Waf user's guide

Carlos Rafael Giani

Contents

1 Getting started

In order to understand the basics of waf, let's create a simple hello world project. It has only one source file, main.cpp:

```
int main()
{
    std::cout << "Hello world";
    return 0;
}
```

Listing 1: the main.cpp hello world code

To build this, we need a waf build script, called a *wscript*. Here is the wscript for the hello world program:

```
srcdir = '.'
blddir = 'build'

def configure(conf):
    conf.check_tool('g++')

def build(bld):
    obj = bld.create_obj('cpp', 'program')
    obj.source = 'main.cpp'
    obj.target = 'hello'
```

Listing 2: The hello world wscript

Let's go line by line through it:

srcdir: the root directory for all source files. The project's entire source code is in this directory (or in a subdirectory).

blddir: where to put the build results. Waf separates build output from source.

configure(conf): this function gets invoked when "waf configure" is called. It handles the necessary configuration steps, such as autodetection. In this script, configure calls *conf.check_tool('g++')*, which autodetects the programs *cpp*, *g++*, *ar*, and *ranlib* (some platforms call the C++ compiler cpp, some g++). This step is necessary for waf to be able to build C++ programs. In Windows, the *msvc* tool would accomplish the same, but using Visual C, and not gcc.

build(bld): this function is called by waf once the actual building starts. In build(), *build objects* are created; here, a build object called "obj" is created via calling *bld.create_obj('cpp', 'program')*. create_obj() takes two arguments: the tool to use for object creation, and a tool-specific type specifier. Tool "cpp" creates an object for building C++ projects and knows the types "shlib", "staticlib", "program", and a few others, but usually these three are used. "shlib" tells "cpp" to build a shared library (.so), "staticlib" means a static library (.a), and "program" builds an executable.

obj.source = 'main.cpp': this line tells the object which source files to use for building. In this case, only main.cpp is used; projects with several files can specify these as one string with the filenames being whitespace-separated.

obj.target = 'hello': the name of the target to build. The resulting executable will be called "hello".

This script is now able to autodetect the g++ compiler (and additional programs like ranlib) and build the hello world program. First *waf configure* is called, which causes waf to call the configure() method. Then, by calling *waf build* (or just *waf*) build() is called, and the program is built. For cleaning the build result without deleting the autoconfiguration results, type *waf clean*. This deletes the build results only; the configuration results remain. For fully removing all autogenerated files, *waf distclean* ist used.

2 The waf building process

Building with waf always follows these steps:

1. Init procedures are called
2. Custom command-line options are added
3. Configuration is done (by calling "waf configure")
4. Actual building takes place (command-line call "waf build", or just "waf")
5. Shutdown procedures are called

As mentioned before, waf scripts are called *wscripts*. They are in fact python modules, and treated as such. A wscript variant is *wscript_build*, which is a wscript with building code only.

2.1 Initialization

init() is called every time waf is run, except when running "waf dist" or "waf distclean". Currently, it has no designated use, but is available for wscripts. However, usually it is omitted in wscripts.

2.2 Custom command-line options

It is possible to add custom command-line options to waf. For example, the hello world wscript with an added option "–foo-path" looks like this:

```
srcdir = '.'
blddir = 'build'

import Params  # necessary for custom options

def set_options(opt):
        opt.add_option('--foo-path', type='string',
                                        help='some path', default='', dest='foopath')

def configure(conf):
    conf.check_tool('g++')
        if Params.g_options.foopath:
                print ''Using foo path %s'' % Params.g_options.foopath

def build(bld):
    obj = bld.create_obj('cpp', 'program')
    obj.source = 'main.cpp'
    obj.target = 'hello'
```

Listing 3: The hello world wscript with the custom option added

This script behaves like the original one, except that the "waf configure –foo-path=/some/path" call now causes waf to print "Using foo path /some/path".
The *Params.g_options* object is in fact a Python *OptionParser*, which is explained in detail in the Python documentation.

2.2.1 Tool options

Some tools add their own commandline options. These tools need to be called in the set_options() function, via the tool_options() functions:

```
opt.tool_options('g++')
```

Listing 4: Adding command-line options of the tool 'g++'

2.2.2 Sub options

In complex projects, it is common that the source is partitioned in subdirectories. If some of the wscripts in these directories contain set_options() functions too, then the *opt.sub_options()* function can be used for recursing into these subdirectories.

```
opt.sub_options('main graphics')
```

Listing 5: Calling the set_options() functions of the wscripts in the directories 'main' and 'graphics'

2.3 Configuration

Waf supplies a "conf" object to the configure() function in a wscript, which contains a Python "env" dictionary. This dictionary contains all *uselib variables*, among others. Additionally, the conf object has functions for creating *enumerators* and *configurators*, and the "check_tool" function for detecting and initializing tools.

2.3.1 Tools

Waf tools are utility code for common tasks. Often, they handle complex autodetection and/or add tools for building. Internally, they reside in wafadmin/Tools/. Some tools are added via the conf.check_tool() function, others are for internal use.

Of crucial importance is the check_tool() function takes two parameters; the first one is the tool name, the second is optional and specifies a path where to look for the tool. At least one tool must be ran in order to be able to build; for example, for being able to build C++ projects using g++, check_tool('g++') must be called. Same with Visual C (check_tool('msvc')) and so on.

Without this call, the first parameter of Build.create_obj() is meaningless (Waf does not know 'cpp', 'cc', etc. unless one C++ tool like 'g++' or 'msvc' is added). To be more specific, g++, gcc, msvc handle the compiler-specific details and add abstract 'cpp' and 'cc' tools to Waf, which are used for building. (This is why internal tools like 'cpp' and 'cc' must not be added manually via check_tool by the wscript.)

The second parameter is a list of paths where to look for the tools mentioned in the first parameter. This is useful for third-party tools not included in waf itself. Usually, this happens when a Waf tool is modified in any way. For example, if one uses a custom "Foo.py" tool, which is located in the same directory as the wscript, use "check_tool('Foo', ['.'])". Omitting the second parameter causes waf to use the internal tools only.

A detailed list of the tools can be found in chapter 3.1.

2.3.2 Uselib variables

Configuration works with so-called *uselib variables*. The configure() functions in the wscripts define the variables, the subsequent build step uses them (this will be explained in more detail in the Build chapter).

uselib variable names all have the same structure: *vartype_varname*

vartype is the type of the uselib variable. For example, CPPPATH is the type for C++ include paths. *varname* is an unique name associated with the contents.

For example, "CPPPATH_FT2" is the uselib variable for the C++ include paths of "FT2".

Here is a list of variable types used in waf, along with a description of their usage and values:

- *LIB* : a library name. Used both for static and dynamic libraries. If a static and a dynamic library with the same name exist, the dynamic one will be used. For example, env['LIB_FT2'] = 'ft2' results in the ld linker flag " -lft2 ". This type also accepts several library names, which must be specified as a list. Example: ['X11' , 'Xxf86vm'] becomes " -lX11 -lXxf86vm ".
- *STATICLIB* : unlike LIB, this type works for static libraries only. If only a dynamic library with the specified name exists, it will not be linked.
- *LIBPATH* : path to a library. Usually used together with LIB. Can accept multiple paths as a list.
- *STATICLIBPATH* : path to a static library. Usually used together with STATICLIB. Can accept multiple paths as a list.
- *CPPPATH*: C/C++ include paths (the "-I" parameters in gcc). Accepts multiple paths as a list.
- *CXXDEFINES*: C/C++ preprocessor defines (the "-D" parameter in gcc). Accepts multiple defines as a list. This one should be used with care, since some platforms limit the amount of characters a command-line call can have. It it therefore usually wiser to use a configuration header with the defines in it. (It also cleans up the waf verbose output.)
- *CCFLAGS*: C compiler flags. The value is directly passed to the compiler. Note that this is compiler-specific, which can be a problem with multiplatform projects. Example: env['CCFLAGS_FT2'] = '-Wall -ansi -pedantic'.
- *CXXFLAGS*: C++ compiler flags. The value is directly passed to the compiler. Note that this is compiler-specific, which can be a problem with multiplatform projects. Example: env['CXXFLAGS_FT2'] = '-Wall -ansi -pedantic'.
- *LINKFLAGS*: Linker flags. The value is directly passed to the linker. Note that this is linker-specific, which can be a problem with multiplatform projects. Example: env['LINKFLAGS_FT2'] = '-fPIC'.

The configure() function in a wscript is free to define the uselib variable contents in any way. Manual setting is not common, however. Usually, *configurators* and *tools* are used. There are configurators for autodetecting C++ headers, libraries, pkg-config packages, config-tools (like *sdl-config*), and others. Waf tools (not to be confused with the aforementioned config-tools) are general utility packages which can also include autodetection features; in fact, many tools use configurators for internal autodetection.

2.3.3 Global uselib variables

These uselib variables are valid everywhere; all build objects include these. They work exactly like normal uselib variables, but lack a name. For example, env['CXXFLAGS'] += ['-Wall'] causes waf to add the '-Wall' C++ compiler flag to *all* build objects. A common use of these special uselib variables is strict compiler behaviour. With gcc, this is achieved by using the flags '-ansi' and '-pedantic'. So, env['CXXFLAGS'] += ['-Wall -ansi -pedantic'] puts gcc in the most strict mode, penalizing almost all violations of the ISO C++ standard.

Note: As seen above, it is recommended to append (+=) and not assign (=) a new value. waf may have written something in the global uselib variables already, and this would be lost by assigning the new value. Also do not forget to put the string in square brackets.

2.3.4 Configurators

As mentioned before, these are used for automating the detection of various components like libraries, C++ headers, programs. They also fill the corresponding uselib variables with the autodetection results. The basic usage is the same with all configurators:

1. Create the configurator
2. Fill in the values necessary for autodetecting
3. call the configurator's run() function

Afterwards, the env dictionary contains the resulting uselib variables, if the autodetection was successful (for example, if the desired header was found). A failure can have two results, depending on the value of the configurator's *mandatory* variable: Print a warning and continue, or stop waf and print an error message.

```
headerconf = conf.create_header_configurator()
headerconf.name = 'GL/gl.h'
headerconf.path = ['/usr/X11R6/include','/usr/local/include']
headerconf.mandatory = 1
headerconf.message = 'This projects requires OpenGL.'
headerconf.run()
```

Listing 6: Example header configurator usage

Configurators have variables, which need to be set with correct values for accomplishing the task. Some variables are the same for all (or almost all) configurators, some are configurator-specific. The common variables are:

- *mandatory* : if this is 1, a failure in autodetection results in waf stopping and printing an error message defined in the *message* variable (see below). 0 means that a warning will be emitted, but waf is not stopped. Default is 0. 1 should be used for components absolutely necessary for this project, 0 for optional parts (for example, an audio player with optional support for mp3 decoding).
- *message*: the message to be printed if *mandatory* is set to 1 and the autodetection fails. This is useful for helping out the user, for example when autodetection of library FT2 fails, the message could contain some suggestions where to download this library and how to install it, or how to install it in popular Unix distributions.
- *define*: the define to be added to the configure header once autoconfiguration is completed.
- *uselib*: the unique name of the uselib variable to set. Configurator results will be stored in uselib variables with this name.

Here is a list of the configurators included in waf:

- *Header configurator*
  Searches for a particular C++ header in a list of paths, and puts its path in a CPPPATH uselib variable if found.

  This configurator works by test-compiling this code ("someheader.h" is the wanted header):

  ```
  // "header_code" contents are inserted here
  #include "someheader.h"
  int main()
  {
    // custom_code contents are inserted here
    return 0;
  }
  ```

header_code and custom_code are explained below.

Its specific members:

- *name* : the name of the C++ header, including the file extension ("shared_ptr.hpp" for example).
- *path* : the list of paths where to look for the header. An example would be ['/usr/X11R6/include', '/usr/local/include'].
- *nosystem* : if this is set to 1, the standard include paths are not appended, that is the header will not be searched in these standard paths. Default is 0.
- *header_code*: The code specified in this member will be inserted into the testcode *before* the searched header is included (see the code above). This is necessary because some headers expect other headers to be included before, other headers expect some preprocessor defines etc. For example, *jpeglib.h* expects *stdio.h* to be included before. Default value is ' '.
- *custom_code*: Custom code which will be inserted into the testcode's main() function. Default value is ' '.
- *libs*: Libraries to be linked to the testprogram. Necessary when testing for *boost.asio* for example; its header(s) require the *pthread* library to be linked, else a linker error occurs. Default value is ' '.
- *libpath*: Paths where to find the aforementioned libraries. Default value is ' '.

Note that this configurator is capable of autodeducing a *uselib* variable name; if *uselib* isn't set, it uses the contents of *name* in uppercase, with the symbols '.', ':', '/' replaced by an underscore '_'. So, in case of GL/gl.h *uselib* is set to 'GL_GL_H'.

Created by calling *conf.create_header_configurator()*.

- *Library configurator*
  Searches for a library in a list of paths. It looks for both static and shared libraries (DLLs in Windows). Its specific members:
  - *name* : the name of the library, **not** including the platform-specific pre- and postfix (e.g. 'foo' instead of 'libfoo.so').
  - *path* : the list of paths where to look for the library. An example would be ['/usr/X11R6/lib', '/usr/local/lib'].

  Created by calling *conf.create_library_configurator()*.

- *Configure tool configurator*
  Uses the values from configure tools (not to be confused with waf tools). Configure tools are package-specific helpers used for determining the flags necessary for using the package. For example, the SDL (http://www.libsdl.org) has "sdl-config". Calling "sdl-config –cflags" returns the C/C++ compiler flags necessary for using the SDL. This configurator can query such a tool and put its results into the uselib variables. Its specific members:
  - *binary* : the name of the configure tool to use. An example would be "sdl-config".

  Unlike other configurators, this one is *not* capable of auto-deducing a uselib variable name, so it must be set explicitely.

  Created by calling *conf.create_cfgtool_configurator()*.

- *Pkg-config configurator*
  This configurator makes use of the pkg-config system. pkg-config is a popular centralized database for querying compiler flags. It works similar to a configure tool; for example, "pkg-config –cflags alsa" prints the cflags necessary for using ALSA. The flags are stored in .pc files, usually located in /usr/lib/pkgconfig/. Its specific members:

- *name* Name of the .pc file. In the example above, it would be "alsa" (note that the .pc suffix must not be added).
- *version* Minimum version of the package. If the present package is older, the configurator fails. Default value is "" (= all versions are ok).
- *path* Path to the .pc file. Internally, this value is passed to the "PKG_CONFIG_PATH" environment variable when pkg-config is called. Default value is "" - in this case PKG_CONFIG_PATH is not set.
- *binary* Name and path to pkg-config. Default value is "" ("pkg-config" is used).
- *variables* You could also check for extra values in a pkg-config file. Use this value to define which values should be checked and defined. Several formats for this value are supported:
  - string with spaces to separate a list
  - list of values to check (define name will be upper(uselib"_"value_name))
  - a list of [value_name, override define_name]

  Default value is [].

Created by calling *conf.create_pkgconfig_configurator()*.

- *Test configurator*

- *OSX framework configurator*

2.3.5 Sub configurations

Like the sub options, wscripts in subdirectories may contain configure() functions as well. To call these, use *conf.sub_config()*. It works analogous to *opt.sub_options()*.

2.3.6 Configure header

Waf can autogenerate a C/C++ header file with preprocessor defines in it. This is the aforementioned alternative to the CXXDEFINES uselib variable type.

The *conf.write_config_header()* function writes all defines present in the environment (conf.env). New defines can be added by using *conf.add_define()*. This function expects a name for the define as first and its value as second parameter. If a define with the same name already exists, it is overwritten.

All configurators add a define, most can autogenerate a name for one. Its value is 1 if the configurator succeeded or 0 if it failed. These defines' names always start with "HAVE_". So, a library configurator looking for the library "GL" would add the define "HAVE_GL", with 1 as a value if the GL library was found.

It is also possible to use a custom define name. The *define* configurator variable provides this. If it is empty, waf autogenerates the define name (if possible), otherwise it uses the one specified in the *define* variable. See chapter 2.3.4 for details.

There are additional functions for handling defines:

1. *is_defined(define)* : This returns nonzero if the specified define exists, zero otherwise.
2. *get_define(define)* : Returns the value of the specified define, or zero if this define does not exist.

```
conf.write_config_header('config.h')
conf.add_define('DEBUG', 1)
```

Listing 7: Configure header example

The example above adds a define called "DEBUG", and sets its value to 1. The resulting header code:

```
/* configuration created by waf */
#ifndef _CONFIG_H_WAF
#define _CONFIG_H_WAF

#define DEBUG 1

#endif /* _CONFIG_H_WAF */
```

Listing 8: Configure header example

This resulting config.h header would be located in *blddir*/default/config.h (*blddir* is the directory where the build results are put).

2.4 Building

Building works by making use of *build objects*. In wscripts, a build() function always gets an object of type *Build* supplied as the function's only parameter (in the example in chapter 1, the object is called "bld"). This object can create build objects. The wscript has to call the create_obj function in Build to create the right build objects, and supply them with data. As an example this build() function is used:

```
def build(bld):
    obj = bld.create_obj('cpp', 'program')
    obj.source = 'example.cpp'
    obj.target = 'example'
    obj.uselib = 'ABC FT2'
```

Listing 9: The example build() function

bld is the object of type Build. The first parameter of create_obj specifies the *tool* to be used for building. Waf has several tools for building various kinds of projects (see chapter 3.1 for a list of all tools).
The second parameter is tool-specific. In this case, it defines which type to use. The cpp tool knows these types: program, staticlib, shlib, bundle, plugin (the last two being OSX specific).
After this call, an object is returned. Waf also registers this object in an internal list. When all objects are created and properly set with valid values, waf calls the tools, which create *tasks*. These tasks handle calling the compiler, copying the build results, deleting generated files when calling *./waf (dist)clean* etc.

build() functions in subdirectories can be called by using *bld.add_subdirs()*. This function accepts multiple whitespace-separated directories as one string, like 'src src2'.

The following lines set the build object data:

- *source* contains all source files to be processed. Common are the whitespace-separated and end-of-line separated representations. An example of the former would be 'a.cpp b.cpp', and one of the latter would be

  ```
  """
  a.cpp
  b.cpp
  """
  ```

  Often, one does not want to specify all source files, instead simply telling in which dir to look for them is desired. For this, *find_sources_in_dirs()* exist. So, writing *obj.find_sources_in_dirs(".")* instead of *obj.source = 'example.cpp'* above are equivalent. This is especially handy if the project has one directory containing many source files.

- *target* specifies the target name. The type of the target is directly related to the type of the build. In case of C++, the target is an executable if the type is "program", a shared library (DLL in windows) if "shlib", a static library if "staticlib", an OSX bundle if "bundle" and a OSX plugin if "plugin" (the latter two equal "shlib" in Unix and Windows). Note that the target name should not include platform-specific pre- and suffixes, since Waf attaches them automatically; a shlib target "foo" will result in a shared library "libfoo.so" in Unix, and "foo.dll" in Windows.

- *uselib* is the key to making use of autoconfiguration. Any uselib variable whose name is in this string will be attached to this build object. Multiple names are whitespace-separated. In the example above, the uselibs "ABC" and "FT2" are attached to the build object. In case uselibs do not exist, waf ignores this silently.
  Continuing with the example above, if a "CPPPATH_FT2" uselib variable exists, its contents are attached to the build object's c++ include path list. This is how autoconfiguration works; configure() creates uselib variables, build() adds their contents to the build objects.

The three lines above are not all members available in build objects. Here is the full list:

- *source, target, uselib*: already explained above.

- *name*: Build objects can (and should) be named. This is for internal references, and especially handy for the *uselib_local* variable.

- *uselib_local*: Similar to *uselib*, this defines the local dependencies of a build object. If program A depends on static library B, both being created in the same project, and library B is named 'b' (e.g. obj.name = 'b'), then program A should use "obj.uselib_local = 'b' ". This way waf makes sure B is built *before* A, and B gets linked to A.

  Note: uselib_local must be in the right order, e.g. if a build object uses the libraries A and B, and lib A uses parts of lib B, uselib_local must be 'A B'. This is necessary for ensuring proper linking.

- *includes*: a string of whitespace-separated paths where project specific C/C++ header files are kept. This is C/C++ specific and necessary for tracking header dependencies; if one of the headers stored in one of these paths is modified, waf rebuilds the source files using the modified file(s).

- *unit_test*: if this is set to 1, this build object classifies as a unit test. Default value is 0. Only useful if the result of the build object is an executable. See chapter 3.2, *unit tests* for more.

2.5 Shutdown

shutdown() is called every time waf is run, except when running "waf dist" or "waf distclean". It is useful for performing unit tests and/or copying files. It is optional, and can be omitted.

3 Additional waf features

3.1 Waf tools

Here is a list of tools included in waf:

- *bison*

- *cs*

- *docbook*

- *flex*
- *g++* Tool for building C++ projects using the GNU C++ compiler. Adds a build tool "cpp" capable of building various types of C++ projects. The type is specified in create_obj after 'cpp'. For example, a C++ project of the "program" type will be created like this: obj = bld.create_obj('cpp', 'program')
  The supported types:
  - *program* Executables (they get a ".exe" suffix in Windows).
  - *shlib* Shared libraries (called DLLs - dynamic link libraries - in Windows). Pre- and suffix vary between platforms; Unix and Unix derivatives use "lib" as pre- and ".so" as suffix, Windows uses no pre- and ".dll" as suffix.
  - *staticlib* Static libraries. These have the same naming convention in Unix as shared libraries, in Windows they end with ".lib" instead of ".dll".
  - *bundle* OSX-specific (equals shlib on other platforms).
  - *plugin* OSX-specific (equals shlib on other platforms).
- *gcc* Tool for building C projects using the GNU C compiler. Adds a build tool 'cc' capable of building the same types g++ supports.
- *Gnome*
- *java*
- *KDE3*
- *msvc* A C++ building tool, using Microsoft's Visual C compiler. It adds both a "cc" and a "cpp" build tool, since it is able to build both C and C++ projects. Like g++, it supports various types of C/C++ projects. These types equal those of g++, as does create_obj() call.

  CAUTION: Do not use msvc and gcc/g++ at the same time. Doing so results in undefined behaviour, since both msvc and gcc/g++ try to set the cc/cpp build tools. If you want to support multiple compilers, make it possible to select one via a command-line option instead.
- *osx*
- *Ocaml*
- *Qt3* Qt3 support tool. Needs a C++ tool, so for using the Qt3 tool, 'g++' or 'msvc' need to be initialized via check_tool() as well.
  This tool provides a build tool "qt3" for use with bld.create_obj(). The build types equal the g++/msvc ones. So, for building a Qt3 program, use *bld.create_obj('qt3', 'program')*.
  The tool tries to autodetect Qt3. However, some installations cannot be autodetected because of unusual paths and/or different naming. For this, the tool also adds custom command-line options (visible via calling "waf –help"). Support for automatic moc calling is also included; however, the C++ source files must have this line at the end:

  ```
  #include "headername.moc"
  ```

  This is necessary to let waf know when to invoke moc. A class using Qt signals is always defined in a header, and its methods are defined in a source file. For example, a class "Foo" is being defined in "foo.h" and its methods defined in a file "foo.cpp". Foo contains some Qt slots, so moc has to process the header. For this, the line

  ```
  #include "foo.moc"
  ```

needs to be added at the end of foo.cpp. (foo.moc, because the header name is "foo", omitting the extension). Note: it is important that waf can actually reach the header, so it has to be ensured that the include paths are correct. *obj.includes* may have to be adjusted for this (see chapter 2.4 for an explanation of *obj.includes*). Also, the Qt3 tool **must** be called in the *set_options()* function by calling 'opt.tool_options('Qt3')' (see chapter 2.2.1 for more about *opt.tool_options()*.

- *Qt4* Qt4 support tool, behaves like the Qt3 one. The Qt3 and Qt4 tools can be used in the same wscript (e.g. calling *conf.check_tool("g++ Qt3 Qt4")* is valid). Like the Qt3 tool, it must be called in the *set_options()* function.
- *Tex*
- *yacc*

Note that these are not all tools present in wafadmin/Tools/; the internal ones have been left out.

3.2 Unit tests

Unit testing works in waf by making use of the *unit_test* variable in build objects (this variable has been explained in chapter 2.4 above). The unit test is meant to be ran in the shutdown() function. You need to import the "UnitTest" module first. Using it is quite straightforward:

1. First create a unit_test instance.
2. Call unit_test.run(); this calls all projects marked as unit tests (e.g. obj.unit_test set to 1). Note that run() does not output anything.
3. Now, either read the test results manually (they are stored in the unit_test instance), or call unit_test.printt_results(), which writes the results to stdout with formatting.

Here is an example:

```
def shutdown():
        import UnitTest

        unittest = UnitTest.unit_test()
        unittest.run()
        unittest.print_results()
```

The output from the print_results() call in this example:

```
Running unit tests

default/src/testprogram ....... OK

Successful tests:      1 (100.0%)
Failed tests:          0 (0.0%)
Erroneous tests:       0 (0.0%)

Total number of tests: 1

Unit tests finished
```

unit_test has some members, *returncode_ok* for determining the unit test "OK" return code, and the others are filled in with data by the run() method.

- *returncode_ok*: The "OK" return code. If a unit test returns this code, it is counted as being successful, else the unit test run will be seen as failed. Default value is 0. Note that if you want to set another code, this has to be done *before* run() is called.
- *num_tests_ok*: Contains the amount of successful unit test runs.
- *num_tests_failed*: Contains the amount of failed unit test runs.
- *num_tests_err*: Contains the amount of failed erroneous test runs (for example a crashed unit test).
- *total_num_tests*: The total amount of unit tests. Equals num_tests_ok + num_tests_failed + num_tests_err.
- *max_label_length*: Labels are names for the unit test; in the example above, "default/src/testprogram" is a label. *max_label_length* contains the largest label; this is useful for pretty-print output.