# SCons User Guide 0.92

Steven Knight

**SCons User Guide 0.92**
by Steven Knight

Revision 0.92.D001 (2003/08/19 19:27:56) Edition
Published 2003

# Table of Contents

# Chapter 1. Preface

Thank you for taking the time to read about SCons. SCons is a next-generation software construction tool, or make tool--that is, a software utility for building software (or other files) and keeping built software up-to-date whenever the underlying input files change.

The most distinctive thing about SCons is that its configuration files are actually *scripts*, written in the Python programming language. This is in contrast to most alternative build tools, which typically invent a new language to configure the build. SCons still has a learning curve, of course, because you have to know what functions to call to set up your build properly, but the underlying syntax used should be familiar to anyone who has ever looked at a Python script.

Paradoxically, using Python as the configuration file format makes SCons *easier* for non-programmers to learn than the cryptic languages of other build tools, which are usally invented by programmers for other programmers. This is in no small part to the consistency and readability that are built in to Python. It just so happens that making a real, live scripting language the basis for the configuration files makes it a snap for more accomplished programmers to do more complicated things with builds, as necessary.

## Why SCons?

SCons is a response to a perennial problem: building software is harder than it should be. In a nutshell: the old, reliable model of the venerable and ubiquitous Make program has had a hard time keeping up with how complicated building software has become. The fact that Make has kept up as well as it has is impressive, and a testament to how the simplicity. But anyone who has wrestled with Automake and Autoconf to try to guarantee that a bit of software will build correctly on multiple platforms can tell you that it takes a lot of work to get right.

## SCons Principles

There are a few overriding principles we try to live up to in designing and implementing SCons:

Correctness

First and foremost, by default, SCons guarantees a correct build even if it means sacrificing performance a little. We strive to guarantee the build is correct regardless of how the software being built is structured, how it may have been written, or how unusual the tools are that build it.

Performance

Given that the build is correct, we try to make SCons build software as quickly as possible. In particular, wherever we may have needed to slow down the default SCons behavior to guarantee a correct build, we also try to make it easy to speed up SCons through optimization options that let you trade off guaranteed correctness for speed.

Convenience

SCons tries to do as much for you out of the box as reasonable, including detecting the right tools on your system and using them correctly to build the software.

In a nutshell, we try hard to make SCons just "do the right thing" and build software correctly, with a minimum of hassles.

## History

`SCons` originated with a design that was submitted to the Software Carpentry design competition in 2000.

`SCons` is the direct descendant of a Perl utility called `Cons`. `Cons` in turn based some of its ideas on `Jam`, a build tool from Perforce Systems.

XXX

## Acknowledgements

`SCons` would not exist without a lot of help from a lot of people, many of whom may not even be aware that they helped or served as inspiration. So in no particular order, and at the risk of leaving out someone:

First and foremost, `SCons` owes a tremendous debt to Bob Sidebotham, the original author of the classic Perl-based `Cons` tool which Bob first released to the world back around 1996. Bob's work on Cons classic provided the underlying architecture and model of specifying a build configuration using a real scripting language. My real-world experience working on Cons informed many of the design decisions in SCons, including the improved parallel build support, making Builder objects easily definable by users, and separating the build engine from the wrapping interface.

Greg Wilson was instrumental in getting `SCons` started as a real project when he initiated the Software Carpentry design competition in February 2000. Without that nudge, marrying the advantages of the Cons classic architecture with the readability of Python might have just stayed no more than a nice idea.

The entire `SCons` team have been absolutely wonderful to work with, and `SCons` would be nowhere near as useful a tool without the energy, enthusiasm and time people have contributed over the past few years. The "core team" of Chad Austin, Anthony Roach, Charles Crain, Steve Leblanc, Greg Spencer and Christoph Wiedemann have been great about reviewing my (and other) changes and catching problems before they get in the code base. Of particular technical note: Anthony's outstanding and innovative work on the tasking engine has given `SCons` a vastly superior parallel build model; Charles has been the master of the crucial Node infrastructure; Christoph's work on the Configure infrastructure has added crucial Autoconf-like functionality; and Greg has provided excellent support for Microsoft Visual Studio.

Special thanks to David Snopek for contributing his underlying "Autocons" code that formed the basis of Christoph's work with the Configure functionality. David was extremely generous in making this code available to `SCons`, given that he initially released it under the GPL and `SCons` is released under a less-restrictive MIT-style license.

`SCons` has received contributions from many other people, of course: Matt Balvin (extending long command-line support on Win32), Allen Bierbaum (extensions and fixes to Options), Steve Christensen (help text sorting and function action signature fixes), Michael Cook (avoiding losing signal bits from executed commands), Derrick 'dman' Hudson (), Alex Jacques (work on the Win32 scons.bat file), Stephen Kennedy (performance enhancements), Lachlan O'Dea (SharedObject() support for masm and normalized paths for the WhereIs() function), Damyan Pepper (keeping output like Make), Jeff Petkau (significant fixes for CacheDir and other areas), Stefan Reichor (Ghostscript support), Zed Shaw (Append() and Replace() environment methods), Terrel Shumway (build and test fixes, as well as the SCons Wiki), sam th (dynamic checks for utilities) and Moshe Zadke (Debian packaging).

Thanks to Peter Miller for his splendid change management system, `Aegis`, which has provided the `SCons` project with a robust development methodology from day one, and which showed me how you could integrate incremental regression tests into a practical development cycle (years before eXtreme Programming arrived on the scene).

And last, thanks to Guido van Rossum for his elegant scripting language, which is the basis not only for the `SCons` implementation, but for the interface itself.

## Contact

The best way to contact people involved with SCons, including the author, is through the SCons mailing lists.

If you want to ask general questions about how to use `SCons` send email to `scons-users@lists.sourceforge.net`.

If you want to contact the `SCons` development community directly, send email to `scons-devel@lists.sourceforge.net`.

If you want to receive announcements about `SCons`, join the low-volume `scons-announce@lists.sourceforge.net` mailing list.

# Chapter 2. Simple Builds

Here's the famous "Hello, World!" program in C:

```
int
main()
{
    printf("Hello, world!\n");
}
```

And here's how to build it using `SCons`. Enter the following into a file named `SConstruct`:

```
env = Environment()
env.Program('hello.c')
```

That's it. Now run the `scons` command to build the program. On a POSIX-compliant system like Linux or UNIX, you'll see something like:

```
% scons
cc -c hello.c -o hello.o
cc -o hello hello.o
```

On a Windows system with the Microsoft Visual C++ compiler, you'll see something like:

```
C:\>scons
cl /Fohello.obj hello.c
link /Fohello.exe hello.obj
```

First, notice that you only need to specify the name of the source file, and that `SCons` deduces the names of the object and executable files correctly from the base of the source file name.

Second, notice that the same input `SConstruct` file, without any changes, generates the correct output file names on both systems: `hello.o` and `hello` on POSIX systems, `hello.obj` and `hello.exe` on Windows systems. This is a simple example of how `SCons` makes it extremely easy to write portable software builds.

(Note that we won't provide duplicate side-by-side POSIX and Windows output for all of the examples in this guide; just keep in mind that, unless otherwise specified, any of the examples should work equally well on both types of systems.)

## The `SConstruct` File

If you're used to build systems like `Make` you've already figured out that the `SConstruct` file is the `SCons` equivalent of a `Makefile`. That is, the `SConstruct` file is the input file that `SCons` reads to control the build.

There is, however, an important difference between an `SConstruct` file and a `Makefile`: the `SConstruct` file is actually a Python script. If you're not already familiar with Python, don't worry. This User's Guide will introduce you step-by-step to the relatively small amount of Python you'll need to know to be able to use `SCons` effectively. And Python is very easy to learn.

One aspect of using Python as the scripting language is that you can put comments in your `SConstruct` file using Python's commenting convention; that is, everything between a '#' and the end of the line will be ignored:

```
env = Environment()    # Create an environment.
# Arrange to build the "hello" program.
env.Program('hello.c')
```

You'll see throughout the remainder of this Guide that being able to use the power of a real scripting language can greatly simplify the solutions to complex requirements of real-world builds.

## Compiling Multiple Source Files

You've just seen how to configure `SCons` to compile a program from a single source file. It's more common, of course, that you'll need to build a program from many input source files, not just one. To do this, you need to put the source files in a Python list (enclosed in square brackets), like so:

```
env = Environment()
env.Program(['prog.c', 'file1.c', 'file2.'])
```

A build of the above example would look like:

```
% scons
cc -c file1.c -o file1.o
cc -c file2.c -o file2.o
cc -c prog.c -o prog.o
cc -o prog prog.o file1.o file2.o
```

Notice that `SCons` deduces the output program name from the first source file specified in the list--that is, because the first source file was `prog.c`, `SCons` will name the resulting program `prog` (or `prog.exe` on a Windows system). If you want to specify a different program name, then you slide the list of source files over to the right to make room for the output program file name. (`SCons` puts the output file name to the left of the source file names so that the order mimics that of an assignment statement: "program = source files".) This makes our example:

```
env = Environment()
env.Program('program', ['main.c', 'file1.c', 'file2.'])
```

On Linux, a build of this example would look like:

```
% scons
cc -c file1.c -o file1.o
cc -c file2.c -o file2.o
cc -c main.c -o main.o
cc -o program main.o file1.o file2.o
```

Or on Windows:

```
C:\>scons
cl /Fofile1.obj file1.c
cl /Fofile2.obj file2.c
cl /Fomain.obj main.c
link /Foprogram.exe main.obj file1.obj file2.obj
```

## Keeping SConstruct Files Easy to Read

One drawback to the use of a Python list for source files is that each file name must be enclosed in quotes (either single quotes or double quotes). This can get cumbersome and difficult to read when the list of file names is long. Fortunately, SCons and Python provide a number of ways to make sure that the SConstruct file stays easy to read.

To make long lists of file names easier to deal with, SCons provides a Split function that takes a quoted list of file names, with the names separated by spaces or other white-space characters, and turns it into a list of separate file names. Using the Split function turns the previous example into:

```
env = Environment()
env.Program('program', Split('main.c file1.c file2.'))
```

Putting the call to the Split function inside the env.Program call can also be a little unwieldy. A more readable alternative is to assign the output from the Split call to a variable name, and then use the variable when calling the env.Program function:

```
env = Environment()
list = Split('main.c file1.c file2.')
env.Program('program', list)
```

Lastly, the Split function doesn't care how much white space separates the file names in the quoted string. This allows you to create lists of file names that span multiple lines, which often makes for easier editing:

```
env = Environment()
list = Split('main.c
              file1.c
              file2.c')
env.Program('program', list)
```

## Keyword Arguments

SCons also allows you to identify the output file and input source files using Python keyword arguments. The output file is known as the *target*, and the source file(s) are known (logically enough) as the *source*. The Python syntax for this is:

```
env = Environment()
list = Split('main.c file1.c file2.')
env.Program(target = 'program', source = list)
```

Whether or not you choose to use keyword arguments to identify the target and source files is purely a personal choice; SCons functions the same either way.

## Compiling Multiple Programs

In order to compile multiple programs within the same SConstruct file, simply call the env.Program method multiple times, once for each program you need to build:

```
env = Environment()
env.Program('foo.c')
env.Program('bar', ['bar1.c', 'bar2.c'])
```

SCons would then build the programs as follows:

```
% scons
cc -c bar1.c -o bar1.o
cc -c bar2.c -o bar2.o
cc -o bar bar1.o bar2.o
cc -c foo.c -o foo.o
cc -o foo foo.o
```

Notice that `SCons` does not necessarily build the programs in the same order in which you specify them in the `SConstruct` file. `SCons` does, however, recognize that the individual object files must be built before the resulting program can be built. We'll discuss this in greater detail in the "Dependencies" section, below.

## Sharing Source Files Between Multiple Programs

It's common to re-use code by sharing source files between multiple programs. One way to do this is to create a library from the common source files, which can then be linked into resulting programs. (Creating libraries is discussed in section XXX, below.)

A more straightforward, but perhaps less convenient, way to share source files between multiple programs is simply to include the common files in the lists of source files for each program:

```
env = Environment()
env.Program(Split('foo.c common1.c common2.c'))
env.Program('bar', Split('bar1.c bar2.c common1.c common2.c'))
```

`SCons` recognizes that the object files for the `common1.c` and `common2.c` source files each only need to be built once, even though the files are listed multiple times:

```
% scons
cc -c bar1.c -o bar1.o
cc -c bar2.c -o bar2.o
cc -c common1.c -o common1.o
cc -c common2.c -o common2.o
cc -o bar bar1.o bar2.o common1.o common2.o
cc -c foo.c -o foo.o
cc -o foo foo.o common1.o common2.o
```

If two or more programs share a lot of common source files, repeating the common files in the list for each program can be a maintenance problem when you need to change the list of common files. You can simplify this by creating a separate Python list to hold the common file names, and concatenating it with other lists using the Python + operator:

```
common = ['common1.c', 'common2.c']
foo_files = ['foo.c'] + common
bar_files = ['bar1.c', 'bar2.c'] + common
env = Environment()
env.Program('foo', foo_files)
env.Program('bar', bar_files)
```

This is functionally equivalent to the previous example.

# Chapter 3. Construction Environments

It is rare that all of the software in a large, complicated system needs to be built the same way. For example, different source files may need different options enabled on the command line, or different executable programs need to be linked with different libraries. `SCons` accomodates these different build requirements by allowing you to create and configure multiple `construction environments` that control how the software is built. Technically, a `construction environment` is an object that has a number of associated `construction variables`, each with a name and a value. (A `construction environment` also has an attached set of `Builder` methods, about which we'll learn more later.)

A `construction environment` is created by the `Environment` method which you have already seen. What you haven't seen, though, is that when you initialize a `construction environment`, you can set the values of the environment's `construction variables` to control how a program is built. For example:

```
env = Environment(CC = 'gcc',
                  CCFLAGS = '-O2')

env.Program('foo.c')
```

This example, rather than using the default, explicitly specifies use of the GNU C compiler `gcc`, and further specifies that the `-O2` (optimization level two) flag should be used when compiling the object file. So a run from this example would look like:

```
% scons
gcc -c -O2 foo.c -o foo.o
gcc -o foo foo.o
```

## Multiple `Construction Environments`

So far, all of our examples have created a single `construction environment` named `env`. `env`, however, is simply a Python variable name, and you can use any other variable name that you like. For example:

```
my_env = Environment(CC = 'gcc',
                     CCFLAGS = '-O2')

my_env.Program('foo.c')
```

This opens up the possibility of using multiple `construction environments`, each with a separate variable name. We can then use these separate `construction environments` to build different programs in different ways:

```
opt = Environment(CCFLAGS = '-O2')
dbg = Environment(CCFLAGS = '-g')

opt.Program('foo', 'foo.c')

dbg.Program('bar', 'bar.c')
```

```
% scons
cc -c -O2 bar.c -o bar.o
cc -o bar bar.o
cc -c -g foo.c -o foo.o
cc -o foo foo.o
```

We can even use multiple `construction environments` to build multiple versions of a single program. If you do this by simply trying to use the `Program` builder with both environments, though, like this:

```
opt = Environment(CCFLAGS = '-O2')
dbg = Environment(CCFLAGS = '-g')

opt.Program('foo', 'foo.c')

dbg.Program('foo', 'foo.c')
```

Then `SCons` generates the following error:

```
% scons
scons: *** Two different environments were specified for the same target:  foo.o
File "SConstruct", line 6, in ?
```

This is because the two `Program` calls have each implicitly told `SCons` to generate an object file named `foo.o`, one with a `CCFLAGS` value of `-O2` and one with a `CCFLAGS` value of `-g`. To avoid this problem, we must explicitly specify that each environment compile `foo.c` to a separately-named object file using the `Object` call, like so:

```
opt = Environment(CCFLAGS = '-O2')
dbg = Environment(CCFLAGS = '-g')

o = opt.Object('foo-opt', 'foo.c')
opt.Program(o)

d = dbg.Object('foo-dbg', 'foo.c')
dbg.Program(d)
```

Notice that each call to the `Object` builder returns a value, an internal `SCons` object that represents the file that will be built. We then use that object as input to the `Program` builder. This avoids having to specify explicitly the object file name in multiple places, and makes for a compact, readable `SConstruct` file. Our `SCons` output then looks like:

```
% scons
cc -c -g foo.c -o foo-dbg.o
cc -o foo-dbg foo-dbg.o
cc -c -O2 foo.c -o foo-opt.o
cc -o foo-opt foo-opt.o
```

## Copying `Construction Environments`

Sometimes you want more than one `construction environment` to share the same values for one or more variables. Rather than always having to repeat all of the common variables when you create each `construction environment`, you can use the `Copy` method to create a copy of a `construction environment`.

Like the `Environment` call that creates a `construction environment`, the `Copy` method takes `construction variable` assignments, which will override the values in the copied `construction environment`. For example, suppose we want to use `gcc` to create three versions of a program, one optimized, one debug, and one with neither. We could do this by creating a "base" `construction environment` that sets `CC` to `gcc`, and then creating two copies, one which sets `CCFLAGS` for optimization and the other with sets `CCFLAGS` for debugging:

```
env = Environment(CC = 'gcc')
opt = env.Copy(CCFLAGS = '-O2')
dbg = env.Copy(CCFLAGS = '-g')

e = opt.Object('foo', 'foo.c')

o = opt.Object('foo-opt', 'foo.c')
opt.Program(o)

d = dbg.Object('foo-dbg', 'foo.c')
dbg.Program(d)
```

Then our output would look like:

```
% scons
gcc -c foo.c -o foo.o
gcc -o foo foo.o
gcc -c -g foo.c -o foo-dbg.o
gcc -o foo-dbg foo-dbg.o
gcc -c -O2 foo.c -o foo-opt.o
gcc -o foo-opt foo-opt.o
```

## Fetching Values From a `Construction Environment`

You can fetch individual construction variables using the normal syntax for accessing individual named items in a Python dictionary:

```
env = Environment()
print "CC is:", env['CC']
```

This example `SConstruct` file doesn't build anything, but because it's actually a Python script, it will print the value of `CC` for us:

```
% scons
CC is: cc
```

A `construction environment`, however, is actually a Python object with associated methods, etc. If you want to have direct access to only the dictionary of construction variables, you can fetch this using the `Dictionary` method:

```
env = Environment(FOO = 'foo', BAR = 'bar')
dict = env.Dictionary()
for key, value in dict.items():
    print "key = %s, value = %s % (key, value)
```

This `SConstruct` file will print the dictionary items for us as follows:

```
% scons
key = FOO, value = foo
key = BAR, value = bar
```

## Modifying a `Construction Environment`

`SCons` provides various methods that support modifying existing values in a `construction environment`.

### Replacing Values in a `Construction Environment`

You can replace existing construction variable values using the `Replace` method:

```
env = Environment(CCFLAGS = '-DDEFINE1)
env.Program('foo.c')
env.Replace(CCFLAGS = '-DDEFINE2')
env.Program('bar.c')
```

The replaced value completely overwrites

% **scons**
gcc -DDEFINE2 -c bar.c -o bar.o
gcc -o bar bar.o
gcc -DDEFINE2 -c foo.c -o foo.o
gcc -o foo foo.o

### Appending to the End of Values in a `Construction Environment`

You can append a value to an existing construction variable using the `Append` method:

```
env = Environment(CCFLAGS = '-DMY_VALUE')
env.Append(CCFLAGS = ' -DLAST')
env.Program('foo.c')
```

% **scons**
gcc -DMY_VALUE -DLAST -c foo.c -o foo.o
gcc -o foo foo.o

### Appending to the Beginning of Values in a `Construction Environment`

You can append a value to the beginning an existing construction variable using the `Prepend` method:

```
env = Environment(CCFLAGS = '-DMY_VALUE')
env.Prepend(CCFLAGS = '-DFIRST ')
env.Program('foo.c')
```

% **scons**
gcc -DFIRST -DMY_VALUE -c foo.c -o foo.o
gcc -o foo foo.o

# Chapter 4. Building and Linking with Libraries

One of the more useful ways in which you can use multiple construction environments is to link programs with different sets of libraries.

## Building Libraries

You build your own libraries by specifying `Library` instead of `Program`:

```
env = Environment()
env.Library('foo', ['f1.c', 'f2.c', 'f3.c'])
```

`SCons` uses the appropriate library prefix and suffix for your system. So on POSIX or Linux systems, the above example would build as follows (although `ranlib` may not be called on all systems):

```
% scons
cc -c f1.c -o f1.o
cc -c f2.c -o f2.o
cc -c f3.c -o f3.o
ar r libfoo.a f1.o f2.o f3.o
ranlib libfoo.a
```

On a Windows system, a build of the above example would look like:

```
C:\>scons
cl /Fof1.obj f1.c
cl /Fof2.obj f2.c
cl /Fof3.obj f3.c
lib /nologo /OUT:foo.lib f1.obj f2.obj f3.obj
```

The rules for the target name of the library are similar to those for programs: if you don't explicitly specify a target library name, `SCons` will deduce one from the name of the first source file specified, and `SCons` will add an appropriate file prefix and suffix if you leave them off.

## Linking with Libraries

Usually, you build a library because you want to link it with one or more programs. You link libraries with a program by specifying the libraries in the `LIBS` construction variable, and by specifying the directory in which the library will be found in the `LIBPATH` construction variable:

```
env = Environment(LIBS = 'foo', LIBPATH = '.')
env.Library('foo', ['f1.c', 'f2.c', 'f3.c'])
env.Program('prog.c')
```

Notice, of course, that you don't need to specify a library prefix (like `lib`) or suffix (like `.a` or `.lib`). `SCons` uses the correct prefix or suffix for the current system.

On a POSIX or Linux system, a build of the above example would look like:

```
% scons
cc -c f1.c -o f1.o
cc -c f2.c -o f2.o
cc -c f3.c -o f3.o
```

```
ar r libfoo.a f1.o f2.o f3.o
ranlib libfoo.a
cc -c prog.c -o prog.o
cc -o prog -L. -lfoo prog.o
```

On a Windows system, a build of the above example would look like:

```
C:\>scons
cl /Fof1.obj f1.c
cl /Fof2.obj f2.c
cl /Fof3.obj f3.c
lib /nologo /OUT:foo.lib f1.obj f2.obj f3.obj
cl /Foprog.obj prog.c
link /OUT:prog.exe /LIBPATH:. foo.lib prog.obj
```

As usual, notice that `SCons` has taken care of constructing the correct command lines to link with the specified library on each system.

## Finding Libraries: the `LIBPATH` Construction Variable

By default, the linker will only look in certain system-defined directories for libraries. `SCons` knows how to look for libraries in directories that you specify with the `LIBPATH` construction variable. `LIBPATH` consists of a list of directory names, like so:

```
env = Environment(LIBS = 'm',
                  LIBPATH = ['/usr/lib', '/usr/local/lib'])
env.Program('prog.c')
```

Using a Python list is preferred because it's portable across systems. Alternatively, you could put all of the directory names in a single string, separated by the system-specific path separator character: a colon on POSIX systems:

```
LIBPATH = '/usr/lib:/usr/local/lib'
```

or a semi-colon on Windows systems:

```
LIBPATH = 'C:\lib;D:\lib'
```

When the linker is executed, `SCons` will create appropriate flags so that the linker will look for libraries in the same directories as `SCons`. So on a POSIX or Linux system, a build of the above example would look like:

```
% scons
cc -c prog.c -o prog.o
cc -o prog -L/usr/lib -L/usr/local/lib -lm prog.o
```

On a Windows system, a build of the above example would look like:

```
C:\>scons
cl /Foprog.obj prog.c
link /nologo /OUT:program.exe /LIBPATH:\usr\lib;\usr\local\lib m.lib prog.obj
```

Note again that `SCons` has taken care of the system-specific details of creating the right command-line options.

# Chapter 5. Dependencies

So far we've seen how `SCons` handles one-time builds. But the real point of a build tool like `SCons` is to rebuild only the necessary things when source files change--or, put another way, `SCons` should *not* waste time rebuilding things that have already been built. You can see this at work simply be re-invoking `SCons` after building our simple `hello` example:

```
% scons
cc -c hello.c -o hello.o
cc -o hello hello.o
% scons
%
```

The second time it is executed, `SCons` realizes that the `hello` program is up-to-date with respect to the current `hello.c` source file, and avoids rebuilding it. You can see this more clearly by naming the `hello` program explicitly on the command line:

```
% scons hello
cc -c hello.c -o hello.o
cc -o hello hello.o
% scons hello
scons: 'hello' is up to date.
%
```

Note that `SCons` reports `"...is up to date"` only for target files named explicitly on the command line, to avoid cluttering the output.

## Source File Signatures

The other side of avoiding unnecessary rebuilds is the fundamental build tool behavior of *rebuilding* things when a source file changes, so that the built software is up to date. `SCons` keeps track of this through a `signature` for each source file, and allows you to configure whether you want to use the source file contents or the modification time (timestamp) as the signature.

### MD5 Source File Signatures

By default, `SCons` keeps track of whether a source file has changed based on the file's contents, not the modification time. This means that you may be surprised by the default `SCons` behavior if you are used to the `Make` convention of forcing a rebuild by updating the file's modification time (using the `touch` command, for example):

```
% scons hello
cc -c hello.c -o hello.o
cc -o hello hello.o
% touch hello.c
% scons hello
scons: 'hello' is up to date.
%
```

Even though the file's modification time has changed, `SCons` realizes that the contents of the `hello.c` file have *not* changed, and therefore that the `hello` program need not be rebuilt. This avoids unnecessary rebuilds when, for example, someone rewrites the contents of a file without making a change. But if the contents of the file really do change, then `SCons` detects the change and rebuilds the program as required:

```
% scons hello
cc -c hello.c -o hello.o
cc -o hello hello.o
% edit hello.c
  [CHANGE THE CONTENTS OF hello.c]
% scons hello
cc -c hello.c -o hello.o
cc -o hello hello.o
%
```

Note that you can, if you wish, specify this default behavior (MD5 signatures) explicitly using the `SourceSignatures` function as follows:

```
env = Environment()
env.Program('hello.c')
SourceSignatures('MD5')
```

## Source File Time Stamps

If you prefer, you can configure `SCons` to use the modification time of source files, not the file contents, when deciding if something needs to be rebuilt. To do this, call the `SourceSignatures` function as follows:

```
env = Environment()
env.Program('hello.c')
SourceSignatures('timestamp')
```

This makes `SCons` act like `Make` when a file's modification time is updated (using the `touch` command, for example):

```
% scons hello
cc -c hello.c -o hello.o
cc -o hello hello.o
% touch hello.c
% scons hello
cc -c hello.c -o hello.o
cc -o hello hello.o
%
```

# Target File Signatures

As you've just seen, `SCons` uses signatures to decide whether a target file is up to date or must be rebuilt. When a target file depends on another target file, `SCons` allows you to separately configure how the signatures of an "intermediate" target file is used when deciding if a dependent target file must be rebuilt.

## Build Signatures

Modifying a source file will cause not only its direct target file to be rebuilt, but also the target file(s) that depend on that direct target file. In our example, changing the contents of the `hello.c` file causes the `hello.o` file to be rebuilt, which in turn causes the `hello` program to be rebuilt:

```
% scons hello
cc -c hello.c -o hello.o
cc -o hello hello.o
% edit hello.c
  [CHANGE THE CONTENTS OF hello.c]
% scons hello
cc -c hello.c -o hello.o
cc -o hello hello.o
%
```

What's not obvious, though, is that `SCons` internally handles the signature of the target file(s) (`hello.o` in the above example) differently from the signature of the source file (`hello.c`). By default, `SCons` tracks whether a target file must be rebuilt by using a `build signature` that consists of the combined signatures of all the files that go into making the target file. This is efficient because the accumulated signatures actually give `SCons` all of the information it needs to decide if the target file is out of date.

If you wish, you can specify this default behavior (build signatures) explicitly using the `TargetSignatures` function:

```
env = Environment()
env.Program('hello.c')
TargetSignatures('build')
```

## File Contents

Sometimes a source file can be changed in such a way that the contents of the rebuilt target file(s) will be exactly the same as the last time the file was built. If so, then any other target files that depend on such a built-but-not-changed target file actually need not be rebuilt. You can have `SCons` realize that a dependent target file need not be rebuilt in this situation using the `TargetSignatures` function as follows:

```
env = Environment()
env.Program('hello.c')
TargetSignatures('content')
```

So if, for example, a user were to only change a comment in a C file, then the rebuilt `hello.o` file would be exactly the same as the one previously built (assuming the compiler doesn't put any build-specific information in the object file). `SCons` would then realize that it would not need to rebuild the `hello` program as follows:

```
% scons hello
cc -c hello.c -o hello.o
cc -o hello hello.o
% edit hello.c
  [CHANGE A COMMENT IN hello.c]
% scons hello
cc -c hello.c -o hello.o
%
```

In essence, `SCons` has "short-circuited" any dependent builds when it realizes that a target file has been rebuilt to exactly the same file as the last build. So configured, `SCons` does take some extra processing time to scan the contents of the target (`hello.o`) file, but this may save time if the rebuild that was avoided would have been very time-consuming and expensive.

## Implicit Dependencies: The `CPPPATH` Construction Variable

Now suppose that our "Hello, World!" program actually has a `#include` line to include the `hello.h` file in the compilation:

```
#include "hello.h"
int
main()
{
    printf("Hello, %s!\n", string);
}
```

And, for completeness, the `hello.h` file looks like this:

```
#define string    "world"
```

In this case, we want `SCons` to recognize that, if the contents of the `hello.h` file change, the `hello` program must be recompiled. To do this, we need to modify the `SConstruct` file like so:

```
env = Environment(CPPPATH = '.')
hello = env.Program('hello.c')
```

The `CPPPATH` assignment in the `Environment` call tells `SCons` to look in the current directory (`'.'`) for any files included by C source files (`.c` or `.h` files). With this assignment in the `SConstruct` file:

```
% scons hello
cc -I. -c hello.c -o hello.o
cc -o hello hello.o
% scons hello
scons: `hello' is up to date.
% edit hello.h
  [CHANGE THE CONTENTS OF hello.h]
% scons hello
cc -I. -c hello.c -o hello.o
cc -o hello hello.o
%
```

First, notice that `SCons` added the `-I.` argument from the `CPPPATH` variable so that the compilation would find the `hello.h` file in the local directory.

Second, realize that `SCons` knows that the `hello` program must be rebuilt because it scans the contents of the `hello.c` file for the `#include` lines that indicate another file is being included in the compilation. `SCons` records these as *implicit dependencies* of the target file, Consequently, when the `hello.h` file changes, `SCons` realizes that the `hello.c` file includes it, and rebuilds the resulting `hello` program that depends on both the `hello.c` and `hello.h` files.

Like the `LIBPATH` variable, the `CPPPATH` variable may be a list of directories, or a string separated by the system-specific path separate character (':' on POSIX/Linux, ';' on Windows). Either way, `SCons` creates the right command-line options so that the followin example:

```
env = Environment(CPPPATH = ['include', '/home/project/inc'])
hello = env.Program('hello.c')
```

Will look like this on POSIX or Linux:

```
% scons hello
cc -Iinclude -I/home/project/inc -c hello.c -o hello.o
cc -o hello hello.o
```

And like this on Windows:

```
% scons hello
cl /Iinclude /I\home\project\inc /Fohello.obj hello.c
link /OUT:hello.exe hello.obj
```

## Caching Implicit Dependencies

Scanning each file for `#include` lines does take some extra processing time. When you're doing a full build of a large system, the scanning time is usually a very small percentage of the overall time spent on the build. You're most likely to notice the scanning time, however, when you *rebuild* all or part of a large system: `SCons` will likely take some extra time to "think about" what must be built before it issues the first build command (or decides that everything is up to date and nothing must be rebuilt).

In practice, having `SCons` scan files saves time relative to the amount of potential time lost to tracking down subtle problems introduced by incorrect dependencies. Nevertheless, the "waiting time" while `SCons` scans files can annoy individual developers waiting for their builds to finish. Consequently, `SCons` lets you cache the implicit dependencies that its scanners find, for use by later builds. You do this either by specifying the `--implicit-cache` option on the command line:

```
% scons --implicit-cache hello
cc -c hello.c -o hello.o
cc -o hello hello.o
% scons hello
scons: 'hello' is up to date.
```

Or by setting the `implicit_cache` option in an `SConscript` file:

```
SetOption('implicit_cache', 1)
```

`SCons` does not cache implicit dependencies like this by default because XXX

XXX

### The `--implicit-deps-changed` Option

XXX

### The `--implicit-deps-unchanged` Option

XXX

## The `Ignore` Method

Sometimes it makes sense to not rebuild a program, even if a dependency file changes. In this case, you would tell `SCons` specifically to ignore a dependency as follows:

```
env = Environment()
hello = env.Program('hello.c')
env.Ignore(hello, 'hello.h')
```

```
% scons hello
cc -c hello.c -o hello.o
cc -o hello hello.o
% scons hello
scons: 'hello' is up to date.
% edit hello.h
  [CHANGE THE CONTENTS OF hello.h]
% scons hello
scons: 'hello' is up to date.
```

Now, the above example is a little contrived, because it's hard to imagine a real-world situation where you wouldn't to rebuild `hello` if the `hello.h` file changed. A more realistic example might be if the `hello` program is being built in a directory that is shared between multiple systems that have different copies of the `stdio.h` include file. In that case, `SCons` would notice the differences between the different systems' copies of `stdio.h` and would rebuild `hello` each time you change systems. You could avoid these rebuilds as follows:

```
env = Environment()
hello = env.Program('hello.c')
env.Ignore(hello, '/usr/include/stdio.h')
```

## The `Depends` Method

On the other hand, sometimes a file depends on another file that has no `SCons` scanner will detect. For this situation, `SCons` allows you to specific explicitly that one file depends on another file, and must be rebuilt whenever that file changes. This is specified using the `Depends` method:

```
env = Environment()
hello = env.Program('hello.c')
env.Depends(hello, 'other_file')
```

```
% scons hello
cc -c hello.c -o hello.o
cc -o hello hello.o
% scons hello
scons: 'hello' is up to date.
% edit other_file
  [CHANGE THE CONTENTS OF other_file]
% scons hello
cc -c hello.c -o hello.o
cc -o hello hello.o
```

# Chapter 6. Default Targets

As mentioned previously, `SCons` will build every target in or below the current directory by default--that is, when you don't explicitly specify one or more targets on the command line. Sometimes, however, you may want to specify explicitly that only certain programs should be built by default. You do this with the `Default` function:

```
env = Environment()
hello = env.Program('hello.c')
env.Program('goodbye.c')
Default(hello)
```

This `SConstruct` file knows how to build two programs, `hello` and `goodbye`, but only builds the `hello` program by default:

```
% scons
cc -c hello.c -o hello.o
cc -o hello hello.o
% scons
% scons goodbye
cc -c goodbye.c -o goodbye.o
cc -o goodbye goodbye.o
%
```

Note that, even when you use the `Default` function in your `SConstruct` file, you can still explicitly specify the current directory (`.`) on the command line to tell `SCons` to build everything in (or below) the current directory:

```
% scons .
cc -c goodbye.c -o goodbye.o
cc -o goodbye goodbye.o
cc -c hello.c -o hello.o
cc -o hello hello.o
%
```

You can also call the `Default` function more than once, in which case each call adds to the list of targets to be built by default:

```
env = Environment()
prog1 = env.Program('prog1.c')
Default(prog1)
prog2 = env.Program('prog2.c')
prog3 = env.Program('prog3.c')
Default(prog3)
```

Or you can specify more than one target in a single call to the `Default` function:

```
env = Environment()
prog1 = env.Program('prog1.c')
prog2 = env.Program('prog2.c')
prog3 = env.Program('prog3.c')
Default(prog1, prog3)
```

Either of these last two examples will build only the `prog1` and `prog3` programs by default:

```
% scons
```

```
cc -c prog1.c -o prog1.o
cc -o prog1 prog1.o
cc -c prog3.c -o prog3.o
cc -o prog3 prog3.o
% scons .
cc -c prog2.c -o prog2.o
cc -o prog2 prog2.o
%
```

Lastly, if for some reason you don't want any targets built by default, you can use the Python `None` variable:

```
env = Environment()
prog1 = env.Program('prog1.c')
prog2 = env.Program('prog2.c')
Default(None)
```

Which would produce build output like:

```
% scons
scons: *** No targets specified and no Default() targets found.  Stop.
% scons .
cc -c prog1.c -o prog1.o
cc -o prog1 prog1.o
cc -c prog2.c -o prog2.o
cc -o prog2 prog2.o
%
```

# Chapter 7. Providing Build Help

It's often very useful to be able to give users some help that describes the specific targets, build options, etc., that can be used for the build. `SCons` provides the `Help` function to allow you to specify this help text:

```
Help("""
Type: 'scons program' to build the production program,
      'scons debug' to build the debug version.
""")
```

(Note the above use of the Python triple-quote syntax, which comes in very handy for specifying multi-line strings like help text.)

When the `SConstruct` or `SConscript` files contain such a call to the `Help` function, the specified help text will be displayed in response to the `SCons -h` option:

```
% scons -h
Type: 'scons program' to build the production program,
     'scons debug' to build the debug version.
```

If there is no `Help` text in the `SConstruct` or `SConscript` files, `SCons` will revert to displaying its standard list that describes the `SCons` command-line options. This list is also always displayed whenever the `-H` option is used.

# Chapter 8. Installing Files in Other Directories

Once a program is built, it is often appropriate to install it in another directory for public use. You use the `Install` method to arrange for a program, or any other file, to be copied into a destination directory:

```
env = Environment()
hello = env.Program('hello.c')
env.Install('/usr/bin', hello)
```

Note, however, that installing a file is still considered a type of file "build." This is important when you remember that the default behavior of `SCons` is to build files in or below the current directory. If, as in the example above, you are installing files in a directory outside of the top-level `SConstruct` file's directory tree, you must specify that directory (or a higher directory, such as `/`) for it to install anything there:

```
% scons
cc -c hello.c -o hello.o
cc -o hello hello.o
% scons /usr/bin
Install file: "hello" as "/usr/bin/hello"
```

It can, however, be cumbersome to remember (and type) the specific destination directory in which the program (or any other file) should be installed. This is an area where the `Alias` function comes in handy, allowing you, for example, to create a pseudo-target named `install` that can expand to the specified destination directory:

```
env = Environment()
hello = env.Program('hello.c')
env.Install('/usr/bin', hello)
env.Alias('install', '/usr/bin')
```

This then yields the more natural ability to install the program in its destination as follows:

```
% scons install
Install file: "hello" as "/usr/bin/hello"
```

## Installing Multiple Files in a Directory

You can install multiple files into a directory simply by calling the `Install` function multiple times:

```
env = Environment()
hello = env.Program('hello.c')
goodbye = env.Program('goodbye.c')
env.Install('/usr/bin', hello)
env.Install('/usr/bin', goodbye])
env.Alias('install', '/usr/bin')
```

Or, more succinctly, listing the multiple input files in a list (just like you can do with any other builder):

```
env = Environment()
hello = env.Program('hello.c')
goodbye = env.Program('goodbye.c')
env.Install('/usr/bin', [hello, goodbye])
```

```
env.Alias('install', '/usr/bin')
```

Either of these two examples yields:

```
% scons install
cc -c goodbye.c -o goodbye.o
cc -o goodbye goodbye.o
cc -c hello.c -o hello.o
cc -o hello hello.o
Install file: "goodbye" as "/usr/bin/goodbye"
Install file: "hello" as "/usr/bin/hello"
```

## Installing a File Under a Different Name

The `Install` method preserves the name of the file when it is copied into the destination directory. If you need to change the name of the file when you copy it, use the `InstallAs` function:

```
env = Environment()
hello = env.Program('hello.c')
env.InstallAs('/usr/bin/hello-new', hello)
env.Alias('install', '/usr/bin')
```

This installs the `hello` program with the name `hello-new` as follows:

```
% scons install
cc -c hello.c -o hello.o
cc -o hello hello.o
Install file: "hello" as "/usr/bin/hello-new"
```

## Installing Multiple Files Under Different Names

Lastly, if you have multiple files that all need to be installed with different file names, you can either call the `InstallAs` function multiple times, or as a shorthand, you can supply same-length lists for the both the target and source arguments:

```
env = Environment()
hello = env.Program('hello.c')
goodbye = env.Program('goodbye.c')
env.InstallAs(['/usr/bin/hello-new',
               '/usr/bin/goodbye-new',
              [hello, goodbye])
```

In this case, the `InstallAs` function loops through both lists simultaneously, and copies each source file into its corresponding target file name:

```
% scons install
cc -c goodbye.c -o goodbye.o
cc -o goodbye goodbye.o
cc -c hello.c -o hello.o
cc -o hello hello.o
Install file: "goodbye" as "/usr/bin/goodbye-new"
Install file: "hello" as "/usr/bin/hello-new"
```

# Chapter 9. Preventing Removal of Targets

By default, `SCons` removes targets before building them. Sometimes, however, this is not what you want. For example, you may want to update a library incrementally, not by having it deleted and then rebuilt from all of the constituent object files. In such cases, you can use the `Precious` method to prevent `SCons` from removing the target before it is built:

```
env = Environment(XXX NEED LIBRARY FLAGS
                  LIBFLAGS = '-r')
lib = env.Library('foo', ['f1.c', 'f2.c', 'f3.c'])
env.Precious(lib)
```

XXX:

```
% scons
XXX ANY INPUT HERE?
```

`SCons` will still delete files marked as `Precious` when the `-c` option is used.

# Chapter 10. Hierarchical Builds

The source code for large software projects rarely stays in a single directory, but is nearly always divided into a hierarchy of directories. Organizing a large software build using `SCons` involves creating a hierarchy of build scripts using the `SConscript` function.

## `SConscript` Files

As we've already seen, the build script at the top of the tree is called `SConstruct`. The top-level `SConstruct` file can use the `SConscript` function to include other subsidiary scripts in the build. These subsidiary scripts can, in turn, use the `SConscript` function to include still other scripts in the build. By convention, these subsidiary scripts are usually named `SConscript`. For example, a top-level `SConstruct` file might arrange for four subsidiary scripts to be included in the build as follows:

```
SConscript(['drivers/display/SConscript',
            'drivers/mouse/SConscript',
            'parser/SConscript',
            'utilities/SConscript'])
```

In this case, the `SConstruct` file lists all of the `SConscript` files in the build explicitly. (Note, however, that not every directory in the tree necessarily has an `SConscript` file.) Alternatively, the `drivers` subdirectory might contain an intermediate `SConscript` file, in which case the `SConscript` call in the top-level `SConstruct` file would look like:

```
SConscript(['drivers/SConscript',
            'parser/SConscript',
            'utilities/SConscript'])
```

And the subsidiary `SConscript` file in the `drivers` subdirectory would look like:

```
SConscript(['display/SConscript',
            'mouse/SConscript'])
```

Whether you list all of the `SConscript` files in the top-level `SConstruct` file, or place a subsidiary `SConscript` file in intervening directories, or use some mix of the two schemes, is up to you and the needs of your software.

## Path Names Are Relative to the `SConscript` Directory

Subsidiary `SConscript` files make it easy to create a build hierarchy because all of the file and directory names in a subsidiary `SConscript` files are interpreted relative to the directory in which the `SConscript` file lives. Typically, this allows the `SConscript` file containing the instructions to build a target file to live in the same directory as the source files from which the target will be built, making it easy to update how the software is built whenever files are added or deleted (or other changes are made).

For example, suppose we want to build two programs `prog1` and `prog2` in two separate directories with the same names as the programs. One typical way to do this would be with a top-level `SConstruct` file like this:

```
SConscript(['prog1/SConscript',
            'prog2/SConscript'])
```

And subsidiary `SConscript` files that look like this:

```
env = Environment()
env.Program('prog1', ['main.c', 'foo1.c', 'foo2.c'])
```

And this:

```
env = Environment()
env.Program('prog2', ['main.c', 'bar1.c', 'bar2.c'])
```

Then, when we run `SCons` in the top-level directory, our build looks like:

```
% scons
cc -c prog1/foo1.c -o prog1/foo1.o
cc -c prog1/foo2.c -o prog1/foo2.o
cc -c prog1/main.c -o prog1/main.o
cc -o prog1/prog1 prog1/main.o prog1/foo1.o prog1/foo2.o
cc -c prog2/bar1.c -o prog2/bar1.o
cc -c prog2/bar2.c -o prog2/bar2.o
cc -c prog2/main.c -o prog2/main.o
cc -o prog2/prog2 prog2/main.o prog2/bar1.o prog2/bar2.o
```

Notice the following: First, you can have files with the same names in multiple directories, like main.c in the above example. Second, unlike standard recursive use of `Make`, `SCons` stays in the top-level directory and issues commands

## Top-Level Path Names in Subsidiary `SConscript` Files

If you need to use a file from another directory, it's sometimes more convenient to specify the path to a file in another directory from the top-level `SConstruct` directory, even when you're using that file in a subsidiary `SConscript` file in a subdirectory. You can tell `SCons` to interpret a path name as relative to the top-level `SConstruct` directory, not the local directory of the `SConscript` file, by appending a `#` (hash mark) to the beginning of the path name:

```
env = Environment()
env.Program('prog', ['main.c', '#lib/foo1.c', 'foo2.c'])
```

In this example, the `lib` directory is directly underneath the top-level `SConstruct` directory. If the above `SConscript` file is in a subdirectory named `src/prog`, the output would look like:

```
% scons
cc -c lib/foo1.c -o lib/foo1.o
cc -c src/prog/foo2.c -o src/prog/foo2.o
cc -c src/prog/main.c -o src/prog/main.o
cc -o src/prog/prog prog/main.o lib/foo1.o prog/foo2.o
```

(Notice that the `lib/foo1.o` object file is built in the same directory as its source file. See section XXX, below, for information about how to build the object file in a different subdirectory.)

## Absolute Path Names

Of course, you can always specify an absolute path name for a file--for example:

```
env = Environment()
env.Program('prog', ['main.c', '/usr/joe/lib/foo1.c', 'foo2.c'])
```

Which, when executed, would yield:

```
% scons
cc -c /usr/joe/lib/foo1.c -o /usr/joe/lib/foo1.o
cc -c src/prog/foo2.c -o src/prog/foo2.o
cc -c src/prog/main.c -o src/prog/main.o
cc -o src/prog/prog prog/main.o /usr/joe/lib/foo1.o prog/foo2.o
```

(As was the case with top-relative path names, notice that the `/usr/joe/lib/foo1.o` object file is built in the same directory as its source file. See section XXX, below, for information about how to build the object file in a different subdirectory.)

## Sharing Environments (and Other Variables) Between `SConscript` Files

In the previous example, each of the subsidiary `SConscript` files created its own construction environment by calling `Environment` separately. This obviously works fine, but if each program must be built with the same construction variables, it's cumbersome and error-prone to initialize separate construction environments in the same way over and over in each subsidiary `SConscript` file.

`SCons` supports the ability to *export* variables from a parent `SConscript` file to its subsidiary `SConscript` files, which allows you to share common initialized values throughout your build hierarchy.

### Exporting Variables

There are two ways to export a variable, such as a construction environment, from one `SConscript` file, so that it may be used by other `SConscript` files. First, you can call the `Export` function with a list of variables, or a string white-space separated variable names. Each call to `Export` adds one or more variables to a global list of variables that are available for import by other `SConscript` files.

```
env = Environment()
Export('env')
```

XXX

```
env = Environment()
debug = ARGUMENTS['debug']
Export('env', 'debug')
```

XXX

```
Export('env debug')
```

Second, you can specify a list of variables to exported as a second argument to the `SConscript` function call:

```
SConscript('src/SConscript', 'env')
```

Or as the `exports` keyword argument:

```
SConscript('src/SConscript', exports='env')
```

These calls export the specified variables to only the listed `SConscript` files. You may, however, specify more than one `SConscript` file in a list:

```
SConscript(['src1/SConscript',
            'src2/SConscript'], exports='env')
```

This is functionally equivalent to calling the `SConscript` function multiple times with the same `exports` argument, one per `SConscript` file.

## Importing Variables

XXX

```
Import('env')
env.Program('prog', ['prog.c'])
```

XXX

```
Import('env', 'debug')
env = env.Copy(DEBUG = debug)
env.Program('prog', ['prog.c'])
```

Which is exactly equivalent to:

```
Import('env debug')
env = env.Copy(DEBUG = debug)
env.Program('prog', ['prog.c'])
```

XXX

## Returning Values From an `SConscript` File

XXX

```
obj = env.Object('foo.c')
Return('obj')
```

XXX

```
objs = []
for subdir in ['foo', 'bar']:
    o = SConscript('%s/SConscript' % subdir)
    objs.append(o)
env.Library('prog', objs)
```

XXX

# Chapter 11. Separating Source and Build Directories

It's often useful to keep any built files completely separate from the source files. This is usually done by creating one or more separate *build directories* that are used to hold the built objects files, libraries, and executable programs, etc. for a specific flavor of build. `SCons` provides two ways to do this, one through the `SConscript` function that we've already seen, and the second through a more flexible `BuildDir` function.

## Specifying a Build Directory as Part of an `SConscript` Call

The most straightforward way to establish a build directory uses the fact that the usual way to set up a build hierarcy is to have an `SConscript` file in the source subdirectory. If you then pass a `build_dir` argument to the `SConscript` function call:

```
SConscript('src/SConscript', build_dir='build')
```

`SCons` will then build all of the files in the `build` subdirectory:

```
% ls -1 src
SConscript
hello.c
% scons
cc -c build/hello.c -o build/hello.o
cc -o build/hello build/hello.o
% ls -1 build
hello
hello.c
hello.o
```

But wait a minute--what's going on here? `SCons` created the object file `build/hello.o` in the `build` subdirectory, as expected. But even though our `hello.c` file lives in the `src` subdirectory, `SCons` has actually compiled a `build/hello.c` file to create the object file.

What's happened is that `SCons` has *duplicated* the `hello.c` file from the `src` subdirectory to the `build` subdirectory, and built the program from there. The next section explains why `SCons` does this.

## Why `SCons` Duplicates Source Files in a Build Directory

`SCons` duplicates source files in build directories because it's the most straightforward way to guarantee a correct build *regardless of include-file directory paths*, and the `SCons` philosophy is to, by default, guarantee a correct build in all cases. Here is an example of an end case where duplicating source files in a build directory is necessary for a correct build:

XXX

```
env = Environmnet()
```

XXX

```
% scons
cc -c build/hello.c -o build/hello.o
cc -o build/hello build/hello.o
```

## Telling `SCons` to Not Duplicate Source Files in the Build Directory

In most cases, however, having `SCons` place its target files in a build subdirectory *without* duplicating the source files works just fine. You can disable the default `SCons` behavior by specifying `duplicate=0` when you call the `SConscript` function:

```
SConscript('src/SConscript', build_dir='build', duplicate=0)
```

When this flag is specified, `SCons` uses the build directory like most people expect--that is, the output files are placed in the build directory while the source files stay in the source directory:

```
% ls -1 src
SConscript
hello.c
% scons
cc -c src/hello.c -o build/hello.o
cc -o build/hello build/hello.o
% ls -1 build
hello
hello.o
```

## The `BuildDir` Function

Use the `BuildDir` function to establish that target files should be built in a separate directory from the source files:

```
BuildDir('build', 'src')
env = Environment()
env.Program('build/hello.c')
```

Note that when you're not using an `SConscript` file in the `src` subdirectory, you must actually specify that the program must be built from the `build/hello.c` file that `SCons` will duplicate in the `build` subdirectory.

XXX

When using the `BuildDir` function directly, `SCons` still duplicates the source files in the build directory by default:

```
% ls src
hello.c
% scons
cc -c build/hello.c -o build/hello.o
cc -o build/hello build/hello.o
% ls -1 build
hello
hello.c
hello.o
```

You can specify the same `duplicate=0` argument that you can specify for an `SConscript` call:

```
BuildDir('build', 'src', duplicate=0)
env = Environment()
env.Program('build/hello.c')
```

In which case `SCons` will disable duplication of the source files:

```
% ls src
hello.c
% scons
cc -c src/hello.c -o build/hello.o
cc -o build/hello build/hello.o
% ls -1 build
hello
hello.o
```

## Using `BuildDir` With an `SConscript` File

Even when using the `BuildDir` function, it's much more natural to use it with a subsidiary `SConscript` file. For example, if the `src/SConscript` looks like this:

```
env = Environment()
env.Program('hello.c')
```

Then our `SConscript` file could look like:

```
BuildDir('build', 'src')
SConscript('build/SConscript')
```

Yielding the following output:

```
% ls -1 src
SConscript
hello.c
% scons
cc -c build/hello.c -o build/hello.o
cc -o build/hello build/hello.o
% ls -1 build
hello
hello.c
hello.o
```

Notice that this is completely equivalent to the use of `SConscript` that we learned about in the previous section.

## Why You'd Want to Call `BuildDir` Instead of `SConscript`

XXX

# Chapter 12. Variant Builds

The `BuildDir` function now gives us everything we need to show how easy it is to create variant builds using `SCons`. Suppose, for example, that we want to build a program for both Windows and Linux platforms, but that we want to build it in a shared directory with separate side-by-side build directories for the Windows and Linux versions of the program.

```
platform = ARGUMENT.get('OS', Platform())

include = "#export/$PLATFORM/include"
lib = "#export/$PLATFORM/lib"
bin = "#export/$PLATFORM/bin"

env = Environment(PLATFORM = platform,
                  CPPPATH = [include],
                  LIB = lib,
                  LIBS = '-lworld')

Export('env')

SConscript('src/SConscript', build_dir='build/$PLATFORM')

#
#BuildDir("#build/$PLATFORM", 'src')
#SConscript("build/$PLATFORM/hello/SConscript")
#SConscript("build/$PLATFORM/world/SConscript")
```

This SConstruct file, when run on a Linux system, yields:

```
$ scons OS=linux
Install build/linux/world/world.h as export/linux/include/world.h
cc -Iexport/linux/include -c build/linux/hello/hello.c -o build/linux/hello/hello.o
cc -Iexport/linux/include -c build/linux/world/world.c -o build/linux/world/world.o
ar r build/linux/world/libworld.a build/linux/world/world.o
ar: creating build/linux/world/libworld.a
ranlib build/linux/world/libworld.a
Install build/linux/world/libworld.a as export/linux/lib/libworld.a
cc -o build/linux/hello/hello build/linux/hello/hello.o -Lexport/linux/lib -lworld
Install build/linux/hello/hello as export/linux/bin/hello
```

The same SConstruct file on Windows would build:

```
C:\test\>scons OS=linux
Install build\linux\world\world.h as export\linux\include\world.h
cl /Iexport\linux\include /Fobuild\linux\hello\hello.obj build\linux\hello\hello.c
cl /Iexport\linux\include /Fobuild\linux\world\world.obj build\linux\world\world.c
XXX
ar r build\linux\world\world.lib build\linux\world\world.obj
Install build\linux\world\world.lib as export\linux\lib\libworld.a
link /Fobuild\linux\hello\hello.exe build\linux\hello\hello.obj -Lexport\linux\lib world.lib
Install build\linux\hello\hello.exe as export\linux\bin\hello.exe
```

```
env = Environment(OS = )
for os in ['newell', 'post']:
    SConscript('src/SConscript', build_dir='build/' + os)
```

```
% scons
```

# Chapter 13. Built-In Builders

SCons provides the ability to build a lot of different types of files right "out of the box." So far, we've been using SCons' ability to build programs, objects and libraries to illustrate much of the underlying functionality of SCons This section will describe all of the different types of files that you can build with SCons, and the built-in Builder objects used to build them.

## Programs: the Program Builder

As we've seen, the Program Builder is used to build an executable program. The source argument is one or more source-code files or object files, and the target argument is the name of the executable program name to be created. For example:

```
env = Environment()
env.Program('prog', 'file1.o')
```

Will create the prog executable on a POSIX system, the prog.exe executable on a Windows system.

The target file's prefix and suffix may be omitted, and the values from the $PROGPREFIX and $PROGSUFFIX construction variables will be appended appropriately. For example:

```
env = Environment(PROGPREFIX='my', PROGSUFFIX='.xxx')
env.Program('prog', ['file1.o', 'file2.o'])
```

Will create a program named myprog.xxx regardless of the system on which it is run.

If you omit the target, the base of the first input file name specified because the base of the target program created. For example:

```
env = Environment()
env.Program(['hello.c', 'goodbye.c'])
```

Will create the hello executable on a POSIX system, the hello.exe executable on a Windows system.

## Object-File Builders

SCons provides separate Builder objects to create both static and shared object files.

### The StaticObject Builder

XXX

```
XXX
```

XXX

### The `SharedObject` Builder

XXX

```
XXX
```

XXX

### The `Object` Builder

XXX

```
XXX
```

Creates a static object file.

## Library Builders

`SCons` provides separate Builder objects to create both static and shared libraries.

### The `StaticLibrary` Builder

XXX

```
XXX
```

XXX

### The `SharedLibrary` Builder

XXX

### The `Library` Builder

XXX

```
XXX
```

XXX

Creates a static library file.

## Pre-Compiled Headers: the `PCH` Builder

XXX

## Microsoft Visual C++ Resource Files: the `RES` Builder

XXX

## Source Files

By default `SCons` supports two Builder objects that know how to build source files from other input files.

### The `CFile` Builder

XXX

```
XXX
```

XXX

### The `CXXFile` Builder

XXX

```
XXX
```

XXX

## Documents

`SCons` provides a number of Builder objects for creating different types of documents.

### The `DVI` Builder

XXX

```
XXX
```

XXX

### The `PDF` Builder

XXX

### The `PostScript` Builder

XXX

```
XXX
```

XXX

## Archives

SCons provides Builder objects for creating two different types of archive files.

### The Tar Builder

The Tar Builder object uses the tar utility to create archives of files and/or directory trees:

```
env = Environment()
env.Tar('out1.tar', ['file1', 'file2'])
env.Tar('out2', 'directory')
```

% **scons .**
tar -c -f out1.tar file1 file2
tar -c -f out2.tar directory

One common requirement when creating a tar archive is to create a compressed archive using the -z option. This is easily handled by specifying the value of the TARFLAGS variable when you create the construction environment. Note, however, that the -c used to to instruct tar to create the archive is part of the default value of TARFLAGS, so you need to set it both options:

```
env = Environment(TARFLAGS = '-c -z')
env.Tar('out.tar.gz', 'directory')
```

% **scons .**
tar -c -z -f out.tar.gz directory

you may also wish to set the value of the TARSUFFIX construction variable to your desired suffix for compress tar archives, so that SCons can append it to the target file name without your having to specify it explicitly:

```
env = Environment(TARFLAGS = '-c -z',
                  TARSUFFIX = '.tgz')
env.Tar('out', 'directory')
```

% **scons .**
tar -c -z -f out.tgz directory

### The Zip Builder

The Zip Builder object creates archives of files and/or directory trees in the ZIP file format. Python versions 1.6 or later contain an internal zipfile module that SCons will use. In this case, given the following SConstruct file:

```
env = Environment()
env.Zip('out', ['file1', 'file2'])
```

Your output will reflect the fact that an internal Python function is being used to create the output ZIP archive:

```
% scons .
zip("out.zip", ["file1", "file2"])
```

If you're using Python version 1.5.2 to run `SCons`, then `SCons` will try to use an external `zip` program as follows:

```
% scons .
zip /home/my/project/zip.out file1 file2
```

## Java

`SCons` provides Builder objects for creating various types of Java output files.

### Building Class Files: the `Java` Builder

The `Java` builder takes one or more input `.java` files and turns them into one or more `.class` files Unlike most builders, however, the `Java` builder takes target and source *directories*, not files, as input.

```
env = Environment()
env.Java(target = 'classes', source = 'src')
```

The `Java` builder will then search the specified source directory tree for all `.java` files, and pass any out-of-date

XXX

### The `Jar` Builder

The `Jar` builder object XXX

```
env = Environment()
env.Java(target = 'classes', source = 'src')
env.Jar(target = '', source = 'classes')
```

XXX

### Building C header and stub files: the `JavaH` Builder

XXX

```
XXX
```

XXX

## Building RMI stub and skeleton class files: the `RMIC` Builder

XXX

```
XXX
```

XXX

# Chapter 14. Writing Your Own Builders

Although `SCons` provides many useful methods for building common software products: programs, libraries, documents. you frequently want to be able to build some other type of file not supported directly by `SCons` Fortunately, `SCons` makes it very easy to define your own `Builder` objects for any custom file types you want to build. (In fact, the `SCons` interfaces for creating `Builder` objects are flexible enough and easy enough to use that all of the the `SCons` built-in `Builder` objects are created the mechanisms described in this section.)

## Writing Builders That Execute External Commands

The simplest `Builder` to create is one that executes an external command. For example, if we want to build an output file by running the contents of the input file through a command named `foobuild`, creating that `Builder` might look like:

```
bld = Builder(action = 'foobuild < $TARGET > $SOURCE')
```

All the above line does is create a free-standing `Builder` object. The next section will show us how to actually use it.

## Attaching a Builder to a `Construction Environment`

A `Builder` object isn't useful until it's attached to a `construction environment` so that we can call it to arrange for files to be built. This is done through the `BUILDERS` `construction variable` in an environment. The `BUILDERS` variable is a Python dictionary that maps the names by which you want to call various `Builder` objects to the objects themselves. For example, if we want to call the `Builder` we just defined by the name `Foo`, our `SConstruct` file might look like:

```
bld = Builder(action = 'foobuild < $TARGET > $SOURCE')
env = Environment(BUILDERS = {'Foo' : bld})
```

With the `Builder` so attached to our `construction environment` we can now actually call it like so:

```
env.Foo('file.foo', 'file.input')
```

Then when we run `SCons` it looks like:

```
% scons
foobuild < file.input > file.foo
```

Note, however, that the default `BUILDERS` variable in a `construction environment` comes with a default set of `Builder` objects already defined: `Program`, `Library`, etc. And when we explicitly set the `BUILDERS` variable when we create the `construction environment`, the default `Builder`s are no longer part of the environment:

```
bld = Builder(action = 'foobuild < $TARGET > $SOURCE')
env = Environment(BUILDERS = {'Foo' : bld})
env.Foo('file.foo', 'file.input')
env.Program('hello.c')
```

```
% scons
scons: Reading SConscript files ...
```

```
other errors
Traceback (most recent call last):
  File "/usr/lib/scons/SCons/Script/__init__.py", line 901, in main
    _main()
  File "/usr/lib/scons/SCons/Script/__init__.py", line 762, in _main
    SCons.Script.SConscript.SConscript(script)
  File "/usr/lib/scons/SCons/Script/SConscript.py", line 207, in SConscript
    exec _file_ in stack[-1].globals
  File "SConstruct", line 4, in ?
    env.Program('hello.c')
scons: Environment instance has no attribute 'Program'
```

To be able use both our own defined `Builder` objects and the default `Builder` objects in the same `construction environment`, you can either add to the `BUILDERS` variable using the `Append` function:

```
env = Environment()
bld = Builder(action = 'foobuild < $TARGET > $SOURCE')
env.Append(BUILDERS = {'Foo' : bld})
env.Foo('file.foo', 'file.input')
env.Program('hello.c')
```

Or you can explicitly set the appropriately-named key in the `BUILDERS` dictionary:

```
env = Environment()
bld = Builder(action = 'foobuild < $TARGET > $SOURCE')
env['BUILDERS']['Foo'] = bld
env.Foo('file.foo', 'file.input')
env.Program('hello.c')
```

Either way, the same `construction environment` can then use both the newly-defined `Foo` `Builder` and the default `Program` `Builder`:

```
% scons
foobuild < file.input > file.foo
cc -c hello.c -o hello.o
cc -o hello hello.o
```

## Letting `SCons` Handle The File Suffixes

By supplying additional information when you create a `Builder`, you can let `SCons` add appropriate file suffixes to the target and/or the source file. For example, rather than having to specify explicitly that you want the `Foo` `Builder` to build the `file.foo` target file from the `file.input` source file, you can give the `.foo` and `.input` suffixes to the `Builder`, making for more compact and readable calls to the `Foo` `Builder`:

```
bld = Builder(action = 'foobuild < $TARGET > $SOURCE',
              suffix = '.foo',
              src_suffix = '.input')
env = Environment(BUILDERS = {'Foo' : bld})
env.Foo('file1')
env.Foo('file2')
```

```
% scons
foobuild < file1.input > file1.foo
```

foobuild < file2.input > file2.foo

You can also supply a `prefix` keyword argument if it's appropriate to have `SCons` append a prefix to the beginning of target file names.

## Builders That Execute Python Functions

In `SCons`, you don't have to call an external command to build a file. You can, instead, define a Python function that a `Builder` object can invoke to build your target file (or files). Such a `builder function` definition looks like:

```
def build_function(target, source, env):
    # XXX
    return None
```

The arguments of a `builder function` are:

target

A list of Node objects representing the target or targets to be built by this builder function. The file names of these target(s) may be extracted using the Python `str` funcion.

source

A list of Node objects representing the sources to be used by this builder function to build the targets. The file names of these source(s) may be extracted using the Python `str` funcion.

env

The `construction environment` used for building the target(s). The builder function may use any of the environment's construction variables in any way to affect how it builds the targets.

The builder function must return a `0` or `None` value if the target(s) are built successfully. The builder function may raise an exception or return any non-zero value to indicate that the build is unsuccessful,

Once you've defined the Python function that will build your target file, defining a `Builder` object for it is as simple as specifying the name of the function, instead of an external command, as the `Builder`'s `action` argument:

```
def build_function(target, source, env):
    # XXX
    return None
bld = Builder(action = build_function,
              suffix = '.foo',
              src_suffix = '.input')
env = Environment(BUILDERS = {'Foo' : bld})
env.Foo('file')
```

And notice that the output changes slightly, reflecting the fact that a Python function, not an external command, is now called to build the target file:

% **scons**
build_function("file.foo", "file.input")

## Builders That Create Actions Using a `Generator`

`SCons` Builder objects can create an action "on the fly" by using a function called a `generator`. This provides a great deal of flexibility XXX A `generator` looks like:

```
def generate_actions(source, target, env, for_signature):
    return XXX
```

The arguments of a `generator` are:

source

A list of Node objects representing the sources to be built by the command or other action generated by this function. The file names of these source(s) may be extracted using the Python `str` funcion.

target

A list of Node objects representing the target or targets to be built by the command or other action generated by this function. The file names of these target(s) may be extracted using the Python `str` funcion.

env

The `construction environment` used for building the target(s). The generator may use any of the environment's construction variables in any way to determine what command or other action to return.

for_signature

A flag that specifies whether the generator is being called to contribute to a build signature, as opposed to actually executing the command. XXX

The `generator` must return a command string or other action that will be used to build the specified target(s) from the specified source(s).

Once you've defined a `generator`, you create a `Builder` to use it by specifying the generator keyword argument instead of `action`.

```
bld = Builder(generator = generate_actions,
              suffix = '.foo',
              src_suffix = '.input')
env = Environment(BUILDERS = {'Foo' : bld})
env.Foo('file')
```

```
% scons
XXX
```

Note that it's illegal to specify both an `action` and a `generator` for a `Builder`.

## Builders That Modify the Target or Source Lists Using an `Emitter`

`SCons` supports the ability for a Builder to modify the lists of target(s) from the specified source(s).

```
def modify_targets(XXX):
    return XXX
bld = Builder(action = 'XXX',
              suffix = '.foo',
              src_suffix = '.input',
              emitter = modify_targets)
```

```
env = Environment(BUILDERS = {'Foo' : bld})
env.Foo('file')
```

% **scons**
XXX

```
bld = Builder(action = 'XXX',
              suffix = '.foo',
              src_suffix = '.input',
              emitter = 'MY_EMITTER')
def modify1(XXX):
    return XXX
def modify2(XXX):
    return XXX
env1 = Environment(BUILDERS = {'Foo' : bld},
                   MY_EMITTER = modify1)
env2 = Environment(BUILDERS = {'Foo' : bld},
                   MY_EMITTER = modify2)
env1.Foo('file1')
env2.Foo('file2')
```

## Builders That Use Other Builders

XXX

```
env = Environment()
env.SourceCode('.', env.BitKeeper('XXX')
env.Program('hello.c')
```

% **scons**
XXX

# Chapter 15. Not Writing a Builder: The `Command` Builder

Creating a `Builder` and attaching it to a `construction environment` allows for a lot of flexibility when you want to re-use actions to build multiple files of the same type. This can, however, be cumbersome if you only need to execute one specific command to build a single file (or group of files). For these situations, `SCons` supports a `Command Builder` that arranges for a specific action to be executed to build a specific file or files. This looks a lot like the other builders (like `Program`, `Object`, etc.), but takes as an additional argument the command to be executed to build the file:

```
env = Environment()
env.Command('foo.out', 'foo.in', "sed 's/x/y/' < $TARGET > $SOURCE")
```

```
% scons .
sed 's/x/y/' < foo.in > foo.out
```

This is often more convenient than creating a `Builder` object and adding it to the `BUILDERS` variable of a `construction environment`

Note that the action you

```
env = Environment()
def build(target, source, env)
    XXX
    return None
env.Command('foo.out', 'foo.in', build)
```

```
% scons .
build("foo.out", "foo.in")
```

# Chapter 16. SCons Actions

XXX

## XXX

XXX

# Chapter 17. Writing Scanners

XXX

## XXX

XXX

# Chapter 18. Building From Code Repositories

Often, a software project will have one or more central repositories, directory trees that contain source code, or derived files, or both. You can eliminate additional unnecessary rebuilds of files by having `SCons` use files from one or more code repositories to build files in your local build tree.

## The `Repository` Method

It's often useful to allow multiple programmers working on a project to build software from source files and/or derived files that are stored in a centrally-accessible repository, a directory copy of the source code tree. (Note that this is not the sort of repository maintained by a source code management system like BitKeeper, CVS, or Subversion. For information about using `SCons` with these systems, see the section, "Fetching Files From Source Code Management Systems," below.) You use the `Repository` method to tell `SCons` to search one or more central code repositories (in order) for any source files and derived files that are not present in the local build tree:

```
env = Environment()
env.Program('hello.c')
Repository('/usr/repository1', '/usr/repository2')
```

Multiple calls to the `Repository` method will simply add repositories to the global list that `SCons` maintains, with the exception that `SCons` will automatically eliminate the current directory and any non-existent directories from the list.

## Finding source files in repositories

The above example specifies that `SCons` will first search for files under the `/usr/repository1` tree and next under the `/usr/repository2` tree. `SCons` expects that any files it searches for will be found in the same position relative to the top-level directory XXX In the above example, if the `hello.c` file is not found in the local build tree, `SCons` will search first for a `/usr/repository1/hello.c` file and then for a `/usr/repository1/hello.c` file to use in its place.

So given the `SConstruct` file above, if the `hello.c` file exists in the local build directory, `SCons` will rebuild the `hello` program as normal:

% **scons**
gcc -c hello.c -o hello.o
gcc -o hello hello.o

If, however, there is no local `hello.c` file, but one exists in `/usr/repository1`, `SCons` will recompile the `hello` program from the source file it finds in the repository:

% **scons**
gcc -c /usr/repository1/hello.c -o hello.o
gcc -o hello hello.o

And similarly, if there is no local `hello.c` file and no `/usr/repository1/hello.c`, but one exists in `/usr/repository2`:

% **scons**
gcc -c /usr/repository2/hello.c -o hello.o
gcc -o hello hello.o

## Finding the `SConstruct` file in repositories

`SCons` will also search in repositories for the `SConstruct` file and any specified `SConscript` files. This poses a problem, though: how can `SCons` search a repository tree for an `SConstruct` file if the `SConstruct` file itself contains the information about the pathname of the repository? To solve this problem, `SCons` allows you to specify repository directories on the command line using the `-Y` option:

```
% scons -Y /usr/repository1 -Y /usr/repository2
```

When looking for source or derived files, `SCons` will first search the repositories specified on the command line, and then search the repositories specified in the `SConstruct` or `SConscript` files.

## Finding derived files in repositories

If a repository contains not only source files, but also derived files (such as object files, libraries, or executables), `SCons` will perform its normal MD5 signature calculation to decide if a derived file in a repository is up-to-date, or the derived file must be rebuilt in the local build directory. For the `SCons` signature calculation to work correctly, a repository tree must contain the `.sconsign` files that `SCons` uses to keep track of signature information.

Usually, this would be done by a build integrator who would run `SCons` in the repository to create all of its derived files and `.sconsign` files, or who would `SCons` in a separate build directory and copying the resulting tree to the desired repository:

```
% cd /usr/repository1
% scons
gcc -c hello.c -o hello.o
gcc -o hello hello.o
```

(Note that this is safe even if the `SConstruct` file lists `/usr/repository1` as a repository, because `SCons` will remove the current build directory from its repository list for that invocation.)

Now, with the repository populated, we only need to create the one local source file we're interested in working with at the moment, and use the `-Y` option to tell `SCons` to fetch any other files it needs from the repository:

```
% cd $HOME/build
% edit hello.c
% scons -Y /usr/repository1
gcc -c hello.c -o hello.o
gcc -o hello hello.o
XXXXXXX
```

Notice that `SCons` realizes that it does not need to rebuild a local XXX.o file, but instead uses the already-compiled XXX.o file from the repository.

## Guaranteeing local copies of files

If the repository tree contains the complete results of a build, and we try to build from the repository without any files in our local tree, something moderately surprising happens:

```
% mkdir $HOME/build2
% cd $HOME/build2
% scons -Y /usr/all/repository hello
scons: 'hello' is up-to-date.
```

Why does `SCons` say that the `hello` program is up-to-date when there is no `hello` program in the local build directory? Because the repository (not the local directory) contains the up-to-date `hello` program, and `SCons` correctly determines that nothing needs to be done to rebuild that up-to-date copy of the file.

There are, however, many times when you want to ensure that a local copy of a file always exists. A packaging or testing script, for example, may assume that certain generated files exist locally. To tell `SCons` to make a copy of any up-to-date repository file in the local build directory, use the `Local` function:

```
env = Environment()
hello = env.Program('hello.c')
Local(hello)
```

If we then run the same command, `SCons` will make a local copy of the program from the repository copy, and tell you that it is doing so:

```
% scons -Y /usr/all/repository hello
Local copy of hello from /usr/all/repository/hello
scons: 'hello' is up-to-date.
XXXXXX DO WE REALLY REPORT up-to-date, TOO?
```

(Notice that, because the act of making the local copy is not considered a "build" of the `hello` file, `SCons` still reports that it is up-to-date.) XXXXXX DO WE REALLY REPORT up-to-date, TOO?

# Chapter 19. Fetching Files From Source Code Management Systems

X

## Fetching Source Code From BitKeeper

X

```
env = Environment()
env.SourceCode('.', env.BitKeeper('XXX'))
env.Program('hello.c')
```

```
% scons
XXX
```

## Fetching Source Code From CVS

X

```
env = Environment()
env.SourceCode('.', env.CVS('XXX'))
env.Program('hello.c')
```

```
% scons
XXX
```

## Fetching Source Code From RCS

X

```
env = Environment()
env.SourceCode('.', env.RCS())
env.Program('hello.c')
```

```
% scons
XXX
```

## Fetching Source Code From SCCS

X

```
env = Environment()
env.SourceCode('.', env.SCCS())
env.Program('hello.c')
```

```
% scons
XXX
```

## Fetching Source Code From Subversion

X

```
env = Environment()
env.SourceCode('.', env.Subversion('XXX'))
env.Program('hello.c')
```

% **scons**
XXX

# Chapter 20. Caching Built Files

On multi-developer software projects, you can sometimes speed up every developer's builds a lot by allowing them to share the derived files that they build. `SCons` makes this easy, as well as reliable.

## Specifying the Shared Cache Directory

To enable sharing of derived files, use the `CacheDir` function in any `SConscript` file:

```
CacheDir('/usr/local/build_cache')
```

Note that the directory you specify must already exist and be readable and writable by all developers who will be sharing derived files. It should also be in some central location that all builds will be able to access. In environments where developers are using separate systems (like individual workstations) for builds, this directory would typically be on a shared or NFS-mounted file system.

Here's what happens: When a build has a `CacheDir` specified, every time a file is built, it is stored in the shared cache directory along with its MD5 build signature. On subsequent builds, before an action is invoked to build a file, `SCons` will check the shared cache directory to see if a file with the exact same build signature already exists. If so, the derived file will not be built locally, but will be copied into the local build directory from the shared cache directory, like so:

```
% scons
cc -c hello.c -o hello.o
cc -o hello hello.o
% scons -c
Removed hello.o
Removed hello
% scons
Retrieved 'hello.o' from cache
Retrieved 'hello' from cache
%
```

## Keeping Build Output Consistent

One potential drawback to using a shared cache is that your build output can be inconsistent from invocation to invocation, because any given file may be rebuilt one time and retrieved from the shared cache the next time. This can make analyzing build output more difficult, especially for automated scripts that expect consistent output each time.

If, however, you use the `--cache-show` option, `SCons` will print the command line that it *would* have executed to build the file, even when it is retrieving the file from the shared cache. This makes the build output consistent every time the build is run:

```
% scons
cc -c hello.c -o hello.o
cc -o hello hello.o
% scons -c
Removed hello.o
Removed hello
% scons --cache-show
cc -c hello.c -o hello.o
cc -o hello hello.o
```

```
%
```

The trade-off, of course, is that you no longer know whether or not `SCons` has retrieved a derived file from cache or has rebuilt it locally.

## Not Retrieving Files From a Shared Cache

Retrieving an already-built file from the shared cache is usually a significant time-savings over rebuilding the file, but how much of a savings (or even whether it saves time at all) can depend a great deal on your system or network configuration. For example, retrieving cached files from a busy server over a busy network might end up being slower than rebuilding the files locally.

In these cases, you can specify the `--cache-disable` command-line option to tell `SCons` to not retrieve already-built files from the shared cache directory:

```
% scons
cc -c hello.c -o hello.o
cc -o hello hello.o
% scons -c
Removed hello.o
Removed hello
% scons
Retrieved `hello.o' from cache
Retrieved `hello' from cache
% scons -c
Removed hello.o
Removed hello
% scons --cache-disable
cc -c hello.c -o hello.o
cc -o hello hello.o
%
```

## Populating a Shared Cache With Already-Built Files

Sometimes, you may have one or more derived files already built in your local build tree that you wish to make available to other people doing builds. For example, you may find it more effective to perform integration builds with the cache disabled (per the previous section) and only populate the shared cache directory with the built files after the integration build has completed successfully. This way, the cache will only get filled up with derived files that are part of a complete, successful build not with files that might be later overwritten while you debug integration problems.

In this case, you can use the the `--cache-force` option to tell `SCons` to put all derived files in the cache, even if the files had already been built by a previous invocation:

```
% scons --cache-disable
cc -c hello.c -o hello.o
cc -o hello hello.o
% scons -c
Removed hello.o
Removed hello
% scons --cache-disable
cc -c hello.c -o hello.o
cc -o hello hello.o
% scons --cache-force
```

```
% scons -c
Removed hello.o
Removed hello
% scons
Retrieved `hello.o' from cache
Retrieved `hello' from cache
%
```

Notice how the above sample run demonstrates that the `--cache-disable` option avoids putting the built `hello.o` and `hello` files in the cache, but after using the `--cache-force` option, the files have been put in the cache for the next invocation to retrieve.

# Chapter 21. Alias Targets

We've already seen how you can use the `Alias` function to create a target named `install`:

```
env = Environment()
hello = env.Program('hello.c')
env.Install('/usr/bin', hello)
env.Alias('install', '/usr/bin')
```

You can then use this alias on the command line to tell `SCons` more naturally that you want to install files:

```
% scons install
Install file: "hello" as "/usr/bin/hello"
```

Like other `Builder` methods, though, the `Alias` method returns an object representing the alias being built. You can then use this object as input to anothother `Builder`. This is especially useful if you use such an object as input to another call to the `Alias` `Builder`, allowing you to create a hierarchy of nested aliases:

```
env = Environment()
p = env.Program('hello.c')
l = env.Library('hello.c')
env.Install('/usr/bin', p)
env.Install('/usr/lib', l)
ib = env.Alias('install-bin', '/usr/bin')
il = env.Alias('install-lib', '/usr/lib')
env.Alias('install', [ib, il])
```

This example defines separate `install`, `install-bin`, and `install-lib` aliases, allowing you finer control over what gets installed:

```
% scons install-bin
Install file: "hello" as "/usr/bin/hello"
% scons install-lib
Install file: "libhello.a" as "/usr/lib/libhello.a"
% scons -c /
% scons install
Install file: "hello" as "/usr/bin/hello"
Install file: "libhello.a" as "/usr/lib/libhello.a"
```

# Chapter 22. How to Run `SCons`

XXX

## Selective Builds

XXX

## Overriding Construction Variables

XXX

## The `SCONSFLAGS` Environment Variable

XXX

# Chapter 23. Troubleshooting

XXX

## XXX

XXX

# Appendix A. Complex `SCons` Example

XXX

## XXX

XXX

# Appendix B. Converting From Make

XXX

## Differences Between `Make` and `SCons`

XXX

## Advantages of `SCons` Over `Make`

XXX

# Appendix C. Converting From Cons

XXX

## Differences Between `Cons` and `SCons`

XXX

## Advantages of `SCons` Over `Cons`

XXX

# Appendix D. Converting From Ant

XXX

## Differences Between `Ant` and `SCons`

XXX

## Advantages of `SCons` Over `Ant`

XXX