`ccbuild` - A strict developer's build utility

A. Bram Neijt

2.0.1

Abstract

This document is a general usage manual to `ccbuild`. It will introduce ways of using `ccbuild`. It will also explain `ccbuild`'s behaviour in more words then the manual does. The newest version of `ccbuild` can be found at the `http://www.logfish.net/pr/ccbuild/`

Copyright Notice

Contents

Chapter 1

Introduction

Programming should not become an administrative task. With most of the tools on the net I found myself either adding file names to scripts (and removing them) or digging deep into the core of some scripting language to try automate my builds. At first I ended up with a clumsy, to say the least, collection of `find`, `grep`, `sed` and `bash`. Combined they formed a script to generate Makefiles dispatched over multiple directories. The complexity quickly grew out of hand and it left me with a lot of clutter around my source. The second incarnation helped to generate autotools scripts. I had to run `./configure` to get the new scripts to generate real Makefiles, followed by make. If you commonly add and remove files, updates are slow.

It was a depressing way to develop, because most additions to the project I made required me to run all those scripts. On top of that, it would take about 30 seconds, or so it felt.

You would say, well why not use an IDE? Well, trouble here was I wanted to split up my code in separate files... a lot of separate files. Not only do I separate my code into directories, I also use seperate files for functions of the same class. No IDE allowed me to do this without the hassle of having to go through a few menus (create file, register it as source, set is as part of the main program, etc.). Those who did, often used autotools in the background and would have to re-autotool on every added file. Frustrated, I almost gave up on my coding ethics and started to do it the way the masses do: 900+ lines of code in a single file, only accessible through a specially equipped text editor or full scale IDE.

The closest thing to perfection out there was `icmake`. This only requires you to mention the directories you use and will keep your development tree clean, however it had two small drawbacks: you needed to mention your classes and you could not use directories in directories. So, I decided to create a fast and simple program to solve my problem.

Although you have to keep some standards (split source over directories) to get `ccbuild` to work on your program, I found it almost completely eliminated my interaction with the build system. Better yet, it helps with bootstrapping other build systems.

Chapter 2

How it works

To know how to work with `ccbuild` is of course the most important thing (if you want to). So here is a quick and simple review of the internals.

We will now consider what happens when `ccbuild` is called without any arguments.

First `ccbuild` will find all source files in the current directory (using the list of source extensions to find them.) All these files are scanned which gives an in memory list of include statements and whether it has a main function.

If it has a main function, it is considered a binary target: `ccbuild` will try to make a program from this.

To find all the object files that need to be linked to the main program, `ccbuild` will follow all local include statements (warning if any fail). Then it will scan all files in the same directory as the included files. If they are object targets (don't have a `int main` function) they will be compiled and linked to the main program.

The arguments needed to compile an object are gathered by the global includes. Using the ccResolutions file, for every global include the arguments are added.

The needed linker arguments (which would create the "not linking now" warning) are identified and kept back for later use when the program is actually linked. If anything goes wrong here, please mail me and hack the file `src/Compiler/countFirstLinkerArguments.cc` for the meantime. This file contains two simple lists for for options with and options without arguments.

Chapter 3

Using `ccbuild`

3.1 Organizing your source

`ccbuild` will read your local includes (#include "something") and compile any source next to it into your program or library. For every class you want to use, make sure you create a separate directory. Every directory contains source files which define the different members of your class.

Because every member of a class has it's own file, each of these files will have an approximately equal header. To keep us from typing "using namespace" and "include <iostream>" for each of these files, a so called internal header file is created. The internal header file is the only file the member implementation include and is identified by the extension `.ih`.

An example member implementation is given `fileSystem/touch.cc`:

```
#include "fileSystem.ih"

bool FileSystem::touch(std::string const &filename)
{
  ofstream file(filename.c_str(), ios::app);
  bool succes = file.is_open();
  file.close();
  return succes;
}
```

The internal header file, `fileSystem/fileSystem.ih`:

```
#include "fileSystem.hh"
#include <fstream>
```

```
#include "../options/options.hh"

using namespace std;
using namespace bneijt;
```

The header file defines the `FileSystem` class in the `bneijt` namespace and includes only what is needed for it's declaration.

Splitting the source up like this will get you a lot of files, but will make editing and hacking your code simple. The functions are easy to find, quick to open and easy to grasp. Furthermore, version control software will encounter less collisions and patches will merge more easily on quicker moving code.

The main program is in the root of the source. `ccbuild` has the following listing:

```
./fileSystem/touch.cc
./fileSystem/fileSystem.ih
./fileSystem/isDirectory.cc
./fileSystem/cleanPath.cc
./fileSystem/modTime.cc
./fileSystem/fileExists.cc
./fileSystem/isReadable.cc
./fileSystem/absolutePath.cc
./ccResolutions
./string/replace.cc
./string/test.cc
./string/string.ih
./string/toUpper.cc
./string/string.hh
./options/options.hh
./options/options.ih
./options/statics.cc
./ccbuild.cc
```

The top most file is `ccbuild.cc`, which contains a special function: `int main`. `ccbuild` does not care about the arguments the main function takes, but it does care about it being `int main`. This is what `ccbuild` calls a binary target, a file that is the root of a binary.

3.2 Building a program

To build a configured `ccbuild` compatible source tree, simple run `ccbuild` in the directory containing the main program. This will compile all programs in the given directory. However, if you only want to compile one given program, issue the command `ccbuild build mainsource.cc`, where mainsource.cc should be the name of the main source file.

Once the command is issued, `ccbuild` will start reading includes the source does and gather sources it should compile. Any sources it can find will be compiled and linked to the main program. Once the `[LINK] mainsource` line get's done, without any errors, your main program will be done and you can start it with `./mainsource`.

3.3 Cleaning up

For cleaning your sourcetree, `ccbuild` offers two commands: `clean` and `distclean`. Although they might act almost the same, they are implemented quite different.

The `distclean` command is totally source independent: it does not scan sources, nor look for them. Distclean simply removes all `ccbuild` related file in the "o" directory and all ".gch" files everywhere. If the "o" directory is empty after that, the directory is removed as well.

The `clean` command is much more subtle: it reads the sources and removes any objects part of the current source tree. Because it reads the sources, using clean will only remove those sources part of the given or implied main binary target(s). This command will not remove any directories.

General rule is to use the force command when you want to update everything, use the `clean` when you want to remove all files for a local binary target (but not any other binary targets in the local directory) and use `distclean` to remove everything including old objects and pre-compiled headers.

Chapter 4

Moving to `ccbuild`

To be able to use `ccbuild`, you as a developer will have to adhere to some strict(er) rules then using something like autotools. Here is a list of things you should keep in mind when moving to `ccbuild`.

4.1 Strictness to adhere to

- `ccbuild` only reads local includes

  When creating your source, make sure that all sources that `ccbuild` should care about can be found using local includes. This means you should strictly use system wide includes only for actual system wide include files. So any header file which is part or your packages should be included using a local include statement.

- Preprocessing isn't helping

  To speed up `ccbuild`, it does not go around looking for system wide headers. This also means that it won't know all the preprocessing directives from these headers. This results in preprocessor excludes of local headers cannot be used. This is no problem if you are compiling for a single platform, but when you need configuration using preprocessor directives, you're going to get into trouble.

  The only way to keep `ccbuild` from reading these sources is by making sure there is a single space between the # and the `include` statement. So the include `# include "something/hello.hh"` will be ignored by `ccbuild`. To test this, run `ccbuild` in verbose mode (`--verbose`) and watch for the warning which state that the file in not included. You can also use the `deps` command to get a list for all binary targets.

4.2 Setting up your configuration file

To set up your ccResolutions file, it's best to do the following steps:

1 Check your local includes span over your whole source

To make sure ccbuild was able to follow your local includes, use the deps command. This will list all the local and global dependencies of a file. You may also use the dot command to get a graphical interpretation of the same information.

All paths that ccbuild needs to search for local includes should be added to the first line of your ccResolutions file. Using -l in this first line will make ccbuild highlight all compiler output.

```
#& -I../tools -I. -l
```

2 Add packages to your global ccResolutions

You can add a package to your global resolution configuration using the genPkgconfigList.sh tool. This will find all files in the include path of a package's include paths and add them to a resolution file. See the Tools section for more information.

3 Check the global includes are resolved

To make sure the global includes are resolved, use the resolve command.

```
ccbuild resolve |sort >> ccResolutions
```

Now all unresolved global headers are listed in your ccResolutions file. When you run ccbuild now, it won't complain about any global includes missing. However, g++ might complain because the needed extra arguments are not in place. You should now add the needed arguments to your ccResolutions file by using, for example, `'pkg-config --cflags --libs <packagename>'` with the needed package in place.

If you have a lot of resolution rules in your defaults (~/.ccbuild/), then it might be hard to see what your project actually depends on. Passing ccbuild the option --nodefres will cause it to skip loading these files and will allow you to see which resolutions fail. This might give you some hints on what packages your program depends on.

Chapter 5

Moving from `ccbuild`

There will be a day you want to move away from `ccbuild`. When the day comes, you would probably only be able to use `ccbuild` for it's dependency generation commands.

To make `ccbuild` useful in these later stages, `ccbuild` has a few commands to help you cope. Don't forget, you can remove all `ccbuild` generated files using:

```
ccbuild distclean;
rm ./ccResolutions;
```

The build script generation commands only read source and, should not generate any output.

5.1 General build file generation

`ccbuild` can generate a number of different files for different build systems. When you call `ccbuild` with a build generation command without a source file, it will try to create a standalone file for that build system. Which will also contain an all rule.

For most systems however, you don't want the all rule to be defined. So, `ccbuild` allows you to state which source you want a build file for. This will then generate a build file without the all rule. Then simply include this build file into your main build file and write the all rule yourself.

5.2 Generating A-A-P files

One of the most useful generation features is probably the A-A-P file generation. You can use this by calling `ccbuild` with the aap command. This will generate an A-A-P file on the stdout.

The most common way of using this aap file is by generating it for a single binary target using:

```
ccbuild aap mainsource.cc > mainsource.aap
```

Or

```
ccbuild aap src/mainsource.cc > mainsource.aap
```

Then create a main.aap file with the following line:

```
:include mainsource.aap

all : ./mainsource
```

Or

```
:include mainsource.aap

all : ./src/mainsource
```

Add any recipes needed and then use aap to generate the main program.

Chapter 6

Problem solving with ccbuild

This is a collection of possible real usage examples for `ccbuild`. If you don't want to take the time to read the manual pages, this is a more problem oriented listing of the same.

6.1 You changed a class interface

When you change a class interface, a large collection of your code will probably break down. But which parts? Well, a hint of which files will be affected can be seen using `ccbuild check`. However, this won't show you whether these sources still compile or not. The only way to test that is by simply running `ccbuild`.

Solution: Use an editor running in the background (something that returns after using the command). An example is using the gedit command when you already have a window open: new files will be opened in a tab and the gedit command will return immediately. So, using gedit, the easiest way to get an overview of your problems would be: `ccbuild` –brute –xof "gedit"

Appendix A

The tools directory

The `tools` directory contains a few scripts and default files that may come in handy. These are not meant to be used a lot and are mostly there as examples of using `ccbuild` in combination with other programs. These utilities are often crude and come with NO WARRANTY WHATSOEVER.

Bottom line: read them before you use them and enjoy.

A.1 genPkgconfigList.sh

This script will generate a list of includes that might be part of the given package. The scripts needs to get a valid package name as it's first argument. It will then call pkg-config to get a list of include paths used for the package. All these paths are searched and all files found are linked to the package using pkg-config in a way that is compatible with ccResolutions syntax.

This list will be very large, and it's not, generally, a good idea to add this list to your local ccResolutions file. A better way of using this is by adding the file to your ccResolutions.d directory under the name of the package.

Using this is of course a brute way of handling resolutions, because it's much nicer to only resolve the ones you need.

A.2 ccbuildStatusPage.sh

This is a small `ccbuild` status page creation script. All command line arguments you give it will be passed to `ccbuild` directly. It runs "ccbuild check" to check which files are up to date and which are not. Then using AWK it translates this into a small auto-refreshing web page. The web page uses ccbuild.css as it's style sheet.

General usage for a single run is:

```
sh ccbuildStatusPage.sh -C "someproject/src"
```

Then use your favourite browser to open the generated html file: ccbuildstatus.html.

You can easily loop it in the background using:

```
while [[ 1 ]];
  do sleep 5;
  nice sh ccbuildStatusPage.sh -C "someproject/src";
done;
```

By default the up to date files are not shown by using “display: none” in the ccbuild.css. Remove this line from ccbuild.css to show all up to date files as well.

Appendix B

Categorically sorted command line parameters

Here is list of the command line parameters divided over categories. If you think you know `ccbuild`, go down this list. If you don't think a given argument is in the right position, you might need to read up on it. Please refer to the `ccbuild` manual for a full explanation of these flags and arguments.

Command execution influencing arguments (the actual system call): `--compiler`, `-a`, `--args`, `-I`, `--recursive-include`, `--xof`, `--exec-on-fail`, `--xop`, `--exec-on-pass`, `--append` and Resolution arguments.

Global header resolution effecting arguments: `-C`, `--nodefres`, `--addres` and `--nodefargs` (if the default commands contain any of the before mentioned).

Command (build/lib/distclean etc.) effecting arguments: `-s`, `--no-act`, `-p`, `--precompile-ih`, `--precompile-all`, `--brute`, `--loop`, `--verbose`.

Arguments that won't, normally, change the resulting binary or output: `-f`, `--force-update`, `--gnutouch`, `--md5`, `--real-man`, `-l`, `--highlight`, `--xof`, `--exec-on-fail`, `--xop`, `--exec-on-pass`, `--clearpc`, `--append`.

Read only actions are:

- Anything with `-s` or `--no-act`
- The commands: `resolve`, `md5`, `deps`, `dot filename.cc`, `makefile`, `aap`, `check` and `icmake`.