SCons User Guide 0.95

Steven Knight

SCons User Guide 0.95
by Steven Knight

Revision 0.95.D001 (2004/03/08 07:28:28) 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 usually invented by programmers for other programmers. This is in no small part due 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.

`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 in all end cases for a speedier build in the usual cases.

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.

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, Gary Oerbrunner, 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 "Autoscons" 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.

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`:

```
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
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
cc -c -o hello.o hello.c
cc -o hello hello.o
scons: done building targets.
```

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

```
C:\>scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
cl /nologo /c hello.c /Fohello.obj
link /nologo /OUT:hello.exe hello.obj
scons: done building targets.
```

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.)

Cleaning Up After a Build

When using `SCons`, it is unnecessary to add special commands or target names to clean up after a build. Instead, you simply use the `-c` or `--clean` option when you invoke `SCons`, and `SCons` removes the appropriate built files. So if we build our example above and then invoke `scons -c` afterwards, the output on POSIX looks like:

```
% scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
cc -c -o hello.o hello.c
cc -o hello hello.o
scons: done building targets.
% scons -c
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Cleaning targets ...
Removed hello.o
Removed hello
scons: done cleaning targets.
```

And the output on Windows looks like:

```
C:\>scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
cl /nologo /c hello.c /Fohello.obj
link /nologo /OUT:hello.exe hello.obj
scons: done building targets.
C:\>scons -c
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Cleaning targets ...
Removed hello.obj
Removed hello.exe
scons: done cleaning targets.
```

Notice that `SCons` changes its output to tell you that it is `Cleaning targets ...` and `done cleaning targets.`

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:

```
# Arrange to build the "hello" program.
Program('hello.c')    # "hello.c" is the source file.
```

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.

Making the Output Less Verbose

You've already seen how `SCons` prints some messages about what it's doing, surrounding the actual commands used to build the software:

```
C:\>scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
cl /nologo /c hello.c /Fohello.obj
link /nologo /OUT:hello.exe hello.obj
scons: done building targets.
```

These messages emphasize the order in which `SCons` does its work: the configuration files (generically referred to as `SConscript` files) are read and executed first, and only then are the target files built. Among other benefits, these messages help to distinguish between errors that occur while the configuration files are read, and errors that occur while targets are being built.

The drawback, of course, is that these messages clutter the output. Fortunately, they're easily disabled by using the `-Q` option when invoking `SCons`:

```
C:\>scons -Q
cl /nologo /c hello.c /Fohello.obj
link /nologo /OUT:hello.exe hello.obj
```

Because we want this User's Guide to focus on what `SCons` is actually doing, we're going use the `-Q` option to remove these messages from the output of all the remaining examples in this Guide.

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:

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

A build of the above example would look like:

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

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

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

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

Or on Windows:

```
C:\>scons -Q
cl /nologo /c file1.c /Fofile1.obj
cl /nologo /c file2.c /Fofile2.obj
cl /nologo /c main.c /Fomain.obj
link /nologo /OUT:program.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:

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

(If you're already familiar with Python, you'll have realized that this is similar to the `split()` method in the Python standard `string` module. Unlike the `string.split()` method, however, the `Split` function does not require a string as input and will wrap up a single non-string object in a list, or return its argument untouched if it's already a list. This comes in handy as a way to make sure arbitrary values can be passed to `SCons` functions without having to check the type of the variable by hand.)

Putting the call to the `Split` function inside the `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 `Program` function:

```
list = Split('main.c file1.c file2.c')
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:

```
list = Split('main.c
              file1.c
              file2.c')
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:

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

Because the keywords explicitly identify what each argument is, you can actually reverse the order if you prefer:

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

Whether or not you choose to use keyword arguments to identify the target and source files, and the order in which you specify them when using keywords, are purely personal choices; `SCons` functions the same regardless.

Compiling Multiple Programs

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

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

`SCons` would then build the programs as follows:

% **scons -Q**
cc -c -o bar1.o bar1.c
cc -c -o bar2.o bar2.c
cc -o bar bar1.o bar2.o
cc -c -o foo.o foo.c
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:

```
Program(Split('foo.c common1.c common2.c'))
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 resulting object files are each linked in to both of the resulting executable programs:

```
% scons -Q
cc -c -o bar1.o bar1.c
cc -c -o bar2.o bar2.c
cc -c -o common1.o common1.c
cc -c -o common2.o common2.c
cc -o bar bar1.o bar2.o common1.o common2.o
cc -c -o foo.o foo.c
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
Program('foo', foo_files)
Program('bar', bar_files)
```

This is functionally equivalent to the previous example.

Chapter 3. Building and Linking with Libraries

It's often useful to organize large software projects by collecting parts of the software into one or more libraries. `SCons` makes it easy to create libraries and to use them in the programs.

Building Libraries

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

```
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 -Q
cc -c -o f1.o f1.c
cc -c -o f2.o f2.c
cc -c -o f3.o f3.c
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 -Q
cl /nologo /c f1.c /Fof1.obj
cl /nologo /c f2.c /Fof2.obj
cl /nologo /c f3.c /Fof3.obj
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.

Building Static Libraries Explicitly

The `Library` function builds a traditional static library. If you want to be explicit about the type of library being built, you can use the synonym `StaticLibrary` function instead of `Library`:

```
StaticLibrary('foo', ['f1.c', 'f2.c', 'f3.c'])
```

There is no functional difference between the `StaticLibrary` and `Library` functions.

Building Shared (DLL) Libraries

If you want to build a shared library (on POSIX systems) or a DLL file (on Windows systems), you use the `SharedLibrary` function:

```
SharedLibrary('foo', ['f1.c', 'f2.c', 'f3.c'])
```

The output on POSIX:

```
% scons -Q
cc -c -o f1.os f1.c
cc -c -o f2.os f2.c
cc -c -o f3.os f3.c
cc -shared -o libfoo.so f1.os f2.os f3.os
```

And the output on Windows:

```
C:\>scons -Q
cl /nologo /c f1.c /Fof1.obj
cl /nologo /c f2.c /Fof2.obj
cl /nologo /c f3.c /Fof3.obj
link /nologo /dll /out:foo.dll /implib:foo.lib f1.obj f2.obj f3.obj
```

Notice again that `SCons` takes care of building the output file correctly, adding the `-shared` option for a POSIX compilation, and the `/dll` option on Windows.

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:

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

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

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

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

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

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

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

```
C:\>scons -Q
cl /nologo /c prog.c /Foprog.obj
link /nologo /OUT:prog.exe /LIBPATH:\usr\lib /LIBPATH:\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 4. 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 -Q
cc -c -o hello.o hello.c
cc -o hello hello.o
% scons -Q
scons: `.' is up to date.
```

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

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

```
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:

```
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 -Q hello
cc -c -o hello.o hello.c
cc -o hello hello.o
% touch hello.c
% scons -Q hello
cc -c -o hello.o hello.c
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 "intermediate" target files are 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 -Q hello
cc -c -o hello.o hello.c
cc -o hello hello.o
% edit hello.c
```

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

```
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 make `SCons` realize that it does not need to rebuild a dependent target file in this situation using the `TargetSignatures` function as follows:

```
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 -Q hello
cc -c -o hello.o hello.c
cc -o hello hello.o
% edit hello.c
  [CHANGE A COMMENT IN hello.c]
% scons -Q hello
cc -c -o hello.o hello.c
scons: 'hello' is up to date.
```

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:

```
Program('hello.c', CPPPATH = '.')
```

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

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

Will look like this on POSIX or Linux:

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

```
cc -o hello hello.o
```

And like this on Windows:

```
C:\>scons -Q hello.exe
cl /nologo /Iinclude /I\home\project\inc /c hello.c /Fohello.obj
link /nologo /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 can do this by specifying the `--implicit-cache` option on the command line:

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

If you don't want to specify `--implicit-cache` on the command line each time, you can make it the default behavior for your build by setting the `implicit_cache` option in an `SConscript` file:

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

The `--implicit-deps-changed` Option

When using cached implicit dependencies, sometimes you want to "start fresh" and have `SCons` re-scan the files for which it previously cached the dependencies. For example, if you have recently installed a new version of external code that you use for compilation, the external header files will have changed and the previously-cached implicit dependencies will be out of date. You can update them by running `SCons` with the `--implicit-deps-changed` option:

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

In this case, `SCons` will re-scan all of the implicit dependencies and cache updated copies of the information.

The `--implicit-deps-unchanged` Option

By default when caching dependencies, `SCons` notices when a file has been modified and re-scans the file for any updated implicit dependency information. Sometimes, however, you may want to force `SCons` to use the cached implicit dependencies, even if the source files changed. This can speed up a build for example, when you have changed your source files but know that you haven't changed any `#include` lines. In this case, you can use the `--implicit-deps-unchanged` option:

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

In this case, `SCons` will assume that the cached implicit dependencies are correct and will not bother to re-scan changed files. For typical builds after small, incremental changes to source files, the savings may not be very big, but sometimes every bit of improved performance counts.

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:

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

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

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

The `Depends` Method

On the other hand, sometimes a file depends on another file that is not detected by an `SCons` scanner. 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:

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

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

Chapter 5. 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. 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 -Q
gcc -O2 -c -o foo.o foo.c
gcc -o foo foo.o
```

Multiple `Construction Environments`

The real advantage of construction environments is that you can create as many different construction environments as you need, each tailored to a different way to build some piece of software or other file. If, for example, we need to build one program with the `-O2` flag and another with the `-g` (debug) flag, we would do this like so:

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

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

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

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

scons: *** Two environments with different actions 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. SCons can't just decide that one of them should take precedence over the other, so it generates the error. 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 object 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 -Q
cc -g -c -o foo-dbg.o foo.c
cc -o foo-dbg foo-dbg.o
cc -O2 -c -o foo-opt.o foo.c
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 which sets CCFLAGS for debugging:

```
env = Environment(CC = 'gcc')
```

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

env.Program('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 -Q
gcc -c -o foo.o foo.c
gcc -o foo foo.o
gcc -g -c -o foo-dbg.o foo.c
gcc -o foo-dbg foo-dbg.o
gcc -O2 -c -o foo-opt.o foo.c
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 -Q
CC is: cc
scons: '.' is up to date.
```

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 in ['OBJSUFFIX', 'LIBSUFFIX', 'PROGSUFFIX']:
    print "key = %s, value = %s" % (key, dict[key])
```

This `SConstruct` file will print the specified dictionary items for us on POSIX systems as follows:

```
% scons -Q
key = OBJSUFFIX, value = .o
key = LIBSUFFIX, value = .a
key = PROGSUFFIX, value =
scons: '.' is up to date.
```

And on Win32:

```
C:\>scons -Q
key = OBJSUFFIX, value = .obj
key = LIBSUFFIX, value = .lib
key = PROGSUFFIX, value = .exe
scons: '.' is up to date.
```

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 -Q
cc -DDEFINE2 -c -o bar.o bar.c
cc -o bar bar.o
cc -DDEFINE1 -c -o foo.o foo.c
cc -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 -Q
cc -DMY_VALUE -DLAST -c -o foo.o foo.c
cc -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 -Q
cc -DFIRST -DMY_VALUE -c -o foo.o foo.c
cc -o foo foo.o
```

Chapter 6. Controlling the Environment Used to Execute Build Commands

When SCons builds a target file, it does not execute the commands with the same external environment that you used to execute SCons. Instead, it uses the dictionary stored in the ENV construction variable as the external environment for executing commands.

The most important ramification of this behavior is that the PATH environment variable, which controls where the operating system will look for commands and utilities, is not the same as in the external environment from which you called SCons. This means that SCons will not, by default, necessarily find all of the tools that you can execute from the command line.

The default value of the PATH environment variable on a POSIX system is /usr/local/bin:/bin:/usr/bin. The default value of the PATH environment variable on a Win32 system comes from the Windows registry value for the command interpreter. If you want to execute any commands--compilers, linkers, etc.--that are not in these default locations, you need to set the PATH value in the ENV dictionary in your construction environment.

The simplest way to do this is to initialize explicitly the value when you create the construction environment; this is one way to do that:

```
path = ['/usr/local/bin', '/bin', '/usr/bin']
env = Environment(ENV = {'PATH' : path})
```

Propagating PATH From the External Environment

You may want to propagate the external PATH to the execution environment for commands. You do this by initializing the PATH variable with the PATH value from the os.environ dictionary, which is Python's way of letting you get at the external environment:

```
import os
env = Environment(ENV = {'PATH' : os.environ['PATH']})
```

Alternatively, you may find it easier to just propagate the entire external environment to the execution environment for commands. This is simpler to code than explicity selecting the PATH value:

```
import os
env = Environment(ENV = os.environ)
```

Either of these will guarantee that SCons will be able to execute any command that you can execute from the command line. The drawback is that the build can behave differently if it's run by people with different PATH values in their environment--for example, both the /bin and /usr/local/bin directories have different cc commands, then which one will be used to compile programs will depend on which directory is listed first in the user's PATH variable.

Chapter 7. Controlling a Build From the Command Line

`SCons` provides a number of ways that allow the writer of the `SConscript` files to give users a great deal of control over how to run the builds.

Not Having to Specify Command-Line Options Each Time: the `SCONSFLAGS` Environment Variable

Users may find themselves supplying the same command-line options every time they run `SCons`. For example, a user might find that it saves time to specify a value of `-j 2` to run the builds in parallel. To avoid having to type `-j 2` by hand every time, you can set the external environment variable `SCONSFLAGS` to a string containing command-line options that you want `SCons` to use.

If, for example, and you're using a POSIX shell that's compatible with the Bourne shell, and you always want `SCons` to use the `-Q` option, you can set the `SCONSFLAGS` environment as follows:

```
% scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
    ... [build output] ...
scons: done building targets.
% export SCONSFLAGS="-Q"
% scons
    ... [build output] ...
```

Users of `csh`-style shells on POSIX systems can set the `SCONSFLAGS` environment as follows:

```
$ setenv SCONSFLAGS "-Q"
```

Windows users may typically want to set this `SCONSFLAGS` in the appropriate tab of the `System Properties` window.

Getting at Command-Line Targets

`SCons` supports a `COMMAND_LINE_TARGETS` variable that lets you get at the list of targets that the user specified on the command line. You can use the targets to manipulate the build in any way you wish. As a simple example, suppose that you want to print a reminder to the user whenever a specific program is built. You can do this by checking for the target in the `COMMAND_LINE_TARGETS` list:

```
if 'bar' in COMMAND_LINE_TARGETS:
    print "Don't forget to copy `bar' to the archive!"
Default(Program('foo.c'))
Program('bar.c')
```

Then, running `SCons` with the default target works as it always does, but explicity specifying the `bar` target on the command line generates the warning message:

```
% scons -Q
cc -c -o foo.o foo.c
cc -o foo foo.o
% scons -Q bar
```

```
Don't forget to copy 'bar' to the archive!
cc -c -o bar.o bar.c
cc -o bar bar.o
```

Another practical use for the `COMMAND_LINE_TARGETS` variable might be to speed up a build by only reading certain subsidiary `SConscript` files if a specific target is requested.

Controlling the Default Targets

One of the most basic things you can control is which targets `SCons` will build by default--that is, when there are no targets specified on the command line. As mentioned previously, `SCons` will normally 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, or programs in certain directories, 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 -Q
cc -c -o hello.o hello.c
cc -o hello hello.o
% scons -Q
scons: 'hello' is up to date.
% scons -Q goodbye
cc -c -o goodbye.o goodbye.c
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 -Q .
cc -c -o goodbye.o goodbye.c
cc -o goodbye goodbye.o
cc -c -o hello.o hello.c
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 -Q
cc -c -o prog1.o prog1.c
cc -o prog1 prog1.o
cc -c -o prog3.o prog3.c
cc -o prog3 prog3.o
% scons -Q .
cc -c -o prog2.o prog2.c
cc -o prog2 prog2.o
```

You can list a directory as an argument to `Default`:

```
env = Environment()
env.Program(['prog1/main.c', 'prog1/foo.c'])
env.Program(['prog2/main.c', 'prog2/bar.c'])
Default('prog1')
```

In which case only the target(s) in that directory will be built by default:

```
% scons -Q
cc -c -o prog1/foo.o prog1/foo.c
cc -c -o prog1/main.o prog1/main.c
cc -o prog1/main prog1/main.o prog1/foo.o
% scons -Q
scons: `prog1' is up to date.
% scons -Q .
cc -c -o prog2/bar.o prog2/bar.c
cc -c -o prog2/main.o prog2/main.c
cc -o prog2/main prog2/main.o prog2/bar.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 -Q
scons: *** No targets specified and no Default() targets found.  Stop.
% scons -Q .
cc -c -o prog1.o prog1.c
cc -o prog1 prog1.o
cc -c -o prog2.o prog2.c
cc -o prog2 prog2.o
```

Getting at the List of Default Targets

SCons supports a DEFAULT_TARGETS variable that lets you get at the current list of default targets. The DEFAULT_TARGETS variable has two important differences from the COMMAND_LINE_TARGETS variable. First, the DEFAULT_TARGETS variable is a list of internal SCons nodes, so you need to convert the list elements to strings if you want to print them or look for a specific target name. Fortunately, you can do this easily by using the Python map function to run the list through str:

```
prog1 = Program('prog1.c')
Default(prog1)
print "DEFAULT_TARGETS is", map(str, DEFAULT_TARGETS)
```

(Keep in mind that all of the manipulation of the DEFAULT_TARGETS list takes place during the first phase when SCons is reading up the SConscript files, which is obvious if we leave off the -Q flag when we run SCons:)

```
% scons
scons: Reading SConscript files ...
DEFAULT_TARGETS is ['prog1']
scons: done reading SConscript files.
scons: Building targets ...
cc -c -o prog1.o prog1.c
cc -o prog1 prog1.o
scons: done building targets.
```

Second, the contents of the DEFAULT_TARGETS list change in response to calls to the Default: function, as you can see from the following SConstruct file:

```
prog1 = Program('prog1.c')
Default(prog1)
print "DEFAULT_TARGETS is now", map(str, DEFAULT_TARGETS)
prog2 = Program('prog2.c')
Default(prog2)
print "DEFAULT_TARGETS is now", map(str, DEFAULT_TARGETS)
```

Which yields the output:

```
% scons
scons: Reading SConscript files ...
DEFAULT_TARGETS is now ['prog1']
DEFAULT_TARGETS is now ['prog1', 'prog2']
scons: done reading SConscript files.
scons: Building targets ...
cc -c -o prog1.o prog1.c
cc -o prog1 prog1.o
cc -c -o prog2.o prog2.c
cc -o prog2 prog2.o
scons: done building targets.
```

In practice, this simply means that you need to pay attention to the order in which you call the Default function and refer to the DEFAULT_TARGETS list, to make sure that you don't examine the list before you've added the default targets you expect to find in it.

Getting at the List of Build Targets, Regardless of Origin

We've already been introduced to the `COMMAND_LINE_TARGETS` variable, which contains a list of targets specified on the command line, and the `DEFAULT_TARGETS` variable, which contains a list of targets specified via calls to the `Default` method or function. Sometimes, however, you want a list of whatever targets `SCons` will try to build, regardless of whether the targets came from the command line or a `Default` call. You could code this up by hand, as follows:

```
if COMMAND_LINE_TARGETS:
    targets = COMMAND_LINE_TARGETS
else:
    targets = DEFAULT_TARGETS
```

`SCons`, however, provides a convenient `BUILD_TARGETS` variable that eliminates the need for this by-hand manipulation. Essentially, the `BUILD_TARGETS` variable contains a list of the command-line targets, if any were specified, and if no command-line targets were specified, it contains a list of the targets specified via the `Default` method or function.

Because `BUILD_TARGETS` may contain a list of `SCons` nodes, you must convert the list elements to strings if you want to print them or look for a specific target name, just like the `DEFAULT_TARGETS` list:

```
prog1 = Program('prog1.c')
Program('prog2.c')
Default(prog1)
print "BUILD_TARGETS is", map(str, BUILD_TARGETS)
```

Notice how the value of `BUILD_TARGETS` changes depending on whether a target is specified on the command line:

```
% scons -Q
BUILD_TARGETS is ['prog1']
cc -c -o prog1.o prog1.c
cc -o prog1 prog1.o
% scons -Q prog2
BUILD_TARGETS is ['prog2']
cc -c -o prog2.o prog2.c
cc -o prog2 prog2.o
% scons -Q -c .
BUILD_TARGETS is ['.']
Removed prog1.o
Removed prog1
Removed prog2.o
Removed prog2
```

Command-Line `variable=value` Build Options

You may want to control various aspects of your build by allowing the user to specify `variable=value` values on the command line. For example, suppose you want users to be able to build a debug version of a program by running `SCons` as follows:

```
% scons -Q debug=1
```

`SCons` provides an `ARGUMENTS` dictionary that stores all of the `variable=value` assignments from the command line. This allows you to modify aspects of your build in response to specifications on the command line. (Note that unless you want to require that users *always* specify an option, you probably want to use the Python `ARGUMENTS.get()` function, which allows you to specify a default value to be used if there is no specification on the command line.)

The following code sets the `CCFLAGS` construction variable in response to the `debug` flag being set in the `ARGUMENTS` dictionary:

```
env = Environment()
debug = ARGUMENTS.get('debug', 0)
if int(debug):
    env.Append(CCFLAGS = '-g')
env.Program('prog.c')
```

This results in the `-g` compiler option being used when `debug=1` is used on the command line:

```
% scons -Q debug=0
cc -c -o prog.o prog.c
cc -o prog prog.o
% scons -Q debug=0
scons: `.' is up to date.
% scons -Q debug=1
cc -g -c -o prog.o prog.c
cc -o prog prog.o
% scons -Q debug=1
scons: `.' is up to date.
```

Notice that `SCons` keeps track of the last values used to build the object files, and as a result correctly rebuilds the object and executable files only when the value of the `debug` argument has changed.

Controlling Command-Line Build Options

Being able to use a command-line build option like `debug=1` is handy, but it can be a chore to write specific Python code to recognize each such option and apply the values to a construction variable. To help with this, `SCons` supports a class to define such build options easily, and a mechanism to apply the build options to a construction environment. This allows you to control how the build options affect construction environments.

For example, suppose that you want users to set a `RELEASE` construction variable on the command line whenever the time comes to build a program for release, and that the value of this variable should be added to the command line with the appropriate `-D` option (or other command line option) to pass the value to the C compiler. Here's how you might do that by setting the appropriate value in a dictionary for the `CPPDEFINES` construction variable:

```
opts = Options()
opts.Add('RELEASE', 'Set to 1 to build for release', 0)
env = Environment(options = opts,
                  CPPDEFINES={'RELEASE_BUILD' : '${RELEASE}'})
env.Program(['foo.c', 'bar.c'])
```

This `SConstruct` file first creates an `Options` object (the `opts = Options()` call), and then uses the object's `Add` method to indicate that the `RELEASE` option can be set on the command line, and that it's default value will be `0` (the third argument to the

`Add` method). The second argument is a line of help text; we'll learn how to use it in the next section.

We then pass the created `Options` object as an `options` keyword argument to the `Environment` call used to create the construction environment. This then allows a user to set the `RELEASE` build option on the command line and have the variable show up in the command line used to build each object from a C source file:

```
% scons -Q RELEASE=1
cc -DRELEASE_BUILD=1 -c -o bar.o bar.c
cc -DRELEASE_BUILD=1 -c -o foo.o foo.c
cc -o foo foo.o bar.o
```

Providing Help for Command-Line Build Options

To make command-line build options most useful, you ideally want to provide some help text that will describe the available options when the user runs `scons -h`. You could write this text by hand, but `SCons` provides an easier way. `Options` objects support a `GenerateHelpText` method that will, as its name indicates, generate text that describes the various options that have been added to it. You then pass the output from this method to the `Help` function:

```
opts = Options('custom.py')
opts.Add('RELEASE', 'Set to 1 to build for release', 0)
env = Environment(options = opts)
Help(opts.GenerateHelpText(env))
```

`SCons` will now display some useful text when the `-h` option is used:

```
% scons -Q -h

RELEASE: Set to 1 to build for release
    default: 0
    actual: 0

Use scons -H for help about command-line options.
```

Notice that the help output shows the default value, and the current actual value of the build option.

Reading Build Options From a File

Being able to use a command-line build option like `debug=1` is handy, but it can be a chore to write specific Python code to recognize each such option and apply the values to a construction variable. To help with this, `SCons` supports a class to define such build options easily and to read build option values from a file. This allows you to control how the build options affect construction environments. The way you do this is by specifying a file name when you call `Options`, like `custom.py` in the following example:

```
opts = Options('custom.py')
opts.Add('RELEASE', 'Set to 1 to build for release', 0)
env = Environment(options = opts,
                  CPPDEFINES={'RELEASE_BUILD' : '${RELEASE}'})
env.Program(['foo.c', 'bar.c'])
Help(opts.GenerateHelpText(env))
```

This then allows us to control the `RELEASE` variable by setting it in the `custom.py` file:

```
RELEASE = 1
```

Note that this file is actually executed like a Python script. Now when we run `SCons`:

```
% scons -Q
cc -DRELEASE_BUILD=1 -c -o bar.o bar.c
cc -DRELEASE_BUILD=1 -c -o foo.o foo.c
cc -o foo foo.o bar.o
```

And if we change the contents of `custom.py` to:

```
RELEASE = 0
```

The object files are rebuilt appropriately with the new option:

```
% scons -Q
cc -DRELEASE_BUILD=0 -c -o bar.o bar.c
cc -DRELEASE_BUILD=0 -c -o foo.o foo.c
cc -o foo foo.o bar.o
```

Canned Build Options

`SCons` provides a number of functions that provide ready-made behaviors for various types of command-line build options.

True/False Values: the `BoolOption` Build Option

It's often handy to be able to specify an option that controls a simple Boolean variable with a `true` or `false` value. It would be even more handy to accomodate users who have different preferences for how to represent `true` or `false` values. The `BoolOption` function makes it easy to accomodate a variety of common values that represent `true` or `false`.

The `BoolOption` function takes three arguments: the name of the build option, the default value of the build option, and the help string for the option. It then returns appropriate information for passing to the `Add` method of an `Options` object, like so:

```
opts = Options('custom.py')
opts.Add(BoolOption('RELEASE', 0, 'Set to build for release'))
env = Environment(options = opts,
                  CPPDEFINES={'RELEASE_BUILD' : '${RELEASE}'})
env.Program('foo.c')
```

With this build option, the `RELEASE` variable can now be enabled by setting it to the value `yes` or `t`:

```
% scons -Q RELEASE=yes foo.o
cc -DRELEASE_BUILD=1 -c -o foo.o foo.c
```

```
% scons -Q RELEASE=t foo.o
```

```
cc -DRELEASE_BUILD=1 -c -o foo.o foo.c
```

Other values that equate to `true` include `y`, `1`, `on` and `all`.

Conversely, `RELEASE` may now be given a `false` value by setting it to `no` or `f`:

```
% scons -Q RELEASE=no foo.o
cc -DRELEASE_BUILD=0 -c -o foo.o foo.c
```

```
% scons -Q RELEASE=f foo.o
cc -DRELEASE_BUILD=0 -c -o foo.o foo.c
```

Other values that equate to `true` include `n`, `0`, `off` and `none`.

Lastly, if a user tries to specify any other value, `SCons` supplies an appropriate error message:

```
% scons -Q RELEASE=bad_value foo.o

scons: *** Error converting option: RELEASE
Invalid value for boolean option: bad_value
File "SConstruct", line 4, in ?
```

Single Value From a List: the `EnumOption` Build Option

Suppose that we want a user to be able to set a `COLOR` option that selects a background color to be displayed by an application, but that we want to restrict the choices to a specific set of allowed colors. This can be set up quite easily using the `EnumOption`, which takes a list of `allowed_values` in addition to the variable name, default value, and help text arguments:

```
opts = Options('custom.py')
opts.Add(EnumOption('COLOR', 'red', 'Set background color',
                    allowed_values=('red', 'green', 'blue')))
env = Environment(options = opts,
                  CPPDEFINES={'COLOR' : '"${COLOR}"'})
env.Program('foo.c')
```

The user can now explicity set the `COLOR` build option to any of the specified allowed values:

```
% scons -Q COLOR=red foo.o
cc -DCOLOR="red" -c -o foo.o foo.c
% scons -Q COLOR=blue foo.o
cc -DCOLOR="blue" -c -o foo.o foo.c
% scons -Q COLOR=green foo.o
cc -DCOLOR="green" -c -o foo.o foo.c
```

But, almost more importantly, an attempt to set `COLOR` to a value that's not in the list generates an error message:

```
% scons -Q COLOR=magenta foo.o

scons: *** Invalid value for option COLOR: magenta
File "SConstruct", line 5, in ?
```

The `EnumOption` function also supports a way to map alternate names to allowed values. Suppose, for example, that we want to allow the user to use the word `navy` as a synonym for `blue`. We do this by adding a `map` dictionary that will map its key values to the desired legal value:

```
opts = Options('custom.py')
opts.Add(EnumOption('COLOR', 'red', 'Set background color',
                    allowed_values=('red', 'green', 'blue'),
                    map={'navy':'blue'}))
env = Environment(options = opts,
                  CPPDEFINES={'COLOR' : '"${COLOR}"'})
env.Program('foo.c')
```

As desired, the user can then use `navy` on the command line, and `SCons` will translate it into `blue` when it comes time to use the `COLOR` option to build a target:

% **scons -Q COLOR=navy foo.o**
cc -DCOLOR="blue" -c -o foo.o foo.c

By default, when using the `EnumOption` function, arguments that differ from the legal values only in case are treated as illegal values:

% **scons -Q COLOR=Red foo.o**

scons: *** Invalid value for option COLOR: Red
File "SConstruct", line 5, in ?
% **scons -Q COLOR=BLUE foo.o**

scons: *** Invalid value for option COLOR: BLUE
File "SConstruct", line 5, in ?
% **scons -Q COLOR=nAvY foo.o**

scons: *** Invalid value for option COLOR: nAvY
File "SConstruct", line 5, in ?

The `EnumOption` function can take an additional `ignorecase` keyword argument that, when set to `1`, tells `SCons` to allow case differences when the values are specified:

```
opts = Options('custom.py')
opts.Add(EnumOption('COLOR', 'red', 'Set background color',
                    allowed_values=('red', 'green', 'blue'),
                    map={'navy':'blue'},
                    ignorecase=1))
env = Environment(options = opts,
                  CPPDEFINES={'COLOR' : '"${COLOR}"'})
env.Program('foo.c')
```

Which yields the output:

% **scons -Q COLOR=Red foo.o**
cc -DCOLOR="Red" -c -o foo.o foo.c
% **scons -Q COLOR=BLUE foo.o**
cc -DCOLOR="BLUE" -c -o foo.o foo.c
% **scons -Q COLOR=nAvY foo.o**
cc -DCOLOR="blue" -c -o foo.o foo.c

% **scons -Q COLOR=green foo.o**
cc -DCOLOR="green" -c -o foo.o foo.c

Notice that an `ignorecase` value of `1` preserves the case-spelling that the user supplied. If you want `SCons` to translate the names into lower-case, regardless of the case used by the user, specify an `ignorecase` value of `2`:

```
opts = Options('custom.py')
opts.Add(EnumOption('COLOR', 'red', 'Set background color',
                    allowed_values=('red', 'green', 'blue'),
                    map={'navy':'blue'},
                    ignorecase=2))
env = Environment(options = opts,
                  CPPDEFINES={'COLOR' : '"${COLOR}"'})
env.Program('foo.c')
```

Now `SCons` will use values of `red`, `green` or `blue` regardless of how the user spells those values on the command line:

% **scons -Q COLOR=Red foo.o**
cc -DCOLOR="red" -c -o foo.o foo.c
% **scons -Q COLOR=nAvY foo.o**
cc -DCOLOR="blue" -c -o foo.o foo.c
% **scons -Q COLOR=GREEN foo.o**
cc -DCOLOR="green" -c -o foo.o foo.c

Multiple Values From a List: the `ListOption` Build Option

Another way in which you might want to allow users to control build option is to specify a list of one or more legal values. `SCons` supports this through the `ListOption` function. If, for example, we want a user to be able to set a `COLORS` option to one or more of the legal list of values:

```
opts = Options('custom.py')
opts.Add(ListOption('COLORS', 0, 'List of colors',
                    ['red', 'green', 'blue']))
env = Environment(options = opts,
                  CPPDEFINES={'COLORS' : '"${COLORS}"'})
env.Program('foo.c')
```

A user can now specify a comma-separated list of legal values, which will get translated into a space-separated list for passing to the any build commands:

% **scons -Q COLORS=red,blue foo.o**
TypeError: sequence item 0: expected string, int found:
% **scons -Q COLORS=blue,green,red foo.o**
TypeError: sequence item 0: expected string, int found:

In addition, the `ListOption` function allows the user to specify explicit keywords of `all` or `none` to select all of the legal values, or none of them, respectively:

% **scons -Q COLORS=all foo.o**
TypeError: sequence item 0: expected string, int found:
% **scons -Q COLORS=none foo.o**
TypeError: sequence item 0: expected string, int found:

And, of course, an illegal value still generates an error message:

% **scons -Q COLORS=magenta foo.o**
TypeError: sequence item 0: expected string, int found:

Path Names: the `PathOption` Build Option

`SCons` supports a `PathOption` function to make it easy to create a build option to control an expected path name. If, for example, you need to define a variable in the preprocessor that control the location of a configuration file:

```
opts = Options('custom.py')
opts.Add(PathOption('CONFIG', '/etc/my_config', 'Path to configuration file'))
env = Environment(options = opts,
                  CPPDEFINES={'CONFIG_FILE' : '"$CONFIG"'})
env.Program('foo.c')
```

This then allows the user to override the `CONFIG` build option on the command line as necessary:

% **scons -Q foo.o**

scons: *** Path does not exist for option CONFIG: Path to configuration file
File "SConstruct", line 4, in ?
% **scons -Q CONFIG=/usr/local/etc/other_config foo.o**
cc -DCONFIG_FILE="/usr/local/etc/other_config" -c -o foo.o foo.c

Enabled/Disabled Path Names: the `PackageOption` Build Option

Sometimes you want to give users even more control over a path name variable, allowing them to explicitly enable or disable the path name by using `yes` or `no` keywords, in addition to allow them to supply an explicit path name. `SCons` supports the `PackageOption` function to support this:

```
opts = Options('custom.py')
opts.Add(PackageOption('PACKAGE', '/opt/location', 'Location package'))
env = Environment(options = opts,
                  CPPDEFINES={'PACKAGE' : '"$PACKAGE"'})
env.Program('foo.c')
```

When the `SConscript` file uses the `PackageOption` funciton, user can now still use the default or supply an overriding path name, but can now explicitly set the specified variable to a value that indicates the package should be enabled (in which case the default should be used) or disabled:

% **scons -Q foo.o**

scons: *** Path does not exist for option PACKAGE: Location package
File "SConstruct", line 4, in ?
% **scons -Q PACKAGE=/usr/local/location foo.o**
cc -DPACKAGE="/usr/local/location" -c -o foo.o foo.c
% **scons -Q PACKAGE=yes foo.o**
cc -DPACKAGE="1" -c -o foo.o foo.c
% **scons -Q PACKAGE=no foo.o**

```
cc -DPACKAGE="0" -c -o foo.o foo.c
```

Adding Multiple Command-Line Build Options at Once

Lastly, `SCons` provides a way to add multiple build options to an `Options` object at once. Instead of having to call the `Add` method multiple times, you can call the `AddOptions` method with a list of build options to be added to the object. Each build option is specified as either a tuple of arguments, just like you'd pass to the `Add` method itself, or as a call to one of the canned functions for pre-packaged command-line build options. in any order:

```
opts = Options()
opts.AddOptions(
    ('RELEASE', 'Set to 1 to build for release', 0),
    ('CONFIG', 'Configuration file', '/etc/my_config'),
    BoolOption('warnings', 'compilation with -Wall and similiar', 1),
    EnumOption('debug', 'debug output and symbols', 'no',
               allowed_values=('yes', 'no', 'full'),
               map={}, ignorecase=0),  # case sensitive
    ListOption('shared',
               'libraries to build as shared libraries',
               'all',
               names = list_of_libs),
    PackageOption('x11',
                  'use X11 installed here (yes = search some places)',
                  'yes'),
    PathOption('qtdir', 'where the root of Qt is installed', qtdir),
)
```

Chapter 8. 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 your 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
scons: Reading SConscript files ...
scons: done reading SConscript files.

Type: 'scons program' to build the production program,
     'scons debug' to build the debug version.

Use scons -H for help about command-line options.
```

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 9. 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 -Q
cc -c -o hello.o hello.c
cc -o hello hello.o
% scons -Q /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 -Q
cc -c -o hello.o hello.c
cc -o hello hello.o
% scons -Q 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 -Q install
cc -c -o goodbye.o goodbye.c
cc -o goodbye goodbye.o
Install file: "goodbye" as "/usr/bin/goodbye"
cc -c -o hello.o hello.c
cc -o hello hello.o
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 -Q install
cc -c -o hello.o hello.c
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])
env.Alias('install', '/usr/bin')
```

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

```
% scons -Q install
cc -c -o goodbye.o goodbye.c
cc -o goodbye goodbye.o
```

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

Chapter 10. 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()
lib = env.Library('foo', ['f1.c', 'f2.c', 'f3.c'])
env.Precious(lib)
```

Although the output doesn't look any different, `SCons` does not, in fact, delete the target library before rebuilding it:

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

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

Chapter 11. 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 -Q
cc -c -o prog1/foo1.o prog1/foo1.c
cc -c -o prog1/foo2.o prog1/foo2.c
cc -c -o prog1/main.o prog1/main.c
cc -o prog1/prog1 prog1/main.o prog1/foo1.o prog1/foo2.o
cc -c -o prog2/bar1.o prog2/bar1.c
cc -c -o prog2/bar2.o prog2/bar2.c
cc -c -o prog2/main.o prog2/main.c
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 (where the `SConstruct` file lives) and issues commands that use the path names from the top-level directory to the target and source files within the hierarchy.

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 -Q
cc -c -o lib/foo1.o lib/foo1.c
cc -c -o src/prog/foo2.o src/prog/foo2.c
cc -c -o src/prog/main.o src/prog/main.c
cc -o src/prog/prog src/prog/main.o lib/foo1.o src/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 -Q
cc -c -o src/prog/foo2.o src/prog/foo2.c
cc -c -o src/prog/main.o src/prog/main.c
cc -c -o /usr/joe/lib/foo1.o /usr/joe/lib/foo1.c
cc -o src/prog/prog src/prog/main.o /usr/joe/lib/foo1.o src/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 an `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')
```

You may export more than one variable name at a time:

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

Because white space is not legal in Python variable names, the `Export` function will even automatically split a string into separate names for you:

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

Second, you can specify a list of variables to export 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

Once a variable has been exported from a calling `SConscript` file, it may be used in other `SConscript` files by calling the `Import` function:

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

The `Import` call makes the `env` construction environment available to the `SConscript` file, after which the variable can be used to build programs, libraries, etc.

Like the `Export` function, the `Import` function can be used with multiple variable names:

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

And the `Import` function will similarly split a string along white-space into separate variable names:

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

Lastly, as a special case, you may import all of the variables that have been exported by supplying an asterisk to the `Import` function:

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

If you're dealing with a lot of `SConscript` files, this can be a lot simpler than keeping arbitrary lists of imported variables in each file.

Returning Values From an `SConscript` File

Sometimes, you would like to be able to use information from a subsidiary `SConscript` file in some way. For example, suppose that you want to create one library from source files scattered throughout a number of subsidiary `SConscript` files. You can do this by using the `Return` function to return values from the subsidiary `SConscript` files to the calling file.

If, for example, we have two subdirectories `foo` and `bar` that should each contribute a source file to a Library, what we'd like to be able to do is collect the object files from the subsidiary `SConscript` calls like this:

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

We can do this by using the `Return` function in the `foo/SConscript` file like this:

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

(The corresponding `bar/SConscript` file should be pretty obvious.) Then when we run `SCons`, the object files from the subsidiary subdirectories are all correctly archived in the desired library:

```
% scons -Q
cc -c -o bar/bar.o bar/bar.c
cc -c -o foo/foo.o foo/foo.c
ar r libprog.a foo/foo.o bar/bar.o
ranlib libprog.a
```

Chapter 12. 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 hierarchy 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 src
SConscript  hello.c
% scons -Q
cc -c -o build/hello.o build/hello.c
cc -o build/hello build/hello.o
% ls build
SConscript  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, relative references between files, or tool support for putting files in different locations*, and the `SCons` philosophy is to, by default, guarantee a correct build in all cases.

The most direct reason to duplicate source files in build directories is simply that some tools (mostly older vesions) are written to only build their output files in the same directory as the source files. In this case, the choices are either to build the output file in the source directory and move it to the build directory, or to duplicate the source files in the build directory.

Additionally, relative references between files can cause problems if we don't just duplicate the hierarchy of source files in the build directory. You can see this at work in use of the C preprocessor `#include` mechanism with double quotes, not angle brackets:

```
#include "file.h"
```

The *de facto* standard behavior for most C compilers in this case is to first look in the same directory as the source file that contains the `#include` line, then to look in the directories in the preprocessor search path. Add to this that the `SCons` implementation of support for code repositories (described below) means not all of the files will be found in the same directory hierarchy, and the simplest way to make sure that the right include file is found is to duplicate the source files into the build directory, which provides a correct build regardless of the original location(s) of the source files.

Although source-file duplication guarantees a correct build even in these end-cases, it *can* usually be safely disabled. The next section describes how you can disable the duplication of source files in the build directory.

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

In most cases and with most tool sets, `SCons` can place its target files in a build subdirectory *without* duplicating the source files and everything will work 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 src
SConscript
hello.c
% scons -Q
cc -c src/hello.c -o build/hello.o
cc -o build/hello build/hello.o
% ls 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.

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

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

```
% ls 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 -Q
cc -c -o build/hello.o src/hello.c
cc -o build/hello build/hello.o
% ls 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 `SConstruct` file could look like:

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

Yielding the following output:

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

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

Chapter 13. 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 = ARGUMENTS.get('OS', Platform())

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

env = Environment(PLATFORM = platform,
                  BINDIR = bin,
                  INCDIR = include,
                  LIBDIR = lib,
                  CPPPATH = [include],
                  LIBPATH = [lib],
                  LIBS = 'world')

Export('env')

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 -Q OS=linux
Install file: "build/linux/world/world.h" as "export/linux/include/world.h"
cc -Iexport/linux/include -c -o build/linux/hello/hello.o build/linux/hello/hello.c
cc -Iexport/linux/include -c -o build/linux/world/world.o build/linux/world/world.c
ar r build/linux/world/libworld.a build/linux/world/world.o
ranlib build/linux/world/libworld.a
Install file: "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 file: "build/linux/hello/hello" as "export/linux/bin/hello"
```

The same SConstruct file on Windows would build:

```
C:\>scons -Q OS=windows
Install file: "build/windows/world/world.h" as "export/windows/include/world.h"
cl /nologo /Iexport\windows\include /c build\windows\hello\hello.c /Fobuild\windows\hello\hello.obj
cl /nologo /Iexport\windows\include /c build\windows\world\world.c /Fobuild\windows\world\world.obj
lib /nologo /OUT:build\windows\world\world.lib build\windows\world\world.obj
Install file: "build/windows/world/world.lib" as "export/windows/lib/world.lib"
link /nologo /OUT:build\windows\hello\hello.exe /LIBPATH:export\windows\lib world.lib build\windows\hello\hello.obj
Install file: "build/windows/hello/hello.exe" as "export/windows/bin/hello.exe"
```

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 < $SOURCE > $TARGET')
```

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 < $SOURCE > $TARGET')
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 -Q
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 Builders are no longer part of the environment:

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

```
% scons -Q
AttributeError: SConsEnvironment 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 < $SOURCE > $TARGET')
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 < $SOURCE > $TARGET')
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 -Q
foobuild < file.input > file.foo
cc -c -o hello.o hello.c
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 < $SOURCE > $TARGET',
              suffix = '.foo',
              src_suffix = '.input')
env = Environment(BUILDERS = {'Foo' : bld})
env.Foo('file1')
env.Foo('file2')
```

```
% scons -Q
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):
    # Code to build "target" from "source"
    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` function.

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` function.

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):
    # Code to build "target" from "source"
    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 -Q
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 to construct just the right list of commands to build your target. A `generator` looks like:

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

```
        return 'foobuild < %s > %s' % (target[0], source[0])
```

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` function.

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` function.

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.

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`.

```
def generate_actions(source, target, env, for_signature):
    return 'foobuild < %s > %s' % (source[0], target[0])
bld = Builder(generator = generate_actions,
              suffix = '.foo',
              src_suffix = '.input')
env = Environment(BUILDERS = {'Foo' : bld})
env.Foo('file')
```

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

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(target, source, env):
    target.append('new_target')
    source.append('new_source')
    return target, source
bld = Builder(action = 'foobuild $TARGETS - $SOURCES',
              suffix = '.foo',
              src_suffix = '.input',
              emitter = modify_targets)
env = Environment(BUILDERS = {'Foo' : bld})
env.Foo('file')
```

% **scons -Q**
foobuild file.foo new_target - file.input new_source

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

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/' < $SOURCE > $TARGET")
```

```
% scons -Q
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):
    # Whatever it takes to build
    return None
env.Command('foo.out', 'foo.in', build)
```

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

Chapter 16. Writing Scanners

SCons has built-in scanners that know how to look in C, Fortran and IDL source files for information about other files that targets built from those files depend on--for example, in the case of files that use the C preprocessor, the .h files that are specified using #include lines in the source. You can use the same mechanisms that SCons uses to create its built-in scanners to write scanners of your own for file types that SCons does not know how to scan "out of the box."

A Simple Scanner Example

Suppose, for example, that we want to create a simple scanner for .foo files. A .foo file contains some text that will be processed, and can include other files on lines that begin with include followed by a file name:

```
include filename.foo
```

Scanning a file will be handled by a Python function that you must supply. Here is a function that will use the Python re module to scan for the include lines in our example:

```
import re

include_re = re.compile(r'^include\\s+(\\S+)$', re.M)

def kfile_scan(node, env, path, arg):
    contents = node.get_contents()
    return include_re.findall(contents)
```

The scanner function must accept the four specified arguments and return a list of implicit dependencies. Presumably, these would be dependencies found from examining the contents of the file, although the function can perform any manipulation at all to generate the list of dependencies.

node

An SCons node object representing the file being scanned. The path name to the file can be used by converting the node to a string using the str() function, or an internal SCons get_contents() object method can be used to fetch the contents.

env

The construction environment in effect for this scan. The scanner function may choose to use construction variables from this environment to affect its behavior.

path

A list of directories that form the search path for included files for this scanner. This is how SCons handles the CPPPATH and LIBPATH variables.

arg

An optional argument that you can choose to have passed to this scanner function by various scanner instances.

A Scanner object is created using the Scanner function, which typically takes an skeys argument to associate the type of file suffix with this scanner. The Scanner object must then be associated with the SCANNERS construction variable of a construction environment, typically by using the Append method:

```
kscan = Scanner(function = kfile_scan,
```

```
                skeys = ['.k'])
env.Append(SCANNERS = kscan)
```

When we put it all together, it looks like:

```
import re

include_re = re.compile(r'^include\\s+(\\S+)$', re.M)

def kfile_scan(node, env, path):
    contents = node.get_contents()
    includes = include_re.findall(contents)
    return includes

kscan = Scanner(function = kfile_scan,
                skeys = ['.k'])

env = Environment(ENV = {'PATH' : '/usr/local/bin'})
env.Append(SCANNERS = kscan)

env.Command('foo', 'foo.k', 'kprocess < $SOURCES > $TARGET')
```

Chapter 17. 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. 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 -Q
cc -c -o hello.o hello.c
cc -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 -Q
cc -c -o hello.o hello.c
cc -o hello hello.o
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 -Q
cc -c -o hello.o hello.c
```

```
cc -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 -Q -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 -Q
cc -c -o file1.o file1.c
cc -c -o file2.o file2.c
cc -c -o hello.o hello.c
cc -o hello hello.o file1.o file2.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 -Q -Y /usr/repository1
cc -c -o hello.o hello.c
cc -o hello hello.o /usr/repository1/file1.o /usr/repository1/file2.o
```

Notice that `SCons` realizes that it does not need to rebuild local copies file1.o and file2.o files, but instead uses the already-compiled files 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 -Q -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.
```

(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.)

Chapter 18. 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 -Q
cc -c -o hello.o hello.c
cc -o hello hello.o
% scons -Q -c
Removed hello.o
Removed hello
% scons -Q
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 -Q
cc -c -o hello.o hello.c
cc -o hello hello.o
% scons -Q -c
Removed hello.o
Removed hello
% scons -Q --cache-show
cc -c -o hello.o hello.c
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 -Q
cc -c -o hello.o hello.c
cc -o hello hello.o
% scons -Q -c
Removed hello.o
Removed hello
% scons -Q
Retrieved 'hello.o' from cache
Retrieved 'hello' from cache
% scons -Q -c
Removed hello.o
Removed hello
% scons -Q --cache-disable
cc -c -o hello.o hello.c
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 -Q --cache-disable
cc -c -o hello.o hello.c
cc -o hello hello.o
% scons -Q -c
Removed hello.o
Removed hello
% scons -Q --cache-disable
cc -c -o hello.o hello.c
cc -o hello hello.o
% scons -Q --cache-force
scons: '.' is up to date.
% scons -Q -c
Removed hello.o
Removed hello
```

% **scons -Q**
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 19. 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 -Q install
cc -c -o hello.o hello.c
cc -o hello hello.o
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('foo.c')
l = env.Library('bar.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 -Q install-bin
cc -c -o foo.o foo.c
cc -o foo foo.o
Install file: "foo" as "/usr/bin/foo"
% scons -Q install-lib
cc -c -o bar.o bar.c
ar r libbar.a bar.o
ranlib libbar.a
Install file: "libbar.a" as "/usr/lib/libbar.a"
% scons -Q -c /
Removed foo.o
Removed foo
Removed /usr/bin/foo
Removed bar.o
Removed libbar.a
Removed /usr/lib/libbar.a
% scons -Q install
cc -c -o foo.o foo.c
cc -o foo foo.o
Install file: "foo" as "/usr/bin/foo"
cc -c -o bar.o bar.c
ar r libbar.a bar.o
ranlib libbar.a
```

```
Install file: "libbar.a" as "/usr/lib/libbar.a"
```

Appendix A. Handling Common Tasks

There is a common set of simple tasks that many build configurations rely on as they become more complex. Most build tools have special purpose constructs for performing these tasks, but since `SConscript` files are `Python` scripts, you can use more flexible built-in `Python` services to perform these tasks. This appendix lists a number of these tasks and how to implement them in `Python`.

Example A-1. Wildcard globbing to create a list of filenames

```
import glob
files = glob.glob(wildcard)
```

Example A-2. Filename extension substitution

```
import os.path
filename = os.path.splitext(filename)[0]+extension
```

Example A-3. Appending a path prefix to a list of filenames

```
import os.path
filenames = [os.path.join(prefix, x) for x in filenames]
```

or in Python 1.5.2:

```
import os.path
new_filenames = []
for x in filenames:
    new_filenames.append(os.path.join(prefix, x))
```

Example A-4. Substituting a path prefix with another one

```
if filename.find(old_prefix) == 0:
    filename = filename.replace(old_prefix, new_prefix)
```

or in Python 1.5.2:

```
import string
if string.find(filename, old_prefix) == 0:
    filename = string.replace(filename, old_prefix, new_prefix)
```

Example A-5. Filtering a filename list to exclude/retain only a specific set of extensions

```
import os.path
filenames = [x for x in filenames if os.path.splitext(x)[1] in extensions]
```

or in Python 1.5.2:

```
import os.path
new_filenames = []
for x in filenames:
    if os.path.splitext(x)[1] in extensions:
        new_filenames.append(x)
```

Example A-6. The "backtick function": run a shell command and capture the output

```
import os
output = os.popen(command).read()
```