Paradyn Parallel Performance Tools

SymtabAPI Programmer's Guide

Release 7.0
March 2011

Paradyn Project
www.paradyn.org

Computer Sciences Department
University of Wisconsin
Madison, WI 53706-1685

Computer Sciences Department
University of Maryland
College Park, MD 20742

bugs@dyninst.org

Table of 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, and POWER), XCOFF (POWER), and PE (Windows) object file formats. In addition, it also supports the DWARF and stabs debugging formats.

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 allows user 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.

This document describes SymtabAPI, an Application Program Interface (API) for parsing object files or binaries, for querying symbols and debug information, and for interacting with the library to add, remove or change symbols or other useful information.

The SymtabAPI interface is designed to be small and easy to understand, while remaining sufficiently expressive to be useful for a variety of applications. The way we have done this is by providing a simple set of abstractions and a simple way to interact with the API.

2 ABSTRACTIONS

SymtabAPI provides a simple set of abstractions over complicated data structures which makes it straight-forward to use. The SymtabAPI consists of four interfaces : the symbol table interface, the type interface, the line map interface and the local variable 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 the section. The API functions in each of the classes are described in detail in Section 6.

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 sup-

ports. 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.
- *Function* - A `Function` class object identifies a function within the binary object.
- *Variable* - A `Variable` class object identifies variable within the binary object.
- *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.

2.2 Type Interface

The Type interface is responsible for parsing type information from the object file and han-

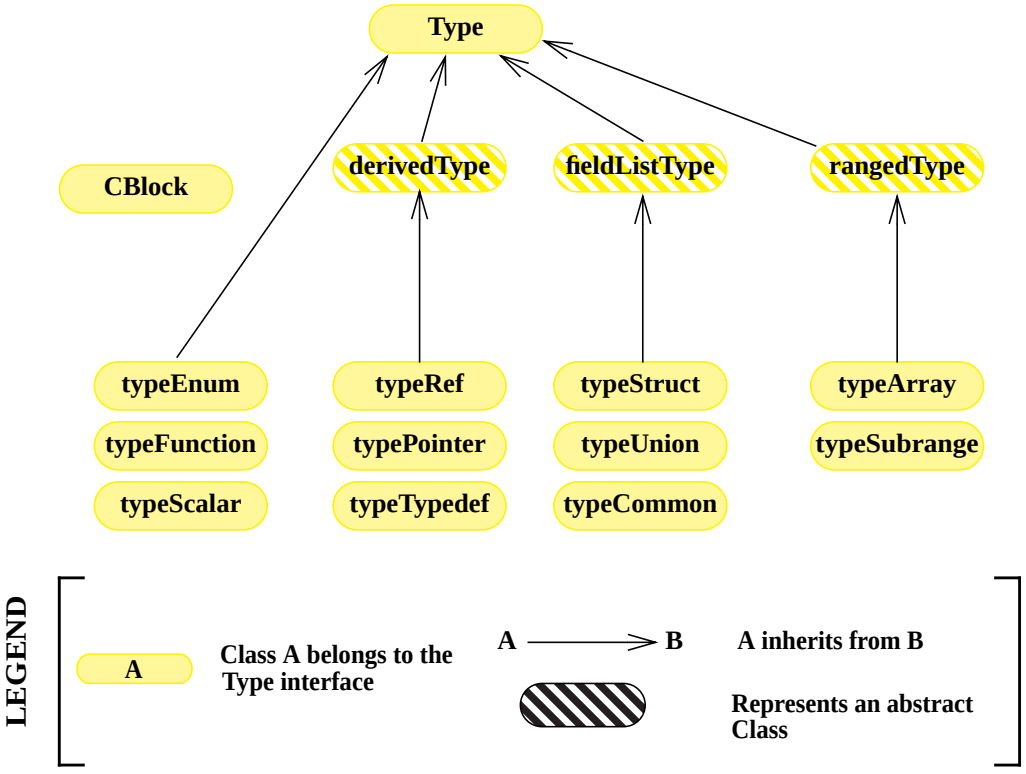

Figure 2: SymtabAPI Type Interface - Class Inheritance Diagram

dling 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.

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).
- *LineNoTuple* - A `LineNoTuple` class object represents a location in source code with a source file, line number in that source file and start column in that line.

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(file, obj);
```

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) name to the first function
       funcs[0]->addMangledName("newname");
}

// search for symbol of any type with demangled (pretty) name "bar".
if (obj->findSymbolByType(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->findModule(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;

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

// 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(obj,"intArray",intType,0,9);

//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(obj, "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( "foo.c", 30, ranges );
```

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->findSymbolByType(syms, "bar", Symbol::ST_FUNCTION);

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

Parsing an XCOFF object file is different than the rest of the file formats as the XCOFF format also allows archives to contain dynamic libraries. The following examples shows how to open an XCOFF archive using SymtabAPI on AIX.

```
//Name the object file to be parsed
std::string file = "libfoo.a"

// Declare a pointer to an object of type Archive. This represents the archive.
Archive *archive = NULL;

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

// Try to parse the archive file; error handling omitted for simplicity
Archive::openArchive(file, archive);

// Get the Symtab object for the desired object
archive->getMember("bar.o", obj);
// Obj now contains a handle to the desired binary object "bar.o"
```

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.
- *`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 solve various problems caused by the need to resolve unique names for pro-

gramming entities in many modern programming languages. For example, a function foo with signature `void foo()` has a mangled name `_Z8foov` when compiled with gcc.

- *`Pretty Name`* - A pretty name for a symbol represents a user-level symbolic name for a symbol. For example, `foo` can be a pretty name for a function declared as `void foo()`.
- *`Typed Name`* - A typed name for a symbol represents the user-level symbolic name complete with the signature. For example, `void foo()` can be a typed name for the function foo.
- *`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 variablethat 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 Section 6 are under the C++ namespace `Dyninst::SymtabAPI`. To access them a user should refer to them using the `Dyninst::` and `SymtabAPI::` prefixs, 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 - STATIC COMPONENTS

This section describes the five interfaces of SymtabAPI. Each of the subsections represent a different interface.

6.1 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
- Getting relocation, exception information
- Getting header information

The primary classes for this interface are Symtab, Module, Symbol, Archive and ExceptionBlock.

6.1.1 Class Symtab

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.

6.1.1.1 File opening/parsing

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

This factory method[1] 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. The error is set to `Object_Parsing. printSymtabError()` should be called to get more error details.

```
static bool openFile(Symtab *&obj, char *mem_image, size_t size)
```

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

1. "Factory method" is an object-oriented design pattern term that describes a method that is responsible for constructing new objects. SymtabAPI uses factory methods to allow it to return an error when constructing an object.

file in memory to be parsed. `size` indicates the size of the image. The Symtab object is returned in `obj` if the parsing succeeds.

Returns true if the file is parsed without an error, else returns false. The error is set to `Object_Parsing. printSymtabError()` should be called to get more error details.

6.1.1.2 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 atleast 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 atleast one data 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 atleast one new region that is added to the object file or else returns false.

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

Find a section wih name `sname` in the binary. It returns true with `reg` containing a handle to the section, if there is a section with that name or else returns false with `reg` set to NULL.

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

Find a section with starting offset `soff` in the binary. It returns true with `reg` containing a handle to the section, if there is a section at that offset or else returns false with `reg` set to NULL.

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

Find the region handle which contains this offset. Returns the region handle if some region contains the offset in the object file; otherwise returns NULL. If this method is called on a Symtab object representing a .o file, the behavior is undefined.

6.1.1.3 Module, Function, Variable, and Symbol lookup

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

This method searches for a module with name `name`. `ret` contains the handle to the module found.

If the module exists return true, or else return 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`.

```
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 = false)
```

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 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`.

```
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 findVariableByOffset(Variable *&ret, const Offset offset)
```

This method returns the Variable object at `offset`. 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 = false)
```

This method finds and returns a vector of Variables 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 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 findSymbolByType(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.

```
bool getAllSymbolsByType(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 atleast one such symbol or else returns false with the error set to `No_Such_Symbol`.

6.1.1.4 Function, Variable, and Symbol insertion and modification

```
Function *createFunction(string name, Offset offset, unsigned size, Module *mod)
```

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(string name, Offset offset, unsigned size, Module *mod)
```

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. `isDynamic` represents whether the symbol belongs to the dynamic symbol table; if false, the symbol is added to the static symbol table. 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 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 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 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.

6.1.1.5 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 ExceptionBlocks.

```
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.

6.1.1.6 Symtab information

```
bool isExec() const
```

This method returns true if the Object is an executable or else returns false.

```
bool isStripped()
```

This method returns true if the Object is stripped or else returns false.

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

ObjectType getObjectType() const
```

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

```
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.

```
const string &file() const
```

This method returns the full path name of the object file. Returns an empty string if the object file is a memory image.

```
const string &name() const
```

This method returns the name of the object file. Returns an empty string if the object file is a memory image.

```
char *mem_image() const
```

This method returns the pointer to the object file in memory. Returns NULL if its a file on disk and not a memory image.

```
Offset imageOffset() const
Offset dataOffset() const
```

These methods return the code/data segment address values for the object, offset from the start (base) of the binary.

```
Offset imageLength() const
Offset dataLength() const
```

These methods return the code/data segment length values for the binary.

```
char* image_ptr () const
char* data_ptr () const
```

These methods return a pointer to the start of the code or the data section for the binary.

```
unsigned getNumberOfSymbols() const
```

This method return the total number of symbols present in the binary

The number returned represents all the symbols present in the `.symtab` section of the binary.

```
unsigned getAddressWidth() const
```

This method returns the address width in bytes for the object file. On 32-bit systems this function will return 4, and on 64-bit systems this function will return 8.

```
Offset getLoadOffset() const
```

This method returns the offset of the load section of the object file.

```
Offset getEntryOffset() const
```

This method returns the entry point offset of the object file.

```
Offset getBaseOffset() const
```

This method returns the base address of the object file.

6.1.1.7 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.

6.1.1.8 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.

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

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

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

Returns all the bulit-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.

- `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_Module` - No matching module exists with the given inputs
- `No_Such_Function` - No matching module exists with the given inputs
- `No_Such_Variable` - No matching variable 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
- `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
- `No_Error` - Previous operation did not result in failur.

```
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.

6.1.1.9 Deprecated methods

```
bool findSymbolByType(vector <Symbol *> &ret, const string name,
                        Symbol::SymbolType sType, bool isMangled = false,
                        bool isRegex = false, bool checkCase = false)
```

This method finds and returns a vector of symbols with type `sType` whose names match the given Mangled/Pretty name based on the `isMangled` flag. 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 .

This method is deprecated; please use the alternate `findSymbolByType` defined above.

```
bool findFuncByEntryOffset(vector<Symbol *> &ret, const Offset offset)
```

This method returns a vector of function symbols with the given offset `offset`.

Returns true on success and false if there is no symbol with the given offset. The error value is set to `No_Such_Symbol`.

This method is deprecated; please use `findFunctionByEntryOffset` defined above.

6.1.2 Class Module

This class represents a module which is part of the image. This class also gives all the information pertaining to a module.

```
typedef enum{
            lang_Unknown,
            lang_Assembly,
            lang_C,
            lang_CPlusPlus,
            lang_GnuCPlusPlus,
            lang_Fortran,
            lang_Fortran_with_pretty_debug,
            lang_CMFortran
}supportedLanguages
```

This represents the language of the source file which has been compiled into the object file.

6.1.2.1 Module information

```
bool isShared() const
```

This method returns true if the module is shared, else return false.

```
const string& fullName() const
```

This method returns the full path name for the module.

There is always a file name corresponding to a module.

```
const string& fileName() const
```

This method returns the file name for the module.

There is always a file name corresponding to a module.

```
bool setName(string newName)
```

This method sets the full name of the module to `newName`. Returns true on success and false on failure.

```
supportedLanguages language() const
bool setLanguage(supportedLanguages lang)
```

These methods return the language for the module / set it to `lang`.

```
Offset addr() const
```

This method returns the offset for the module from the start of the object file.

```
Symtab *exec() const
```

Returns the Symtab object to which this module belongs to.

6.1.2.2 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 = false)
```

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 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.

```
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 findVariableByOffset(Variable *&ret, const Offset offset)
```

This method returns the Variable object at offset. 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 = false)
```

This method finds and returns a vector of Variables 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 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.

```
bool getAllSymbolsByType(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`.

6.1.2.3 Line number information

```
bool hasLineInformation()
```

Return true if the module has line information present or else it returns false.

```
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<LineNoTuple> &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.

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

This method returns the line map corresponding to the module. Returns NULL if there is no line information existing for the module.

6.1.2.4 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.

6.1.2.5 Deprecated

```
bool findSymbolByType(vector <Symbol *> &ret, const string name,
                      Symbol::SymbolType sType, bool isMangled = false,
                      bool isRegex = false, bool checkCase = false)
```

This method finds and returns a vector of symbols within the module with type `sType` where the name matches the given Mangled/Pretty name based on the `isMangled` flag. 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 .

This method is deprecated; use the replacement `findSymbolByType` defined above.

6.1.3 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.

```
const Module *getModule () const
```

This method returns the module handle to which this symbol belongs. If the symbol does not belong to any module NULL is returned.

```
Offset getOffset() const
```

This method returns the offset associated with this function.

```
unsigned getSize() const
```

This method returns the size of the function (as reported by the symbol table).

```
vector<string> &getAllMangledNames() const
vector<string> &getAllPrettyNames() const
vector<string> &getAllTypedNames() const
```

This method returns a list of appropriate (mangled/unmangled/typed) names associated with this function, including all aliases (alternate names).

```
Region *getRegion() const
```

This method returns a handle to the region in which this function is present. Returns NULL if the symbol is not associated with a region.

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

This method returns the vector of `Symbols` 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 = false)
```

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 = false)
```

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 = false)
```

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.

```
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.

```
Type *getReturnType()
```

Retrive the return type of a function.

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

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

6.1.4 Class Variable

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

```
const Module *getModule () const
```

This method returns the module handle to which this symbol belongs. If the symbol does not belong to any module NULL is returned.

```
Offset getOffset() const
```

This method returns the offset associated with this function.

```
unsigned getSize() const
```

This method returns the size of the function (as reported by the symbol table).

```
vector<string> &getAllMangledNames() const
vector<string> &getAllPrettyNames() const
vector<string> &getAllTypedNames() const
```

This method returns a the list of appropriate (mangled/unmangled/typed) names associated with this function, including all aliases (alternate names).

```
Region *getRegion() const
```

This method returns a handle to the region in which this function is present. Returns NULL if the symbol is not associated with a region.

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

This method returns the vector of `Symbols` 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 = false)
```

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 = false)
```

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 = false)
```

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.

```
Type *getType()
```

Retrive the return type of a variable.

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

Sets the type of the variable to `type`.

6.1.5 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.

```
typedef enum {
            ST_UNKNOWN,
            ST_FUNCTION,
            ST_OBJECT,
            ST_MODULE,
            ST_SECTION,
            ST_TLS,
            ST_DELETED,
            ST_NOTYPE
} SymbolType
```

This represents the type for a symbol. `ST_UNKNOWN` represents an unknown type, `ST_FUNCTION` represents a function type. `ST_OBJECT` represents the variable type, `ST_MODULE` represents the module type, `ST_SECTION` represents sections, `ST_TLS` represents thread local storage, `ST_DELETED` represents deleted symbols, and `ST_NOTYPE` represents the existence of no type.

```
typedef enum {
            SL_UNKNOWN,
            SL_GLOBAL,
            SL_LOCAL,
            SL_WEAK
} SymbolLinkage

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

6.1.5.1 Symbol information

```
Symbol (const string name, SymbolType stype,
        SymbolLinkage slinkage, SymbolVisibility visibility,
        Offset addr, Module *mod = NULL, Region *reg = NULL,
        unsigned size = 0, bool isDynamic = false,
        bool isAbsolute = true)
```

This method creates an object of class Symbol with the mangled name `name`, the symbol type `stype`, the symbol linkage `slinkage`, the symbol visibility `visibility`, and the offset `addr`. The remaining fields are optional and assigned defaults if not present. If `isDynamic` is true the symbol belongs to the dynamic symbol table (ELF only); the default is to place the symbol in the static symbol table.

The void pointer can be used to extend the Symbol class to store any additional information. This is particularly useful when converting between Symbol to any other object.

```
bool operator== (const Symbol &sym) const
```

This method compares this symbol with sym.

Returns true if all the members of the object `sym` are same as the members of this object.

```
const string& getMangledName () const
```

This method returns the primary mangled name for the symbol. Refer to Section 4 for more details on a mangled name.

```
const string& getPrettyName() const
```

This method returns the primary pretty (unmangled) name for the symbol. Refer to Section 4 for more details on a mangled name.

```
const string& getTypedName() const
```

This method returns the primary typed name for the symbol. Refer to Section 4 for more details on a typed name.

```
const Module *getModule () const
```

This method returns the module handle to which this symbol belongs. If the symbol does not belong to any module NULL is returned.

```
SymbolType getType() const
```

This method returns the type of the symbol.

```
SymbolLinkage getLinkage() const
```

This method returns the linkage associated with this symbol.

```
Offset getOffset() const
```

This method returns the offset associated with this symbol.

```
unsigned getSize() const
```

This method returns the size occupied by a symbol in bytes.

```
Section *getRegion() const
```

This method returns a handle to the section in which this symbol is present. Returns NULL if the symbol is not associated with a section.

```
bool isInDynSymtab() const
```

Returns true if the symbol belongs to the dynamic symbol table

```
bool isInSymtab() const
```

Returns true if the symbol belongs to the regular symbol table.

```
bool isAbsolute() const
```

Returns true if the symbol has the absolute flag set.

```
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.

```
Function *getFunction()
```

```
Variable *getVariable()
```

Return the `Function` or `Variable` that contains this symbol if such exists; otherwise return NULL.

6.1.5.2 Symbol modification

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

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

```
bool setSize (unsigned ns)
```

The method changes the size of the symbol to `ns`. Returns true if it succeeds.

```
bool setOffset(Offset newOffset)
```

This method changes the offset of the symbol to `newAddr`. Returns true if it succeeds.

```
bool setType(SymbolType sType)
```

This method sets the type of the symbol to `sType`. Returns true if it succeeds, else returns false.

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

This sets the version file name for the symbol. Returns true on success.

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

This sets the version names for this symbols to the names in `vers`. Returns true on success.

```
bool setMangledName(string name
```

This method sets the mangled name of the symbol to `name`.

`bool setPrettyName(string name`

This method sets the mangled name of the symbol to `name`.

`bool setTypedName(string name`

This method sets the mangled name of the symbol to `name`.

6.1.5.3 Deprecated

`Offset getAddr() const`

Please use the equivalent `getOffset` instead.

`Region *getSec() const`

Please use the equivalent `getRegion` instead.

`const string &getName() const`

Please use the equivalent `getMangledName` instead.

`const string &getModuleName () const`

Please use `getModule` instead.

`bool setModuleName (string moduleName)`

Please use `setModule` instead.

`bool setAddr(Offset newAddr)`

Please use `setOffset` instead.

6.1.6 Class Archive

This is used only on AIX and 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. `print-SymtabError()` 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. `printSymtab-Error()` 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 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.

6.1.7 Class ExceptionBlock

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

```
bool hasTry() const
```

This method returns true if the exception block has a try block, else returns false.

```
Offset tryStart() const
```

This methods returns the start offset of the try block.

If the exception block does not have a try block it returns 0.

```
Offset tryEnd() const
```

This method returns the end offset of the try block.

If the exception block does not have a try block it returns 0.

```
Offset trySize() const
```

This method returns the size of the try block. This returns 0 in case the exception block does not have a try block.

```
Offset catchStart() const
```

This method returns the start address 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.

6.1.8 Class Region

This class represents a Region present in the object file. This conatins information of sections and the segments(for ELF only) in case none of the sections fall under it.

```
enum perm_t{
            RP_R, RP_RW, RP_RX, RP_RWX }
```

This enum represents the permissions of a region. `RP_R` indicates that it is just readable, `RP_RW` indicates that is both read and write, `RP_RX` indicates that it has both read and execute permissions and finally `RP_RWX` indicates that it has all read, write and execute permissions

```
enum RegionType {
            RT_TEXT, RT_DATA, RT_TEXTDATA, RT_SYMTAB, RT_STRTAB ,
            RT_BSS, RT_SYMVERSIONS, RT_SYMVERDEF, RT_SYMVERNEEDED,
            RT_REL, RT_RELA, RT_PLTREL, RT_PLTRELA, RT_DYNAMIC, RT_HASH,
RT_GNU_HASH, RT_OTHER };
```

This enum represents the type of the region whether it is text, data, text and data, symbol table, string table, bss, symbol versions, symbol version definitions, symbol version requirements, relocations (2 types), PLT relocations (2 types), hash symbols (2 types), dynamic or other type following the order specified above.

```
static bool createRegion( Offset diskOff, perm_t perms, region_t regType,
                          unsigned long diskSize = 0, Offset memOff = 0,
                          unsigned long memSize = 0, std::string name = "",
                          char *rawDataPtr = NULL);
```

This factory method creates a new region with disk offset `diskOff`, permissions `perms`, region type `regType`, disk size `diskSize`, memory offset `memOff`, memory size `memSize`, name `name` and the raw data pointer `rawDataPtr`.

```
unsigned getRegionNumber() const;
```

This method returns the region number index for the region.

```
bool setRegionNumber(unsigned sidnumber);
```

This method is used to set a region number for a section. This is mostly used by the rewritter where regions are to be added.

```
std::string getRegionName() const;
```

Retrieve the symbolic name of a Region.

```
Offset getRegionAddr() const;
```

Get the Offset of the Region from the start of the object file. This generally returns the disk offset on all ELF/XCOFF platforms and the memory offset on windows which is the most accurate value.

```
unsigned long getRegionSize() const
```

Gets the region size of this region. This returns the disk size on all ELF/XCOFF platforms and the memory size on windows which is the most accurate value.

```
void *getPtrToRawData() const;
```

Retrive the pointer to the raw data of the section

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

Set the raw data pointer of the section 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 section. Implicitly changes the size of the section.

```
Offset getDiskOffset() const
```

Returns the disk offset for this region.

```
unsigned long getDiskSize() const
```

Returns the disk size for this region.

```
Offset getMemOffset() const
```

Returns the memoryoffset for this region.

```
unsigned long getMemSize() const
```

Returns the memory size for this region.

```
bool isBSS() const;
bool isText() const;
bool isData() const;
```

Methods to query the type of the Region.

```
perm_t getRegionPermissions() const
```

This returns the permissions for this region.

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

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

```
RegionType getRegionType() const
```

This returns the region type for this region.

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

Return true if the offset falls within the region data or else returns false.

```
bool isLoadable() const;
```

Return true if the section is loaded into memory at execution time.

```
bool setLoadable() const;
```

This method sets the region to be loadable into memory at load time. Returns true on success.

```
bool isDirty() const;
```

Return true if the raw data of the section has been changed.

```
std::vector<relocationEntry> &getRelocations();
```

Get relocations pertaining this the section. Returns an empty vector if there are no relocations for this section

```
bool addRelocationEntry(Offset relocationAddr, Symbol *dynref,
                        unsigned long relType);
```

Add a relocation entry for this section. `dynref` represents the dynamic symbol with respect to which the relocation should be made. `relType` is one of the numerous relocation types which specifies type of relocation to be applied.

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

Patch the raw data for this section. `buf` reprsents the buffer to be patched with at offset `off` and size `size`.

6.2 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 object file.

To look up types or to add new types, the object file should already be parsed and should have a Symtab handle to the object file. The rest of the section describes the classes that are part of the type interface.

6.2.1 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 the 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();
```

```
const 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 truie on success and false on failure.

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

DataClass getType()
```

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 isStructure-Type() 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 object 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.

6.2.2 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 = NULL)
```

These factory methods create a new enumerated type. There are two variations to this function. `consts` supplies the names and Id's 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.

```
vector< pair<string, int> *> *getConstants();
```

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

```
bool setName(string &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. For type compatibility rules, see 7

6.2.3 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. `type-Function` 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. For type compatibility rules see 7.

```
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.

6.2.4 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. For type compatibility rules see 7.

6.2.5 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.

6.2.6 Class fieldListType

This class represents a container type. It is one of the three categories of types as described in Section . 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.

```
bool addField(string &fieldname, Type *type,
                              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`. Returns true on success and false on failure.

```
bool addField(string &fieldname, Type *type,int num
                              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`. Returns true on success and false on failure.

```
bool addField(Field *fld)
```

This method adds a new field `fld` to the container type.Returns true on success and false on failure.

```
bool addField(Field *fld, int num)
```

This method adds a field `fld` after field `num` to the container type. Returns true on success and false on failure.

6.2.6.1 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 `fieldList-Type`, 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. For type compatibility rules see 7

6.2.6.2 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. For type compatibility rules see 7.

6.2.6.3 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

6.2.6.4 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.

6.2.7 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 . 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.

6.2.7.1 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. For type compatibility rules see 7

```
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.

6.2.7.2 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 typedef'ed 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. For type compatibility rules see 7

6.2.7.3 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, int size,
                                                Symtab * obj = NULL)
```

This factory method creates a new type, named *name*, which is a reference to objects of type `ptr`. 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 isCompatible(Type *type)
```

This method returns true if the ref type is compatible with the given type `type` or else returns false. For type compatibility rules see 7

6.2.8 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 . 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.

6.2.8.1 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. For type compatibility rules see 7

6.2.8.2 Class typeArray

This class represents an Array type. As a subclass of class rangedType, all methods in ranged-Type 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. For type compatibility rules see 7

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

This method returns the base type of this array type.

6.3 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.

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.

6.3.1 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(vector< pair<unsigned long, unsigned long> > & ranges,
                                    string &lineSource, unsigned int LineNo)
```

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

This method 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.

```
bool addLine(string &lineSource, unsigned int lineNo,
                        unsigned int lineOffset, Offset lowInclusiveAddr,
                        Offset highExclusiveAddr, Symtab *obj = NULL)
```

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

If `obj` is `NULL` it is just added to that line map or else the new line is added back to the library.

```
bool addAddressRange(Offset lowInclusiveAddr, Offset highExclusiveAddr,
                               string &lineSource, unsigned int lineNo,
                               unsigned int lineOffset = 0, Symtab *obj = NULL);
```

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

If `obj` is `NULL` it is just added to that line map or else the new line is added back to the library.

```
LineInformation::const_iterator begin()
```

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 6.3.3 gives more information on how to use `begin()` for iterating over the line information.

```
LineInformation::const_iterator end()
```

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 6.3.3 gives more information on how to use `end()` for iterating over the line information.

6.3.2 Class LineNoTuple

LineNoTuple is tuple of three objects: source file where a line is present, line number in that source file and start column in that line. It's members can be accessed directly.

```
const chat *first
```

This member gives the source file for the line

```
unsigned int second
```

This member gives the line number for the line

```
unsigned int column
```

This member gives the start column for that line.

6.3.3 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 "foo" within the object.
obj->findModule(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;
}
```

6.4 Local Variable Interface

This section describes the local variable interface for the SymtabAPI library. Currently this interface has the following capabilities

- Look up local variables and parameters within an function.

- Look up local variables local to a particular scope.
- Add new local variables.

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

```
typedef enum {
            storageAddr,
            storageReg,
            storageRegOffset,
} storageClass;
```

This represents a handle that encodes how a variable is stored and gives information on how to access it. If the value is `storageAddr` then absolute address of the variable is available. If the value is `storageReg` then the register holds the variable value. If the value is `storageRegOffset` then the address of the variable can be calculated as `$reg+address`.

```
typedef enum{
            storageAddrRef,
            storageRegRef,
} StorageRefClass;
```

This gives information on whether a variable can be accessed through an address or a register. If the value is `storageAddrRef` then the variable can be accessed through a pointer and the address of that pointer is available. If the value is `storageRegRef` then the variable can be accessed through a register which holds the pointer.

```
typedef struct{
      storageClass stClass;         //storage class for the variable
      storageRef refClass;          //storage reference for the variable
      int reg;                      //Register in which the local variable will
                                    //be stored
} VariableLocation;
```

This represents a particular scope for a variable where it is local. Gives information on how to access the variable based on the location.

6.4.1 Class localVar

This represents a local variable or parameter within a function.

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

This method returns the name of the local variable.

```
Type *getType()
```

This method returns the type of the local variable.

```
string &getFileName()
```

This method returns the filename where this local variable was declared.

```
int getLineNum()
```

This method returns the line number at which the local variable was declared.

```
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 where the variable is in scope. These are represented by the loc_t objects which give information on the storage class and how to access it.

7 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.

7.1 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 libraries' symbol tables will contain offsets rather than absolute addresses. These offsets can be added to the libraries' 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 and Solaris platforms 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 `createAddressLookup` factor constructors. The `ProcessReader` class is discussed in more detail in Section 7.1.1.

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 or Solaris 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 or Solaris 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.

```
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 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' 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 the AIX system, 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 will return the data load address on AIX systems only, all other supported operating systems return zero.

This function returns `true` on success and `false` otherwise.

7.1.1 Class ProcessReader

The implementation of the `AddressLookup` on Linux and Solaris requires it to be able to read from the target process' address space. By default, reading from another process on the same system this is done through the OS's 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 and Solaris systems.

```
PID pid
```

The `pid` member variable of `ProcessReader` refers to the PID of the process being read from.

```
ProcessReader(PID pid_)
```

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

```
virtual bool start() = 0
```

This function is called by `AddressLookup` before it begins a batch of reads from a process. Multiple `readAddressSpace` calls may follow the call to begin, which are then concluded with a call to `done`. This function may be useful for any initialization that needs to be done before reads. Note that this function signifies the start of a batch of reads, and multiple batches of reads occur during the lifetime of a `ProcessReader`.

This function should return `true` on success and `false` on error.

```
virtual bool readAddressSpace(Address traced, unsigned amount, void *inSelf) = 0
```

This function should read `amount` bytes from the address at `traced` into the buffer pointed to by `inSelf`.

This function should return `true` on success and `false` on error.

```
virtual bool done() = 0
```

This function is called to signify the completion of a batch of reads. It is guaranteed to be paired with a `start` call.

This function should return `true` on success and `false` on error.

Appendix A: Building SymtabAPI

This appendix describes how to build SymtabAPI from source code, which can be downloaded from http://www.paradyn.org or http://www.dyninst.org.

Building on Unix

Building SymtabAPI on UNIX platforms is a four step process that involves: unpacking the SymtabAPI source, installing any SymtabAPI dependencies, configuring paths in make.config.local, and running the build.

SymtabAPI's source code is packaged in a tar.gz format. If your SymtabAPI source tarball is called `symtab_src_1.0.tar.gz`, then you could extract it with the command `gunzip symtab_src_1.0.tar.gz; tar -xvf symtab_src_1.0.tar`. This will create two directories: core and scripts.

SymtabAPI has several dependencies, depending on what platform you are using, which must be installed before SymtabAPI can be built. Note that for most of these packages Symtab needs to be able to access the package's include files, which means that development versions are required. If a version number is listed for a packaged, then there are known bugs that may affect Symtab with earlier versions of the package.

Linux/x86	libdwarf-20060327 libelf libxml2
Linux/IA-64	libdwarf-20060327

	libunwind-0.98.5 libelf libxml2
Linux/x86-64	libdwarf-20060327 libelf libxml2
Solaris/Sparc	libxml2
AIX/Power	libxml2
Windows/x86	libxml2

At the time of this writing the Linux packages could be found at:

- libdwarf - http://reality.sgiweb.org/davea/dwarf.html
- libelf - http://www.mr511.de/software/english.html
- libunwind - http://www.hpl.hp.com/research/linux/libunwind/download.php4
- libxml2 - http://xmlsoft.org/downloads.html

Once the dependencies for SymtabAPI have been installed, SymtabAPI must be configured to know where to find these packages. This is done through SymtabAPI's `core/make.config.local` file. This file must be written in GNU Makefile syntax and must specify directories for each dependency. Specifically, LIBDWARFDIR, LIBELFDIR and LIBXML2DIR variables must be set. LIBDWARFDIR should be set to the absolute path of libdwarf library where `dwarf.h` and `libdwarf.h` files reside. LIBELFDIR should be set to the absolute path where `libelf.a` and `libelf.so` files are located. Finally, LIBXML2DIR to the absolute path where libxml2 is located.

The next thing is to set DYNINST_ROOT, PLATFORM, and LD_LIBRARY_PATH environment variables. DYNINST_ROOT should be set to the path of the directory that contains core and scripts subdirectories.

PLATFORM should be set to one of the following values depending upon what operating system you are running on:

alpha-dec-osf5.1	Tru64 UNIX on the Alpha processor
i386-unknown-linux2.4	Linux 2.4/2.6 on an Intel x86 processor
ia64-unknown-linux2.4	Linux 2.4/2.6 on an IA-64 processor
rs6000-ibm-aix5.1	AIX Version 5.1
sparc-sun-solaris2.9	Solaris 2.9 on a SPARC processor
x86_64-unknown-linux2.4	Linux 2.4/2.6 on an AMD-64 processor

LD_LIBRARY_PATH variable should be set in a way that it includes libdwarf home directory/lib and ${DYNINST_ROOT}/${PLATFORM}/lib directories.

Once make.config.local is set you are ready to build SymtabAPI. Change to the core directory and execute the command make SymtabAPI. This will build the SymtabAPI library. Successfully

built binaries will be stored in a directory named after your platform at the same level as the core directory.

Building on Windows

SymtabAPI for Windows is built with Microsoft Visual Studio 2003 project and solution files. Building SymtabAPI for Windows is similar to UNIX in that it is a four step process: unpack the SymtabAPI source code, install SymtabAPI's package dependencies, configure Visual Studio to use the dependencies, and run the build system.

SymtabAPI's source code is distributed as part of a tar.gz package. Most popular unzipping programs are capable of handling this format. Extracting the Symtab tarball results in two directories: core and scripts.

Symtab for Windows depends on Microsoft's Debugging Tools for Windows, which could be found at http://www.microsoft.com/whdc/devtools/debugging/default.mspx at the time of this writing. Download these tools and install them at an appropriate location. Make sure to do a custom install and install the SDK, which is not always installed by default. For the rest of this section, we will assume that the Debugging Tools are installed at `c:\program files\Debugging Tools for Windows`. If this is not the case, then adjust the following instruction appropriately.

Once the Debugging Tools are installed, Visual Studio must be configured to use them. We need to add the Debugging Tools include and library directories to Visual Studios search paths. In Visual Studio 2003 select `Options...` from the `tools` menu. Next select `Projects` and `VC++ Directories` from the pane on the left. You should see a list of directories that are sorted into categories such as 'Executable files', 'Include files', etc. The current category can be changed with a drop down box in the upper right hand side of the Dialog.

First, change to the 'Library files' category, and add an entry that points to `C:\Program Files\Debugging Tools for Windows\sdk\lib\i386`. Make sure that this entry is above Visual Studio's default search paths.

Next, Change to the 'Include files' category and make a new entry in the list that points to `C:\Program Files\Debugging Tools for Windows\sdk\inc`. Also make sure that this entry is above Visual Studio's default search paths. Some users have had a problem where Visual Studio cannot find the cvconst.h file. You may need to add the directory containing this file to the include search path. We have seen it installed at `$(VCInstallDir)/../Visual Studio SDKs/DIA SDK/include`, although you may need to search for it. You also need to add the libxml2 include path depending on the where the libxml2 is installed on the system.

Once you have installed and configured the Debugging Tools for Windows you are ready to build Symtab. First, you need to create the directories where Dyninst will install its completed build. From the core directory you need to create the directories `../i386-unknown-nt4.0/bin` and `../i386-unknown-nt4.0/lib`. Next open the solution file core/SymtabAPI.sln with Visual Studio. You can then build SymtabAPI by select 'Build Solution' from the build menu. This will build the SymtabAPI library.