Binder Documentation

Release 0.1

Sergey Lyskov

May 13, 2021

Contents

Contents:

CHAPTER 1

About this project

Binder is a tool for automatic generation of Python bindings for C++11 projects using Pybind11 and Clang LibTooling libraries. That is, Binder, takes a C++ project and compiles it into objects and functions that are all usable within Python. Binder is different from prior tools in that it handles special features new in C++11.

Initially Binder was developed to create PyRosetta - Python bindings for Rosetta project. Using such rather large C++ code base (Rosetta have ~3M SLOC of C++11 code) allowed to develop fully automated solution capable of bindings almost any C++11 code without need of human intervention.

1.1 Core features

Automatically determine which types/functions could be bound and generate bindings for the following C++ code:

- functions
- enums
- C++ classes and structs, including:
- read/write access to public data members, including static data members
- class member functions including static functions
- class inner enums
- class operators
- automatically determine which template classes should be instantiated and generate bindings for it
- generate single shared library which on Python-import act as collection of Python modules representing hierarchy of C++ namespaces
- generated bindings for both Python-2.6+ and Python-3+

1.2 Goodies

Bindings generation is guided through config file which allows to configure:

- bindings for which namespace/type/function bindings should be generated
- default return policy for functions returning pointer, lvalue and rvalue references
- additional includes which should be added to generated code
- includes which should be ignored in generated codes
- specify custom bindings functions for type/template
- specify add-on bindings functions which will be called after automatic bindings was generated

CHAPTER 2

Installation

Binder is written in C++11 and must be built before use. This page describes the steps for the build process. Please note that installation require up to ~2.6+ Gb of free disk space.

2.1 Requirements

The following tools need to be present in order to build and use **Binder**

- CMake, https://cmake.org - static compilation requires version 3.13 or above, see below
- Pybind11, RosettaCommons fork: https://github.com/RosettaCommons/pybind11
- [optional] Ninja (or you can use *make* by omitting *-G Ninja* command below)

Binder has experimental support for being *statically compiled* on CentOS8, which additionally requires:

- *libclang-static-build*: https://github.com/deech/libclang-static-build
- CMake version 3.13 or above

Go to *Building Statically (Linux only)* for the build process. Also note a caveat to static compilation: the version of *libclang* that Binder is compiled against may not be compatible with the header files on the system Binder where is run.

2.2 Building

The steps below are encoded in *binder/build.py* and *binder/build-and-run-tests.py* files so for default install you can just run *build-and-run-tests.py* script directly. This section describes how to build a dynamically-linked `binder` executable. To *statically* compile binder, see *Building Statically (Linux only)*.

1. To build Binder execute the following command sequence in shell (replace `$HOME/prefix` and `$HOME/binder` with your paths):

```
# clone Binder
cd $HOME
git clone https://github.com/RosettaCommons/binder.git

# Create build dir
mkdir $HOME/prefix && cd $HOME/prefix

# Clone  LLVM
git clone http://llvm.org/git/llvm.git llvm && cd llvm
git checkout release_60

# Clone Clang
cd $HOME/prefix/llvm/tools
git clone http://llvm.org/git/clang.git clang
cd clang && git checkout release_60


# Clone Clang extra tools
cd $HOME/prefix/llvm/tools/clang/tools
git clone http://llvm.org/git/clang-tools-extra.git extra
cd extra && git checkout release_60

# Create symlink pointing to binder/src dir
ln -s $HOME/binder/source $HOME/prefix/llvm/tools/clang/tools/extra/binder

# Create ``llvm/tools/clang/tools/extra/CMakeLists.txt`` file with content: ``add_
↪subdirectory(binder)``
echo 'add_subdirectory(binder)' > $HOME/prefix/llvm/tools/clang/tools/extra/
↪CMakeLists.txt

# Build Binder
mkdir $HOME/prefix/build && cd $HOME/prefix/build
cmake -G Ninja -DCMAKE_BUILD_TYPE=Release -DLLVM_ENABLE_EH=1 -DLLVM_ENABLE_RTTI=ON ../
↪llvm && ninja

# At this point, if all above steps is successful, binder should be at
# $HOME/prefix/build/bin/binder
```

CHAPTER 3

Installation with pre-installed LLVM

3.1 Requirements

The basic dependencies for this type of installation are very similar to these described above and include:

- CMake, version 3.4.3 or higher from https://cmake.org
- C++ compiler with c++11 support, e.g. gcc from https://gcc.gnu.org/
- make or Ninja
- llvm with development packages (headers)
- clang with development packages (headers)

The installation process of the required packages varies from system to system. On the RHEL7/RHEL8/Fedora22+/Ubuntu18+ systems binder can be compiled with the llvm, clang and dependent packages available for these systems from their default repositories

For RHEL7/RHEL8/Fedora22+:

- To install the needed packages run as root

```
yum install clang clang-devel llvm-devel llvm-static clang-libs
```

- If a newer or specific version of the llvm/clang is needed, it can be installed as root

```
yum install clang8.0 clang8.0-devel llvm8.0-devel llvm8.0-static clang8.0-libs
```

  to obtain a specific version (8.0 in this case).

- If the option above is not sufficient, or the available packages are outdated, for the CentOS/RHEL/Fedora and compatible systems the llvm-toolset-7.0 toolset (or later) from https://www.softwarecollections.org/en/scls/rhscl/llvm-toolset-7.0/ provides LLVM of version 7.0. To install it run as root

```
yum install llvm-toolset-7.0*
```

  Then the compilation can be performed using the following shell

```
scl enable llvm-toolset-7.0 bash
```

- Please note that binder requires cmake of version 3, therefore for some older systems package cmake3 should be installed and used instead of cmake.

```
yum install cmake3
```

For Ubuntu18+ run, an example for LLVM/Clang 10:

```
sudo apt-get update
sudo apt-get -y install  clang-10 llvm-10 libclang-10-dev llvm-10-dev
sudo apt-get -y install  cmake make gcc g++
```

For MacOSX:

Make sure the XCode is installed. If needed, install cmake, python and other utilities, e.g. using homebrew:

```
brew install wget coreutils xz pybind11 cmake
```

Note: the pybind11 version from https://github.com/RosettaCommons/pybind11 should be preffered , but pybind11 version from homebrew might work as well.

Download and install the llvm+clang from the official site, e.g. using `wget` and add the location of llvm config to the $PATH:

```
wget --no-verbose https://github.com/llvm/llvm-project/releases/download/
↪llvmorg-11.0.0/clang+llvm-11.0.0-x86_64-apple-darwin.tar.xz
tar -xJf clang+llvm-11.0.0-x86_64-apple-darwin.tar.xz
export PATH=$PATH:$(pwd)/clang+llvm-11.0.0-x86_64-apple-darwin/bin
```

3.2 Building

To build `binder` run

```
cmake CMakeLists.txt -DCMAKE_INSTALL_PREFIX:PATH=/home/user/whereiwanttohaveit/
make
make install
```

To perform the build with a specific version of LLVM, the location of LLVM and CLANG directories should be set simultaneously via the location of their cmake configurations, i.e.

```
cmake CMakeLists.txt   -DLLVM_DIR=/usr/lib64/llvm8.0/lib/cmake/llvm -DClang_DIR=/usr/
↪lib64/llvm8.0/lib/cmake/clang
```

Alternatively, the location of the llvm-config script could be set.

```
cmake CMakeLists.txt   -DLLVMCONFIG=/usr/lib64/llvm7.0/bin/llvm-config
```

As an example with Ubuntu 18.04 and llvm-10:

```
cmake CMakeLists.txt   -DLLVM_DIR=/usr/lib/llvm-10 -DClang_DIR=/usr/lib/llvm-10
```

3.3 Using `binder` built with pre-installed LLVM

Under some circumstances (e.g. on system where the default compiller is not clang) `binder` might emit error messages like

```
/usr/lib/gcc/x86_64-redhat-linux/10/../../../../include/c++/10/bits/cxxabi_init_
↪exception.h:38:10: fatal error: 'stddef.h' file not found
#include <stddef.h>
         ^~~~~~~~~~
1 error generated.
```

and similar, see https://clang.llvm.org/docs/FAQ.html. To fix this issue, `binder` should be pointed to the location of the appropriate clang includes. This can be archived using the clang options that are passed to binder after `--` flag, e.g.

```
binder ...binder...options...  -- -x c++  ...other...options...   -iwithsysroot/where/
↪the/directory/with/includes/is/
```

See https://clang.llvm.org/docs/ClangCommandLineReference.html for details. If `binder` was build with some older versions of LLVM, one could also set the location of the headers with the `C_INCLUDE_PATH` and `CPLUS_INCLUDE_PATH` environment variables, e.g.

```
export CPLUS_INCLUDE_PATH=/where/the/directory/with/includes/is/
```

3.4 Building Statically (Linux only)

The first step in the static build is to build the `libclang` statically following the instructions from https://github.com/deech/libclang-static-build. For this quite a recent version of cmake is needed (3.13+). If the version of cmake form the used distribution is too old (e.g. as in the CentOS8) a precompilled package from the CMake site from https://cmake.org/ can be used instead.

The static build requires some other static libraries to be present in the system. For the CentOS8 install `libstdc++-static` and `ncurses-compat-libs` runnign as root:

```
sudo yum install libstdc++-static ncurses-compat-libs
```

Set the environment variable `LIBCLANG_STATIC_BUILD_DIR` to the path of *libclang-static-build*. Then build `binder` with the following procedure:

```
cmake CMakeLists.txt -DSTATIC=on -DLLVMCONFIG="${LIBCLANG_STATIC_BUILD_DIR}/build/_
↪deps/libclang_prebuilt-src/bin/llvm-config" -DLLVM_LIBRARY_DIR="${LIBCLANG_STATIC_
↪BUILD_DIR}/lib" -DCMAKE_INSTALL_PREFIX:PATH=/home/user/whereiwanttohaveit/

make

make install
```

CHAPTER 4

The Basics

In this section we describe basic Binder usage.

4.1 Principles

In order to create shared library that will provide bindings to C++ code we need to:

1. Gather data about what classes/functions are available and acquire in-depth information of class heritage, member functions and standalone functions type signatures.
2. Generate bindings code
3. Compile code into shared library

Binder is tool that aims to automate steps 1 and 2.

4.1.1 Preparing the input file

In order to feed Binder information about our underlying C++ code we need to create special C++ include file that in turn includes all header files from our project. For example: suppose that our C++ project contain the following header files: *frutes/apple.hpp*, *frutes/orange.hpp* and *vegetables/tomato.hpp*. In this case our special include file should contain something like this:

```
#include <frutes/apple.hpp>
#include <frutes/orange.hpp>
#include <vegetables/tomato.hpp>
```

Note: Make sure to specify complete-relative-to-project-root path to includes and use `#include <file>` and avoid using `#include "file"` form. That way Binder will be able to determine correct include paths for each include which is essential for generating correct include sets on step 2.

For small projects such file could be simply typed by-hands and for large project it might be more practical to use a scripts to do so.

4.1.2 Running the Binder

After the input file is ready the next step is to run Binder. Assuming that our include file containing all headers from the project is named as `all_includes.hpp` it could be done as:

```
binder --root-module my_project --prefix $HOME/my_project/bindings/source \
  --bind my_root_namespace \
  all_includes.hpp \
  -- -std=c++11 -I$HOME/my_project/include -I$HOME/extra/some_libs \
  -DMY_PROJECT_DEFINE -DNDEBUG
```

Note that we have to specify project-wide include path so Binder could find includes specifies in `all_includes.hpp` as well as path to any additional C++ include headers that is used in project.

Most big project will probably require fine tunning of bindings generation process. This can be done by creating Binder config file and specifying it when calling Binder as `--config my_project.config`. For detailed reference of config file options please see *Configuration*.

4.1.3 Compiling generated code

If all goes well and Binder finished its run without error the path specified by `--prefix` option should contain generated source code and auxiliary files:

`<root_module_name>.sources` list of generated source files

`<root_module_name>.cpp` main file for binding code

`<root_module_name>.modules` file containing the list of Python modules that were generated

CHAPTER 5

Configuration

Binder provides two ways to supply configuration options: command-line and config file.

5.1 Command line options

`--root-module` specify name of generated Python root module. This name is also used as prefix for various Binder output files. Typically the following files will be generated: `<root-module>.cpp`, `<root-module>.sources`, `<root-module>.modules`.

`--max-file-size` specify maximum file size in bytes exceeding which Binder will split generated sources into multiple files.

`--prefix` name/path prefix for generated files.

`--bind` list of namespaces that need to binded. Works in conjunction with similar config file directives.

`--skip` list of namespaces that should be skipped. Works in conjunction with similar config file directives.

`--config` specify config file to use.

`--single-file` if specified instruct Binder to put generated sources into single large file. This might be useful for small projects.

`--flat` if specified instruct Binder to write generate code files into single directory. Generated files will be named as `<root-module>.cpp`, `<root-module>_1.cpp`, `<root-module>_2.cpp`, `...` etc.

`--suppress-errors` if the generated bindings codes are correct but there are some fatal errors from clang and you want to get rid of them. This situation can happen when you would like to generate binding codes for a small part of a huge project and the you cannot include all the required header files with `-I` to the command.

`--annotate-includes` [debug] if specified Binder will comment each include with type name which trigger it inclusion.

`--trace` [debug] if specified instruct Binder to add extra debug output before binding each type. This might be useful when debugging generated code that produce seg-faults during python import.

5.2 Config file options

Config file is text file containing either comment-line (starts with #) or directive line started with either + or − signs followed by a directive's name and optional parameters. Some directives will accept only the + while others could be used with both prefixes.

5.2.1 Config file directives:

- `namespace`, specify if functions/classes/enums from particular namespace should be bound. Could be used with both + and − prefixes. This directive works recursively so for example if you specify `+namespace root` and later `-namespace root::a` then all objects in `root` will be bound with exception of `root::a` and its descendants.

```
-namespace boost
+namespace utility
```

- `class`, specify if particular class/struct should be bound. Purpose of this directive is to allow developer to cherry-pick particular class from otherwise binded/skipped namespaces and mark it for binding/skipping.

```
-class utility::pointer::ReferenceCount
-class std::__weak_ptr
```

- `function`, specify if particular function should be bound. This could be used for both template and normal function.

```
-function ObjexxFCL::FArray<std::string>::operator-=
-function core::id::swap
```

- `include`, directive to control C++ include directives. Force Binder to either skip adding particular include into generated source files (− prefix) or force Binder to always add some include files into each generated file. Normally Binder could automatically determine which C++ header files is needed in order to specify type/functions but in some cases it might be useful to be able to control this process. For example forcing some includes is particularly useful when you want to provide custom-binder-functions with either `+binder` or `+add_on_binder` directives.

```
-include <boost/format/internals.hpp>
+include <python/PyRosetta/binder/stl_binders.hpp>
```

- `include_for_class`, directive to control C++ include directives on a per-class basis. Force Binder to add particular include into generated source files when a given target class is present. This allows the inclusion of custom binding code, which may then be referenced with either `+binder` or `+add_on_binder` directives.

```
+include_for_class example::class <example/class_binding.hpp>
```

- `include_for_namespace`, directive to control C++ include directives on a per-namespace basis. Force Binder to add particular include into generated source files when generating bindings for specified namespace. This allows the inclusion of custom binding code, which may then be referenced with either `+binder`, `+add_on_binder`, `binder_for_namespace` or `add_on_binder_for_namespace` directives.

```
+include_for_namespace aaaa::bbbb <aaaa/bbbb/namespace_binding.hpp>
```

- `binder`, specify custom binding function for particular concrete or template class. In the example below all specializations of template std::vector will be handled by `binder::vector_binder` function. For template classes binder function should be a template function taking the same number of types as original type and

having the following type signature: pybind11 module, then std::string for each template argument provided. So for `std::vector` it will be:

```
template <typename T, class Allocator>
vector_binder(pybind11::module &m, std::string const &name, std::string const & /
↪*allocator name*/) {...}
```

- `+add_on_binder`, similar to `binder`: specify custom binding function for class/struct that will be called *after* Binder generated code bound it. This allow developer to create extra bindings for particular type (bind special Python methods, operators, etc.)

```
+binder std::vector my_binders::vector_binder
+binder std::map    my_binders::map_binder

+add_on_binder numeric::xyzVector rosetta_binders::xyzVector_add_on_binder
```

- `+binder_for_namespace`, similar to `binder`: specify custom binding function for namespace. Call to specified function will be generated _instead_ of generating bindings for namaspace.

```
+binder_for_namespace aaaa binder_for_namespace_aaaa
```

- `+add_on_binder_for_namespace`, similar to `add_on_binder`: specify custom binding function for namespace that will be called *before* Binder generated code bound it. This allow developer to create extra bindings for particular namespace.

```
+add_on_binder_for_namespace aaaa::bbbb binder_for_namespace_aaaa_bbbb
```

- `default_static_pointer_return_value_policy`, specify return value policy for static functions returning pointer to objects. Default is *pybind11::return_value_policy::automatic*.
- `default_static_lvalue_reference_return_value_policy`, specify return value policy for static functions returning l-value reference. Default is *pybind11::return_value_policy::automatic*.
- `default_static_rvalue_reference_return_value_policy`, specify return value policy for static functions returning r-value reference. Default is *pybind11::return_value_policy::automatic*.
- `default_member_pointer_return_value_policy`, specify return value policy for member functions returning pointer to objects. Default is *pybind11::return_value_policy::automatic*.
- `default_member_lvalue_reference_return_value_policy`, specify return value policy for member functions returning l-value reference. Default is *pybind11::return_value_policy::automatic*.
- `default_member_rvalue_reference_return_value_policy`, specify return value policy for member functions returning r-value reference. Default is *pybind11::return_value_policy::automatic*.
- `default_call_guard`, optionally specify a call guard applied to all function definitions. See pybind11 documentation. Default None.

```
+default_member_pointer_return_value_policy           pybind11::return_value_
↪policy::reference
+default_member_lvalue_reference_return_value_policy  pybind11::return_value_
↪policy::reference_internal
+default_member_rvalue_reference_return_value_policy  pybind11::return_value_
↪policy::move
+default_call_guard pybind11::gil_scoped_release
```

CHAPTER 6

Examples

This section is to talk about examples of how you would use binder.

Notes: - All python code should be run in the directory with the generated `.so` file.

6.1 Basic outline

These examples follow this general workflow:

1. Make a file that includes all `#include` lines
    - They must use <> not `""`
2. Generate bindings with binder
    - Set the namespace(s) to bind with the flag `--bind`
3. Compile the cpp files into objects separately
4. Link all generated objects into one file with the suffix `.so`
5. Try importing into python!

6.2 Simple struct

There are three examples of how to build bindings in the `example_struct` folder.

- `make_bindings_via_cmake.py`
- `make_bindings_via_bash.sh`
- `make_bindings_via_bash_and_stl.sh`

Their names are self explanatory, but I would highly recommend that for your own applications that you follow the `python` & `cmake` workflow.

Each script's final running lines also imports the `test_struct` module and prints a variable or two of it to prove that it is working.

This example/tutorial will walk you through the step-by-step of both `via_bash` scripts, because they will help you better understand what needs to be done to generate bindings. Upon understanding the more manual `bash` method, the `cmake` code should make much more sense.

The rest of "Simple struct" will also take you through generating pybind11 stl bindings (like making bindings for *std::vector* -> python list) and how to use binder's bindings for *std::vector* to access *std::vector* objects without converting them to python lists. This allows us to benefit from the speed of C++!

6.2.1 Building bindings basics

Using the g++/bash example we will go through how to generate bindings for this simple struct.

```
#include <string>
#include <vector>

namespace testers {
struct test_my_struct {
    int an_int;
    std::string a_string;
    std::vector<int> a_vector;
    float a_float;

    test_my_struct() {
            an_int = 27;
            a_string = "TEST_STRING";
            a_vector = std::vector<int>{1,2,3,4,5};
            a_float = 88.88;
    }

    void
    increment_int() {
            ++an_int;
    }

    void
    add_float() {
            a_float += 22.22;
    }

    void
    append_vec() {
            a_vector.push_back(a_vector.back()+1);
    }

};
}
```

1. First we have to generate a file that combines all of the `#includes` our project.
    - Remember, all `#includes` must use <>

```
grep -rh "#include" include/* | sort -u > all_bash_includes.hpp
```

2. Next we have to generate the bindings via `binder`

```
$PWD/../../build/llvm-4.0.0/build_4.0.0*/bin/binder \
  --root-module test_struct \
  --prefix $PWD/bash_bindings/ \
  --bind testers \
  all_bash_includes.hpp \
  -- -std=c++11 -I$PWD/include \
  -DNDEBUG
```

A skeleton of this would be:

```
$PWD/../../build/llvm-4.0.0/build_4.0.0*/bin/binder \
  --root-module ${my_python_module_name} \
  --prefix ${where_i_want_to_build_this} \
  --bind ${my_namespaces_to_build} \
  ${my_all_includes_file} \
  -- -std=c++11 -I${any_directores_to_include_for_compiler} \
  -DNDEBUG
```

3. Now that we have build bindings, we have to compile our bindings into `object files`.

First go into the directory where we build the bindings (set by `--prefix`) and then run the command:

```
pybase=`which python3`
g++ \
  -O3 \
  -I${pybase::-12}/include/python3.6m -I$PWD/../../../build/pybind11/include -I$PWD/..
↪/include \
  -I$PWD/../../../source -shared  \
  -std=c++11  -c test_struct.cpp  \
  -o test_struct.o -fPIC
```

NOTE ^^ Your python directory may be different slightly, you can find out yours using the shell command:

```
python -c "from distutils.sysconfig import get_python_inc; print(get_python_inc())"
```

Again, a skeleton of this command would be:

```
pybase=`which python3`
g++ \
  -O3 \
  -I${my_python_include_directory} -I${pybind11_include_directory} -I${my_project_
↪directory} \
  -I${binder_source_directory} -shared  \
  -std=c++11  -c ${bindings_code_to_build_object_file_from}  \
  -o ${output_object_file_name} -fPIC
```

4. Do this again for the other generated `.cpp` file
    - **All `.cpp` files to compile are located in the `.sources` file.**
5. Link together all of the compiled `object files`

```
g++ -o test_struct.so -shared test_struct/test_struct.o test_struct.o
```

6. Try running via python

```
python3 -c "import sys; sys.path.append('.'); import test_struct; f = test_struct.
↪testers.test_my_struct(); print(f.an_int)"
```

This should yeild: `27`

6.2.2 Binding STL via pybind11

You may notice how ever that this will still fail:

```
python3 -c "import sys; sys.path.append('.'); import test_struct; f = test_struct.
↪testers.test_my_struct(); print(f.a_float); f.add_float(); print(f.a_float);␣
↪print(f.a_vector)"
```

This fails because python does not understand how to interact with the std library classes like `std::vector`. You can get around this by remaking your bindings with this config file. **However**, you must note that when you are returning vectors into your python environment, or pushing lists to the c++ side, there is a performance penalty when pybind11 converts from `python list[]` -> `std::vector`, and vice-versa. This can be a problem when dealing with larger lists/vectors.

If performance is critical, it is advised that most work is done via c++, and you just use python as the "glue". For example, the following command does not fail, because python never has to "see" the std::vector and all of the work is done in the C++ layer.

```
python3 -c "import sys; sys.path.append('.'); import test_struct; f = test_struct.
↪testers.test_my_struct(); print(f.a_float); f.add_float(); print(f.a_float); f.
↪append_vec()"
```

But before you discount this approach completely, give it a try! It may still yeild performance improvements!

You can do this by adding a config file, and altering your binder compile command to read the config command like this:

my_config_file.cfg

```
+include <pybind11/stl.h>
```

New binder compile command

```
pybase=`which python3`
$PWD/../../build/llvm-4.0.0/build_4.0.0*/bin/binder \
  --root-module test_struct \
  --prefix $PWD/bash_bindings/ \
  --bind testers --config my_config_file.cfg \
  all_bash_includes.hpp \
  -- -std=c++11 -I$PWD/include -I$PWD/../../../build/pybind11/include -I${pybase::-12}
↪/include/python3.6m \
  -DNDEBUG
```

As an example of how the pybind11 bindings work, try running the command:

```
python3 -c "import sys; sys.path.append('.'); import test_struct; f = test_struct.
↪testers.test_my_struct(); print(f.a_float); f.add_float(); print(f.a_float);␣
↪print(f.a_vector)"
```

This will now run and print `[1, 2, 3, 4, 5]` at the end!

6.2.3 Binding STL via Binder

Binder allows us to add another layer so that we can interact directly with *std::vector*' for improved performance. This is sort of a hybrid between the above pybind11 implementation, and full on c++ code. There are a few things that have

to be changed though, before this will work.

changes to allow for binder bindings

1. We must add a function that returns the `std::vector<>` type of interest.
    - **std::vector bindings will be optimized out unless we add this function**

```
std::vector<int>
get_a_vector() {
        return a_vector;
}
```

2. We must make a config file that tells binder to build the vector bindings
    - you can also move the `--bind` commandline flags here by using the format `+namespace {what to bind}`.

```
+include <stl_binders.hpp>
+namespace testers
+binder std::vector binder::vector_binder
```

Now if we run the following command

```
python3 -c "import sys; sys.path.append('.'); import test_struct; f = test_struct.
↪testers.test_my_struct(); print(f.a_float); f.add_float(); print(f.a_float);␣
↪print(f.a_vector)"
```

this will print `vector_int[1, 2, 3, 4, 5]` at the end!, you can see, that unlike how pybind11 returns a python list, we have a statically typed list that can only take ints (much like c++).

in case you were curious, if you try to append a float to this list by using a command like `f.a_vector.append(22.22)`. You will see an error that looks similar to this:

```
Traceback (most recent call last):
  File "<string>", line 1, in <module>
TypeError: append(): incompatible function arguments. The following argument types␣
↪are supported:
    1. (self: test_struct.std.vector_int, x: int) -> None

Invoked with: vector_int[1, 2, 3, 4, 5], 22.22
```

CHAPTER 7

Debugging and troubleshooting

This section is dedicated to the description of problems that might appear while creating the python bindings with binder and the ways to avoid them.

Below are some helpful tips that might help to make the bindings.

7.1 Inconsistencies

Binder moves down the `all_includes_file` file sequentially, sometimes ending up with errors. This is almost always caused by the `all_includes_file` being different between runs. The order should not be important, but nail it down to at least be consistent, and then move on to the next step.

7.2 Build failures

Even when the bindings were generated successfully, there might be compilation errors when building the modules from the generated sources. Quite often the errors are caused by the implementation of the C++ standard library, when the headers of the standard library include each other, or include implementation-specific headers. Many cases like that are already handled in the functions from the `source/types.cpp` file, using the knowledge of the existing STL implementations. However some cases might still be missing, e.g. for the newest or not wide-spread versions of STL. An example of debugging for these cases is described below.

On systems with GNU STL, the compilation errors for the cases not handled by the `source/types.cpp`, would manifest itself with an abundance of long and cryptic messages

For instance, the compilation could fail with the following error messages:

```
FAILED: CMakeFiles/statvec.dir/std/complex.o
In file included from std/complex.cpp:1:0:
/usr/include/c++/7/bits/stl_construct.h: In function 'void std::_Destroy(_
↪ForwardIterator, _ForwardIterator)':
**long and cryptic error message**
```

The ways to handle this error:

1. Rebuild bindings adding the flag `--annotate-includes` which will provide much more information on the binded classes.
2. Since the includes from the `bits` directory should not appear in the generated code, one can grep for `bits` in the generated codes, i.e. `grep -r "bits" cmake_bindings/*` could yield:

```
cmake_bindings/std/complex.cpp:#include <bits/stl_construct.h> // std::_Construct
cmake_bindings/std/complex.cpp:#include <bits/stl_construct.h> // std::_Destroy
cmake_bindings/std/complex.cpp:#include <bits/stl_construct.h> // std::_Destroy_
↪aux
cmake_bindings/std/complex.cpp:#include <bits/stl_construct.h> // std::_Destroy_
↪aux<true>::__destroy
cmake_bindings/std/complex.cpp:#include <bits/stl_construct.h> // std::_Destroy_n_
↪aux
cmake_bindings/std/complex.cpp:#include <bits/stl_uninitialized.h> // std::__
↪uninitialized_copy
cmake_bindings/std/complex.cpp:#include <bits/stl_uninitialized.h> // std::__
↪uninitialized_copy<false>::__uninit_copy
cmake_bindings/std/complex.cpp:#include <bits/stl_uninitialized.h> // std::__
↪uninitialized_copy<true>::__uninit_copy
cmake_bindings/std/complex.cpp:#include <bits/stl_uninitialized.h> // std::__
↪uninitialized_copy_a
cmake_bindings/std/complex.cpp:#include <bits/stl_uninitialized.h> // std::__
↪uninitialized_default_1
cmake_bindings/std/complex.cpp:#include <bits/stl_uninitialized.h> // std::__
↪uninitialized_default_n
cmake_bindings/std/complex.cpp:#include <bits/stl_uninitialized.h> // std::__
↪uninitialized_default_n_1
cmake_bindings/std/complex.cpp:#include <bits/stl_uninitialized.h> // std::__
↪uninitialized_default_n_1<false>::__uninit_default_n
cmake_bindings/std/complex.cpp:#include <bits/stl_uninitialized.h> // std::__
↪uninitialized_default_n_1<true>::__uninit_default_n
cmake_bindings/std/complex.cpp:#include <bits/stl_uninitialized.h> // std::__
↪uninitialized_default_n_a
cmake_bindings/std/complex.cpp:#include <bits/stl_uninitialized.h> // std::__
↪uninitialized_default_novalue_1
cmake_bindings/std/complex.cpp:#include <bits/stl_uninitialized.h> // std::__
↪uninitialized_default_novalue_n_1
cmake_bindings/std/complex.cpp:#include <bits/stl_uninitialized.h> // std::__
↪uninitialized_fill
cmake_bindings/std/complex.cpp:#include <bits/stl_uninitialized.h> // std::__
↪uninitialized_fill_n
cmake_bindings/std/complex.cpp:#include <bits/stl_uninitialized.h> // std::__
↪uninitialized_move_if_noexcept_a
cmake_bindings/std/complex.cpp:#include <bits/stl_uninitialized.h> //␣
↪std::uninitialized_copy
```

   The important information in the output is the `std::` types/functions without the leading underscores. Those are STL-implementation independent types/functions that should be defined elsewhere, not in the headers from the `bits` directory. In this particular example, the function of interest is `std::uninitialized_copy`.

   A quick search in the C++ documentation at https://en.cppreference.com or other resources tells that this function is defined in the <memory> header. Therefore, this information should be hardcoded into the binder.

3. The internal binder function that handles the STL library mappings is located in `source/types.cpp:add_relevant_include_for_decl`. Briefly, the function has a map with the STL headers and the types those contain. That should look similar to this:

```
{ "<algorithm>", {"std::move_backward", "std::iter_swap", "std::min"} },
{ "<exception>", {"std::nested_exception"} }
```

If there is a need to make a simple change, like in our case, the map for the `<memory>` can be added like this:

```
{ "<algorithm>", {"std::move_backward", "std::iter_swap", "std::min"} },
{ "<exception>", {"std::nested_exception"} },
{ "<memory>", {"std::uninitialized_copy"} },
```

In addition to that, to ensure a better portability, some of the implementation-specific headers are replaced in binder with the standard ones. The map that holds the replacements is located in the `source/types.cpp` file as well. It should look similar to this:

```
static vector< std::pair<string, string> > const include_map = {
 make_pair("<bits/ios_base.h>",      "<ios>"),
 make_pair("<bits/istream.tcc>",     "<istream>"),
 make_pair("<bits/ostream.tcc>",     "<ostream>"),
 make_pair("<bits/postypes.h>",      "<ios>"),
```

4. After the changes are done, the binder executable should be recompilled and re-used to create the desired bindings. In some cases, many iterations of the described procedure will be needed till all the STL types/functions will be mapped to the correct includes.

   If this fixes your problem please let us know, or make a pull request!

CHAPTER 8

Testing

This section describes the testing suite for binder. The testing suite has two implementations, both using the same set of tests located in the `test` subdirectory. The first implementation is located inside the script `build-and-run-tests.py` and is designed to be used for the builds inside the LLVM source tree. This implementation is briefly described above. The second implementation uses `cmake/ctest` and is used in the CI with the external LLVM installation. To configure this testing suite, use

```
cmake ...  -DBINDER_ENABLE_TEST=ON ...
```

Multiple python versions can be and should be used in parallel. The versions are set as a comma-separated list passed by the `BINDER_TEST_PYTHON_VERSIONS` option. The default list is `0,2,3`. The Python version "0" corresponds to a "plain diff" of the output vs. reference. The versions of Python2/Python3 will be the first versions found by cmake, see https://cmake.org/cmake/help/latest/module/FindPython.html and https://cmake.org/cmake/help/latest/module/FindPython3.html for your cmake version. To use specific python versions one can use the following

```
cmake .... -DBINDER_TEST_PYTHON_VERSIONS=0,2.7.15,3.8.0,3.7.0 ...
```

The generated codes will be compiled and loaded using the corresponding interpreter. With an option `-DBINDER_MOCK_TEST=ON` one can mock the code generation by binder. In this case the reference codes will be used in the python tests.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search