# AMBuild Tutorial

Writing project files with AMBuild is fairly easy. This tutorial will guide you through making simple AMBuild scripts to compile and package a C++ project.

For information on the full API, see AMBuild API.

## Contents

## Simple Project

To begin, let's say we have a sample project with the following files:

```
$ ls
goodbye.cpp  helpers.cpp  README.txt
```

To start, we need to generate a default AMBuild configure script. This is the script that will perform the "configure" step for your build. You can generate one with the following command:

```
$ ambuild --new-project
$ ls
AMBuildScript  configure.py  goodbye.cpp  helpers.cpp  README.txt
```

The configure script simply invokes AMBuild. It can be modified (as we'll see later) to take extra command line options.

```
$ cat configure.py
# vim: set sts=2 ts=8 sw=2 tw=99 noet:
import sys
from ambuild2 import run

prep = run.BuildParser(sys.path[0], api='2.2')
prep.Configure()
```

Now, we're ready to actually make a build script for our project. The master build script must be a file called `AMBuildScript`, and it must be written in Python (http://python.org/) syntax. The full Python API on your system is available to AMBuild scripts, but the important aspect we'll deal with here is the AMBuild API.

The first step is to tell AMBuild to detect the first available C or C++ compiler. This is done with the following line:

```
cxx = builder.DetectCxx()
```

With just this line in your build script, you can now try to configure build. You should see something like:

```
$ mkdir build
$ cd build
$ python ../configure.py
Checking CC compiler (vendor test gcc)... ['cc', 'test.c', '-o', 'test']
found gcc version 4.7
Checking CXX compiler (vendor test gcc)... ['c++', 'test.cpp', '-o', 'testp']
found gcc version 4.7
$
```

If you get an error - it's likely you don't have a compiler installed. Make sure gcc, clang, or Microsoft Visual Studio is available where appropriate.

Now, we're ready to complete our AMBuildScript:

```
program = cxx.Program("hello")
program.sources = [
  'main.cpp',
  'helpers.cpp',
]
builder.Add(program)
```

The `builder` object is an instance of an AMBuild *context* - more about this is in the AMBuild API documentation. Every AMBuild script has access to a `builder`. The `builder.cxx` object has information about the C/C++ compiler for the configure session. The `Program()` method will return an object used to create C++ compilation tasks. In this case, we're asking to build an executable that will be named 'hello' (or `hello.exe` on Windows). You can also specified shared libraries with `Library`, and static libraries with `StaticLibrary`.

You can attach a list of source files to your Program via the `sources` attribute. Finally, use `builder.Add` to take your C++ configuration and construct the necessary dependency graph and build steps.

Now, we can actually attempt to build. First, let's make sure AMBuild computed our graph and dependencies correctly:

```
$ python ../configure.py
$ ambuild --show-graph
 : mkdir "hello"
 - hello/hello
   - c++ main.o helpers.o -o hello
     - hello/main.o
       - [gcc] -> c++ -H -c /home/dvander/projects/ambuild/ambuild2/main.cpp -o main.o
         - /home/dvander/projects/ambuild/ambuild2/main.cpp
     - hello/helpers.o
       - [gcc] -> c++ -H -c /home/dvander/projects/ambuild/ambuild2/helpers.cpp -o helpers.o
         - /home/dvander/projects/ambuild/ambuild2/helpers.cpp
$ ambuild --show-steps
mkdir -p hello
task 0: [gcc] -> c++ -H -c /home/dvander/projects/ambuild/ambuild2/main.cpp -o main.o
  -> hello/main.o
task 1: [gcc] -> c++ -H -c /home/dvander/projects/ambuild/ambuild2/helpers.cpp -o helpers.o
  -> hello/helpers.o
```

```
task 2: c++ main.o helpers.o -o hello
  -> hello/hello
```

It looks good! Now we can build:

```
$ ambuild
mkdir -p hello
Spawned task master (pid: 15563)
Spawned worker (pid: 15564)
Spawned worker (pid: 15565)
[15564] c++ -H -c /home/dvander/projects/ambuild/ambuild2/helpers.cpp -o helpers.o
[15565] c++ -H -c /home/dvander/projects/ambuild/ambuild2/main.cpp -o main.o
[15565] c++ main.o helpers.o -o hello
[15565] Child process terminating normally.
[15564] Child process terminating normally.
[15563] Child process terminating normally.
Build succeeded.
$ ./hello/hello
Hello!
```

Note that AMBuild gives each C++ binary its own folder. For example, if you build a static library called `egg.a`, a shared library called `egg.so`, and an executable called `egg` all in the same folder, AMBuild will actually perform each of these builds in separate folders, and the binary paths will look like:

- `egg.a/egg.a`
- `egg.so/egg.so`
- `egg/egg`

This is to allow complex build scenarios where the same files are rebuilt multiple times.

## Packaging

Now that our project builds, let's add to our build script so that we can create a build package. We'd like to make a folder we can zip or tar for distribution, with the following files:

- `README.txt`, our readme
- `hello`, our final binary

First, we have to add a step to the build to create the distribution folder:

```
dist_folder = builder.AddFolder('dist')
```

The return value from `AddFolder` is a dependency graph node, that we can use as an input to future steps.

Now we can copy our files:

```
outputs = builder.Add(program)

folder = builder.AddFolder('dist')
builder.AddCopy(os.path.join(builder.sourcePath, 'README.txt'), folder)
builder.AddCopy(outputs.binary, folder)
```

`README.txt` can be copied directly from the source tree. To copy the executable, we use the return value of `builder.Add()`. We could construct its path ourselves, but having the dependency object already available is much more convenient.

Now, when we build, we see:

```
[5952] cp "/home/dvander/projects/ambuild/ambuild2/README.txt" "./dist/README.txt"
Spawned worker (pid: 5953)
[5952] c++ -H -c /home/dvander/projects/ambuild/ambuild2/helpers.cpp -o helpers.o
[5954] c++ -H -c /home/dvander/projects/ambuild/ambuild2/main.cpp -o main.o
[5954] c++ main.o helpers.o -o hello
[5954] cp "hello/hello" "./dist/hello"
```

Since copying `README.txt` has no dependencies, it can execute in parallel with other jobs, even before compilation has finished. It won't be copied again unless `README.txt` changes. However the copy of `hello/hello` has to occur last. We can see that it succeeded with:

```
$ ls -l dist/
total 12
-rwxr-xr-x 1 dvander dvander 7036 Oct 16 22:32 hello
-rw-r--r-- 1 dvander dvander   23 Oct 16 22:32 README.txt
```

It is also possible to add a step to execute a command like "tar" or "zip", but there's a complication. There must be a dependency to every file that would be included in the command, otherwise, the commands might occur out of order. We are still looking into easier ways to automate this.

## Multiple Scripts

Non-trivial projects usually need more than one build script. AMBuild allows build scripts to nest; any script can run another script. Each script gets its own `builder`, known internally as a *context*. All jobs are created within a context and associated with that context. This allows AMBuild to reparse a minimal number of build scripts when a build script changes.

Contexts, by default, are associated with the folder they exist in relative to the source tree. For example, a build script in `/source-tree/src/game/AMBuild` will have a context associated with `src/game`. This folder structure is mirrored in the build folder, and all jobs occur within the context's local folder. For example, let's move our packaging into a separate script, `PackageScript`:

```
# PackageScript
import os

builder.SetBuildFolder('dist')
builder.AddCopy(os.path.join(builder.sourcePath, 'README.txt'), '.')
builder.AddCopy(Hello.binary, '.')
```

Then we modify our main `AMBuildScript`:

```
# AMBuildScript
cxx = builder.DetectCxx()

program = cxx.Program("hello")
program.sources = [
  'main.cpp',
  'helpers.cpp',
]
outputs = builder.Add(program)

builder.Build(
  ['PackageScript'],
  { 'Hello': outputs }
)
```

The first parameter is an array of script paths to run, and the second is a dictionary of global variables to give each script. Note that since our PackageScript is in the root of the source tree, by default its build folder is '.', so we manually override its build folder.

When PackageScript is parsed during the configure step, all of its jobs will automatically be configured to occur inside a dist folder within the build folder, so '.' actually refers to ./dist/.

If you have only one script, you can use Build instead. This also lets scripts return a value. For example:

```
# AMBuildScript
folders = builder.Build('MakeFolders')
```

```
# MakeFolders
folders = [
  builder.AddFolder('egg'),
  builder.AddFolder('yam'),
  builder.AddFolder('plant'),
]

# Magic variable; assigning to "rvalue" will propagate the value
# back up to Build().
rvalue = folders
```

It is possible to pass variables to other scripts (they appear as global variables), or they can be attached to the builder object and they will automatically be propagated down. It's useful to propagate compiler objects this way, so they don't have to be re-detected and reconfigured in every subsequent script. For more information, see AMBuild API.

For example, to pass our compiler around, we can use:

```
# AMBuildScript
builder.cxx = builder.DetectCxx()
```

Instead. This does not work for multi-architecture builds (for example, a single build producing x86 and x86_64 binaries), so if you support those, you will need a list of compilers instead.

## Custom Options

It is possible to add custom options to the configure step using Python's optparse (http://docs.python.org/2/library/optparse.html) module. Recall the default configure.py that AMBuild generates:

```
# vim: set sts=2 ts=8 sw=2 tw=99 noet:
import sys, ambuild2.run

parser = run.BuildParser(sys.path[0], api='2.2')
parser.Configure()
```

The parser object has an options attribute, which is an instance of argparse.ArgumentParser. You can add to it, for example,

```
# vim: set sts=2 ts=8 sw=2 tw=99 noet:
import sys, ambuild2.run

parser = run.BuildParser(sys.path[0], api='2.2')
parser.options.add_argument('--enable-debug', action='store_true', dest='debug', default=False,
                            help='Enable debugging symbols')
parser.options.add_argument('--enable-optimize', action='store_true', dest='opt', default=False,
                            help='Enable optimization')
parser.Configure()
```

These options can be accessed from any `builder` object, like so:

```
if builder.options.debug:
  cxx.cflags += ['-O0', '-ggdb3']
  cxx.cdefines += ['DEBUG']
if builder.options.opt:
  cxx.cflags += ['-O3']
  cxx.cdefines += ['NDEBUG']
```

## Weak Dependencies

Sometimes it is useful to force build steps to occur in distinct phases. Normally, this would be the antithesis of what we want: the dependency graph should precisely and perfectly represent dependencies, and there should be no need to enforce order manually. That's true, but there are situations that warrant relaxing how we construct the graph.

One example is generated headers in particular pose a problem. If we created dependencies on a "generate headers" task, then generating new headers would trigger recompiling every source file - even ones that never included those headers. Furthermore, if we created dependencies on each individual generated header, we'd have a huge dependency graph - 50 includes and 800 source files would mean 80,000 dependency links. AMBuild solves the first problem in the same way tup (http://gittup.org/tup/) does. We do not yet attempt to solve the second problem, though it is planned for the future.

First, we introduce the concept of a *weak dependency*. A weak dependency is one that theoretically exists, and must exist for ordering, but does not propagate damage. For example, let's say that `hello.cpp` has a weak dependency on `generated.h`. If `hello.cpp` doesn't `#include "generated.h"`, then no changes to `generated.h` should ever trigger a rebuild of `hello.cpp`. However, if `hello.cpp` is changed to include `generated.h`, then the weak dependency ensures those jobs are executed in the right order. (It is illegal in AMBuild to depend on a generated file without having an explicit dependency.) The weak dependency can then be upgraded to a strong dependency, and possibly downgraded again later if the `#include` is removed.

As an example, let's say we have two steps: generating headers and actual compilation. First, we add a shell command and save its outputs in a global variable.

```
generated_headers = builder.AddCommand(
  argv = ['python', os.path.join(builder.buildPath, 'tools', 'buildbot', 'generate_headers.py')],
  inputs = [os.path.join(builder.sourcePath, '.hg', 'dirstate')],
  outputs = ['sourcemod_auto_version.h']
)
```

In another build script, assuming we communicated the `headers` object through, we could add:

```
library = cxx.Library('cstrike')
library.sources = ['cstrike.cpp', 'smsdk_ext.cpp']
library.sourcedeps += generated_headers
builder.Add(library)
```

And now when our generated headers change, we are guaranteed that if our library's sources need to be recompiled, they will be compiled after the headers are generated.

## Many Source Groups

Projects may have complex components that form a single, monolithic unit. For example, portions of code may need certain compile options that other areas do not need. This can be accomplished in AMBuild using modules. For example, the root of a project might look like this:

```
program = cxx.Program('sample')

builder.Build(['cairo/AMBuild', 'gtk/AMBuild'], {
  'program': program
})

builder.Add(program)
```

Then, `cairo/AMBuild` can do:

```
module = program.Module(builder, 'cairo')
module.sources += [
  'file.cc',
]
module.cxxflags += ['-Wno-flag-needed-for-cairo']
```

The module will build as part of the entire binary.

## Reconfiguring

Reconfiguring can happen for two reasons. One is if you change some properties of the build, for example, configuring a debug build over an existing optimized build. Another is if a build script changes, AMBuild will automatically reconfigure the build using the previous configure options.

### Output Cleaning

When a reconfigure occurs, AMBuild will produce a new dependency graph alongside the old dependency graph, and these graphs are then merged. Any generated files in the old graph that are not present in the new graph are removed from the file system. This is necessary to ensure that builds do not become inconsistent or corrupt: every incremental build should be identical to a clean build.

For example, our object folder for our original script might look like:

```
-rw-rw-r-- 1 dvander dvander  933 Nov 11 02:26 goodbye.o
-rw-rw-r-- 1 dvander dvander 1232 Nov 11 02:26 helpers.o
-rwxrwxr-x 1 dvander dvander 8509 Nov 11 02:26 sample
```

If we comment out 'goodbye.cpp' from `AMBuildScript`, and build again:

```
Reparsing build scripts.
Checking CC compiler (vendor test gcc)... ['cc', 'test.c', '-o', 'test']
found gcc version 4.7
Checking CXX compiler (vendor test gcc)... ['c++', '-fno-exceptions', '-fno-rtti', 'test.cpp', '-o', 'testp']
found gcc version 4.7
Removing old output: sample/goodbye.o
```

```
Spawned taskmaster (pid: 41899)
Spawned worker (pid: 41900)
[41900] c++ helpers.o -o sample
Build succeeded.
```

And indeed, `goodbye.o` is gone:

```
-rw-rw-r-- 1 dvander dvander 1232 Nov 11 02:26 helpers.o
-rwxrwxr-x 1 dvander dvander 8473 Nov 11 02:28 sample
```

## Minimal Reparsing

Unlike the normal dependency graph, AMBuild scripts can depend on each other. For example, a nested script may propagate a graph object back to the root script, which then propagates it down again. This creates an implicit cycle: if either script changes, the entire cycle must be reparsed. Avoiding this is extremely difficult as it is easy to construct situations in which a minimal reparse algorithm would fail. Thus, at the moment, AMBuild performs full reparses instead.

If this becomes a performance bottleneck, which it might in a huge project with hundreds of build scripts, we would like to implement minimal reparsing. It would likely require transitioning to a more restricted API that disallows (or discourages) arbitrary dataflow between build scripts. To solve the problem mentioned earlier, we would need specific functions that allow AMBuild to track script inter-dependencies.

Even with full reparsing however, AMBuild will only remove or rebuild individual jobs that have changed. If you add a single .cpp file to a source list, a full reparse will only result in that file being built and its binary relinked (and of course, any tasks depending on that binary).

## Refactoring

Sometimes, it is useful to see whether changing build scripts would actually change the dependency graph. For example, refactoring them, or tracking down excessive rebuild problems. In Tup, this can be done with a "refactoring" build, and AMBuild (sort of) supports this feature. It is still new and may not catch all problems.

An example of a refactoring build where the dependency graph has changed:

```
dvander@linux64:~/temp$ ambuild --refactor obj-linux-x86_64
Reparsing build scripts.
Checking CC compiler (vendor test gcc)... ['cc', 'test.c', '-o', 'test']
found gcc version 4.7
Checking CXX compiler (vendor test gcc)... ['c++', '-fno-exceptions', '-fno-rtti', 'test.cpp', '-o', 'testp']
found gcc version 4.7
New output introduced: sample
Failed to reparse build scripts.
```