool findVariablesByName(std::vector<Variable *> &ret,
                         const std::string name,
                         NameType nameType = anyName,
                         bool isRegex = false,
                         bool checkCase = true)
```

This method finds and returns a vector of `Variable`s whose names match the given pattern. The `nameType` parameter determines which names are searched: mangled, pretty, typed, or any (note: a `Variable` may not have a typed name). If the `isRegex` flag is set a regular expression match is performed with the symbol names. `checkCase` is applicable only if `isRegex` has been set. This indicates if the case be considered while performing regular expression matching. `ret` contains the list of matching `Variable`s, if any. Returns `true` if it finds variables that match the given name, otherwise returns `false`. The error value is set to `No_Such_Variable`.

```
bool getAllVariables(vector<Variable *> &ret)
```

This method returns all variables in the object file. Returns `true` on success and `false` if there are no modules. The error value is set to `No_Such_Variable`.

```
bool findSymbol(vector <Symbol *> &ret,
                const string name,
                Symbol::SymbolType sType,
                NameType nameType = anyName,
                bool isRegex = false,
                bool checkCase = false)
```

This method finds and returns a vector of symbols with type `sType` whose names match the given name. The `nameType` parameter determines which names are searched: mangled, pretty, typed, or any. If the `isRegex` flag is set a regular expression match is performed with the symbol names. `checkCase` is applicable only if `isRegex` has been set. This indicates if the case be considered while performing regular expression matching. `ret` contains the list of matched symbols if any. Returns `true` if it finds symbols with the given attributes. or else returns `false`. The error value is set `to` `No_Such_Function` / `No_Such_Variable`/ `No_Such_Module`/ `No_Such_Symbol` based on the type.

```
const vector<Symbol *> *findSymbolByOffset(Offset offset)
```

Return a pointer to a vector of `Symbol`s with the specified offset. The pointer belongs to `Symtab` and should not be modified or freed.

```
bool getAllSymbols(vector<Symbol *> &ret)
```

This method returns all symbols. Returns `true` on success and `false` if there are no symbols. The error value is set to `No_Such_Symbol`.

```
bool getAllSymbolsByType(vector<Symbol *> &ret,
                         Symbol::SymbolType sType)
```

This method returns all symbols whose type matches the given type `sType`. Returns `true` on success and `false` if there are no symbols with the given type. The error value is set to `No_Such_Symbol`.

```
bool getAllUndefinedSymbols(std::vector<Symbol *> &ret)
```

This method returns all symbols that reference symbols in other files (e.g., external functions or variables). Returns `true` if there is at least one such symbol or else returns `false` with the error set to `No_Such_Symbol`.

6.1.4 Region lookup

```
bool getCodeRegions(std::vector<Region *>&ret)
```

This method finds all the code regions in the object file. Returns `true` with `ret` containing the code regions if there is at least one code region in the object file or else returns `false`.

```
bool getDataRegions(std::vector<Region *>&ret)
```

This method finds all the data regions in the object file. Returns `true` with `ret` containing the data regions if there is at least one data region in the object file or else returns `false`.

```
bool getMappedRegions(std::vector<Region *>&ret)
```

This method finds all the loadable regions in the object file. Returns `true` with `ret` containing the loadable regions if there is at least one loadable region in the object file or else returns `false`.

```
bool getAllRegions(std::vector<Region *>&ret)
```

This method retrieves all the regions in the object file. Returns `true` with `ret` containing the regions.

```
bool getAllNewRegions(std::vector<Region *>&ret)
```

This method finds all the new regions added to the object file. Returns `true` with `ret` containing the regions if there is at least one new region that is added to the object file or else returns `false`.

```
bool findRegion(Region *&reg,
                string sname)
```

Find a region (ELF section) wih name `sname` in the binary. Returns `true` if found, with `reg` set to the region pointer. Otherwise returns `false` with `reg` set to `NULL`.

```
bool findRegion(Region *&reg,
                const Offset addr,
                const unsigned long size)
```

Find a region (ELF section) with a memory offset of `addr` and memory size of `size`. Returns `true` if found, with `reg` set to the region pointer. Otherwise returns `false` with `reg` set to `NULL`.

```
bool findRegionByEntry(Region *&reg,
                       const Offset soff)
```

Find a region (ELF section) with a memory offset of `addr`. Returns `true` if found, with `reg` set to the region pointer. Otherwise returns `false` with `reg` set to `NULL`.

```
Region *findEnclosingRegion(const Offset offset)
```

Find the region (ELF section) whose virtual address range contains `offset`. Returns the region if found; otherwise returns `NULL`.

6.1.5 Insertion and modification

```
bool emit(string file)
```

Creates a new file using the specified name that contains all changes made by the user.

```
bool addLibraryPrereq(string lib)
```

Add a library dependence to the file such that when the file is loaded, the library will be loaded as well. Cannot be used for static binaries.

```
Function *createFunction(std::string name,
                         Offset offset,
                         size_t size,
                         Module *mod = NULL)
```

This method creates a `Function` and updates all necessary data structures (including creating Symbols, if necessary). The function has the provided mangled name, offset, and size, and is added to the Module `mod`. Symbols representing the function are added to the static and dynamic symbol tables. Returns the pointer to the new `Function` on success or `NULL` on failure.

```
Variable *createVariable(std::string name,
                         Offset offset,
                         size_t size,
                         Module *mod = NULL)
```

This method creates a `Variable` and updates all necessary data structures (including creating Symbols, if necessary). The variable has the provided mangled name, offset, and size, and is added to the Module `mod`. Symbols representing the variable are added to the static and dynamic symbol tables. Returns the pointer to the new `Variable` on success or `NULL` on failure.

```
bool addSymbol(Symbol *newsym)
```

This method adds a new symbol `newsym` to all of the internal data structures. The primary name of the `newsym` must be a mangled name. Returns `true` on success and `false` on failure. A new copy of `newsym` is not made. `newsym` must not be deallocated after adding it to symtabAPI. We suggest using `createFunction` or `createVariable` when possible.

```
bool addSymbol(Symbol *newsym,
               Symbol *referringSymbol)
```

This method adds a new dynamic symbol `newsym` which refers to `referringSymbol` to all of the internal data structures. `newsym` must represent a dynamic symbol. The primary name of the newsym must be a mangled name. All the required version names are allocated automatically. Also if the `referringSymbol` belongs to a shared library which is not currently a dependency, the shared library is added to the list of dependencies implicitly. Returns `true` on success and `false` on failure. A new copy of `newsym` is not made. `newsym` must not be deallocated after adding it to symtabAPI.

```
bool deleteFunction(Function *func)
```

This method deletes the `Function` `func` from all of symtab's data structures. It will not be available for further queries. Return `true` on success and `false` if `func` is not owned by the `Symtab`.

```
bool deleteVariable(Variable *var)
```

This method deletes the variable `var` from all of symtab's data structures. It will not be available for further queries. Return `true` on success and `false` if `var` is not owned by the `Symtab`.

```
bool deleteSymbol(Symbol *sym)
```

This method deletes the symbol `sym` from all of symtab's data structures. It will not be available for further queries. Return `true` on success and `false` if func is not owned by the `Symtab`.

```
bool addRegion(Offset vaddr,
               void *data,
               unsigned int dataSize,
               std::string name,
               Region::RegionType rType_,
               bool loadable = false,
               unsigned long memAlign = sizeof(unsigned),
               bool tls = false)
```

Creates a new region using the specified parameters and adds it to the file.

```
Offset getFreeOffset(unsigned size)
```

Find a contiguous region of unused space within the file (which may be at the end of the file) of the specified size and return an offset to the start of the region. Useful for allocating new regions.

```
bool addRegion(Region *newreg);
```

Adds the provided region to the file.

6.1.6 Catch and Exception block lookup

```
bool getAllExceptions(vector<ExceptionBlock *> &exceptions)
```

This method retrieves all the exception blocks in the `Object` file. Returns `false` if there are no exception blocks else returns `true` with exceptions containing a vector of `ExceptionBlock`s.

```
bool findException(ExceptionBlock &excp,
                   Offset addr)
```

This method returns the exception block in the binary at the offset `addr`. Returns `false` if there is no exception block at the given offset else returns `true` with `excp` containing the exception block.

```
bool findCatchBlock(ExceptionBlock &excp,
                    Offset addr,
                    unsigned size = 0)
```

This method returns `true` if the address range `[addr, addr+size]` contains a catch block, with `excp` pointing to the appropriate block, else returns `false`.

6.1.7 Symtab information

```
typedef enum {
    obj_Unknown,
    obj_SharedLib,
    obj_Executable,
    obj_RelocatableFile,
} ObjectType;
```

```
bool isCode(const Offset where) const
```

This method checks if the given offset `where` belongs to the text section. Returns `true` if that is the case or else returns `false`.

```
bool isData(const Offset where) const
```

This method checks if the given offset `where` belongs to the data section. Returns `true` if that is the case or else returns `false`.

```
bool isValidOffset(const Offset where) const
```

This method checks if the given offset `where` is valid. For an offset to be valid it should be aligned and it should be a valid code offset or a valid data offset. Returns `true` if it succeeds or else returns `false`.

6.1.8 Line number information

```
bool getAddressRanges(vector<pair<Offset, Offset> > & ranges,
                      string lineSource,
                      unsigned int LineNo)
```

This method returns the address ranges in `ranges` corresponding to the line with line number `lineNo` in the source file `lineSource`. Searches all modules for the given source. Return `true` if at least one address range corresponding to the line number was found and returns `false` if none found.

```
bool getSourceLines(vector<LineNoTuple> &lines,
                    Offset addressInRange)
```

This method returns the source file names and line numbers corresponding to the given address `addressInRange`. Searches all modules for the given source. Return `true` if at least one tuple corresponding to the offset was found and returns `false` if none found.

```
bool addLine(string lineSource,
             unsigned int lineNo,
             unsigned int lineOffset,
             Offset lowInclusiveAddr,
             Offset highExclusiveAddr)
```

This method adds a new line to the line map. `lineSource` represents the source file name. `lineNo` represents the line number. Returns `true` on success and `false` on error.

```
bool addAddressRange(Offset lowInclusiveAddr,
                     Offset highExclusiveAddr,
                     string lineSource,
                     unsigned int lineNo,
                     unsigned int lineOffset = 0);
```

This method adds an address range `[lowInclusiveAddr, highExclusiveAddr)` for the line with line number `lineNo` in source file `lineSource` at offset `lineOffset`. Returns `true` on success and `false` on error.

6.1.9 Type information

```
void parseTypesNow()
```

Forces SymtabAPI to perform type parsing instead of delaying it to when needed.

```
bool findType(Type *&type,
              string name)
```

Performs a look up among all the built-in types, standard types and user-defined types and returns a handle to the found type with name `name`. Returns `true` if a type is found with type containing the handle to the type, else return `false`.

```
bool addType(Type * type)
```

Adds a new type `type` to symtabAPI. Return `true` on success.

```
static std::vector<Type *> * getAllstdTypes()
```

Returns all the standard types that normally occur in a program.

```
static std::vector<Type *> * getAllbuiltInTypes()
```

Returns all the built-in types defined in the binary.

```
bool findLocalVariable(vector<localVar *> &vars,
                       string name)
```

The method returns a list of local variables named name within the object file. Returns `true` with `vars` containing a list of `localVar` objects corresponding to the local variables if found or else returns `false`.

```
bool findVariableType(Type *&type,
                      std::string name)
```

This method looks up a global variable with name `name` and returns its type attribute. Returns `true` if a variable is found or returns `false` with type set to `NULL`.

```
typedef enum ... SymtabError
```

`SymtabError` can take one of the following values.

SymtabError enum	Meaning
Obj_Parsing	An error occurred during object parsing(internal error).
Syms_To_Functions	An error occurred in converting symbols to functions(internal error).
Build_Function_Lists	An error occurred while building function lists(internal error).
No_Such_Function	No matching function exists with the given inputs.
No_Such_Variable	No matching variable exists with the given inputs.
No_Such_Module	No matching module exists with the given inputs.
No_Such_Symbol	No matching symbol exists with the given inputs.
No_Such_Region	No matching region exists with the given inputs.
No_Such_Member	No matching member exists in the archive with the given inputs.
Not_A_File	Binary to be parsed may be an archive and not a file.
Not_An_Archive	Binary to be parsed is not an archive.
Duplicate_Symbol	Duplicate symbol found in symbol table.
Export_Error	Error occurred during export of modified symbol table.
Emit_Error	Error occurred during generation of modified binary.
Invalid_Flags	Flags passed are invalid.
Bad_Frame_Data	Stack walking DWARF information has bad frame data.
No_Frame_Entry	No stack walking frame data found in debug information for this location.
Frame_Read_Error	Failed to read stack frame data.
Multiple_Region_Matches	Multiple regions match the provided data.
No_Error	Previous operation did not result in failure.

```
static SymtabError getLastSymtabError()
```

This method returns an error value for the previously performed operation that resulted in a failure. SymtabAPI sets a global error value in case of error during any operation. This call returns the last error that occurred while performing any operation.

```
static string printError(SymtabError serr)
```

This method returns a detailed description of the enum value serr in human readable format.

6.2 Class Module

This class represents the concept of a single source file. Currently, Modules are only identified for the executable file; each shared library is made up of a single Module, ignoring any source file information that may be present. We also create a single module, called `DEFAULT_MODULE`, for each Symtab that contains any symbols for which module information was unavailable. This may be compiler template code, or files produced without module information.

supportedLanguages	Meaning
lang_Unknown	Unknown source language
lang_Assembly	Raw assembly code
lang_C	C source code
lang_CPlusPlus	C++ source code
lang_GnuCPlusPlus	C++ with GNU extensions
lang_Fortran	Fortran source code
lang_Fortran_with_pretty_debug	Fortran with debug annotations
lang_CMFortran	Fortran with CM extensions

Method name	Return type	Method description
isShared	bool	True if the module is for a shared library, false for an executable.
fullName	std::string &	Name, including path, of the source file represented by the module.
fileName	std::string &	Name, not including path, of the source file represented by the module.
language	supportedLanguages	The source language used by the Module.
addr	Offset	Offset of the start of the module, as reported by the symbol table, assuming contiguous modules.
exec	Symtab *	Symtab object that contains the module.

6.2.1 Function, Variable, Symbol lookup

```
bool findFunctionByEntryOffset(Function *&ret,
                               const Offset offset)
```

This method returns the `Function` object that begins at `offset`. Returns `true` on success and `false` if there is no matching function. The error value is set to `No_Such_Function`.

```
typedef enum {
    mangledName,
    prettyName,
    typedName,
    anyName
} NameType;
```

```
bool findFunctionsByName(vector<Function> &ret,
                         const string name,
                         Symtab::NameType nameType = anyName,
                         bool isRegex = false,
                         bool checkCase = true)
```

This method finds and returns a vector of `Functions` whose names match the given pattern. The `nameType` parameter determines which names are searched: mangled, pretty, typed, or any. If the

`isRegex` flag is set a regular expression match is performed with the symbol names. `checkCase` is applicable only if `isRegex` has been set. This indicates if the case be considered while performing regular expression matching. `ret` contains the list of matching `Function`s, if any. Returns `true` if it finds functions that match the given name, otherwise returns `false`. The error value is set to `No_Such_Function`.

```
bool getAllFunctions(vector<Function *> &ret)
```

This method returns all functions in the object file. Returns `true` on success and `false` if there are no modules. The error value is set to `No_Such_Function`.

```
bool findVariablesByOffset(std::vector<Variable *> &ret,
                           const Offset offset)
```

This method returns a vector of `Variable`s with the specified offset. There may be more than one variable at an offset if they have different sizes. Returns `true` on success and `false` if there is no matching variable. The error value is set to `No_Such_Variable`.

```
bool findVariablesByName(vector<Function> &ret,
                         const string &name,
                         Symtab::NameType nameType,
                         bool isRegex = false,
                         bool checkCase = true)
```

This method finds and returns a vector of `Variable`s whose names match the given pattern. The `nameType` parameter determines which names are searched: mangled, pretty, typed, or any (note: a `Variable` may not have a typed name). If the `isRegex` flag is set a regular expression match is performed with the symbol names. `checkCase` is applicable only if `isRegex` has been set. This indicates if the case be considered while performing regular expression matching. `ret` contains the list of matching `Variables`, if any. Returns `true` if it finds variables that match the given name, otherwise returns `false`. The error value is set to `No_Such_Variable`.

```
bool getAllSymbols(vector<Symbol *> &ret)
```

This method returns all symbols. Returns `true` on success and `false` if there are no symbols. The error value is set to `No_Such_Symbol`.

```
bool getAllSymbolsByType(vector<Symbol *> &ret,
                         Symbol::SymbolType sType)
```

This method returns all symbols whose type matches the given type `sType`. Returns `true` on success and `false` if there are no symbols with the given type. The error value is set to `No_Such_Symbol`.

6.2.2 Line number information

```
bool getAddressRanges(vector<pair<unsigned long, unsigned long> > & ranges,
                      string lineSource, unsigned int lineNo)
```

This method returns the address ranges in `ranges` corresponding to the line with line number `lineNo` in the source file `lineSource`. Searches only this module for the given source. Return `true` if at least one address range corresponding to the line number was found and returns false if none found.

```
bool getSourceLines(vector<Statement *> &lines,
                    Offset addressInRange)
```

This method returns the source file names and line numbers corresponding to the given address `addressInRange`. Searches only this module for the given source. Return `true` if at least one tuple corresponding to the offset was found and returns `false` if none found. The `Statement` class used to be named `LineNoTuple`; backwards compatibility is provided via typedef.

```
LineInformation *getLineInformation() const
```

This method returns the line map (section 7.1) corresponding to the module. Returns `NULL` if there is no line information existing for the module.

```
bool getStatements(std::vector<Statement *> &statements)
```

Returns all line information (section 7.2) available for the module.

6.2.3 Type information

```
bool findType(Type * &type,
              string name)
```

This method performs a look up and returns a handle to the named `type`. This method searches all the built-in types, standard types and user-defined types within the module. Returns `true` if a type is found with type containing the handle to the type, else return `false`.

```
bool findLocalVariable(vector<localVar *> &vars,
                       string name)
```

The method returns a list of local variables within the module with name `name`. Returns `true` with vars containing a list of `localVar` objects corresponding to the local variables if found or else returns `false`.

```
bool findVariableType(Type *&type,
                      std::string name)
```

This method looks up a global variable with name `name` and returns its type attribute. Returns `true` if a variable is found or returns `false` with `type` set to `NULL`.

6.3 Class FunctionBase

The `FunctionBase` class provides a common interface that can represent either a regular function or an inlined function.

Method name	Return type	Method description
getModule	const Module *	Module this function belongs to.
getSize	unsigned	Size encoded in the symbol table; may not be actual function size.
getRegion	Region *	Region containing this function.
getReturnType	Type *	Type representing the return type of the function.
getName	std::string	Returns primary name of the function (first mangled name or DWARF name)

```
bool setModule (Module *module)
```

This function changes the module to which the function belongs to `module`. Returns `true` if it succeeds.

```
bool setSize (unsigned size)
```

This function changes the size of the function to `size`. Returns `true` if it succeeds.

```
bool setOffset (Offset offset)
```

The method changes the offset of the function to `offset`. Returns `true` if it succeeds.

```
bool addMangledName(string name,
                    bool isPrimary)
```

This method adds a mangled name `name` to the function. If `isPrimary` is `true` then it becomes the default name for the function. This method returns `true` on success and `false` on failure.

```
bool addPrettyName(string name,
                   bool isPrimary)
```

This method adds a pretty name `name` to the function. If `isPrimary` is `true` then it becomes the default name for the function. This method returns `true` on success and `false` on failure.

```
bool addTypedName(string name,
                  bool isPrimary)
```

This method adds a typed name `name` to the function. If `isPrimary` is `true` then it becomes the default name for the function. This method returns `true` on success and `false` on failure.

```
bool getLocalVariables(vector<localVar *> &vars)
```

This method returns the local variables in the function. `vars` contains the list of variables found. If there is no debugging information present then it returns `false` with the error code set to `NO_DEBUG_INFO` accordingly. Otherwise it returns `true`.

```
std::vector<VariableLocation> &getFramePtr()
```

This method returns a list of frame pointer offsets (abstract top of the stack) for the function. See the `VariableLocation` class description for more information.

```
bool getParams(vector<localVar *> &params)
```

This method returns the parameters to the function. `params` contains the list of parameters. If there is no debugging information present then it returns `false` with the error code set to `NO_DEBUG_INFO` accordingly. Returns `true` on success.

```
bool findLocalVariable(vector<localVar *> &vars,
                       string name)
```

This method returns a list of local variables within a function that have name `name`. `vars` contains the list of variables found. Returns `true` on success and `false` on failure.

```
bool setReturnType(Type *type)
```

Sets the return type of a function to `type`.

```
FunctionBase* getInlinedParent()
```

Gets the function that this function is inlined into, if any. Returns `NULL` if there is no parent.

```
const InlineCollection& getInlines()
```

Gets the set of functions inlined into this one (possibly empty).

6.4 Class Function

The `Function` class represents a collection of symbols that have the same address and a type of `ST_FUNCTION`. When appropriate, use this representation instead of the underlying `Symbol` objects.

Method name	Return type	Method description
getModule	const Module *	Module this function belongs to.
getOffset	Offset	Offset in the file associated with the function.
getSize	unsigned	Size encoded in the symbol table; may not be actual function size.
mangled_names_begin	Aggregate::name_iter	Beginning of a range of unique names of symbols pointing to this function.
mangled_names_end	Aggregate::name_iter	End of a range of unique names of symbols pointing to this function.
pretty_names_begin	Aggregate::name_iter	As above, but prettified with the demangler.
pretty_names_end	Aggregate::name_iter	As above, but prettified with the demangler.
typed_names_begin	Aggregate::name_iter	As above, but including full type strings.
typed_names_end	Aggregate::name_iter	As above, but including full type strings.
getRegion	Region *	Region containing this function.
getReturnType	Type *	Type representing the return type of the function.

```
bool getSymbols(vector<Symbol *> &syms) const
```

This method returns the vector of `Symbol`s that refer to the function.

```
bool setModule (Module *module)
```

This function changes the module to which the function belongs to `module`. Returns `true` if it succeeds.

```
bool setSize (unsigned size)
```

This function changes the size of the function to `size`. Returns `true` if it succeeds.

```
bool setOffset (Offset offset)
```

The method changes the offset of the function to `offset`. Returns `true` if it succeeds.

```
bool addMangledName(string name,
                    bool isPrimary)
```

This method adds a mangled name `name` to the function. If `isPrimary` is `true` then it becomes the default name for the function. This method returns `true` on success and `false` on failure.

```
bool addPrettyName(string name,
                   bool isPrimary)
```

This method adds a pretty name `name` to the function. If `isPrimary` is `true` then it becomes the default name for the function. This method returns `true` on success and `false` on failure.

```
bool addTypedName(string name,
                  bool isPrimary)
```

This method adds a typed name `name` to the function. If `isPrimary` is `true` then it becomes the default name for the function. This method returns `true` on success and `false` on failure.

```
bool getLocalVariables(vector<localVar *> &vars)
```

This method returns the local variables in the function. `vars` contains the list of variables found. If there is no debugging information present then it returns `false` with the error code set to `NO_DEBUG_INFO` accordingly. Otherwise it returns `true`.

```
std::vector<VariableLocation> &getFramePtr()
```

This method returns a list of frame pointer offsets (abstract top of the stack) for the function. See the `VariableLocation` class description for more information.

```
bool getParams(vector<localVar *> &params)
```

This method returns the parameters to the function. `params` contains the list of parameters. If there is no debugging information present then it returns `false` with the error code set to `NO_DEBUG_INFO` accordingly. Returns `true` on success.

```
bool findLocalVariable(vector<localVar *> &vars,
                       string name)
```

This method returns a list of local variables within a function that have name `name`. `vars` contains the list of variables found. Returns `true` on success and `false` on failure.

```
bool setReturnType(Type *type)
```

Sets the return type of a function to `type`.

6.5 Class InlinedFunction

The `InlinedFunction` class represents an inlined function, as found in DWARF information. Its interface is almost entirely inherited from `FunctionBase`.

```
std::pair<std::string, Dyninst::Offset> getCallsite()
```

Returns the file and line corresponding to the call site of an inlined function.

6.6 Class Variable

The `Variable` class represents a collection of symbols that have the same address and represent data.

Method name	Return type	Method description
getOffset	Offset	Offset associated with this variable.
getSize	unsigned	Size of this variable in the symbol table.
mangled_names_begin	Aggregate::name_iter	Beginning of a range of unique names of symbols pointing to this variable.
mangled_names_end	Aggregate::name_iter	End of a range of unique names of symbols pointing to this variable.
getType	Type *	Type of this variable, if known.
getModule	const Module *	Module that contains this variable.
getRegion	Region *	Region that contains this variable.

```
bool getSymbols(vector<Symbol *> &syms) const
```

This method returns the vector of `Symbol`s that refer to the variable.

```
bool setModule (Module *module)
```

This method changes the module to which the variable belongs. Returns `true` if it succeeds.

```
bool setSize (unsigned size)
```

This method changes the size of the variable to `size`. Returns `true` if it succeeds.

```
bool setOffset (Offset offset)
```

The method changes the offset of the variable. Returns `true` if it succeeds.

```
bool addMangledName(string name,
                    bool isPrimary)
```

This method adds a mangled name `name` to the variable. If `isPrimary` is `true` then it becomes the default name for the variable. This method returns `true` on success and `false` on failure.

```
bool addPrettyName(string name,
                   bool isPrimary)
```

This method adds a pretty name `name` to the variable. If `isPrimary` is `true` then it becomes the default name for the variable. This method returns `true` on success and `false` on failure.

```
bool addTypedName(string name,
                  bool isPrimary)
```

This method adds a typed name `name` to the variable. If `isPrimary` is `true` then it becomes the default name for the variable. This method returns `true` on success and `false` on failure.

```
bool setType(Type *type)
```

Sets the type of the variable to `type`.

6.7 Class Symbol

The `Symbol` class represents a symbol in the object file. This class holds the symbol information such as the mangled, pretty and typed names, the module in which it is present, type, linkage, offset and size.

SymbolType	Meaning
ST_UNKNOWN	Unknown type
ST_FUNCTION	Function or other executable code sequence
ST_OBJECT	Variable or other data object
ST_MODULE	Source file declaration
ST_SETION	Region declaration
ST_TLS	Thread-local storage declaration
ST_DELETED	Deleted symbol
ST_NOTYPE	Miscellaneous symbol

SymbolLinkage	Meaning
SL_UNKNOWN	Unknown linkage
SL_GLOBAL	Process-global symbol
SL_LOCAL	Process-local (e.g., static) symbol
SL_WEAK	Alternate name for a function or variable

The following two types are platform-specific:

```
typedef enum {
    SV_UNKNOWN,
    SV_DEFAULT,
    SV_INTERNAL,
    SV_HIDDEN,
    SV_PROTECTED
} SymbolVisibility;


typedef enum {
    TAG_UNKNOWN,
    TAG_USER,
    TAG_LIBRARY,
    TAG_INTERNAL
} SymbolTag;
```

Method name	Return type	Method description
getMangledName	string	Raw name of the symbol in the symbol table, including name mangling.
getPrettyName	string	Demangled name of the symbol with parameters (for functions) removed.
getTypedName	string	Demangled name of the symbol including full function parameters.
getModule	Module *	The module, if any, that contains the symbol.
getType	SymbolType	The symbol type (as defined above) of the symbol.
getLinkage	SymbolLinkage	The linkage (as defined above) of the symbol.
getVisibility	SymbolVisibility	The visibility (as defined above) of the symbol.
tag	SymbolTag	The tag (as defined above) of the symbol.
getOffset	Offset	The offset of the object the symbols refers to.
getSize	unsigned	The size of the object the symbol refers to.
getRegion	Region *	The region containing the symbol.
getIndex	int	The index of the symbol within the symbol table.
getStrIndex	int	The index of the symbol name in the string table.
isInDynSymtab	bool	If true, the symbol is dynamic and can be used as the target of an intermodule reference. Implies isInSymtab is false.
isInSymtab	bool	If true, the symbol is static. Implies isInDynSymtab is false.
isAbsolute	bool	If true, the offset encoded in the symbol is an absolute value rather than an offset.
isFunction	bool	If true, the symbol refers to a function.
getFunction	Function *	The Function that contains this symbol if such a Function exists.
isVariable	bool	If true, the symbol refers to a variable.
getVariable	Variable *	The Variable that contains this symbol if such a Variable exists.
getSymtab	Symtab *	The Symtab that contains this symbol.
getPtrOffset	Offset	For binaries with an OPD section, the offset in the OPD that contains the function pointer data structure for this symbol.
getLocalTOC	Offset	For platforms with a TOC register, the expected TOC for the object referred to by this symbol.
isCommonStorage	bool	True if the symbol represents a common section (Fortran).

```
SYMTAB_EXPORT Symbol(const std::string& name,
                     SymbolType type,
                     SymbolLinkage linkage,
                     SymbolVisibility visibility,
                     Offset offset,
                     Module *module = NULL,
                     Region *region = NULL,
                     unsigned size = 0,
                     bool dyamic = false,
                     bool absolute = false,
                     int index = -1,
                     int strindex = -1,
```

```
                    bool commonStorage = false)
```

Symbol creation interface:

name The mangled name of the symbol.

type The type of the symbol as specified above.

linkage The linkage of the symbol as specified above.

visibility The visibility of the symbol as specified above.

offset The offset within the file that the symbol refers to.

module The source code module the symbol should belong to; default is no module.

region The region the symbol belongs to; if left unset this will be determined if a new binary is generated.

size The size of the object the symbol refers to; defaults to 0.

dynamic If true, the symbol belongs to the dynamic symbol table (ELF) and may be used as the target of inter-module references.

absolute If true, the offset specified is treated as an absolute value rather than an offset.

index The index in the symbol table. If left unset, it will be determined when generating a new binary.

strindex The index in the string table that contains the symbol name. If left unset, it will be determined when generating a new binary.

commonStorage If true, the symbol references common storage (Fortran).

```
bool getVersionFileName(std::string &fileName)
```

This method retrieves the file name in which this symbol is present. Returns `false` if this symbol does not have any version information present otherwise returns `true`.

```
bool getVersions(std::vector<std::string> *&vers)
```

This method retrieves all the version names for this symbol. Returns `false` if the symbol does not have any version information present.

```
bool getVersionNum(unsigned &verNum)
```

This method retrieves the version number of the symbol. Returns `false` if the symbol does not have any version information present.

6.7.1 Symbol modification

Most elements of a `Symbol` can be modified using the functions below. Each returns `true` on success and `false` otherwise.

```
bool setSize (unsigned size)
bool setOffset (Offset newOffset)
bool setMangledName (string name)
bool setType (SymbolType sType)
bool setModule (Module *module)
bool setRegion (Region *region)
bool setDynamic (bool dyn)
bool setAbsolute (bool absolute)
bool setCommonStorage (bool common)
bool setFunction (Function *func)
bool setVariable (Variable *var)
bool setIndex (int index)
bool setStrIndex (int index)
bool setPtrOffset (Offset ptr)
bool setLocalTOC (Offset toc)
bool setVersionNum (unsigned num)
bool setVersionFileName (std::string &fileName)
bool setVersions (std::vector<std::string> &vers)
```

6.8 Class Archive

This is used only on ELF platforms. This class represents an archive. This class has information of all the members in the archives.

```
static bool openArchive(Archive *&img,
                        string name)
```

This factory method creates a new `Archive` object for an archive file on disk. This object serves as a handle to the parsed archive file. `name` represents the name of the archive to be parsed. The `Archive` object is returned in `img` if the parsing succeeds. This method returns `false` if the given file is not an archive. The error is set to `Not_An_Archive`. This returns `true` if the archive is parsed without an error. `printSymtabError()` should be called to get more error details.

```
static bool openArchive(Archive *&img,
                        char *mem_image,
                        size_t size)
```

This factory method creates a new `Archive` object for an archive file in memory. This object serves as a handle to the parsed archive file. `mem_image` represents the pointer to the archive to be parsed. `size` represents the size of the memory image. The `Archive` object is returned in `img` if the parsing succeeds. This method returns `false` if the given file is not an archive. The error is set to

`Not_An_Archive`. This returns `true` if the archive is parsed without an error. `printSymtabError()` should be called to get more error details. This method is not supported currently on all ELF platforms.

```
bool getMember(Symtab *&img,
               string member_name)
```

This method returns the member object handle if the member exists in the archive. `img` corresponds to the object handle for the member. This method returns `false` if the member with name `member_name` does not exist else returns `true`.

```
bool getMemberByOffset(Symtab *&img,
                       Offset memberOffset)
```

This method returns the member object handle if the member exists at the start offset `memberOffset` in the archive. `img` corresponds to the object handle for the member. This method returns `false` if the member with name `member_name` does not exist else returns `true`.

```
bool getAllMembers(vector <Symtab *> &members)
```

This method returns all the member object handles in the archive. Returns `true` on success with `members` containing the `Symtab` Objects for all the members in the archive.

```
bool isMemberInArchive(string member_name)
```

This method returns `true` if the member with name `member_name` exists in the archive or else returns `false`.

```
bool findMemberWithDefinition(Symtab *&obj,
                              string name)
```

This method retrieves the member in an archive which contains the definition to a symbol with mangled name `name`. Returns `true` with `obj` containing the `Symtab` handle to that member or else returns `false`.

```
static SymtabError getLastError()
```

This method returns an error value for the previously performed operation that resulted in a failure. SymtabAPI sets a global error value in case of error during any operation. This call returns the last error that occurred while performing any operation.

```
static string printError(SymtabError serr)
```

This method returns a detailed description of the enum value `serr` in human readable format.

6.9 Class Region

This class represents a contiguous range of code or data as encoded in the object file. For ELF, regions represent ELF sections.

perm_t	Meaning
RP_R	Read-only data
RP_RW	Read/write data
RP_RX	Read-only code
RP_RWX	Read/write code

RegionType	Meaning
RT_TEXT	Executable code
RT_DATA	Read/write data
RT_TEXTDATA	Mix of code and data
RT_SYMTAB	Static symbol table
RT_STRTAB	String table used by the symbol table
RT_BSS	0-initialized memory
RT_SYMVERSIONS	Versioning information for symbols
RT_SYMVERDEF	Versioning information for symbols
RT_SYMVERNEEDED	Versioning information for symbols
RT_REL	Relocation section
RT_RELA	Relocation section
RT_PLTREL	Relocation section for PLT (inter-library references) entries
RT_PLTRELA	Relocation section for PLT (inter-library references) entries
RT_DYNAMIC	Decription of library dependencies
RT_HASH	Fast symbol lookup section
RT_GNU_HASH	GNU-specific fast symbol lookup section
RT_OTHER	Miscellaneous information

Method name	Return type	Method description
getRegionNumber	unsigned	Index of the region in the file, starting at 0.
getRegionName	std::string	Name of the region (e.g. .text, .data).
getPtrToRawData	void *	Read-only pointer to the region's raw data buffer.
getDiskOffset	Offset	Offset within the file where the region begins.
getDiskSize	unsigned long	Size of the region's data in the file.
getMemOffset	Offset	Location where the region will be loaded into memory, modified by the file's base load address.
getMemSize	unsigned long	Size of the region in memory, including zero padding.
isBSS	bool	Type query for uninitialized data regions (zero disk size, non-zero memory size).
isText	bool	Type query for executable code regions.
isData	bool	Type query for initialized data regions.
getRegionPermissions	perm_t	Permissions for the region; perm_ t is defined above.
getRegionType	RegionType	Type of the region as defined above.
isLoadable	bool	True if the region will be loaded into memory (e.g., code or data), false otherwise (e.g., debug information).
isDirty	bool	True if the region's raw data buffer has been modified by the user.

```
static Region *createRegion(Offset diskOff,
                            perm_t perms,
                            RegionType regType,
                            unsigned long diskSize = 0,
                            Offset memOff = 0,
                            unsigned long memSize = 0,
                            std::string name = "",
                            char *rawDataPtr = NULL,
                            bool isLoadable = false,
                            bool isTLS = false,
                            unsigned long memAlign = sizeof(unsigned))
```

This factory method creates a new region with the provided arguments. The `memOff` and `memSize` parameters identify where the region should be loaded in memory (modified by the base address of the file); if `memSize` is larger than `diskSize` the remainder will be zero-padded (e.g., bss regions).

```
bool isOffsetInRegion(const Offset &offset) const
```

Return `true` if the offset falls within the region data.

```
void setRegionNumber(unsigned index) const
```

Sets the region index; the value must not overlap with any other regions and is not checked.

```
bool setPtrToRawData(void *newPtr,
                     unsigned long rawsize)
```

Set the raw data pointer of the region to `newPtr`. `rawsize` represents the size of the raw data buffer. Returns `true` if success or `false` when unable to set/change the raw data of the region. Implicitly changes the disk and memory sizes of the region.

```
bool setRegionPermissions(perm_t newPerms)
```

This sets the regions permissions to `newPerms`. Returns `true` on success.

```
bool setLoadable(bool isLoadable)
```

This method sets whether the region is loaded into memory at load time. Returns `true` on success.

```
bool addRelocationEntry(Offset relocationAddr,
                        Symbol *dynref,
                        unsigned long relType,
                        Region::RegionType rtype = Region::RT_REL)
```

Creates and adds a relocation entry for this region. The symbol `dynref` represents the symbol used by he relocation, `relType` is the (platform-specific) relocation type, and `rtype` represents whether the relocation is REL or RELA (ELF-specific).

```
vector<relocationEntry> &getRelocations()
```

Get the vector of relocation entries that will modify this region. The vector should not be modified.

```
bool addRelocationEntry(const relocationEntry& rel)
```

Add the provided relocation entry to this region.

```
bool patchData(Offset off,
               void *buf,
               unsigned size);
```

Patch the raw data for this region. `buf` represents the buffer to be patched at offset `off` and size `size`.

6.9.1 REMOVED

The following methods were removed since they were inconsistent and dangerous to use.

`Offset getRegionAddr() const`

Please use `getDiskOffset` or `getMemOffset` instead, as appropriate.

`unsigned long getRegionSize() const`

Please use `getDiskSize` or `getMemSize` instead, as appropriate.

6.10 Relocation Information

This class represents object relocation information.

`Offset target_addr() const`

Specifies the offset that will be overwritten when relocations are processed.

`Offset rel_addr() const`

Specifies the offset of the relocation itself.

`Offset addend() const`

Specifies the value added to the relocation; whether this value is used or not is specific to the relocation type.

`const std::string name() const`

Specifies the user-readable name of the relocation.

`Symbol *getDynSym() const`

Specifies the symbol whose final address will be used in the relocation calculation. How this address is used is specific to the relocation type.

`unsigned long getRelType() const`

Specifies the platform-specific relocation type.

6.11 Class ExceptionBlock

This class represents an exception block present in the object file. This class gives all the information pertaining to that exception block.

Method name	Return type	Method description
hasTry	bool	True if the exception block has a try block.
tryStart	Offset	Start of the try block if it exists, else 0.
tryEnd	Offset	End of the try block if it exists, else 0.
trySize	Offset	Size of the try block if it exists, else 0.
catchStart	Offset	Start of the catch block.

```
bool contains(Offset addr) const
```

This method returns `true` if the offset `addr` is contained with in the try block. If there is no try block associated with this exception block or the offset does not fall within the try block, it returns `false`.

6.12 Class localVar

This represents a local variable or parameter of a function.

Method name	Return type	Method description
getName	string &	Name of the local variable or parameter.
getType	Type *	Type associated with the variable.
getFileName	string &	File where the variable was declared, if known.
getLineNum	int	Line number where the variable was declared, if known.

```
vector<VariableLocation> &getLocationLists()
```

A local variable can be in scope at different positions and based on that it is accessible in different ways. Location lists provide a way to encode that information. The method retrieves the location list, specified in terms of `VariableLocation` structures (section 6.13) where the variable is in scope.

6.13 Class VariableLocation

The `VariableLocation` class is an encoding of the location of a variable in memory or registers.

```
typedef enum {
   storageUnset,
   storageAddr,
   storageReg,
```

```
   storageRegOffset
} storageClass;

typedef enum {
   storageRefUnset,
   storageRef,
   storageNoRef
} storageRefClass;

struct VariableLocation  {
    storageClass stClass;
    storageRefClass refClass;
    MachRegister mr_reg;
    long frameOffset;
    Address lowPC;
    Address hiPC;
}
```

A `VariableLocation` is valid within the address range represented by `lowPC` and `hiPC`. If these are 0 and (Address) -1, respectively, the `VariableLocation` is always valid.

The location represented by the `VariableLocation` can be determined by the user as follows:

- stClass == storageAddr

  refClass == storageRef The frameOffset member contains the address of a pointer to the variable.

  refClass == storageNoRef The frameOffset member contains the address of the variable.

- stClass == storageReg

  refClass == storageRef The register named by mr_reg contains the address of the variable.

  refClass == storageNoRef The register named by mr_reg member contains the variable.

- stClass == storageRegOffset

  refClass == storageRef The address computed by adding frameOffset to the contents of mr_reg contains a pointer to the variable.

  refClass == storageNoRef The address computed by adding frameOffset to the contents of mr_reg contains the variable.

7 API Reference - Line Number Interface

This section describes the line number interface for the SymtabAPI library. Currently this interface has the following capabilities:

- Look up address ranges for a given line number.
- Look up source lines for a given address.
- Add new line information. This information will be available for lookup, but will not be included with an emitted object file.

In order to look up or add line information, the user/application must have already parsed the object file and should have a Symtab handle to the object file. For more information on line information lookups through the Symtab class refer to Section 6. The rest of this section describes the classes that are part of the line number interface.

7.1 Class LineInformation

This class represents an entire line map for a module. This contains mappings from a line number within a source to the address ranges.

```
bool getAddressRanges(const char * lineSource,
                      unsigned int LineNo,
                      std::vector<AddressRange> & ranges)
```

This methos returns the address ranges in `ranges` corresponding to the line with line number `lineNo` in the source file `lineSource`. Searches within this line map. Return `true` if at least one address range corresponding to the line number was found and returns `false` if none found.

```
bool getSourceLines(Offset addressInRange,
                    std::vector<Statement *> & lines)
bool getSourceLines(Offset addressInRange,
                    std::vector<LineNoTuple> & lines)
```

These methods returns the source file names and line numbers corresponding to the given address `addressInRange`. Searches within this line map. Return `true` if at least one tuple corresponding to the offset was found and returns `false` if none found. Note that the order of arguments is reversed from the corresponding interfaces in `Module` and `Symtab`.

```
bool addLine(const char * lineSource,
             unsigned int lineNo,
             unsigned int lineOffset,
             Offset lowInclusiveAddr,
             Offset highExclusiveAddr)
```

This method adds a new line to the line Map. `lineSource` represents the source file name. `lineNo` represents the line number.

```
bool addAddressRange(Offset lowInclusiveAddr,
                     Offset highExclusiveAddr,
                     const char* lineSource,
                     unsigned int lineNo,
                     unsigned int lineOffset = 0);
```

This method adds an address range `[lowInclusiveAddr, highExclusiveAddr)` for the line with line number `lineNo` in source file `lineSource`.

`LineInformation::const_iterator begin() const`

This method returns an iterator pointing to the beginning of the line information for the module. This is useful for iterating over the entire line information present in a module. An example described in Section 7.3 gives more information on how to use `begin()` for iterating over the line information.

`LineInformation::const_iterator end() const`

This method returns an iterator pointing to the end of the line information for the module. This is useful for iterating over the entire line information present in a module. An example described in Section 7.3 gives more information on how to use `end()` for iterating over the line information.

7.2 Class Statement

A `Statement` is the base representation of line information.

Method name	Return type	Method description
startAddr	Offset	Starting address of this line in the file.
endAddr	Offset	Ending address of this line in the file.
getFile	std::string	File that contains the line.
getLine	unsigned int	Line number.
getColumn	unsigned int	Starting column number.

For backwards compatibility, this class may also be referred to as a `LineNoTuple`, and provides the following legacy member variables. They should not be used and will be removed in a future version of SymtabAPI.

Member	Return type	Method description
first	const char *	Equivalent to getFile.
second	unsigned int	Equivalent to getLine.
column	unsigned int	Equivalent to getColumn.

7.3 Iterating over Line Information

The `LineInformation` class also provides the ability for iterating over its data (line numbers and their corresponding address ranges). The following example shows how to iterate over the line information for a given module using SymtabAPI.

```
//Example showing how to iterate over the line information for a given module.
using namespace Dyninst;
using namespace SymtabAPI;
```

```
//Obj represents a handle to a parsed object file using symtabAPI
//Module handle for the module
Module *mod;

//Find the module \lq foo\rq within the object.
obj->findModuleByName(mod, "foo");

// Get the Line Information for module foo.
LineInformation *info = mod->getLineInformation();

//Iterate over the line information
LineInformation::const_iterator iter;
for( iter = info->begin(); iter != info->end(); iter++)
{
// First component represents the address range for the line
const std::pair<Offset, Offset> addrRange = iter->first;

//Second component gives information about the line itself.
LineNoTuple lt = iter->second;
}
```

8 API Reference - Type Interface

This section describes the type interface for the SymtabAPI library. Currently this interface has the following capabilities:

- Look up types within an object file.
- Extend the types to create new types and add them to the Symtab file representation. These types will be available for lookup but will not be added if a new object file is produced.

The rest of the section describes the classes that are part of the type interface.

8.1 Class Type

The class `Type` represents the types of variables, parameters, return values, and functions. Instances of this class can represent language predefined types (e.g. `int`, `float`), already defined types in the Object File or binary (e.g., structures compiled into the binary), or newly created types (created using the create factory methods of the corresponding type classes described later in this section) that are added to SymtabAPI by the user.

As described in Section 2.2, this class serves as a base class for all the other classes in this interface. An object of this class is returned from type look up operations performed through the Symtab class described in Section 6. The user can then obtain the specific type object from the generic Type class object. The following example shows how to get the specific object from a given `Type` object returned as part of a look up operation.

```
//Example shows how to retrieve a structure type object from a given ‘‘Type’’ object
using namespace Dyninst;
using namespace SymtabAPI;

//Obj represents a handle to a parsed object file using symtabAPI
//Find a structure type in the object file
Type *structType = obj->findType(‘‘structType1’’);

// Get the specific typeStruct object
typeStruct *stType = structType->isStructType();
```

`string &getName()`

This method returns the name associated with this type. Each of the types is represented by a symbolic name. This method retrieves the name for the type. For example, in the example above "structType1" represents the name for the `structType` object.

`bool setName(string zname)`

This method sets the name of this type to name. Returns `true` on success and `false` on failure.

```
typedef enum{
    dataEnum,
    dataPointer,
    dataFunction,
    dataSubrange,
    dataArray,
    dataStructure,
    dataUnion,
    dataCommon,
    dataScalar,
    dataTypedef,
    dataReference,
    dataUnknownType,
    dataNullType,
    dataTypeClass
} dataClass;
```

`dataClass getDataClass()`

This method returns the data class associated with the type. This value should be used to convert this generic type object to a specific type object which offers more functionality by using the corresponding query function described later in this section. For example, if this method returns `dataStructure` then the `isStructureType()` should be called to dynamically cast the `Type` object to the `typeStruct` object.

`typeId_t getID()`

This method returns the ID associated with this type. Each type is assigned a unique ID within the object file. For example an integer scalar built-in type is assigned an ID -1.

`unsigned getSize()`

This method returns the total size in bytes occupied by the type.

`typeEnum *getEnumType()`

If this `Type` hobject represents an enum type, then return the object casting the `Type` object to `typeEnum` otherwise return `NULL`.

`typePointer *getPointerType()`

If this `Type` object represents an pointer type, then return the object casting the `Type` object to `typePointer` otherwise return `NULL`.

`typeFunction *getFunctionType()`

If this `Type` object represents an `Function` type, then return the object casting the `Type` object to `typeFunction` otherwise return `NULL`.

`typeRange *getSubrangeType()`

If this `Type` object represents a `Subrange` type, then return the object casting the `Type` object to `typeSubrange` otherwise return `NULL`.

`typeArray *getArrayType()`

If this `Type` object represents an `Array` type, then return the object casting the `Type` object to `typeArray` otherwise return `NULL`.

`typeStruct *getStructType()`

If this `Type` object represents a `Structure` type, then return the object casting the `Type` object to `typeStruct` otherwise return `NULL`.

`typeUnion *getUnionType()`

If this `Type` object represents a `Union` type, then return the object casting the `Type` object to `typeUnion` otherwise return `NULL`.

`typeScalar *getScalarType()`

If this `Type` object represents a `Scalar` type, then return the object casting the `Type` object to `typeScalar` otherwise return `NULL`.

`typeCommon *getCommonType()`

If this `Type` object represents a `Common` type, then return the object casting the `Type` object to `typeCommon` otherwise return `NULL`.

`typeTypedef *getTypedefType()`

If this `Type` object represents a `TypeDef` type, then return the object casting the `Type` object to `typeTypedef` otherwise return `NULL`.

`typeRef *getRefType()`

If this `Type` object represents a `Reference` type, then return the object casting the `Type` object to `typeRef` otherwise return `NULL`.

8.2 Class typeEnum

This class represents an enumeration type containing a list of constants with values. This class is derived from `Type`, so all those member functions are applicable. `typeEnum` inherits from the `Type` class.

```
static typeEnum *create(string &name,
                        vector<pair<string, int> *> &consts,
                        Symtab *obj = NULL)
static typeEnum *create(string &name,
                        vector<string> &constNames,
                        Symtab *obj)
```

These factory methods create a new enumerated type. There are two variations to this function. `consts` supplies the names and Ids of the constants of the enum. The first variant is used when user-defined identifiers are required; the second variant is used when system-defined identifiers will be used. The newly created type is added to the `Symtab` object `obj`. If `obj` is `NULL` the type is not added to any object file, but it will be available for further queries.

```
bool addConstant(const string &constname,
                 int value)
```

This method adds a constant to an enum type with name `constName` and value `value`. Returns `true` on success and `false` on failure.

```
std::vector<std::pair<std::string, int> > &getConstants();
```

This method returns the vector containing the enum constants represented by a (name, value) pair of the constant.

```
bool setName(const char* name)
```

This method sets the new name of the enum type to `name`. Returns `true` if it succeeds, else returns `false`.

```
bool isCompatible(Type *type)
```

This method returns `true` if the enum type is compatible with the given type `type` or else returns `false`.

8.3 Class typeFunction

This class represents a function type, containing a list of parameters and a return type. This class is derived from `Type`, so all the member functions of class `Type` are applicable. `typeFunction` inherits from the `Type` class.

```
static typeFunction *create(string &name,
                            Type *retType,
                            vector<Type *> &paramTypes,
                            Symtab *obj = NULL)
```

This factory method creates a new function type with name `name`. `retType` represents the return type of the function and `paramTypes` is a vector of the types of the parameters in order. The the newly created type is added to the `Symtab` object `obj`. If `obj` is `NULL` the type is not added to any object file, but it will be available for further queries.

```
bool isCompatible(Type *type)
```

This method returns `true` if the function type is compatible with the given type `type` or else returns `false`.

`bool addParam(Type *type)`

This method adds a new function parameter with type `type` to the function type. Returns `true` if it succeeds, else returns `false`.

`Type *getReturnType() const`

This method returns the return type for this function type. Returns `NULL` if there is no return type associated with this function type.

`bool setRetType(Type *rtype)`

This method sets the return type of the function type to `rtype`. Returns `true` if it succeeds, else returns `false`.

`bool setName(string &name)`

This method sets the new name of the function type to `name`. Returns `true` if it succeeds, else returns `false`.

`vector< Type *> &getParams() const`

This method returns the vector containing the individual parameters represented by their types in order. Returns `NULL` if there are no parameters to the function type.

8.4 Class typeScalar

This class represents a scalar type. This class is derived from `Type`, so all the member functions of class `Type` are applicable. `typeScalar` inherits from the Type class.

`static typeScalar *create(string &name, int size, Symtab *obj = NULL)`

This factory method creates a new scalar type. The `name` field is used to specify the name of the type, and the `size` parameter is used to specify the size in bytes of each instance of the type. The newly created type is added to the `Symtab` object `obj`. If `obj` is `NULL` the type is not added to any object file, but it will be available for further queries.

`bool isSigned()`

This method returns `true` if the scalar type is signed or else returns `false`.

```
bool isCompatible(Type *type)
```

This method returns `true` if the scalar type is compatible with the given type `type` or else returns `false`.

8.5 Class Field

This class represents a field in a container. For e.g. a field in a structure/union type.

```
typedef enum {
    visPrivate,
    visProtected,
    visPublic,
    visUnknown
} visibility_t;
```

A handle for identifying the visibility of a certain `Field` in a container type. This can represent private, public, protected or unknown(default) visibility.

```
Field(string &name,
      Type *type,
      visibility_t vis = visUnknown)
```

This constructor creates a new field with name `name`, type `type` and visibility `vis`. This newly created `Field` can be added to a container type.

```
const string &getName()
```

This method returns the name associated with the field in the container.

```
Type *getType()
```

This method returns the type associated with the field in the container.

```
int getOffset()
```

This method returns the offset associated with the field in the container.

```
visibility_t getVisibility()
```

This method returns the visibility associated with a field in a container. This returns `visPublic` for the variables within a common block.

8.6 Class fieldListType

This class represents a container type. It is one of the three categories of types as described in Section 2.2. The structure and the union types fall under this category. This class is derived from `Type`, so all the member functions of class `Type` are applicable. `fieldListType` inherits from the `Type` class.

```
vector<Field *> *getComponents()
```

This method returns the list of all fields present in the container. This gives information about the name, type and visibility of each of the fields. Returns `NULL` of there are no fields.

```
void addField(std::string fieldname,
              Type *type,
              int offsetVal = -1,
              visibility_t vis = visUnknown)
```

This method adds a new field at the end to the container type with field name `fieldname`, type `type` and type visibility `vis`.

```
void addField(unsigned num,
              std::string fieldname,
              Type *type,
              int offsetVal = -1,
              visibility_t vis = visUnknown)
```

This method adds a field after the field with number `num` with field name `fieldname`, type `type` and type visibility `vis`.

```
void addField(Field *fld)
```

This method adds a new field `fld` to the container type.

```
void addField(unsigned num,
              Field *fld)
```

This method adds a field `fld` after field `num` to the container type.

8.6.1 Class typeStruct : public fieldListType

This class represents a structure type. The structure type is a special case of the container type. The fields of the structure represent the fields in this case. As a subclass of class `fieldListType`, all methods in `fieldListType` are applicable.

```
static typeStruct *create(string &name,
                          vector<pair<string, Type *>*> &flds,
                          Symtab *obj = NULL)
```

This factory method creates a new struct type. The name of the structure is specified in the `name` parameter. The `flds` vector specifies the names and types of the fields of the structure type. The newly created type is added to the `Symtab` object `obj`. If `obj` is `NULL` the type is not added to any object file, but it will be available for further queries.

```
static typeStruct *create(string &name,
                          vector<Field *> &fields,
                          Symtab *obj = NULL)
```

This factory method creates a new struct type. The name of the structure is specified in the `name` parameter. The `fields` vector specifies the fields of the type. The newly created type is added to the `Symtab` object `obj`. If `obj` is `NULL` the type is not added to any object file, but it will be available for further queries.

```
bool isCompatible(Type *type)
```

This method returns `true` if the struct type is compatible with the given type `type` or else returns `false`.

8.6.2 Class typeUnion

This class represents a union type, a special case of the container type. The fields of the union type represent the fields in this case. As a subclass of class `fieldListType`, all methods in `fieldListType` are applicable. `typeUnion` inherits from the `fieldListType` class.

```
static typeUnion *create(string &name,
                         vector<pair<string, Type *>*> &flds,
                         Symtab *obj = NULL)
```

This factory method creates a new union type. The name of the union is specified in the `name` parameter. The `flds` vector specifies the names and types of the fields of the union type. The newly created type is added to the `Symtab` object `obj`. If `obj` is `NULL` the type is not added to any object file, but it will be available for further queries.

```
static typeUnion *create(string &name,
                         vector<Field *> &fields,
                         Symtab *obj = NULL)
```

This factory method creates a new union type. The name of the structure is specified in the `name` parameter. The `fields` vector specifies the fields of the type. The newly created type is added to the `Symtab` object `obj`. If `obj` is `NULL` the type is not added to any object file, but it will be available for further queries.

```
bool isCompatible(Type *type)
```

This method returns `true` if the union type is compatible with the given type `type` or else returns `false`.

8.6.3 Class typeCommon

This class represents a common block type in fortran, a special case of the container type. The variables of the common block represent the fields in this case. As a subclass of class `fieldListType`, all methods in `fieldListType` are applicable. `typeCommon` inherits from the `Type` class.

```
vector<CBlocks *> *getCBlocks()
```

This method returns the common block objects for the type. The methods of the `CBlock` can be used to access information about the members of a common block. The vector returned by this function contains one instance of `CBlock` for each unique definition of the common block.

8.6.4 Class CBlock

This class represents a common block in Fortran. Multiple functions can share a common block.

```
bool getComponents(vector<Field *> *vars)
```

This method returns the vector containing the individual variables of the common block. Returns `true` if there is at least one variable, else returns `false`.

```
bool getFunctions(vector<Symbol *> *funcs)
```

This method returns the functions that can see this common block with the set of variables described in `getComponents` method above. Returns `true` if there is at least one function, else returns `false`.

8.7 Class derivedType

This class represents a derived type which is a reference to another type. It is one of the three categories of types as described in Section 2.2. The pointer, reference and the typedef types fall under this category. This class is derived from `Type`, so all the member functions of class `Type` are applicable.

```
Type *getConstituentType() const
```

This method returns the type of the base type to which this type refers to.

8.7.1 Class typePointer

This class represents a pointer type, a special case of the derived type. The base type in this case is the type this particular type points to. As a subclass of class `derivedType`, all methods in `derivedType` are also applicable.

```
static typePointer *create(string &name,
                           Type *ptr,
                           Symtab *obj = NULL)
static typePointer *create(string &name,
                           Type *ptr,
                           int size,
                           Symtab *obj = NULL)
```

These factory methods create a new type, named `name`, which points to objects of type `ptr`. The first form creates a pointer whose size is equal to sizeof(void*) on the target platform where the application is running. In the second form, the size of the pointer is the value passed in the `size` parameter. The newly created type is added to the `Symtab` object `obj`. If obj is `NULL` the type is not added to any object file, but it will be available for further queries.

```
bool isCompatible(Type *type)
```

This method returns `true` if the Pointer type is compatible with the given type `type` or else returns `false`.

```
bool setPtr(Type *ptr)
```

This method sets the pointer type to point to the type in `ptr`. Returns `true` if it succeeds, else returns `false`.

8.7.2 Class typeTypedef

This class represents a `typedef` type, a special case of the derived type. The base type in this case is the `Type`. This particular type is typedefed to. As a subclass of class `derivedType`, all methods in `derivedType` are also applicable.

```
static typeTypedef *create(string &name,
                           Type *ptr,
                           Symtab *obj = NULL)
```

This factory method creates a new type called `name` and having the type `ptr`. The newly created type is added to the `Symtab` object `obj`. If `obj` is `NULL` the type is not added to any object file, but it will be available for further queries.

```
bool isCompatible(Type *type)
```

This method returns `true` if the typedef type is compatible with the given type `type` or else returns `false`.

8.7.3 Class typeRef

This class represents a reference type, a special case of the derived type. The base type in this case is the `Type` this particular type refers to. As a subclass of class `derivedType`, all methods in `derivedType` are also applicable here.

```
static typeRef *create(string &name,
                       Type *ptr,
                       Symtab * obj = NULL)
```

This factory method creates a new type, named `name`, which is a reference to objects of type `ptr`. The newly created type is added to the `Symtab` object `obj`. If `obj` is `NULL` the type is not added to any object file, but it will be available for further queries.

```
bool isCompatible(Type *type)
```

This method returns `true` if the ref type is compatible with the given type `type` or else returns `false`.

8.8 Class rangedType

This class represents a range type with a lower and an upper bound. It is one of the three categories of types as described in section 2.2. The sub-range and the array types fall under this category. This class is derived from `Type`, so all the member functions of class `Type` are applicable.

```
unsigned long getLow() const
```

This method returns the lower bound of the range. This can be the lower bound of the range type or the lowest index for an array type.

```
unsigned long getHigh() const
```

This method returns the higher bound of the range. This can be the higher bound of the range type or the highest index for an array type.

8.8.1 Class typeSubrange

This class represents a sub-range type. As a subclass of class `rangedType`, all methods in `rangedType` are applicable here. This type is usually used to represent a sub-range of another type. For example, a `typeSubrange` can represent a sub-range of the array type or a new integer type can be declared as a sub range of the integer using this type.

```
static typeSubrange *create(string &name,
                            int size,
                            int low,
                            int hi,
                            symtab *obj = NULL)
```

This factory method creates a new sub-range type. The name of the type is `name`, and the size is `size`. The lower bound of the type is represented by `low`, and the upper bound is represented by `high`. The newly created type is added to the `Symtab` object `obj`. If `obj` is `NULL` the type is not added to any object file, but it will be available for further queries.

```
bool isCompatible(Type *type)
```

This method returns `true` if this sub range type is compatible with the given type `type` or else returns `false`.

8.8.2 Class typeArray

This class represents an `Array` type. As a subclass of class `rangedType`, all methods in `rangedType` are applicable.

```
static typeArray *create(string &name,
                         Type *type,
                         int low,
                         int hi,
                         Symtab *obj = NULL)
```

This factory method creates a new array type. The name of the type is `name`, and the type of each element is `type`. The index of the first element of the array is `low`, and the last is `high`. The newly created type is added to the `Symtab` object `obj`. If `obj` is `NULL` the type is not added to any object file, but it will be available for further queries.

```
bool isCompatible(Type *type)
```

This method returns `true` if the array type is compatible with the given type `type` or else returns `false`.

```
Type *getBaseType() const
```

This method returns the base type of this array type.

9 API Reference - Dynamic Components

Unlike the static components discussed in Section 6, which operate on files, SymtabAPI's dynamic components operate on a process. The dynamic components currently consist of the Dynamic Address Translation system, which translates between absolute addresses in a running process and static SymtabAPI objects.

9.1 Class AddressLookup

The `AddressLookup` class provides a mapping interface for determining the address in a process where a SymtabAPI object is loaded. A single dynamic library may load at different addresses in different processes. The 'address' fields in a dynamic library's symbol tables will contain offsets rather than absolute addresses. These offsets can be added to the library's load address, which is computed at runtime, to determine the absolute address where a symbol is loaded.

The `AddressLookup` class examines a process and finds its dynamic libraries and executables and each one's load address. This information can be used to map between SymtabAPI objects and absolute addresses.

Each `AddressLookup` instance is associated with one process. An `AddressLookup` object can be created to work with the currently running process or a different process on the same system.

On the Linux platform the `AddressLookup` class needs to read from the process' address space to determine its shared objects and load addresses. By default, `AddressLookup` will attach to another process using a debugger interface to read the necessary information, or simply use `memcpy` if reading from the current process. The default behavior can be changed by implementing a new ProcessReader class and passing an instance of it to the create`AddressLookup` factor constructors. The ProcessReader class is discussed in more detail in Section 9.2.

When an `AddressLookup` object is created for a running process it takes a snapshot of the process' currently loaded libraries and their load addresses. This snapshot is used to answer queries into the `AddressLookup` object, and is not automatically updated when the process loads or unloads libraries. The refresh function can be used to updated an `AddressLookup` object's view of its process.

```
static AddressLookup *createAddressLookup(ProcessReader *reader = NULL)
```

This factory constructor creates a new `AddressLookup` object associated with the process that called this function. The returned `AddressLookup` object should be cleaned with the delete operator when it is no longer needed. If the reader parameter is non-NULL on Linux then the new `AddressLookup` object will use reader to read from the target process. This function returns the new `AddressLookup` object on success and `NULL` on error.

```
static AddressLookup *createAddressLookup(PID pid,
                                          ProcessReader *reader = NULL)
```

This factory constructor creates a new `AddressLookup` object associated with the process referred to by `pid`. The returned `AddressLookup` object should be cleaned with the delete operator when it is no longer needed. If the `reader` parameter is non-NULL on Linux then the new `AddressLookup` object will use it to read from the target process. This function returns the new `AddressLookup` object on success and `NULL` on error.

```
typedef struct {
    std::string name;
    Address codeAddr;
    Address dataAddr;
} LoadedLibrary;

static AddressLookup *createAddressLookup(const std::vector<LoadedLibrary> &ll)
```

This factory constructor creates a new `AddressLookup` associated with a previously collected list of libraries from a process. The list of libraries can initially be collected with the `getLoadAddresses` function. The list can then be used with this function to re-create the `AddressLookup` object, even if the original process no longer exists. This can be useful for off-line address lookups, where only the load addresses are collected while the process exists and then all address translation is done after the process has terminated. This function returns the new `AddressLookup` object on success and `NULL` on error.

```
bool getLoadAddresses(std::vector<LoadedLibrary> &ll)
```

This function returns a vector of LoadedLibrary objects that can be used by the `createAddressLookup(const std::vector<LoadedLibrary> &ll)` function to create a new `AddressLookup` object. This function is usually used as part of an off-line address lookup mechanism. This function returns `true` on success and `false` on error.

```
bool refresh()
```

When a `AddressLookup` object is initially created it takes a snapshot of the libraries currently loaded in a process, which is then used to answer queries into this API. As the process runs more libraries may be loaded and unloaded, and this snapshot may become out of date. An `AddressLookup`'s view of a process can be updated by calling this function, which causes it to examine the process for loaded and unloaded objects and update its data structures accordingly. This function returns `true` on success and `false` on error.

```
bool getAddress(Symtab *tab,
                Symbol *sym,
                Address &addr)
```

Given a `Symtab` object, `tab`, and a symbol, `sym`, this function returns the address, `addr`, where the symbol can be found in the process associated with this `AddressLookup`. This function returns `true` if it was able to successfully lookup the address of `sym` and `false` otherwise.

```
bool getAddress(Symtab *tab,
                Offset off,
                Address &addr)
```

Given a `Symtab` object, `tab`, and an offset into that object, `off`, this function returns the address, `addr`, of that location in the process associated with this `AddressLookup`. This function returns `true` if it was able to successfully lookup the address of sym and `false` otherwise.

```
bool getSymbol(Address addr,
               Symbol * &sym,
               Symtab* &tab,
               bool close = false)
```

Given an address, `addr`, this function returns the `Symtab` object, `tab`, and `Symbol`, `sym`, that reside at that address. If the close parameter is `true` then `getSymbol` will return the nearest symbol that comes before `addr`; this can be useful when looking up the function that resides at an address. This function returns `true` if it was able to find a symbol and `false` otherwise.

```
bool getOffset(Address addr,
               Symtab* &tab,
               Offset &off)
```

Given an address, `addr`, this function returns the `Symtab` object, `tab`, and an offset into `tab`, `off`, that reside at that address. This function returns `true` on success and `false` otherwise.

```
bool getOffset(Address addr,
               LoadedLibrary &lib,
               Offset &off)
```

As above, but returns a `LoadedLibrary` data structure instead of a Symtab.

```
bool getAllSymtabs(std::vector<Symtab *> &tabs)
```

This function returns all `Symtab` objects that are contained in the process represented by this `AddressLookup` object. This will include the process's executable and all shared objects loaded by this process. This function returns `true` on success and `false` otherwise.

```
bool getLoadAddress(Symtab *sym,
                    Address &load_address)
```

Given a `Symtab` object, `sym`, that resides in the process associated with this `AddressLookup`, this function returns `sym`'s load address. On systems where an object can have one load address for its code and one for its data, this function will return the code's load address. Use `getDataLoadAddress` to get the data load address. This function returns `true` on success and `false` otherwise.

```
bool getDataLoadAddress(Symtab *sym,
                        Address &load_addr)
```

Given a Symtab object, `sym`, this function returns the load address of its data section. This function returns `true` on success and `false` otherwise.

9.2 Class ProcessReader

The implementation of the `AddressLookup` on Linux requires it to be able to read from the target process's address space. By default, reading from another process on the same system this is done through the operating system debugger interface. A user can provide their own process reading mechanism by implementing a child of the `ProcessReader` class and passing it to the `AddressLookup` constructors. The API described in this section is an interface that a user can implement. With the exception of the `ProcessReader` constructor, these functions should not be called by user code.

The `ProcessReader` is defined, but not used, on non-Linux systems.

```
ProcessReader()
```

This constructor for a `ProcessReader` should be called by any child class constructor.

```
virtual bool ReadMem(Address traced,
                     void *inSelf,
                     unsigned size) = 0
```

This function should read `size` bytes from the address at `traced` into the buffer pointed to by `inSelf`. This function must return `true` on success and `false` on error.

```
virtual bool GetReg(MachRegister reg,
                    MachRegisterVal &val) = 0
```

This function reads from the register specified by `reg` and places the result in `val`. It must return `true` on success and `false` on failure.