SBuildr

Release 0.6.2

Jan 11, 2020

Contents

A stupid, simple python-based meta-build system for C++ projects.

CHAPTER 1

Installation

1.1 Prerequisites

1. RBuild
   - Install Cargo
   - Run `cargo install rbuild`

1.2 Installing from PyPI

```
pip install sbuildr
```

1.3 Installing from Source

1. Clone the SBuildr source repository.
2. Install locally with `python setup.py install`

CHAPTER 2

A Small Example

For this example, we will assume the following directory structure:

```
minimal_project
├── build.py
├── include
│   └── math.hpp
├── src
│   ├── factorial.cpp
│   ├── factorial.hpp
│   ├── fibonacci.cpp
│   ├── fibonacci.hpp
│   └── utils.hpp
└── tests
    └── test.cpp
```

The corresponding `build.py` file might look like this:

```
#!/usr/bin/env python
import sbuildr
import os

project = sbuildr.Project()

# Build a library using two source files. Note that headers do not have to be␣
↪specified manually.
# Full file paths are only required in cases where a partial path would be ambiguous.
libmath = project.library("math", sources=["factorial.cpp", "fibonacci.cpp"], libs=[
↪"stdc++"])

# Specify that math.hpp is part of the public API for this library.
project.interfaces(["math.hpp"])

# Specify a test for the project using the test.cpp source file. The resulting␣
↪executable will
```

(continues on next page)

(continued from previous page)

```
# be linked against the library created above.
test = project.test("test", sources=["test.cpp"], libs=["stdc++", libmath])

# Enable this script to be used interactively on the command-line
sbuildr.cli(project)
```

The call to the `cli()` function allows us to use the script to build interactively in a shell. For example, to run all tests registered for this project, you can run: `./build.py test`. This will configure the project, build all dependencies, and finally run tests.

To view all available commands, you can run `./build.py --help`

CHAPTER 3

API Documentation

For more information, see the API Documentation

CHAPTER 4

Known Limitations

- SBuildr's header scanning functionality does not take into account preprocessor `#ifdefs`. This means that an `#include` in a `false` branch will still be used as a dependency during builds. Header scanning will also not work for paths containing escaped characters.

4.1 Project

The project is the primary SBuildr's primary interface. It keeps track of all source files, and project targets.

class sbuildr.**Project**(*root: str = None*, *dirs: Set[str] = {}*, *build_dir: str = None*)

PROJECT_API_VERSION = 1

Represents a project. Projects include two default profiles with the following configuration: `release`: `BuildFlags().O(3).std(17).march("native").fpic()` `debug`: `BuildFlags().O(0).std(17).debug().fpic().define("S_DEBUG")`, attaches file suffix "_debug" These can be overridden using the `profile()` function.

Parameters

- **root** – The path to the root directory for this project. All directories and files within the root directory are considered during searches for files. If no root directory is provided, defaults to the containing directory of the script calling this constructor.
- **dirs** – Additional directories outside the root directory that are part of the project. These directories and all contents will be considered during searches for files.
- **build_dir** – The build directory to use. If no build directory is provided, a directory named 'build' is created in the root directory.

build(*targets: List[sbuildr.project.target.ProjectTarget] = None*, *profile_names: List[str] = None*) → float

Builds the specified targets for this project. Configuration should be run prior to calling this function.

Parameters

- **targets** – The targets to build. Defaults to all targets.
- **profile_names** – The profiles for which to build the targets. Defaults to all profiles.

Returns Time elapsed during the build.

clean(*profile_names: List[str] = None*, *nuke: bool = False*, *dry_run: bool = True*)
Removes build directories and project artifacts.

Parameters

- **profile_names** – The profiles for which to remove build directories. Defaults to all profiles.
- **nuke** – Whether to remove all build directories associated with the project, including profile build directories.
- **dry_run** – Whether this is a dry-run, in which case SBuildr will only display which directories would be removed rather than removing them. Defaults to True.

configure(*targets: List[sbuildr.project.target.ProjectTarget] = None*, *profile_names: List[str] = None*, *BackendType: type = <class 'sbuildr.backends.rbuild.RBuildBackend'>*) → None
Configure does 3 things: 1. Finds dependencies for the specified targets. This involves potentially fetching and building dependencies if they do not exist in the cache. 2. Configures the project's build graph after discovering libraries for targets. Before calling configure(), a target's libs/lib_dirs lists are not guaranteed to be complete. 3. Configure the project for build using the specified backend type. This includes generating any build configuration files required by this project's backend.

This function must be called prior to building.

Parameters

- **targets** – The targets for which to configure the project. Defaults to all targets.
- **profile_names** – The names of profiles for which to configure the project. Defaults to all profiles.
- **BackendType** – The type of backend to use. Since SBuildr is a meta-build system, it can support multiple backends to perform builds. For example, RBuild (i.e. `sbuildr.backends.RBuildBackend`) can be used for fast incremental builds. Note that this should be a type rather than an instance of a backend.

executable(*name: str*, *sources: List[str]*, *flags: sbuildr.tools.flags.BuildFlags = <sbuildr.tools.flags.BuildFlags object>*, *libs: List[Union[sbuildr.dependencies.dependency.DependencyLibrary, sbuildr.project.target.ProjectTarget, sbuildr.graph.node.Library]] = []*, *compiler: sbuildr.tools.compiler.Compiler = <sbuildr.tools.compiler.Compiler object>*, *include_dirs: List[str] = []*, *linker: sbuildr.tools.linker.Linker = <sbuildr.tools.linker.Linker object>*, *depends: List[sbuildr.dependencies.dependency.Dependency] = []*, *internal=False*) → sbuildr.project.target.ProjectTarget
Adds an executable target to all profiles within this project.

Parameters

- **name** – The name of the target. This should NOT include platform-dependent extensions.
- **sources** – A list of names or paths of source files to include in this target.
- **flags** – Compiler and linker flags. See sbuildr.BuildFlags for details.
- **libs** – A list containing either `ProjectTarget` s, `DependencyLibrary` s or `Library` s.
- **compiler** – The compiler to use for this target. Defaults to clang.

- **include_dirs** – A list of paths for preprocessor include directories. These directories take precedence over automatically deduced include directories.
- **linker** – The linker to use for this target. Defaults to clang.
- **depends** – Any additional dependencies not already captured in libs. This may include header only packages for example.
- **internal** – Whether this target is internal to the project, in which case it will not be installed.

Returns `sbuildr.project.target.ProjectTarget`

find(*path*) → str

Attemps to locate a path in the project. If no paths were found, or multiple ambiguous paths were found, raises an exception.

Parameters **path** – The path to find. This may be an absolute path, partial path, or file/directory name.

Returns An absolute path to the matching file or directory.

install(*targets: List[sbuildr.project.target.ProjectTarget] = None*, *profile_names: List[str] = None*, *headers: List[str] = None*, *header_install_path: str = '/usr/local/include'*, *library_install_path: str = '/usr/local/lib'*, *executable_install_path: str = '/usr/local/bin'*, *dry_run: bool = True*)

Install the specified targets for the specified profiles.

Parameters

- **targets** – The targets to install. Defaults to all non-internal project targets.
- **profile_names** – The profiles for which to install. Defaults to the "release" profile.
- **headers** – The headers to install. Defaults to all headers that are part of the interface as per *interfaces()* .
- **header_install_path** – The path to which to install headers. This defaults to one of the default locations for the host OS.
- **library_install_path** – The path to which to install libraries. This defaults to one of the default locations for the host OS.
- **executable_install_path** – The path to which to install executables. This defaults to one of the default locations for the host OS.
- **dry_run** – Whether to perform a dry-run only, with no file copying. Defaults to True.

install_profile() → str

Returns the name of the profile for which this project will install targets.

install_targets() → List[sbuildr.project.target.ProjectTarget]

Returns all targets that this project can install.

Returns A list of targets.

interfaces(*headers: List[str], depends: List[sbuildr.dependencies.dependency.Dependency] = []*) → List[str]

Specifies headers that are part of this project's public interface. When running the `install` command on the CLI, the headers specified via this function will be copied to installation directories.

Parameters **headers** – A list of paths to a public headers.

Returns The absolute paths of the discovered headers.

library(*name: str, sources: List[str], flags: sbuildr.tools.flags.BuildFlags = <sbuildr.tools.flags.BuildFlags object>, libs: List[Union[sbuildr.dependencies.dependency.DependencyLibrary, sbuildr.project.target.ProjectTarget, sbuildr.graph.node.Library]] = [], compiler: sbuildr.tools.compiler.Compiler = <sbuildr.tools.compiler.Compiler object>, include_dirs: List[str] = [], linker: sbuildr.tools.linker.Linker = <sbuildr.tools.linker.Linker object>, depends: List[sbuildr.dependencies.dependency.Dependency] = [], internal=False*) → sbuildr.project.target.ProjectTarget

Adds a library target to all profiles within this project.

Parameters

- **name** – The name of the target. This should NOT include platform-dependent extensions.
- **sources** – A list of names or paths of source files to include in this target.
- **flags** – Compiler and linker flags. See sbuildr.BuildFlags for details.
- **libs** – A list containing either `ProjectTarget` s, `DependencyLibrary` s or `Library` s.
- **compiler** – The compiler to use for this target. Defaults to clang.
- **include_dirs** – A list of paths for preprocessor include directories. These directories take precedence over automatically deduced include directories.
- **linker** – The linker to use for this target. Defaults to clang.
- **depends** – Any additional dependencies not already captured in libs. This may include header only packages for example.
- **internal** – Whether this target is internal to the project, in which case it will not be installed.

Returns `sbuildr.project.target.ProjectTarget`

run(*targets: List[sbuildr.project.target.ProjectTarget], profile_names: List[str] = []*) → None

Runs targets from this project.

Parameters

- **targets** – The targets to run.
- **profile_names** – The profiles for which to run the targets.

run_tests(*targets: List[sbuildr.project.target.ProjectTarget] = None, profile_names: List[str] = None*)

Run tests from this project. Runs all tests from the project for all profiles by default.

Parameters

- **targets** – The test targets to run. Raises an exception if the target is not a test target.
- **profile_names** – The profiles for which to run the tests. Defaults to all profiles.

test(*name: str, sources: List[str], flags: sbuildr.tools.flags.BuildFlags = <sbuildr.tools.flags.BuildFlags object>, libs: List[Union[sbuildr.dependencies.dependency.DependencyLibrary, sbuildr.project.target.ProjectTarget, sbuildr.graph.node.Library]] = [], compiler: sbuildr.tools.compiler.Compiler = <sbuildr.tools.compiler.Compiler object>, include_dirs: List[str] = [], linker: sbuildr.tools.linker.Linker = <sbuildr.tools.linker.Linker object>, depends: List[sbuildr.dependencies.dependency.Dependency] = []*) → sbuildr.project.target.ProjectTarget

Adds an executable target to all profiles within this project. Test targets can be automatically built and run by using the `test` command on the CLI.

Parameters

- **name** – The name of the target. This should NOT include platform-dependent extensions.
- **sources** – A list of names or paths of source files to include in this target.
- **flags** – Compiler and linker flags. See sbuildr.BuildFlags for details.
- **libs** – A list containing either `ProjectTarget` s, `DependencyLibrary` s or `Library` s.
- **compiler** – The compiler to use for this target. Defaults to clang.
- **include_dirs** – A list of paths for preprocessor include directories. These directories take precedence over automatically deduced include directories.
- **linker** – The linker to use for this target. Defaults to clang.
- **depends** – Any additional dependencies not already captured in libs. This may include header only packages for example.

Returns `sbuildr.project.target.ProjectTarget`

test_targets() → List[sbuildr.project.target.ProjectTarget]
Returns all targets in this project that are tests.

Returns A list of targets.

uninstall(*targets: List[sbuildr.project.target.ProjectTarget] = None*, *profile_names: List[str] = None*, *headers: List[str] = None*, *header_install_path: str = '/usr/local/include'*, *library_install_path: str = '/usr/local/lib'*, *executable_install_path: str = '/usr/local/bin'*, *dry_run: bool = True*)
Uninstall the specified targets for the specified profiles.

Parameters

- **targets** – The targets to uninstall. Defaults to all non-internal project targets.
- **profile_names** – The profiles for which to uninstall. Defaults to the "release" profile.
- **headers** – The headers to uninstall. Defaults to all headers that are part of the interface as per *interfaces()* .
- **header_install_path** – The path from which to uninstall headers. This defaults to one of the default locations for the host OS.
- **library_install_path** – The path from which to uninstall libraries. This defaults to one of the default locations for the host OS.
- **executable_install_path** – The path from which to uninstall executables. This defaults to one of the default locations for the host OS.
- **dry_run** – Whether to perform a dry-run only, with no file copying. Defaults to True.

4.2 Profile

class sbuildr.**Profile**(*flags: sbuildr.tools.flags.BuildFlags*, *build_dir: str*, *suffix: str*)
Represents a profile in a project. A profile is essentially a set of options applied to targets in the project. For example, a profile can be used to specify that all targets should be built with debug information, and that they should have a "_debug" suffix.

Parameters

- **flags** – The flags to use for this profile. These will be applied to all targets for this profile. Per-target flags always take precedence.

- **build_dir** – An absolute path to the build directory to use.
- **suffix** – A file suffix to attach to all artifacts generated for this profile.

4.3 Build Flags

class sbuildr.**BuildFlags**

Abstract description of compiler and linker flags. These are interpreted by SBuildr's compiler and linker interfaces and converted to concrete command-line flags.

It is possible to add two BuildFlags, in which case the right-hand side takes precedence when flags are set for both instances. For example, `BuildFlags.O(3).fpic() + BuildFlags.O(0)` would result in a value equivalent to: `BuildFlags.O(0).fpic()`

O(*level: Union[int, str]*) → sbuildr.tools.flags.BuildFlags

Sets the optimization level.

Parameters **level** – An integer or string indicating the optimization level. For example, to disable optimization, this would be set to `0` or `"0"`.

Returns self

debug(*use=True*) → sbuildr.tools.flags.BuildFlags

Enables or disables generation of debug information.

Parameters **use** – Whether to generate debug information.

Returns self

define(*macro*) → sbuildr.tools.flags.BuildFlags

Defines the specified macro during compilation. This can be useful to enable/disable code paths using #ifdefs.

Parameters **macro** – The macro to define.

Returns self

fpic(*use=True*) → sbuildr.tools.flags.BuildFlags

Enables or disables generation of position independent code.

Parameters **use** – Whether to generate position independent code.

Returns self

march(*type: str*) → sbuildr.tools.flags.BuildFlags

Sets the microarchitecture.

Parameters **type** – A string describing the CPU microarchitecture.

Returns self

raw(*opts: List[str]*) → sbuildr.tools.flags.BuildFlags

Allows for providing raw options.

Parameters **opts** – A list of options, as strings. These are passed on to the compiler and linker without modification.

Returns self

std(*year: Union[int, str]*) → sbuildr.tools.flags.BuildFlags

Sets the C++ standard.

Parameters **year** – An integer or string indicating the last two digits of the year of the corresponding C++ standard. For example, to use C++11, this would be set to `11` or `"11"`.

Returns self

Index