

SCons User Guide 0.96.95

Steven Knight

SCons User Guide 0.96.95

by Steven Knight

Revision 0.96.95.D001 (2007/02/12 21:41:50) Edition

Published 2004, 2005, 2006, 2007

Copyright © 2004, 2005, 2006, 2007 Steven Knight

SCons User's Guide Copyright (c) 2004, 2005, 2006, 2007 Steven Knight

Table of Contents

Preface	i
SCons Principles	i
A Caveat About This Guide's Completeness.....	i
Acknowledgements	ii
Contact	ii
1. Building and Installing SCons	1
Installing Python	1
Installing SCons From Pre-Built Packages.....	1
Installing SCons on Red Hat (and Other RPM-based) Linux Systems	1
Installing SCons on Debian Linux Systems	2
Installing SCons on Windows Systems.....	2
Building and Installing SCons on Any System	2
Building and Installing Multiple Versions of SCons Side-by-Side	3
Installing SCons in Other Locations.....	3
Building and Installing SCons Without Administrative Privileges.....	3
2. Simple Builds	5
Building Simple C / C++ Programs	5
Building Object Files	6
Simple Java Builds.....	6
Cleaning Up After a Build	6
The SConstruct File.....	7
SConstruct Files Are Python Scripts	7
SCons Functions Are Order-Independent.....	8
Making the SCons Output Less Verbose	8
3. Less Simple Things to Do With Builds	11
Specifying the Name of the Target (Output) File.....	11
Compiling Multiple Source Files	11
Specifying Single Files Vs. Lists of Files	12
Making Lists of Files Easier to Read.....	13
Keyword Arguments	13
Compiling Multiple Programs	14
Sharing Source Files Between Multiple Programs	14
4. Building and Linking with Libraries	17
Building Libraries.....	17
Building Static Libraries Explicitly: the StaticLibrary Builder.....	17
Building Shared (DLL) Libraries: the SharedLibrary Builder	17
Linking with Libraries.....	18
Finding Libraries: the \$LIBPATH Construction Variable	19
5. Node Objects	21
Builder Methods Return Lists of Target Nodes	21
Explicitly Creating File and Directory Nodes	21
Printing Node File Names.....	22
Using a Node's File Name as a String	23
6. Dependencies	25
Deciding When a Source File Has Changed: the SourceSignatures Function	
25	
MD5 Source File Signatures	25
Source File Time Stamps.....	26
Deciding When a Target File Has Changed: the TargetSignatures Function	26
Build Signatures.....	26
File Contents.....	27
Implicit Dependencies: The \$CPPPATH Construction Variable	27
Caching Implicit Dependencies	29
The --implicit-deps-changed Option	29
The --implicit-deps-unchanged Option.....	29

Ignoring Dependencies: the <code>Ignore</code> Method	30
Explicit Dependencies: the <code>Depends</code> Method	30
7. Construction Environments	33
Multiple Construction Environments	33
Copying Construction Environments	34
Fetching Values From a Construction Environment	35
Expanding Values From a Construction Environment	36
Modifying a Construction Environment	37
Replacing Values in a Construction Environment	37
Appending to the End of Values in a Construction Environment	38
Appending to the Beginning of Values in a Construction Environment	38
8. Controlling the External Environment Used to Execute Build Commands	41
Propagating <code>PATH</code> From the External Environment	41
9. Controlling a Build From the Command Line	43
Not Having to Specify Command-Line Options Each Time: the <code>SCONSFLAGS</code> Environment Variable	43
Getting at Command-Line Targets	43
Controlling the Default Targets	44
Getting at the List of Default Targets	45
Getting at the List of Build Targets, Regardless of Origin	46
Command-Line <code>variable=value</code> Build Options	47
Controlling Command-Line Build Options	48
Providing Help for Command-Line Build Options	49
Reading Build Options From a File	49
Canned Build Options	50
True/False Values: the <code>BoolOption</code> Build Option	50
Single Value From a List: the <code>EnumOption</code> Build Option	51
Multiple Values From a List: the <code>ListOption</code> Build Option	53
Path Names: the <code>PathOption</code> Build Option	53
Enabled/Disabled Path Names: the <code>PackageOption</code> Build Option	55
Adding Multiple Command-Line Build Options at Once	55
10. Providing Build Help: the <code>Help</code> Function	57
11. Installing Files in Other Directories: the <code>Install</code> Builder	59
Installing Multiple Files in a Directory	59
Installing a File Under a Different Name	60
Installing Multiple Files Under Different Names	60
12. Platform-Independent File System Manipulation	63
Copying Files or Directories: The <code>Copy</code> Factory	63
Deleting Files or Directories: The <code>Delete</code> Factory	63
Moving (Renaming) Files or Directories: The <code>Move</code> Factory	64
Updating the Modification Time of a File: The <code>Touch</code> Factory	65
Creating a Directory: The <code>Mkdir</code> Factory	65
Changing File or Directory Permissions: The <code>Chmod</code> Factory	65
Executing an action immediately: the <code>Execute</code> Function	66
13. Preventing Removal of Targets	67
Preventing target removal during build: the <code>Precious</code> Function	67
Preventing target removal during clean: the <code>NoClean</code> Function	67
14. Hierarchical Builds	69
<code>SConscript</code> Files	69
Path Names Are Relative to the <code>SConscript</code> Directory	69
Top-Level Path Names in Subsidiary <code>SConscript</code> Files	70
Absolute Path Names	70
Sharing Environments (and Other Variables) Between <code>SConscript</code> Files	71
Exporting Variables	71
Importing Variables	72

Returning Values From an <code>SConscript</code> File.....	72
15. Separating Source and Build Directories.....	75
Specifying a Build Directory as Part of an <code>SConscript</code> Call	75
Why <code>SCons</code> Duplicates Source Files in a Build Directory	75
Telling <code>SCons</code> to Not Duplicate Source Files in the Build Directory	76
The <code>BuildDir</code> Function	76
Using <code>BuildDir</code> With an <code>SConscript</code> File	77
16. Variant Builds	79
17. Writing Your Own Builders	81
Writing Builders That Execute External Commands	81
Attaching a Builder to a Construction Environment.....	81
Letting <code>SCons</code> Handle The File Suffixes.....	82
Builders That Execute Python Functions.....	82
Builders That Create Actions Using a Generator.....	83
Builders That Modify the Target or Source Lists Using an <code>Emitter</code>	84
18. Not Writing a Builder: the <code>Command</code> Builder	87
19. Writing Scanners	89
A Simple Scanner Example.....	89
20. Building From Code Repositories	91
The <code>Repository</code> Method	91
Finding source files in repositories.....	91
Finding <code>#include</code> files in repositories	91
Limitations on <code>#include</code> files in repositories.....	92
Finding the <code>SConstruct</code> file in repositories.....	93
Finding derived files in repositories.....	93
Guaranteeing local copies of files	94
21. Multi-Platform Configuration (<code>Autoconf</code> Functionality).....	97
Configure Contexts.....	97
Checking for the Existence of Header Files.....	97
Checking for the Availability of a Function	98
Checking for the Availability of a Library	98
Checking for the Availability of a <code>typedef</code>	98
Adding Your Own Custom Checks	99
Not Configuring When Cleaning Targets.....	100
22. Caching Built Files	103
Specifying the Shared Cache Directory.....	103
Keeping Build Output Consistent	103
Not Retrieving Files From a Shared Cache.....	104
Populating a Shared Cache With Already-Built Files	104
23. Alias Targets.....	107
24. Java Builds.....	109
Building Java Class Files: the <code>Java</code> Builder	109
How <code>SCons</code> Handles Java Dependencies.....	109
Building Java Archive (<code>.jar</code>) Files: the <code>Jar</code> Builder	110
Building C Header and Stub Files: the <code>JavaH</code> Builder	110
Building RMI Stub and Skeleton Class Files: the <code>RMIC</code> Builder	111
25. Troubleshooting	113
Why is That Target Being Rebuilt? the <code>--debug=explain</code> Option.....	113
What's in That Construction Environment? the <code>Dump</code> Method	114
A. Construction Variables	119
B. Builders	159
C. Tools.....	169
D. Handling Common Tasks	175

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.

A Caveat About This Guide's Completeness

One word of warning as you read through this Guide: Like too much Open Source software out there, the `SCons` documentation isn't always kept up-to-date with the available features. In other words, there's a lot that `SCons` can do that isn't yet covered in this User's Guide. (Come to think of it, that also describes a lot of proprietary software, doesn't it?)

Although this User's Guide isn't as complete as we'd like it to be, our development process does emphasize making sure that the `SCons` man page is kept up-to-date with new features. So if you're trying to figure out how to do something that `SCons`

supports but can't find enough (or any) information here, it would be worth your while to look at the man page to see if the information is covered there. And if you do, maybe you'd even consider contributing a section to the User's Guide so the next person looking for that information won't have to go through the same thing...?

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 Oberbrunner, 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 users@scons.tigris.org.

If you want to contact the SCons development community directly, send email to dev@scons.tigris.org.

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

Chapter 1. Building and Installing sCons

This chapter will take you through the basic steps of installing sCons on your system, and building sCons if you don't have a pre-built package available (or simply prefer the flexibility of building it yourself). Before that, however, this chapter will also describe the basic steps involved in installing Python on your system, in case that is necessary. Fortunately, both sCons and Python are very easy to install on almost any system, and Python already comes installed on many systems.

Installing Python

Because sCons is written in Python, you must obviously have Python installed on your system to use sCons. Before you try to install Python, you should check to see if Python is already available on your system by typing **python** at your system's command-line prompt. You should see something like the following on a UNIX or Linux system that has Python installed:

```
$ python
Python 2.2.2 (#1, Feb 24 2003, 19:13:11)
[GCC 3.2.2 20030222 (Red Hat Linux 3.2.2-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> ^D
```

And on a Windows system with Python installed:

```
C:\>python
Python 2.2.2 (#34, Apr 9 2002, 19:34:33) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> ^Z
```

The >>> is the input prompt for the Python interpreter. The ^D and ^Z represent the CTRL-D and CTRL-Z characters that you will need to type to get out of the interpreter before proceeding to installing sCons.

If Python is not installed on your system, you will see an error message stating something like "command not found" (on UNIX or Linux) or "'python' is not recognized as an internal or external command, operable program or batch file" (on Windows). In that case, you need to install Python before you can install sCons.

The standard location for information about downloading and installing Python is <http://www.python.org/download/>. See that page for information about how to download and install Python on your system.

Installing sCons From Pre-Built Packages

sCons comes pre-packaged for installation on a number of systems, including Linux and Windows systems. You do not need to read this entire section, you should only need to read the section appropriate to the type of system you're running on.

Installing sCons on Red Hat (and Other RPM-based) Linux Systems

sCons comes in RPM (Red Hat Package Manager) format, pre-built and ready to install on Red Hat Linux, Fedora Core, or any other Linux distribution that uses RPM. Your distribution may already have an sCons RPM built specifically for it; many do, including SuSe, Mandrake and Fedora. You can check for the availability of an sCons

RPM on your distribution's download servers, or by consulting an RPM search site like <http://www.rpmfind.net/> or <http://rpm.pbone.net/>.

If your Linux distribution does not already have a specific SCons RPM file, you can download and install from the generic RPM provided by the SCons project. This will install the SCons script(s) in `/usr/bin`, and the SCons library modules in `/usr/lib/scons`.

To install from the command line, simply download the appropriate `.rpm` file, and then run:

```
# rpm -Uvh scons-0.96-1.noarch.rpm
```

Or, you can use a graphical RPM package manager like `gnorpm`. See your package manager application's documentation for specific instructions about how to use it to install a downloaded RPM.

Installing sCons on Debian Linux Systems

Debian Linux systems use a different package management format that also makes it very easy to install SCons.

If your system is connected to the Internet, you can install the latest official Debian package by running:

```
# apt-get install scons
```

Installing sCons on Windows Systems

SCons provides a Windows installer that makes installation extremely easy. Download the `scons-0.95.win32.exe` file from the SCons download page at <http://www.scons.org/download.html>. Then all you need to do is execute the file (usually by clicking on its icon in Windows Explorer). These will take you through a small sequence of windows that will install SCons on your system.

Building and Installing sCons on Any System

If a pre-built SCons package is not available for your system, then you can still easily build and install SCons using the native Python `distutils` package.

The first step is to download either the `scons-0.96.95.tar.gz` or `scons-0.96.95.zip`, which are available from the SCons download page at <http://www.scons.org/download.html>.

Unpack the archive you downloaded, using a utility like `tar` on Linux or UNIX, or `winZip` on Windows. This will create a directory called `scons-0.96.95`, usually in your local directory. Then change your working directory to that directory and install SCons by executing the following commands:

```
# cd scons-0.96.95
# python setup.py install
```

This will build SCons, install the `scons` script in the default system scripts directory (`/usr/local/bin` or `C:\Python2.2\Scripts`), and will install the SCons build engine in an appropriate stand-alone library directory (`/usr/local/lib/scons` or

C:\Python2.2\scons). Because these are system directories, you may need root (on Linux or UNIX) or Administrator (on Windows) privileges to install SCons like this.

Building and Installing Multiple Versions of SCons Side-by-Side

The SCons `setup.py` script has some extensions that support easy installation of multiple versions of SCons in side-by-side locations. This makes it easier to download and experiment with different versions of SCons before moving your official build process to a new version, for example.

To install SCons in a version-specific location, add the `--version-lib` option when you call `setup.py`:

```
# python setup.py install --version-lib
```

This will install the SCons build engine in the `/usr/lib/scons-0.96.95` or `C:\Python2.2\scons-0.96.95` directory, for example.

If you use the `--version-lib` option the first time you install SCons, you do not need to specify it each time you install a new version. The SCons `setup.py` script will detect the version-specific directory name(s) and assume you want to install all versions in version-specific directories. You can override that assumption in the future by explicitly specifying the `--standalone-lib` option.

Installing SCons in Other Locations

You can install SCons in locations other than the default by specifying the `--prefix=` option:

```
# python setup.py install --prefix=/opt/scons
```

This would install the `scons` script in `/opt/scons/bin` and the build engine in `/opt/scons/lib/scons`,

Note that you can specify both the `--prefix=` and the `--version-lib` options at the same time, in which case `setup.py` will install the build engine in a version-specific directory relative to the specified prefix. Adding `--version-lib` to the above example would install the build engine in `/opt/scons/lib/scons-0.96.95`.

Building and Installing SCons Without Administrative Privileges

If you don't have the right privileges to install SCons in a system location, simply use the `--prefix=` option to install it in a location of your choosing. For example, to install SCons in appropriate locations relative to the user's `$HOME` directory, the `scons` script in `$HOME/bin` and the build engine in `$HOME/lib/scons`, simply type:

```
$ python setup.py install --prefix=$HOME
```

You may, of course, specify any other location you prefer, and may use the `--version-lib` option if you would like to install version-specific directories relative to the specified prefix.

Notes

1. <http://www.python.org/download/>

2. <http://www.rpmfind.net/>
3. <http://rpm.pbone.net/>
4. <http://www.scons.org/download.html>
5. <http://www.scons.org/download.html>

Chapter 2. Simple Builds

In this chapter, you will see several examples of very simple build configurations using `SCons`, which will demonstrate how easy it is to use `SCons` to build programs from several different programming languages on different types of systems.

Building Simple C / C++ Programs

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')
```

This minimal configuration file gives `SCons` two pieces of information: what you want to build (an executable program), and the input file from which you want it built (the `hello.c` file). `Program` is a *builder_method*, a Python call that tells `SCons` that you want to build an executable program.

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 -o hello.o -c 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` correctly deduces the names of the object and executable files to be built 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.)

Building Object Files

The `Program` builder method is only one of many builder methods that `SCons` provides to build different types of files. Another is the `Object` builder method, which tells `SCons` to build an object file from the specified source file:

```
Object('hello.c')
```

Now when you run the `scons` command to build the program, it will build just the `hello.o` object file on a POSIX system:

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

And just the `hello.obj` object file on a Windows system (with the Microsoft Visual C++ compiler):

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

Simple Java Builds

`SCons` also makes building with Java extremely easy. Unlike the `Program` and `Object` builder methods, however, the `Java` builder method requires that you specify the name of a destination directory in which you want the class files placed, followed by the source directory in which the `.java` files live:

```
Java('classes', 'src')
```

If the `src` directory contains a single `hello.java` file, then the output from running the `scons` command would look something like this (on a POSIX system):

```
% scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
javac -d classes -sourcepath src src/hello.java
scons: done building targets.
```

We'll cover Java builds in more detail, including building Java archive (`.jar`) and other types of file, in Chapter 24.

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 -o hello.o -c 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.

`SConstruct` Files Are Python Scripts

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.

SCons Functions Are Order-Independent

One important way in which the SConstruct file is not exactly like a normal Python script, and is more like a Makefile, is that the order in which the SCons functions are called in the SConstruct file does *not* affect the order in which SCons actually builds the programs and object files you want it to build.¹ In other words, when you call the Program builder (or any other builder method), you're not telling SCons to build the program at the instant the builder method is called. Instead, you're telling SCons to build the program that you want, for example, a program built from a file named `hello.c`, and it's up to SCons to build that program (and any other files) whenever it's necessary. (We'll learn more about how SCons decides when building or rebuilding a file is necessary in Chapter 6, below.)

SCons reflects this distinction between *calling a builder method like* `Program()` and *actually building the program* by printing the status messages that indicate when it's "just reading" the SConstruct file, and when it's actually building the target files. This is to make it clear when SCons is executing the Python statements that make up the SConstruct file, and when SCons is actually executing the commands or other actions to build the necessary files.

Let's clarify this with an example. Python has a `print` statement that prints a string of characters to the screen. If we put `print` statements around our calls to the Program builder method:

```
print "Calling Program('hello.c')"
```

`Program('hello.c')`

```
print "Calling Program('goodbye.c')"
```

`Program('goodbye.c')`

```
print "Finished calling Program()"
```

Then when we execute SCons, we see the output from the `print` statements in between the messages about reading the SConstruct files, indicating that that is when the Python statements are being executed:

```
% scons
scons: Reading SConscript files ...
Calling Program('hello.c')
Calling Program('goodbye.c')
Finished calling Program()
scons: done reading SConscript files.
scons: Building targets ...
cc -o goodbye.o -c goodbye.c
cc -o goodbye goodbye.o
cc -o hello.o -c hello.c
cc -o hello hello.o
scons: done building targets.
```

Notice also that SCons built the `goodbye` program first, even though the "reading SConstruct" output shows that we called `Program('hello.c')` first in the SConstruct file.

Making the sCons 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: all of 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.

One 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 to use the `-Q` option to remove these messages from the output of all the remaining examples in this Guide.

Notes

1. In programming parlance, the SConstruct file is *declarative*, meaning you tell sCons what you want done and let it figure out the order in which to do it, rather than strictly *imperative*, where you specify explicitly the order in which to do things.

Chapter 3. Less Simple Things to Do With Builds

In this chapter, you will see several examples of very simple build configurations using `SCons`, which will demonstrate how easy it is to use `SCons` to build programs from several different programming languages on different types of systems.

Specifying the Name of the Target (Output) File

You've seen that when you call the `Program` builder method, it builds the resulting program with the same base name as the source file. That is, the following call to build an executable program from the `hello.c` source file will build an executable program named `hello` on POSIX systems, and an executable program named `hello.exe` on Windows systems:

```
Program('hello.c')
```

If you want to build a program with a different name than the base of the source file name, you simply put the target file name to the left of the source file name:

```
Program('new_hello', 'hello.c')
```

(`SCons` requires the target file name first, followed by the source file name, so that the order mimics that of an assignment statement in most programming languages, including Python: `program = source files`.)

Now `SCons` will build an executable program named `new_hello` when run on a POSIX system:

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

And `SCons` will build an executable program named `new_hello.exe` when run on a Windows system:

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

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

A build of the above example would look like:

```
% scons -Q
cc -o file1.o -c file1.c
cc -o file2.o -c file2.c
cc -o main.o -c main.c
cc -o main main.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 (as we’ve seen in the previous section) 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 -o file1.o -c file1.c
cc -o file2.o -c file2.c
cc -o main.o -c 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
```

Specifying Single Files Vs. Lists of Files

We’ve now shown you two ways to specify the source for a program, one with a list of files:

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

And one with a single file:

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

You could actually put a single file name in a list, too, which you might prefer just for the sake of consistency:

```
Program('hello', ['hello.c'])
```

`SCons` functions will accept a single file name in either form. In fact, internally, `SCons` treats all input as lists of files, but allows you to omit the square brackets to cut down a little on the typing when there’s only a single file name.

Important: Although `SCons` functions are forgiving about whether or not you use a string vs. a list for a single file name, Python itself is more strict about treating lists and strings differently. So where `SCons` allows either a string or list:

```
# The following two calls both work correctly:
Program('program1', 'program1.c')
Program('program2', ['program2.c'])
```

Trying to do "Python things" that mix strings and lists will cause errors or lead to incorrect results:

```
common_sources = ['file1.c', 'file2.c']

# THE FOLLOWING IS INCORRECT AND GENERATES A PYTHON ERROR
# BECAUSE IT TRIES TO ADD A STRING TO A LIST:
Program('program1', common_sources + 'program1.c')

# The following works correctly, because it's adding two
# lists together to make another list.
Program('program2', common_sources + ['program2.c'])
```

Making Lists of Files Easier 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)
```

(Note in this example that we used the Python "triple-quote" syntax, which allows a string to contain multiple lines. The three quotes can be either single or double quotes.)

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 -o bar1.o -c bar1.c
cc -o bar2.o -c bar2.c
cc -o bar bar1.o bar2.o
cc -o foo.o -c 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 Chapter 4, 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 -o bar1.o -c bar1.c
cc -o bar2.o -c bar2.c
cc -o common1.o -c common1.c
cc -o common2.o -c common2.c
cc -o bar bar1.o bar2.o common1.o common2.o
cc -o foo.o -c 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 4. 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 -o f1.o -c f1.c
cc -o f2.o -c f2.c
cc -o f3.o -c f3.c
ar rc 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 `StaticLibrary` Builder

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: the `SharedLibrary` Builder

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 -o f1.os -c f1.c
cc -o f2.os -c f2.c
cc -o f3.os -c f3.c
cc -o libfoo.so -shared 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
RegServerFunc(target, source, env)
```

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', 'bar'], 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 -o f1.o -c f1.c
cc -o f2.o -c f2.c
cc -o f3.o -c f3.c
ar rc libfoo.a f1.o f2.o f3.o
ranlib libfoo.a
cc -o prog.o -c prog.c
cc -o prog prog.o -L. -lfoo -lbar
```

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 /Foprogram.obj
link /nologo /OUT:prog.exe /LIBPATH:. foo.lib bar.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.

Note also that, if you only have a single library to link with, you can specify the library name in single string, instead of a Python list, so that:

```
Program('prog.c', LIBS='foo', LIBPATH='.')
```

is equivalent to:

```
Program('prog.c', LIBS=['foo'], LIBPATH='.')
```

This is similar to the way that SCons handles either a string or a list to specify a single source file.

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'
```

(Note that Python requires that the backslash separators in a Windows path name be escaped within strings.)

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 -o prog.o -c 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 /Foprogram.obj
link /nologo /OUT:prog.exe /LIBPATH:\usr\lib /LIBPATH:\usr\local\lib m.lib prog.o
```

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

Chapter 5. Node Objects

Internally, `SCons` represents all of the files and directories it knows about as `Nodes`. These internal objects (not object *files*) can be used in a variety of ways to make your `SConstruct` files portable and easy to read.

Builder Methods Return Lists of Target Nodes

All builder methods return a list of `Node` objects that identify the target file or files that will be built. These returned `Nodes` can be passed as source files to other builder methods,

For example, suppose that we want to build the two object files that make up a program with different options. This would mean calling the `Object` builder once for each object file, specifying the desired options:

```
Object('hello.c', CCFLAGS='-DHELLO')
Object('goodbye.c', CCFLAGS='-DGOODBYE')
```

One way to combine these object files into the resulting program would be to call the `Program` builder with the names of the object files listed as sources:

```
Object('hello.c', CCFLAGS='-DHELLO')
Object('goodbye.c', CCFLAGS='-DGOODBYE')
Program(['hello.o', 'goodbye.o'])
```

The problem with listing the names as strings is that our `SConstruct` file is no longer portable across operating systems. It won't, for example, work on Windows because the object files there would be named `hello.obj` and `goodbye.obj`, not `hello.o` and `goodbye.o`.

A better solution is to assign the lists of targets returned by the calls to the `Object` builder to variables, which we can then concatenate in our call to the `Program` builder:

```
hello_list = Object('hello.c', CCFLAGS='-DHELLO')
goodbye_list = Object('goodbye.c', CCFLAGS='-DGOODBYE')
Program(hello_list + goodbye_list)
```

This makes our `SConstruct` file portable again, the build output on Linux looking like:

```
% scons -Q
cc -o goodbye.o -c -DGOODBYE goodbye.c
cc -o hello.o -c -DHELLO hello.c
cc -o hello hello.o goodbye.o
```

And on Windows:

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

We'll see examples of using the list of nodes returned by builder methods throughout the rest of this guide.

Explicitly Creating File and Directory Nodes

It's worth mentioning here that `SCons` maintains a clear distinction between Nodes that represent files and Nodes that represent directories. `SCons` supports `File` and `Dir` functions that, respectively, return a file or directory Node:

```
hello_c = File('hello.c')
Program(hello_c)

classes = Dir('classes')
Java(classes, 'src')
```

Normally, you don't need to call `File` or `Dir` directly, because calling a builder method automatically treats strings as the names of files or directories, and translates them into the Node objects for you. The `File` and `Dir` functions can come in handy in situations where you need to explicitly instruct `SCons` about the type of Node being passed to a builder or other function, or unambiguously refer to a specific file in a directory tree.

There are also times when you may need to refer to an entry in a file system without knowing in advance whether it's a file or a directory. For those situations, `SCons` also supports an `Entry` function, which returns a Node that can represent either a file or a directory.

```
xyzzzy = Entry('xyzzzy')
```

The returned `xyzzzy` Node will be turned into a file or directory Node the first time it is used by a builder method or other function that requires one vs. the other.

Printing Node File Names

One of the most common things you can do with a Node is use it to print the file name that the node represents. For example, the following `SConstruct` file:

```
hello_c = File('hello.c')
Program(hello_c)

classes = Dir('classes')
Java(classes, 'src')

object_list = Object('hello.c')
program_list = Program(object_list)
print "The object file is:", object_list[0]
print "The program file is:", program_list[0]
```

Would print the following file names on a POSIX system:

```
% scons -Q
The object file is: hello.o
The program file is: hello
cc -o hello.o -c hello.c
cc -o hello hello.o
```

And the following file names on a Windows system:

```
C:\>scons -Q
The object file is: hello.obj
The program file is: hello.exe
cl /nologo /c hello.c /Fohello.obj
```



```
link /nologo /OUT:hello.exe hello.obj
```

Using a Node's File Name as a String

Printing a Node's name as described in the previous section works because the string representation of a Node is the name of the file. If you want to do something other than print the name of the file, you can fetch it by using the builtin Python `str` function. For example, if you want to use the Python `os.path.exists` to figure out whether a file exists while the SConstruct file is being read and executed, you can fetch the string as follows:

```
import os.path
program_list = Program('hello.c')
program_name = str(program_list[0])
if not os.path.exists(program_name):
    print program_name, "does not exist!"
```

Which executes as follows on a POSIX system:

```
% scons -Q
hello does not exist!
cc -o hello.o -c hello.c
cc -o hello hello.o
```


Chapter 6. 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 by re-invoking `SCons` after building our simple `hello` example:

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

Deciding When a Source File Has Changed: the `sourceSignatures` Function

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 -o hello.o -c 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 -o hello.o -c hello.c
```

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

Deciding When a Target File Has Changed: the `TargetSignatures` Function

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

```
cc -o hello.o -c 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 -o hello.o -c hello.c
cc -o hello hello.o
% edit hello.c
[CHANGE A COMMENT IN hello.c]
% scons -Q hello
cc -o hello.o -c 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 -o hello.o -c -I. 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 -o hello.o -c -I. 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 -o hello.o -c -Iinclude -I/home/project/inc 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 -o hello.o -c 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 -o hello.o -c 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 -o hello.o -c 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.

Ignoring Dependencies: 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 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')
```

Explicit Dependencies: 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 specify 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 7. 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 accommodates 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:

```
env = Environment()
```

By default, SCons initializes every new construction environment with a set of construction variables based on the tools that it finds on your system, plus the default set of builder methods necessary for using those tools. The construction variables are initialized with values describing the C compiler, the Fortran compiler, the linker, etc., as well as the command lines to invoke them.

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')
```

The construction environment in this example is still initialized with the same default construction variable values, except that the user has explicitly specified 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. In other words, the explicit initializations of `$CC` and `$CCFLAGS` override the default values in the newly-created construction environment. So a run from this example would look like:

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

```
cc -o bar bar.o
cc -o foo.o -c -O2 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.
File "/home/my/project/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` builder, 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 -o foo-dbg.o -c -g foo.c
cc -o foo-dbg foo-dbg.o
cc -o foo-opt.o -c -O2 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 `Clone` method to create a copy of a construction environment.

Like the `Environment` call that creates a construction environment, the `Clone` 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.Clone(CCFLAGS = '-O2')
dbg = env.Clone(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 -o foo.o -c foo.c
gcc -o foo foo.o
gcc -o foo-dbg.o -c -g foo.c
gcc -o foo-dbg foo-dbg.o
gcc -o foo-opt.o -c -O2 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 an 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 ['OBSUFFIX', '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 Windows:

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

If you want to loop through and print the values of all of the construction variables in a construction environment, the Python code to do that in sorted order might look something like:

```
env = Environment()
dict = env.Dictionary()
keys = dict.keys()
keys.sort()
for key in keys:
    print "construction variable = '%s', value = '%s'" % (key, dict[key])
```

Expanding Values From a Construction Environment

Another way to get information from a construction environment is to use the `subst` method on a string containing `$`-expansions of construction variable names. As a simple example, the example from the previous section that used `env['CC']` to fetch the value of `$CC` could also be written as:

```
env = Environment()
print "CC is:", env.subst('$CC')
```

The real advantage of using `subst` to expand strings is that construction variables in the result get re-expanded until there are no expansions left in the string. So a simple fetch of a value like `$CCCOM`:

```
env = Environment(CCFLAGS = '-DFOO')
print "CCCOM is:", env['CCCOM']
```

Will print the unexpanded value of `$CCCOM`, showing us the construction variables that still need to be expanded:

```
% scons -Q
CCCOM is: $CC $CCFLAGS $CPPFLAGS $_CPPDEFFLAGS $_CPPINCFLAGS -c -o $TARGET $SOURCE
scons: '.' is up to date.
```

Calling the `subst` method on `$CCCOM`, however:

```
env = Environment(CCFLAGS = '-DFOO')
print "CCCOM is:", env.subst('$CCCOM')
```

Will recursively expand all of the `$`-prefixed construction variables, showing us the final output:

```
% scons -Q
CCCOM is: gcc -DFOO -c -o
scons: `.' is up to date.
```

(Note that because we're not expanding this in the context of building something there are no target or source files for `$TARGET` and `$SOURCES` to expand.

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

The replacing value (`-DDEFINE2` in the above example) completely replaces the value in the construction environment:

```
% scons -Q
cc -o foo.o -c -DDEFINE2 foo.c
cc -o foo foo.o
```

You can safely call `Replace` for construction variables that don't exist in the construction environment:

```
env = Environment()
env.Replace(NEW_VARIABLE = 'xyzzzy')
print "NEW_VARIABLE =", env['NEW_VARIABLE']
```

In this case, the construction variable simply gets added to the construction environment:

```
% scons -Q
NEW_VARIABLE = xyzzzy
scons: `.' is up to date.
```

Because the variables aren't expanded until the construction environment is actually used to build the targets, and because `SCons` function and method calls are order-independent, the last replacement "wins" and is used to build all targets, regardless of the order in which the calls to `Replace()` are interspersed with calls to builder methods:

```
env = Environment(CCFLAGS = '-DDEFINE1')
print "CCFLAGS =", env['CCFLAGS']
env.Program('foo.c')

env.Replace(CCFLAGS = '-DDEFINE2')
print "CCFLAGS =", env['CCFLAGS']
env.Program('bar.c')
```

The timing of when the replacement actually occurs relative to when the targets get built becomes apparent if we run `scons` without the `-Q` option:

```
% scons
scons: Reading SConscript files ...
CCFLAGS = -DDEFINE1
CCFLAGS = -DDEFINE2
scons: done reading SConscript files.
scons: Building targets ...
cc -o bar.o -c -DDEFINE2 bar.c
cc -o bar bar.o
cc -o foo.o -c -DDEFINE2 foo.c
cc -o foo foo.o
scons: done building targets.
```

Because the replacement occurs while the `SConscript` files are being read, the `$CCFLAGS` variable has already been set to `-DDEFINE2` by the time the `foo.o` target is built, even though the call to the `Replace` method does not occur until later in the `SConscript` file.

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` then supplies both the `-DMY_VALUE` and `-DLAST` flags when compiling the object file:

```
% scons -Q
cc -o foo.o -c -DMY_VALUE -DLAST foo.c
cc -o foo foo.o
```

If the construction variable doesn't already exist, the `Append` method will create it:

```
env = Environment()
env.Append(NEW_VARIABLE = 'added')
print "NEW_VARIABLE =", env['NEW_VARIABLE']
```

Which yields:

```
% scons -Q
NEW_VARIABLE = added
scons: '.' is up to date.
```

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 then supplies both the `-DFIRST` and `-DMY_VALUE` flags when compiling the object file:

```
% scons -Q
cc -o foo.o -c -DFIRST -DMY_VALUE foo.c
cc -o foo foo.o
```

If the construction variable doesn't already exist, the `Prepend` method will create it:

```
env = Environment()
env.Prepend(NEW_VARIABLE = 'added')
print "NEW_VARIABLE =", env['NEW_VARIABLE']
```

Which yields:

```
% scons -Q
NEW_VARIABLE = added
scons: `.' is up to date.
```


Chapter 8. Controlling the External 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 Windows 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})
```

Assign a dictionary to the `$ENV` construction variable in this way completely resets the external environment so that the only variable that will be set when external commands are executed will be the `PATH` value. If you want to use the rest of the values in `$ENV` and only set the value of `PATH`, the most straightforward way is probably:

```
env['ENV']['PATH'] = ['/usr/local/bin', '/bin', '/usr/bin']
```

Note that `SCons` does allow you to define the directories in the `PATH` in a string, separated by the pathname-separator character for your system (':' on POSIX systems, ';' on Windows):

```
env['ENV']['PATH'] = '/usr/local/bin:/bin:/usr/bin'
```

But doing so makes your `SConscript` file less portable, (although in this case that may not be a huge concern since the directories you list are likely system-specific, anyway).

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 explicitly 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 9. 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 explicitly specifying the `bar` target on the command line generates the warning message:

```
% scons -Q
cc -o foo.o -c foo.c
cc -o foo foo.o
% scons -Q bar
Don't forget to copy 'bar' to the archive!
cc -o bar.o -c 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 `SConstruct` 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 -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q
scons: 'hello' is up to date.
% scons -Q goodbye
cc -o goodbye.o -c 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 -o goodbye.o -c goodbye.c
cc -o goodbye goodbye.o
cc -o hello.o -c 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 -o prog1.o -c prog1.c
cc -o prog1 prog1.o
cc -o prog3.o -c prog3.c
cc -o prog3 prog3.o
% scons -Q .
cc -o prog2.o -c 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 -o prog1/foo.o -c prog1/foo.c
cc -o prog1/main.o -c prog1/main.c
cc -o prog1/main prog1/main.o prog1/foo.o
% scons -Q
scons: `prog1' is up to date.
% scons -Q .
cc -o prog2/bar.o -c prog2/bar.c
cc -o prog2/main.o -c 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 -o prog1.o -c prog1.c
cc -o prog1 prog1.o
cc -o prog2.o -c 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 -o prog1.o -c 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 -o prog1.o -c prog1.c
cc -o prog1 prog1.o
cc -o prog2.o -c 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 -o prog1.o -c prog1.c
cc -o prog1 prog1.o
% scons -Q prog2
BUILD_TARGETS is ['prog2']
cc -o prog2.o -c 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 -o prog.o -c prog.c
cc -o prog prog.o
% scons -Q debug=0
scons: `.` is up to date.
% scons -Q debug=1
cc -o prog.o -c -g 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 its 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 -o bar.o -c -DRELEASE_BUILD=1 bar.c
cc -o foo.o -c -DRELEASE_BUILD=1 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 -o bar.o -c -D['RELEASE_BUILD=', 1] bar.c
cc -o foo.o -c -D['RELEASE_BUILD=', 1] 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 -o bar.o -c -D['RELEASE_BUILD=', 0] bar.c
cc -o foo.o -c -D['RELEASE_BUILD=', 0] 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 accommodate users who have different preferences for how to represent `true` or `false` values. The `BoolOption` function makes it easy to accommodate 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', 'Set to build for release', 0))
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 -o foo.o -c -D['RELEASE_BUILD=', True] foo.c

% scons -Q RELEASE=t foo.o
cc -o foo.o -c -D['RELEASE_BUILD=', True] 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 -o foo.o -c -D['RELEASE_BUILD=', False] foo.c

% scons -Q RELEASE=f foo.o
cc -o foo.o -c -D['RELEASE_BUILD=', False] 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 "/home/my/project/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', 'Set background color', 'red',
                    allowed_values=('red', 'green', 'blue')))
env = Environment(options = opts,
                  CPPDEFINES={'COLOR' : '${COLOR}''})
env.Program('foo.c')
```

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

```
% scons -Q COLOR=red foo.o
cc -o foo.o -c -DCOLOR="red" foo.c
% scons -Q COLOR=blue foo.o
cc -o foo.o -c -DCOLOR="blue" foo.c
% scons -Q COLOR=green foo.o
cc -o foo.o -c -DCOLOR="green" 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 "/home/my/project/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', 'Set background color', 'red',
```

```

        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 -o foo.o -c -DCOLOR="blue" 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 "/home/my/project/SConstruct", line 5, in ?
% scons -Q COLOR=BLUE foo.o

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

scons: *** Invalid value for option COLOR: nAvY
File "/home/my/project/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', 'Set background color', 'red',
                  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 -o foo.o -c -DCOLOR="Red" foo.c
% scons -Q COLOR=BLUE foo.o
cc -o foo.o -c -DCOLOR="BLUE" foo.c
% scons -Q COLOR=nAvY foo.o
cc -o foo.o -c -DCOLOR="blue" foo.c
% scons -Q COLOR=green foo.o
cc -o foo.o -c -DCOLOR="green" 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', 'Set background color', 'red',
                  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 -o foo.o -c -DCOLOR="red" foo.c
% scons -Q COLOR=nAvY foo.o
cc -o foo.o -c -DCOLOR="blue" foo.c
% scons -Q COLOR=GREEN foo.o
cc -o foo.o -c -DCOLOR="green" 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', 'List of colors', 0,
                  ['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
cc -o foo.o -c -DCOLORS="red blue" foo.c
% scons -Q COLORS=blue,green,red foo.o
cc -o foo.o -c -DCOLORS="blue green red" foo.c

```

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
cc -o foo.o -c -DCOLORS="red green blue" foo.c
% scons -Q COLORS=none foo.o
cc -o foo.o -c -DCOLORS="" foo.c

```

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

```

% scons -Q COLORS=magenta foo.o

scons: *** Error converting option: COLORS
Invalid value(s) for option: magenta
File "/home/my/project/SConstruct", line 5, in ?

```

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',
                    'Path to configuration file',
                    '/etc/my_config'))
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
cc -o foo.o -c -DCONFIG_FILE="/etc/my_config" foo.c
% scons -Q CONFIG=/usr/local/etc/other_config foo.o
scons: 'foo.o' is up to date.
```

By default, PathOption checks to make sure that the specified path exists and generates an error if it doesn't:

```
% scons -Q CONFIG=/does/not/exist foo.o

scons: *** Path for option CONFIG does not exist: /does/not/exist
File "/home/my/project/SConstruct", line 6, in ?
```

PathOption provides a number of methods that you can use to change this behavior. If you want to ensure that any specified paths are, in fact, files and not directories, use the PathOption.PathIsFile method:

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

Conversely, to ensure that any specified paths are directories and not files, use the PathOption.PathIsDir method:

```
opts = Options('custom.py')
opts.Add(PathOption('DBDIR',
                    'Path to database directory',
                    '/var/my_dbdir',
                    PathOption.PathIsDir))
env = Environment(options = opts,
                  CPPDEFINES={'DBDIR' : '"$DBDIR"'})
env.Program('foo.c')
```

If you want to make sure that any specified paths are directories, and you would like the directory created if it doesn't already exist, use the PathOption.PathIsDirCreate method:

```
opts = Options('custom.py')
```



```

opts.Add(PathOption('DBDIR',
                    'Path to database directory',
                    '/var/my_dbdir',
                    PathOption.PathIsDirCreate))
env = Environment(options = opts,
                  CPPDEFINES={'DBDIR' : '$DBDIR'})
env.Program('foo.c')

```

Lastly, if you don't care whether the path exists, is a file, or a directory, use the `PathOption.PathAccept` method to accept any path that the user supplies:

```

opts = Options('custom.py')
opts.Add(PathOption('OUTPUT',
                    'Path to output file or directory',
                    None,
                    PathOption.PathAccept))
env = Environment(options = opts,
                  CPPDEFINES={'OUTPUT' : '$OUTPUT'})
env.Program('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',
                      'Location package',
                      '/opt/location'))
env = Environment(options = opts,
                  CPPDEFINES={'PACKAGE' : '$PACKAGE'})
env.Program('foo.c')

```

When the `SConscript` file uses the `PackageOption` function, 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
cc -o foo.o -c -DPACKAGE="/opt/location" foo.c
% scons -Q PACKAGE=/usr/local/location foo.o
cc -o foo.o -c -DPACKAGE="/usr/local/location" foo.c
% scons -Q PACKAGE=yes foo.o
cc -o foo.o -c -D['PACKAGE=', True, ''] foo.c
% scons -Q PACKAGE=no foo.o
cc -o foo.o -c -D['PACKAGE=', False, ''] 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 10. Providing Build Help: the `Help` Function

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.
```

The `SConscript` files may contain multiple calls to the `Help` function, in which case the specified text(s) will be concatenated when displayed. This allows you to split up the help text across multiple `SConscript` files. In this situation, the order in which the `SConscript` files are called will determine the order in which the `Help` functions are called, which will determine the order in which the various bits of text will get concatenated.

Another use would be to make the help text conditional on some variable. For example, suppose you only want to display a line about building a Windows-only version of a program when actually run on Windows. The following `SConstruct` file:

```
env = Environment()

Help("\nType: 'scons program' to build the production program.\n")

if env['PLATFORM'] == 'win32':
    Help("\nType: 'scons windebug' to build the Windows debug version.\n")
```

Will display the completely help text on Windows:

```
C:\>scons -h
scons: Reading SConscript files ...
scons: done reading SConscript files.

Type: 'scons program' to build the production program.

Type: 'scons windebug' to build the Windows debug version.

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

But only show the relevant option on a Linux or UNIX system:

```
% scons -h
scons: Reading SConscript files ...
scons: done reading SConscript files.
```

```
Type: 'scons program' to build the production program.
```

```
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 11. Installing Files in Other Directories: the `Install` Builder

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

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


Chapter 12. Platform-Independent File System Manipulation

SCons provides a number of platform-independent functions, called `factories`, that perform common file system manipulations like copying, moving or deleting files and directories, or making directories. These functions are `factories` because they don't perform the action at the time they're called, they each return an `Action` object that can be executed at the appropriate time.

Copying Files or Directories: The `copy` Factory

Suppose you want to arrange to make a copy of a file, and the `Install` builder isn't appropriate because it may make a hard link on POSIX systems. One way would be to use the `Copy` action factory in conjunction with the `Command` builder:

```
Command("file.out", "file.in", Copy("$TARGET", "$SOURCE"))
```

Notice that the action returned by the `Copy` factory will expand the `$TARGET` and `$SOURCE` strings at the time `file.out` is built, and that the order of the arguments is the same as that of a builder itself--that is, target first, followed by source:

```
% scons -Q
Copy("file.out", "file.in")
```

You can, of course, name a file explicitly instead of using `$TARGET` or `$SOURCE`:

```
Command("file.out", [], Copy("$TARGET", "file.in"))
```

Which executes as:

```
% scons -Q
Copy("file.out", "file.in")
```

The usefulness of the `Copy` factory becomes more apparent when you use it in a list of actions passed to the `Command` builder. For example, suppose you needed to run a file through a utility that only modifies files in-place, and can't "pipe" input to output. One solution is to copy the source file to a temporary file name, run the utility, and then copy the modified temporary file to the target, which the `Copy` factory makes extremely easy:

```
Command("file.out", "file.in",
[
    Copy("tempfile", "$SOURCE"),
    "modify tempfile",
    Copy("$TARGET", "tempfile"),
])
```

The output then looks like:

```
% scons -Q
Copy("tempfile", "file.in")
modify tempfile
Copy("file.out", "tempfile")
```

Deleting Files or Directories: The `Delete` Factory

If you need to delete a file, then the `Delete` factory can be used in much the same way as the `Copy` factory. For example, if we want to make sure that the temporary file in our last example doesn't exist before we copy to it, we could add `Delete` to the beginning of the command list:

```
Command("file.out", "file.in",
[
    Delete("tempfile"),
    Copy("tempfile", "$SOURCE"),
    "modify tempfile",
    Copy("$TARGET", "tempfile"),
])
```

When then executes as follows:

```
% scons -Q
Delete("tempfile")
Copy("tempfile", "file.in")
modify tempfile
Copy("file.out", "tempfile")
```

Of course, like all of these Action factories, the `Delete` factory also expands `$TARGET` and `$SOURCE` variables appropriately. For example:

```
Command("file.out", "file.in",
[
    Delete("$TARGET"),
    Copy("$TARGET", "$SOURCE")
])
```

Executes as:

```
% scons -Q
Delete("file.out")
Copy("file.out", "file.in")
```

(Note, however, that you typically don't need to call the `Delete` factory explicitly in this way; by default, `SCons` deletes its target(s) for you before executing any action.

Moving (Renaming) Files or Directories: The `Move` Factory

The `Move` factory allows you to rename a file or directory. For example, if we don't want to copy the temporary file, we could:

```
Command("file.out", "file.in",
[
    Copy("tempfile", "$SOURCE"),
    "modify tempfile",
    Move("$TARGET", "tempfile"),
])
```

Which would execute as:

```
% scons -Q
Copy("tempfile", "file.in")
modify tempfile
```

```
Move("file.out", "tempfile")
```

Updating the Modification Time of a File: The `Touch` Factory

If you just need to update the recorded modification time for a file, use the `Touch` factory:

```
Command("file.out", "file.in",
[
    Copy("tempfile", "$SOURCE"),
    "modify tempfile",
    Move("$TARGET", "tempfile"),
])
```

Which executes as:

```
% scons -Q
Copy("tempfile", "file.in")
modify tempfile
Move("file.out", "tempfile")
```

Creating a Directory: The `Mkdir` Factory

If you need to create a directory, use the `Mkdir` factory. For example, if we need to process a file in a temporary directory in which the processing tool will create other files that we don't care about, you could:

```
Command("file.out", "file.in",
[
    Delete("tempdir"),
    Mkdir("tempdir"),
    Copy("tempdir/${SOURCE.file}", "$SOURCE"),
    "process tempdir",
    Move("$TARGET", "tempdir/output_file"),
    Delete("tempdir"),
])
```

Which executes as:

```
% scons -Q
Delete("tempdir")
Mkdir("tempdir")
Copy("tempdir/file.in", "file.in")
process tempdir
Move("file.out", "tempdir/output_file")
scons: *** [file.out] No such file or directory
```

Changing File or Directory Permissions: The `Chmod` Factory

To change permissions on a file or directory, use the `Chmod` factory. The permission argument uses POSIX-style permission bits and should typically be expressed as an octal, not decimal, number:

```
Command("file.out", "file.in",
[
    Copy("$TARGET", "$SOURCE"),
    Chmod("$TARGET", 0755),
])
```

Which executes:

```
% scons -Q
Copy("file.out", "file.in")
Chmod("file.out", 0755)
```

Executing an action immediately: the `Execute` Function

We've been showing you how to use Action factories in the `Command` function. You can also execute an Action returned by a factory (or actually, any Action) at the time the `SConscript` file is read by wrapping it up in the `Execute` function. For example, if we need to make sure that a directory exists before we build any targets,

```
Execute(Mkdir('/tmp/my_temp_directory'))
```

Notice that this will create the directory while the `SConscript` file is being read:

```
% scons
scons: Reading SConscript files ...
Mkdir("/tmp/my_temp_directory")
scons: done reading SConscript files.
scons: Building targets ...
scons: `.' is up to date.
scons: done building targets.
```

If you're familiar with Python, you may wonder why you would want to use this instead of just calling the native Python `os.mkdir()` function. The advantage here is that the `Mkdir` action will behave appropriately if the user specifies the `SCons -n` or `-q` options--that is, it will print the action but not actually make the directory when `-n` is specified, or make the directory but not print the action when `-q` is specified.

Chapter 13. Preventing Removal of Targets

There are two occasions when `SCons` will, by default, remove target files. The first is when `SCons` determines that a target file needs to be rebuilt and removes the existing version of the target before executing. The second is when `SCons` is invoked with the `-c` option to "clean" a tree of its built targets. These behaviours can be suppressed with the `Precious` and `NoClean` functions, respectively.

Preventing target removal during build: the `Precious` Function

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(RANLIBCOM="")
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 -o f1.o -c f1.c
cc -o f2.o -c f2.c
cc -o f3.o -c f3.c
ar rc libfoo.a f1.o f2.o f3.o
```

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

Preventing target removal during clean: the `NoClean` Function

By default, `SCons` removes all built targets when invoked with the `-c` option to clean a source tree of built targets. Sometimes, however, this is not what you want. For example, you may want to remove only intermediate generated files (such as object files), but leave the final targets (the libraries) untouched. In such cases, you can use the `NoClean` method to prevent `SCons` from removing a target during a clean:

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

Notice that the `libfoo.a` is not listed as a removed file:

```
% scons -Q
cc -o f1.o -c f1.c
cc -o f2.o -c f2.c
cc -o f3.o -c f3.c
ar rc libfoo.a f1.o f2.o f3.o
% scons -c
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Cleaning targets ...
Removed f1.o
Removed f2.o
Removed f3.o
scons: done cleaning targets.
```


Chapter 14. 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 -o prog1/foo1.o -c prog1/foo1.c
cc -o prog1/foo2.o -c prog1/foo2.c
cc -o prog1/main.o -c prog1/main.c
cc -o prog1/prog1 prog1/main.o prog1/foo1.o prog1/foo2.o
cc -o prog2/bar1.o -c prog2/bar1.c
cc -o prog2/bar2.o -c prog2/bar2.c
cc -o prog2/main.o -c 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 -o lib/foo1.o -c lib/foo1.c
cc -o src/prog/foo2.o -c src/prog/foo2.c
cc -o src/prog/main.o -c 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 Chapter 15, 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 -o src/prog/foo2.o -c src/prog/foo2.c
cc -o src/prog/main.o -c src/prog/main.c
cc -o /usr/joe/lib/foo1.o -c /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 Chapter 15, 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.Clone(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.Clone(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.Clone(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 -o bar/bar.o -c bar/bar.c
cc -o foo/foo.o -c foo/foo.c
ar rc libprog.a foo/foo.o bar/bar.o
ranlib libprog.a
```


Chapter 15. 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 -o build/hello.o -c 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 versions) 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 -o build/hello.o -c 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 -o build/hello.o -c 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 -o build/hello.o -c 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 16. Variant Builds

The `build_dir` keyword argument of the `SConscript` function provides 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')
```

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 -o build/linux/hello/hello.o -c -Iexport/linux/include build/linux/hello/hello.c
cc -o build/linux/world/world.o -c -Iexport/linux/include build/linux/world/world.c
ar rc 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
cl /nologo /Iexport\windows\include /c build\windows\world\world.c /Fobuild\windows
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
Install file: "build/windows/hello/hello.exe" as "export/windows/bin/hello.exe"
```


Chapter 17. 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' object has no attribute 'Program':
File "SConstruct", line 4:
    env.Program('hello.c')
```

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 -o hello.o -c 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 18. 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 19. 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 20. 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.) 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/repository2/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 -o hello.o -c 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 -o hello.o -c /usr/repository1/hello.c
cc -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 -o hello.o -c /usr/repository2/hello.c
cc -o hello hello.o
```

Finding `#include` files in repositories

We've already seen that SCons will scan the contents of a source file for `#include` file names and realize that targets built from that source file also depend on the `#include` file(s). For each directory in the `$CPPPATH` list, SCons will actually search the corresponding directories in any repository trees and establish the correct dependencies on any `#include` files that it finds in repository directory.

Unless the C compiler also knows about these directories in the repository trees, though, it will be unable to find the `#include` files. If, for example, the `hello.c` file in our previous example includes the `hello.h`; in its current directory, and the `hello.h`; only exists in the repository:

```
% scons -Q
cc -o hello.o -c hello.c
hello.c:1: hello.h: No such file or directory
```

In order to inform the C compiler about the repositories, SCons will add appropriate `-I` flags to the compilation commands for each directory in the `$CPPPATH` list. So if we add the current directory to the construction environment `$CPPPATH` like so:

```
env = Environment(CPPPATH = ['.'])
env.Program('hello.c')
Repository('/usr/repository1')
```

Then re-executing SCons yields:

```
% scons -Q
cc -o hello.o -c -I. -I/usr/repository1 hello.c
cc -o hello hello.o
```

The order of the `-I` options replicates, for the C preprocessor, the same repository-directory search path that SCons uses for its own dependency analysis. If there are multiple repositories and multiple `$CPPPATH` directories, SCons will add the repository directories to the beginning of each `$CPPPATH` directory, rapidly multiplying the number of `-I` flags. If, for example, the `$CPPPATH` contains three directories (and shorter repository path names!):

```
env = Environment(CPPPATH = ['dir1', 'dir2', 'dir3'])
env.Program('hello.c')
Repository('/r1', '/r2')
```

Then we'll end up with nine `-I` options on the command line, three (for each of the `$CPPPATH` directories) times three (for the local directory plus the two repositories):

```
% scons -Q
cc -o hello.o -c -Idir1 -I/r1/dir1 -I/r2/dir1 -Idir2 -I/r1/dir2 -I/r2/dir2 -Idir3
cc -o hello hello.o
```

Limitations on `#include` files in repositories

SCons relies on the C compiler's `-I` options to control the order in which the preprocessor will search the repository directories for `#include` files. This causes a problem, however, with how the C preprocessor handles `#include` lines with the file name included in double-quotes.

As we've seen, `SCons` will compile the `hello.c` file from the repository if it doesn't exist in the local directory. If, however, the `hello.c` file in the repository contains a `#include` line with the file name in double quotes:

```
#include "hello.h"
int
main(int argc, char *argv[])
{
    printf(HELLO_MESSAGE);
    return (0);
}
```

Then the C preprocessor will *always* use a `hello.h` file from the repository directory first, even if there is a `hello.h` file in the local directory, despite the fact that the command line specifies `-I` as the first option:

```
% scons -Q
cc -o hello.o -c -I. -I/usr/repository1 /usr/repository1/hello.c
cc -o hello hello.o
```

This behavior of the C preprocessor--always search for a `#include` file in double-quotes first in the same directory as the source file, and only then search the `-I`--can not, in general, be changed. In other words, it's a limitation that must be lived with if you want to use code repositories in this way. There are three ways you can possibly work around this C preprocessor behavior:

1. Some modern versions of C compilers do have an option to disable or control this behavior. If so, add that option to `$CFLAGS` (or `$CXXFLAGS` or both) in your construction environment(s). Make sure the option is used for all construction environments that use C preprocessing!
2. Change all occurrences of `#include "file.h"` to `#include <file.h>`. Use of `#include` with angle brackets does not have the same behavior--the `-I` directories are searched first for `#include` files--which gives `SCons` direct control over the list of directories the C preprocessor will search.
3. Require that everyone working with compilation from repositories check out and work on entire directories of files, not individual files. (If you use local wrapper scripts around your source code control system's command, you could add logic to enforce this restriction there.

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 -o file1.o -c file1.c
cc -o file2.o -c file2.c
cc -o hello.o -c 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 21. Multi-Platform Configuration (Autoconf Functionality)

SCons has integrated support for multi-platform build configuration similar to that offered by GNU Autoconf, such as figuring out what libraries or header files are available on the local system. This section describes how to use this SCons feature.

Note: This chapter is still under development, so not everything is explained as well as it should be. See the SCons man page for additional information.

Configure Contexts

The basic framework for multi-platform build configuration in SCons is to attach a configure context to a construction environment by calling the `Configure` function, perform a number of checks for libraries, functions, header files, etc., and to then call the configure context's `Finish` method to finish off the configuration:

```
env = Environment()
conf = Configure(env)
# Checks for libraries, header files, etc. go here!
env = conf.Finish()
```

SCons provides a number of basic checks, as well as a mechanism for adding your own custom checks.

Note that SCons uses its own dependency mechanism to determine when a check needs to be run—that is, SCons does not run the checks every time it is invoked, but caches the values returned by previous checks and uses the cached values unless something has changed. This saves a tremendous amount of developer time while working on cross-platform build issues.

The next sections describe the basic checks that SCons supports, as well as how to add your own custom checks.

Checking for the Existence of Header Files

Testing the existence of a header file requires knowing what language the header file is. A configure context has a `CheckCHeader` method that checks for the existence of a C header file:

```
env = Environment()
conf = Configure(env)
if not conf.CheckCHeader('math.h'):
    print 'Math.h must be installed!'
    Exit(1)
if conf.CheckCHeader('foo.h'):
    conf.env.Append('-DHAS_FOO_H')
env = conf.Finish()
```

Note that you can choose to terminate the build if a given header file doesn't exist, or you can modify the construction environment based on the existence of a header file.

If you need to check for the existence a C++ header file, use the `CheckCXXHeader` method:

```
env = Environment()
```

```
conf = Configure(env)
if not conf.CheckCXXHeader('vector.h'):
    print 'vector.h must be installed!'
    Exit(1)
env = conf.Finish()
```

Checking for the Availability of a Function

Check for the availability of a specific function using the `CheckFunc` method:

```
env = Environment()
conf = Configure(env)
if not conf.CheckFunc('strcpy'):
    print 'Did not find strcpy(), using local version'
    conf.env.Append('-Dstrcpy=my_local_strcpy')
env = conf.Finish()
```

Checking for the Availability of a Library

Check for the availability of a library using the `CheckLib` method. You only specify the basename of the library, you don't need to add a `lib` prefix or a `.a` or `.lib` suffix:

```
env = Environment()
conf = Configure(env)
if not conf.CheckLib('m'):
    print 'Did not find libm.a or m.lib, exiting!'
    Exit(1)
env = conf.Finish()
```

Because the ability to use a library successfully often depends on having access to a header file that describes the library's interface, you can check for a library *and* a header file at the same time by using the `CheckLibWithHeader` method:

```
env = Environment()
conf = Configure(env)
if not conf.CheckLibWithHeader('m', 'math.h'):
    print 'Did not find libm.a or m.lib, exiting!'
    Exit(1)
env = conf.Finish()
```

This is essentially shorthand for separate calls to the `CheckHeader` and `CheckLib` functions.

Checking for the Availability of a typedef

Check for the availability of a typedef by using the `CheckType` method:

```
env = Environment()
conf = Configure(env)
if not conf.CheckType('off_t'):
    print 'Did not find off_t typedef, assuming int'
    conf.env.Append(CCFLAGS = '-Doff_t=int')
env = conf.Finish()
```

You can also add a string that will be placed at the beginning of the test file that will be used to check for the typedef. This provides a way to specify files that must be included to find the typedef:

```
env = Environment()
conf = Configure(env)
if not conf.CheckType('off_t', '#include <sys/types.h>\n'):
    print 'Did not find off_t typedef, assuming int'
    conf.env.Append(CCFLAGS = '-Doff_t=int')
env = conf.Finish()
```

Adding Your Own Custom Checks

A custom check is a Python function that checks for a certain condition to exist on the running system, usually using methods that sCons supplies to take care of the details of checking whether a compilation succeeds, a link succeeds, a program is runnable, etc. A simple custom check for the existence of a specific library might look as follows:

```
mylib_test_source_file = """
#include <mylib.h>
int main(int argc, char **argv)
{
    MyLibrary mylib(argc, argv);
    return 0;
}
"""

def CheckMyLibrary(context):
    context.Message('Checking for MyLibrary...')
    result = context.TryLink(mylib_test_source_file, '.c')
    context.Result(result)
    return result
```

The `Message` and `Result` methods should typically begin and end a custom check to let the user know what's going on: the `Message` call prints the specified message (with no trailing newline) and the `Result` call prints ok if the check succeeds and failed if it doesn't. The `TryLink` method actually tests for whether the specified program text will successfully link.

(Note that a custom check can modify its check based on any arguments you choose to pass it, or by using or modifying the configure context environment in the `context.env` attribute.)

This custom check function is then attached to the `configure` context by passing a dictionary to the `Configure` call that maps a name of the check to the underlying function:

```
env = Environment()
conf = Configure(env, custom_tests = {'CheckMyLibrary' : CheckMyLibrary})
```

You'll typically want to make the check and the function name the same, as we've done here, to avoid potential confusion.

We can then put these pieces together and actually call the `CheckMyLibrary` check as follows:

```
mylib_test_source_file = """
#include <mylib.h>
int main(int argc, char **argv)
```

```
{
    MyLibrary mylib(argc, argv);
    return 0;
}
"""

def CheckMyLibrary(context):
    context.Message('Checking for MyLibrary... ')
    result = context.TryLink(mylib_test_source_file, '.c')
    context.Result(result)
    return result

env = Environment()
conf = Configure(env, custom_tests = {'CheckMyLibrary' : CheckMyLibrary})
if not conf.CheckMyLibrary():
    print 'MyLibrary is not installed!'
    Exit(1)
env = conf.Finish()

# We would then add actual calls like Program() to build
# something using the "env" construction environment.
```

If MyLibrary is not installed on the system, the output will look like:

```
% scons
scons: Reading SConscript file ...
Checking for MyLibrary... failed
MyLibrary is not installed!
```

If MyLibrary is installed, the output will look like:

```
% scons
scons: Reading SConscript file ...
Checking for MyLibrary... failed
scons: done reading SConscript
scons: Building targets ...
.
.
.
```

Not Configuring When Cleaning Targets

Using multi-platform configuration as described in the previous sections will run the configuration commands even when invoking **scons -c** to clean targets:

```
% scons -Q -c
Checking for MyLibrary... ok
Removed foo.o
Removed foo
```

Although running the platform checks when removing targets doesn't hurt anything, it's usually unnecessary. You can avoid this by using the `GetOption();` method to check whether the `-c` (clean) option has been invoked on the command line:

```
env = Environment()
if not env.GetOption('clean'):
    conf = Configure(env, custom_tests = {'CheckMyLibrary' : CheckMyLibrary})
    if not conf.CheckMyLibrary():
        print 'MyLibrary is not installed!'
```

```
        Exit(1)  
env = conf.Finish()
```

```
% scons -Q -c  
Removed foo.o  
Removed foo
```


Chapter 22. 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 -o hello.o -c 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 -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q -c
Removed hello.o
Removed hello
% scons -Q --cache-show
cc -o hello.o -c 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 -o hello.o -c 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 -o hello.o -c 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 `--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 -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q -c
Removed hello.o
Removed hello
% scons -Q --cache-disable
cc -o hello.o -c 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 23. 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 -o hello.o -c 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 another 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 -o foo.o -c foo.c
cc -o foo foo.o
Install file: "foo" as "/usr/bin/foo"
% scons -Q install-lib
cc -o bar.o -c bar.c
ar rc 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 -o foo.o -c foo.c
cc -o foo foo.o
Install file: "foo" as "/usr/bin/foo"
cc -o bar.o -c bar.c
ar rc libbar.a bar.o
ranlib libbar.a
Install file: "libbar.a" as "/usr/lib/libbar.a"
```


Chapter 24. Java Builds

So far, we've been using examples of building C and C++ programs to demonstrate the features of *SCons*. *SCons* also supports building Java programs, but Java builds are handled slightly differently, which reflects the ways in which the Java compiler and tools build programs differently than other languages' tool chains.

Building Java Class Files: the Java Builder

The basic activity when programming in Java, of course, is to take one or more `.java` files containing Java source code and to call the Java compiler to turn them into one or more `.class` files. In *SCons*, you do this by giving the Java Builder a target directory in which to put the `.class` files, and a source directory that contains the `.java` files:

```
Java('classes', 'src')
```

If the `src` directory contains three `.java` source files, then running *SCons* might look like this:

```
% scons -Q
javac -d classes -sourcepath src src/Example1.java src/Example2.java src/Example3.java
```

SCons will actually search the `src` directory tree for all of the `.java` files. The Java compiler will then create the necessary class files in the `classes` subdirectory, based on the class names found in the `.java` files.

How *SCons* Handles Java Dependencies

In addition to searching the source directory for `.java` files, *SCons* actually runs the `.java` files through a stripped-down Java parser that figures out what classes are defined. In other words, *SCons* knows, without you having to tell it, what `.class` files will be produced by the `javac` call. So our one-liner example from the preceding section:

```
Java('classes', 'src')
```

Will not only tell you reliably that the `.class` files in the `classes` subdirectory are up-to-date:

```
% scons -Q
javac -d classes -sourcepath src src/Example1.java src/Example2.java src/Example3.java
% scons -Q classes
scons: 'classes' is up to date.
```

But it will also remove all of the generated `.class` files, even for inner classes, without you having to specify them manually. For example, if our `Example1.java` and `Example3.java` files both define additional classes, and the class defined in `Example2.java` has an inner class, running **`scons -c`** will clean up all of those `.class` files as well:

```
% scons -Q
javac -d classes -sourcepath src src/Example1.java src/Example2.java src/Example3.java
% scons -Q -c classes
Removed classes/Example1.class
Removed classes/AdditionalClass1.class
Removed classes/Example2$Inner2.class
```

```
Removed classes/Example2.class
Removed classes/Example3.class
Removed classes/AdditionalClass3.class
```

Building Java Archive (.jar) Files: the Jar Builder

After building the class files, it's common to collect them into a Java archive (.jar) file, which you do by calling the `Jar` Builder method. If you want to just collect all of the class files within a subdirectory, you can just specify that subdirectory as the `Jar` source:

```
Java(target = 'classes', source = 'src')
Jar(target = 'test.jar', source = 'classes')
```

`SCons` will then pass that directory to the `jar` command, which will collect all of the underlying .class files:

```
% scons -Q
javac -d classes -sourcepath src src/Example1.java src/Example2.java src/Example3.java
jar cf test.jar classes
```

If you want to keep all of the .class files for multiple programs in one location, and only archive some of them in each .jar file, you can pass the `Jar` builder a list of files as its source. It's extremely simple to create multiple .jar files this way, using the lists of target class files created by calls to the `Java` builder as sources to the various `Jar` calls:

```
prog1_class_files = Java(target = 'classes', source = 'prog1')
prog2_class_files = Java(target = 'classes', source = 'prog2')
Jar(target = 'prog1.jar', source = prog1_class_files)
Jar(target = 'prog2.jar', source = prog2_class_files)
```

This will then create `prog1.jar` and `prog2.jar` next to the subdirectories that contain their .java files:

```
% scons -Q
javac -d classes -sourcepath prog1 prog1/Example1.java prog1/Example2.java
javac -d classes -sourcepath prog2 prog2/Example3.java prog2/Example4.java
jar cf prog1.jar classes/Example1.class classes/Example2.class
jar cf prog2.jar classes/Example3.class classes/Example4.class
```

Building C Header and Stub Files: the JavaH Builder

You can generate C header and source files for implementing native methods, by using the `JavaH` Builder. There are several ways of using the `JavaH` Builder. One typical invocation might look like:

```
classes = Java(target = 'classes', source = 'src/pkg/sub')
JavaH(target = 'native', source = classes)
```

The source is a list of class files generated by the call to the `Java` Builder, and the target is the output directory in which we want the C header files placed. The target gets converted into the `-d` when `SCons` runs `javah`:


```
% scons -Q
javac -d classes -sourcepath src/pkg/sub src/pkg/sub/Example1.java src/pkg/sub/Example2.java
javah -d native -classpath classes pkg.sub.Example1 pkg.sub.Example2 pkg.sub.Example3
```

In this case, the call to `javah` will generate the header files `native/pkg_sub_Example1.h`, `native/pkg_sub_Example2.h` and `native/pkg_sub_Example3.h`. Notice that `SCons` remembered that the class files were generated with a target directory of `classes`, and that it then specified that target directory as the `-classpath` option to the call to `javah`.

Although it's more convenient to use the list of class files returned by the Java Builder as the source of a call to the `JavaH` Builder, you *can* specify the list of class files by hand, if you prefer. If you do, you need to set the `$JAVACLASSDIR` construction variable when calling `JavaH`:

```
Java(target = 'classes', source = 'src/pkg/sub')
class_file_list = ['classes/pkg/sub/Example1.class',
                  'classes/pkg/sub/Example2.class',
                  'classes/pkg/sub/Example3.class']
JavaH(target = 'native', source = class_file_list, JAVACLASSDIR = 'classes')
```

The `$JAVACLASSDIR` value then gets converted into the `-classpath` when `SCons` runs `javah`:

```
% scons -Q
javac -d classes -sourcepath src/pkg/sub src/pkg/sub/Example1.java src/pkg/sub/Example2.java
javah -d native -classpath classes pkg.sub.Example1 pkg.sub.Example2 pkg.sub.Example3
```

Lastly, if you don't want a separate header file generated for each source file, you can specify an explicit File Node as the target of the `JavaH` Builder:

```
classes = Java(target = 'classes', source = 'src/pkg/sub')
JavaH(target = File('native.h'), source = classes)
```

Because `SCons` assumes by default that the target of the `JavaH` builder is a directory, you need to use the `File` function to make sure that `SCons` doesn't create a directory named `native.h`. When a file is used, though, `SCons` correctly converts the file name into the `javah -o` option:

```
% scons -Q
javac -d classes -sourcepath src/pkg/sub src/pkg/sub/Example1.java src/pkg/sub/Example2.java
javah -o native.h -classpath classes pkg.sub.Example1 pkg.sub.Example2 pkg.sub.Example3
```

Building RMI Stub and Skeleton Class Files: the `RMIC` Builder

You can generate Remote Method Invocation stubs by using the `RMIC` Builder. The source is a list of directories, typically returned by a call to the Java Builder, and the target is an output directory where the `_Stub.class` and `_Skel.class` files will be placed:

```
classes = Java(target = 'classes', source = 'src/pkg/sub')
RMIC(target = 'outdir', source = classes)
```

As it did with the `JavaH` Builder, `SCons` remembers the class directory and passes it as the `-classpath` option to `rmic`:

```
% scons -Q  
javac -d classes -sourcepath src/pkg/sub src/pkg/sub/Example1.java src/pkg/sub/Example2.java  
rmic -d outdir -classpath classes pkg.sub.Example1 pkg.sub.Example2
```

This example would generate the files `outdir/pkg/sub/Example1_Skel.class`, `outdir/pkg/sub/Example1_Stub.class`, `outdir/pkg/sub/Example2_Skel.class` and `outdir/pkg/sub/Example2_Stub.class`.

Chapter 25. Troubleshooting

The experience of configuring any software build tool to build a large code base usually, at some point, involves trying to figure out why the tool is behaving a certain way, and how to get it to behave the way you want. SCons is no different.

Why is That Target Being Rebuilt? the `--debug=explain` Option

Let's take a simple example of a misconfigured build that causes a target to be rebuilt every time SCons is run:

```
# Intentionally misspell the output file name in the
# command used to create the file:
Command('file.out', 'file.in', 'cp $SOURCE file.oout')
```

(Note to Windows users: The POSIX `cp` command copies the first file named on the command line to the second file. In our example, it copies the `file.in` file to the `file.out` file.)

Now if we run SCons multiple on this example, we see that it re-runs the `cp` command every time:

```
% scons -Q
cp file.in file.oout
% scons -Q
cp file.in file.oout
% scons -Q
cp file.in file.oout
```

In this example, the underlying cause is obvious: we've intentionally misspelled the output file name in the `cp` command, so the command doesn't actually build the `file.out` file that we've told SCons to expect. But if the problem weren't obvious, it would be helpful to specify the `--debug=explain` option on the command line to have SCons tell us very specifically why it's decided to rebuild the target:

```
% scons -Q --debug=explain
scons: building 'file.out' because it doesn't exist
cp file.in file.oout
```

If this had been a more complicated example involving a lot of build output, having SCons tell us that it's trying to rebuild the target file because it doesn't exist would be an important clue that something was wrong with the command that we invoked to build it.

The `--debug=explain` option also comes in handy to help figure out what input file changed. Given a simple configuration that builds a program from three source files, changing one of the source files and rebuilding with the `--debug=explain` option shows very specifically why SCons rebuilds the files that it does:

```
% scons -Q
cc -o file1.o -c file1.c
cc -o file2.o -c file2.c
cc -o file3.o -c file3.c
cc -o prog file1.o file2.o file3.o
% edit file2.c
[CHANGE THE CONTENTS OF file2.c]
% scons -Q --debug=explain
scons: rebuilding 'file2.o' because 'file2.c' changed
cc -o file2.o -c file2.c
scons: rebuilding 'prog' because 'file2.o' changed
```

```
cc -o prog file1.o file2.o file3.o
```

This becomes even more helpful in identifying when a file is rebuilt due to a change in an implicit dependency, such as an included `.h` file. If the `file1.c` and `file3.c` files in our example both included a `hello.h` file, then changing that included file and re-running `SCons` with the `--debug=explain` option will pinpoint that it's the change to the included file that starts the chain of rebuilds:

```
% scons -Q
cc -o file1.o -c -I. file1.c
cc -o file2.o -c -I. file2.c
cc -o file3.o -c -I. file3.c
cc -o prog file1.o file2.o file3.o
% edit hello.h
[CHANGE THE CONTENTS OF hello.h]
% scons -Q --debug=explain
scons: rebuilding 'file1.o' because 'hello.h' changed
cc -o file1.o -c -I. file1.c
scons: rebuilding 'file3.o' because 'hello.h' changed
cc -o file3.o -c -I. file3.c
scons: rebuilding 'prog' because:
        'file1.o' changed
        'file3.o' changed
cc -o prog file1.o file2.o file3.o
```

What's in That Construction Environment? the `Dump` Method

When you create a construction environment, `SCons` populates it with construction variables that are set up for various compilers, linkers and utilities that it finds on your system. Although this is usually helpful and what you want, it might be frustrating if `SCons` doesn't set certain variables that you expect to be set. In situations like this, it's sometimes helpful to use the construction environment `Dump` method to print all or some of the construction variables. Note that the `Dump` method *returns* the representation of the variables in the environment for you to print (or otherwise manipulate):

On a POSIX system with `gcc` installed, this might generate:

```
% scons
scons: Reading SConscript files ...
{ 'BUILDERS': {},
  'CONFIGUREDIRENTRY': '#/.sconf_temp',
  'CONFIGURELOG': '#/config.log',
  'CPPSUFFIXES': [ '.c',
                   '.C',
                   '.cxx',
                   '.cpp',
                   '.c++',
                   '.cc',
                   '.h',
                   '.H',
                   '.hxx',
                   '.hpp',
                   '.hh',
                   '.F',
                   '.fpp',
                   '.FPP',
                   '.m',
                   '.mm',
                   '.S',
                   '.spp',
```

```

        '.SPP'],
'DSUFFIXES': ['.d'],
'Dir': <SCons.Defaults.Variable_Method_Caller instance at 0xb7c43bec>,
'Dirs': <SCons.Defaults.Variable_Method_Caller instance at 0xb7c43c0c>,
'ENV': {'PATH': '/usr/local/bin:/opt/bin:/bin:/usr/bin'},
'ESCAPE': <function escape at 0xb7b66c34>,
'File': <SCons.Defaults.Variable_Method_Caller instance at 0xb7c43c2c>,
'IDLSUFFIXES': ['.idl', '.IDL'],
'INSTALL': <function installFunc at 0xb7c41f0c>,
'INSTALLSTR': <function installStr at 0xb7c41f44>,
'LATEXSUFFIXES': ['.tex', '.ltx', '.latex'],
'LIBPREFIX': 'lib',
'LIBPREFIXES': '$LIBPREFIX',
'LIBSUFFIX': '.a',
'LIBSUFFIXES': ['$LIBSUFFIX', '$SHLIBSUFFIX'],
'MAXLINELENGTH': 128072,
'OBJPREFIX': "",
'OBJSUFFIX': '.o',
'PLATFORM': 'posix',
'PROGPREFIX': "",
'PROGSUFFIX': "",
'PSPAWN': <function piped_env_spawn at 0xb7b66fb4>,
'RDirs': <SCons.Defaults.Variable_Method_Caller instance at 0xb7c43c4c>,
'SCANNERS': [],
'SHELL': 'sh',
'SHLIBPREFIX': '$LIBPREFIX',
'SHLIBSUFFIX': '.so',
'SHOBJPREFIX': '$OBJPREFIX',
'SHOBJSUFFIX': '$OBJSUFFIX',
'SPAWN': <function spawnvpe_spawn at 0xb7b66a74>,
'TEMPFILE': <class SCons.Platform.TempFileMunge at 0xb7bd37ac>,
'TEMPFILEPREFIX': '@',
'TOOLS': [],
'_CPPDEFFLAGS': '${_defines(CPPDEFPREFIX, CPPDEFINES, CPPDEFSUFFIX, __env__)}',
'_CPPINCFLAGS': '${( ${_concat(INCPREFIX, CPPPATH, INCSUFFIX, __env__, RDirs, TA
)_LIBDIRFLAGS': '${( ${_concat(LIBDIRPREFIX, LIBPATH, LIBDIRSUFFIX, __env__, RDi
)_LIBFLAGS': '${_concat(LIBLINKPREFIX, LIBS, LIBLINKSUFFIX, __env__)}',
'_RPATH': '$RPATH',
'_concat': <function _concat at 0xb7c41fb4>,
'_defines': <function _defines at 0xb7c47064>,
'_installStr': <function installStr at 0xb7c41f44>,
'_stripixes': <function _stripixes at 0xb7c4702c>}
scons: done reading SConscript files.
scons: Building targets ...
scons: '.' is up to date.
scons: done building targets.

```

On a Windows system with Visual C++ the output might look like:

```

C:\>scons
scons: Reading SConscript files ...
{ 'BUILDERS': {'Object': <SCons.Builder.CompositeBuilder instance at 0xb7b6024c>
'CC': 'cl',
'CCCOM': <SCons.Action.FunctionAction instance at 0xb7b6086c>,
'CCCOMFLAGS': '$CPPFLAGS $CPPDEFFLAGS $CPPINCFLAGS /c $SOURCES /Fo$TARGET $CO
'CCFLAGS': ['/nologo'],
'CCPCHFLAGS': ['${(PCH and "/Yu%s /Fp%s"%(PCHSTOP or "",File(PCH))) or ""}'],
'CCPDBFLAGS': ['${(PDB and "/Z7") or ""}'],
'FILESUFFIX': '.c',
'CFLAGS': [],
'CONFIGUREDIR': '#/.sconf_temp',
'CONFIGURELOG': '#/config.log',
'CPPDEFPREFIX': '/D',
'CPPDEFSUFFIX': "",
'CPPSUFFIXES': [ '.c',

```

```

        '.C',
        '.cxx',
        '.cpp',
        '.c++',
        '.cc',
        '.h',
        '.H',
        '.hxx',
        '.hpp',
        '.hh',
        '.F',
        '.fpp',
        '.FPP',
        '.m',
        '.mm',
        '.S',
        '.spp',
        '.SPP'],
'CXX': '$CC',
'CXXCOM': '$CXX $CXXFLAGS $CCCOMFLAGS',
'CXXFILESUFFIX': '.cc',
'CXXFLAGS': ['$CCFLAGS', '$(', '/TP', '$)'],
'DSUFFIXES': ['.d'],
'Dir': <SCons.Defaults.Variable_Method_Caller instance at 0xb7c58bec>,
'Dirs': <SCons.Defaults.Variable_Method_Caller instance at 0xb7c58c0c>,
'ENV': { 'INCLUDE': 'C:\\Program Files\\Microsoft Visual Studio\\VC98\\include',
        'LIB': 'C:\\Program Files\\Microsoft Visual Studio\\VC98\\lib',
        'PATH': 'C:\\Program Files\\Microsoft Visual Studio\\Common\\tools\\WI',
        'PATHEXT': '.COM;.EXE;.BAT;.CMD',
        'SystemRoot': 'C:/WINDOWS'},
'ESCAPE': <function escape at 0xb7bc917c>,
'File': <SCons.Defaults.Variable_Method_Caller instance at 0xb7c58c2c>,
'IDLSUFFIXES': ['.idl', '.IDL'],
'INCPREFIX': '/I',
'INCSUFFIX': "",
'INSTALL': <function installFunc at 0xb7c56f0c>,
'INSTALLSTR': <function installStr at 0xb7c56f44>,
'LATEXSUFFIXES': ['.tex', '.ltx', '.latex'],
'LIBPREFIX': "",
'LIBPREFIXES': ['$LIBPREFIX'],
'LIBSUFFIX': '.lib',
'LIBSUFFIXES': ['$LIBSUFFIX'],
'MAXLINELENGTH': 2048,
'MSVS': {'VERSION': '6.0', 'VERSIONS': ['6.0']},
'MSVS_VERSION': '6.0',
'OBJPREFIX': "",
'OBJSUFFIX': '.obj',
'PCHCOM': '$CXX $CXXFLAGS $CPPFLAGS $_CPPDEFFLAGS $_CPPINCFLAGS /c $SOURCES /Fo',
'PCHPDBFLAGS': ['$${(PDB and "/Yd") or ""}'],
'PLATFORM': 'win32',
'PROGPREFIX': "",
'PROGSUFFIX': '.exe',
'PSPAWN': <function piped_spawn at 0xb7bc90d4>,
'RC': 'rc',
'RCCOM': '$RC $_CPPDEFFLAGS $_CPPINCFLAGS $RCFLAGS /fo$TARGET $SOURCES',
'RCFLAGS': [],
'RDirs': <SCons.Defaults.Variable_Method_Caller instance at 0xb7c58c4c>,
'SCANNERS': [],
'SHCC': '$CC',
'SHCCCOM': <SCons.Action.FunctionAction instance at 0xb7b608cc>,
'SHCCFLAGS': ['$CCFLAGS'],
'SHCFLAGS': ['$CFLAGS'],
'SHCXX': '$CXX',
'SHCXXCOM': '$SHCXX $SHCXXFLAGS $CCCOMFLAGS',
'SHCXXFLAGS': ['$CXXFLAGS'],
'SHELL': None,

```

```

'SHLIBPREFIX': '',
'SHLIBSUFFIX': '.dll',
'SHOBJPREFIX': '$OBJPREFIX',
'SHOBJSUFFIX': '$OBJSUFFIX',
'SPAWN': <function spawn at 0xb7bc9144>,
'STATIC_AND_SHARED_OBJECTS_ARE_THE_SAME': 1,
'TEMPFILE': <class SCons.Platform.TempFileMunge at 0xb7be87ac>,
'TEMPFILEPREFIX': '@',
'TOOLS': ['msvc'],
'_CPPDEFFLAGS': '${_defines(CPPDEFPREFIX, CPPDEFINES, CPPDEFSUFFIX, __env__)}',
'_CPPINCFLAGS': '$( ${_concat(INCPREFIX, CPPPATH, INCSUFFIX, __env__, RDirs, TA
'_LIBDIRFLAGS': '$( ${_concat(LIBDIRPREFIX, LIBPATH, LIBDIRSUFFIX, __env__, RDi
'_LIBFLAGS': '${_concat(LIBLINKPREFIX, LIBS, LIBLINKSUFFIX, __env__)}',
'_concat': <function _concat at 0xb7c56fb4>,
'_defines': <function _defines at 0xb7c5c064>,
'_installStr': <function installStr at 0xb7c56f44>,
'_stripixes': <function _stripixes at 0xb7c5c02c>}
scons: done reading SConscript files.
scons: Building targets ...
scons: '.' is up to date.
scons: done building targets.

```

The construction environments in these examples have actually been restricted to just gcc and Visual C++, respectively. In a real-life situation, the construction environments will likely contain a great many more variables.

To make it easier to see just what you're interested in, the `Dump` method allows you to specify a specific construction variable that you want to display. For example, it's not unusual to want to verify the external environment used to execute build commands, to make sure that the `PATH` and other environment variables are set up the way they should be. You can do this as follows:

Which might display the following when executed on a POSIX system:

```

% scons
scons: Reading SConscript files ...
{'PATH': '/usr/local/bin:/opt/bin:/bin:/usr/bin'}
scons: done reading SConscript files.
scons: Building targets ...
scons: '.' is up to date.
scons: done building targets.

```

And the following when executed on a Windows system:

```

C:\>scons
scons: Reading SConscript files ...
{ 'INCLUDE': 'C:\\Program Files\\Microsoft Visual Studio\\VC98\\include',
  'LIB': 'C:\\Program Files\\Microsoft Visual Studio\\VC98\\lib',
  'PATH': 'C:\\Program Files\\Microsoft Visual Studio\\Common\\tools\\WIN95;C:\\F
  'PATHEXT': '.COM;.EXE;.BAT;.CMD',
  'SystemRoot': 'C:/WINDOWS' }
scons: done reading SConscript files.
scons: Building targets ...
scons: '.' is up to date.
scons: done building targets.

```


Appendix A. Construction Variables

This appendix contains descriptions of all of the construction variables that are *potentially* available "out of the box" in this version of SCons. Whether or not setting a construction variable in a construction environment will actually have an effect depends on whether any of the Tools and/or Builders that use the variable have been included in the construction environment.

In this appendix, we have appended the initial \$ (dollar sign) to the beginning of each variable name when it appears in the text, but left off the dollar sign in the left-hand column where the name appears for each entry.

AR

The static library archiver.

ARCOM

The command line used to generate a static library from object files.

ARCOMSTR

The string displayed when an object file is generated from an assembly-language source file. If this is not set, then \$ARCOM (the command line) is displayed.

```
env = Environment(ARCOMSTR = "Archiving $TARGET")
```

ARFLAGS

General options passed to the static library archiver.

AS

The assembler.

ASCOM

The command line used to generate an object file from an assembly-language source file.

ASCOMSTR

The string displayed when an object file is generated from an assembly-language source file. If this is not set, then \$ASCOM (the command line) is displayed.

```
env = Environment(ASCOMSTR = "Assembling $TARGET")
```

ASFLAGS

General options passed to the assembler.

ASPPCOM

The command line used to assemble an assembly-language source file into an object file after first running the file through the C preprocessor. Any options specified in the \$ASFLAGS and \$CPPFLAGS construction variables are included on this command line.

ASPPCOMSTR

The string displayed when an object file is generated from an assembly-language source file after first running the file through the C preprocessor. If this is not set, then \$ASPPCOM (the command line) is displayed.

```
env = Environment(ASPPCOMSTR = "Assembling $TARGET")
```

ASPPFLAGS

General options when assembling an assembly-language source file into an object file after first running the file through the C preprocessor. The default is to use the value of \$ASFLAGS.

BIBTEX

The bibliography generator for the TeX formatter and typesetter and the LaTeX structured formatter and typesetter.

BIBTEXCOM

The command line used to call the bibliography generator for the TeX formatter and typesetter and the LaTeX structured formatter and typesetter.

BIBTEXCOMSTR

The string displayed when generating a bibliography for TeX or LaTeX. If this is not set, then \$BIBTEXCOM (the command line) is displayed.

```
env = Environment(BIBTEXCOMSTR = "Generating bibliography $TARGET")
```

BIBTEXFLAGS

General options passed to the bibliography generator for the TeX formatter and typesetter and the LaTeX structured formatter and typesetter.

BITKEEPER

The BitKeeper executable.

BITKEEPERCOM

The command line for fetching source files using BitKeeper.

BITKEEPERCOMSTR

The string displayed when fetching a source file using BitKeeper. If this is not set, then \$BITKEEPERCOM (the command line) is displayed.

BITKEEPERGET

The command (\$BITKEEPER) and subcommand for fetching source files using BitKeeper.

BITKEEPERGETFLAGS

Options that are passed to the BitKeeper **get** subcommand.

BUILDERS

A dictionary mapping the names of the builders available through this environment to underlying Builder objects. Builders named Alias, CFile, CXXFile, DVI, Library, Object, PDF, PostScript, and Program are available by default. If you initialize this variable when an Environment is created:

```
env = Environment(BUILDERS = {'NewBuilder' : foo})
```

the default Builders will no longer be available. To use a new Builder object in addition to the default Builders, add your new Builder object like this:

```
env = Environment()  
env.Append(BUILDERS = {'NewBuilder' : foo})
```

or this:

```
env = Environment()
env['BUILDERS']['NewBuilder'] = foo
```

CC

The C compiler.

CCCOM

The command line used to compile a C source file to a (static) object file. Any options specified in the \$CFLAGS, \$CCFLAGS and \$CPPFLAGS construction variables are included on this command line.

CCCOMSTR

The string displayed when a C source file is compiled to a (static) object file. If this is not set, then \$CCCOM (the command line) is displayed.

```
env = Environment(CCCOMSTR = "Compiling static object $TARGET")
```

CCFLAGS

General options that are passed to the C and C++ compilers.

CCPCHFLAGS

Options added to the compiler command line to support building with precompiled headers. The default value expands to the appropriate Microsoft Visual C++ command-line options when the \$PCH construction variable is set.

CCPDBFLAGS

Options added to the compiler command line to support storing debugging information in a Microsoft Visual C++ PDB file. The default value expands to appropriate Microsoft Visual C++ command-line options when the \$PDB construction variable is set.

The Visual C++ compiler option that SCons uses by default to generate PDB information is /Z7. This works correctly with parallel (-j) builds because it embeds the debug information in the intermediate object files, as opposed to sharing a single PDB file between multiple object files. This is also the only way to get debug information embedded into a static library. Using the /Zi instead may yield improved link-time performance, although parallel builds will no longer work.

You can generate PDB files with the /Zi switch by overriding the default \$CCPDBFLAGS variable as follows:

```
import SCons.Util
env['CCPDBFLAGS'] = SCons.Util.CLVar(['${(PDB and "/Zi /Fd%s" % File(PDB)) or ""}']
```

An alternative would be to use the /Zi to put the debugging information in a separate .pdb file for each object file by overriding the \$CCPDBFLAGS variable as follows:

```
env['CCPDBFLAGS'] = '/Zi /Fd${TARGET}.pdb'
```

CCVERSION

The version number of the C compiler. This may or may not be set, depending on the specific C compiler being used.

CFILESUFFIX

The suffix for C source files. This is used by the internal CFile builder when generating C files from Lex (.l) or YACC (.y) input files. The default suffix, of course, is `.c` (lower case). On case-insensitive systems (like Windows), SCons also treats `.C` (upper case) files as C files.

CFLAGS

General options that are passed to the C compiler (C only; not C++).

`_concat`

A function used to produce variables like `$_CPPINCFLAGS`. It takes four or five arguments: a prefix to concatenate onto each element, a list of elements, a suffix to concatenate onto each element, an environment for variable interpolation, and an optional function that will be called to transform the list before concatenation.

```
env['_CPPINCFLAGS'] = '$( ${_concat(INCPREFIX, CPPPATH, INCSUFFIX, __env__, RDirs)}
```

CONFIGUREDIR

The name of the directory in which Configure context test files are written. The default is `.sconf_temp` in the top-level directory containing the SConstruct file.

CONFIGURELOG

The name of the Configure context log file. The default is `config.log` in the top-level directory containing the SConstruct file.

`_CPPDEFFLAGS`

An automatically-generated construction variable containing the C preprocessor command-line options to define values. The value of `$_CPPDEFFLAGS` is created by appending `$CPPDEFPREFIX` and `$CPPDEFSUFFIX` to the beginning and end of each directory in `$CPPDEFINES`.

CPPDEFINES

A platform independent specification of C preprocessor definitions. The definitions will be added to command lines through the automatically-generated `$_CPPDEFFLAGS` construction variable (see above), which is constructed according to the type of value of `$CPPDEFINES`:

If `$CPPDEFINES` is a string, the values of the `$CPPDEFPREFIX` and `$CPPDEFSUFFIX` construction variables will be added to the beginning and end.

```
# Will add -Dxyz to POSIX compiler command lines,
# and /Dxyz to Microsoft Visual C++ command lines.
env = Environment(CPPDEFINES='xyz')
```

If `$CPPDEFINES` is a list, the values of the `$CPPDEFPREFIX` and `$CPPDEFSUFFIX` construction variables will be appended to the beginning and end of each element in the list. If any element is a list or tuple, then the first item is the name being defined and the second item is its value:

```
# Will add -DB=2 -DA to POSIX compiler command lines,
# and /DB=2 /DA to Microsoft Visual C++ command lines.
env = Environment(CPPDEFINES=[('B', 2), 'A'])
```

If `$CPPDEFINES` is a dictionary, the values of the `$CPPDEFPREFIX` and `$CPPDEFSUFFIX` construction variables will be appended to the beginning and end of each item from the dictionary. The key of each dictionary item is a name being defined to the dictionary item's corresponding value; if the value is `None`, then the name is defined without an explicit value. Note that the resulting flags are

sorted by keyword to ensure that the order of the options on the command line is consistent each time `scons` is run.

```
# Will add -DA -DB=2 to POSIX compiler command lines,
# and /DA /DB=2 to Microsoft Visual C++ command lines.
env = Environment(CPPDEFINES={'B':2, 'A':None})
```

CPPDEFPREFIX

The prefix used to specify preprocessor definitions on the C compiler command line. This will be appended to the beginning of each definition in the `$CPPDEFINES` construction variable when the `$_CPPDEFFLAGS` variable is automatically generated.

CPPDEFSUFFIX

The suffix used to specify preprocessor definitions on the C compiler command line. This will be appended to the end of each definition in the `$CPPDEFINES` construction variable when the `$_CPPDEFFLAGS` variable is automatically generated.

CPPFLAGS

User-specified C preprocessor options. These will be included in any command that uses the C preprocessor, including not just compilation of C and C++ source files via the `$CCCOM`, `$SHCCCOM`, `$CXXCOM` and `$SHCXXCOM` command lines, but also the `$FORTRANPPCOM`, `$SHFORTRANPPCOM`, `$F77PPCOM` and `$SHF77PPCOM` command lines used to compile a Fortran source file, and the `$ASPPCOM` command line used to assemble an assembly language source file, after first running each file through the C preprocessor. Note that this variable does *not* contain `-I` (or similar) include search path options that `scons` generates automatically from `$CPPPATH`. See `$_CPPINCFLAGS`, below, for the variable that expands to those options.

\$_CPPINCFLAGS

An automatically-generated construction variable containing the C preprocessor command-line options for specifying directories to be searched for include files. The value of `$_CPPINCFLAGS` is created by appending `$INCPREFIX` and `$INCSUFFIX` to the beginning and end of each directory in `$CPPPATH`.

CPPPATH

The list of directories that the C preprocessor will search for include directories. The C/C++ implicit dependency scanner will search these directories for include files. Don't explicitly put include directory arguments in `CCFLAGS` or `CXXFLAGS` because the result will be non-portable and the directories will not be searched by the dependency scanner. Note: directory names in `CPPPATH` will be looked-up relative to the `SConscript` directory when they are used in a command. To force `scons` to look-up a directory relative to the root of the source tree use `#`:

```
env = Environment(CPPPATH='#/include')
```

The directory look-up can also be forced using the `Dir()` function:

```
include = Dir('include')
env = Environment(CPPPATH=include)
```

The directory list will be added to command lines through the automatically-generated `$_CPPINCFLAGS` construction variable, which is constructed by appending the values of the `$INCPREFIX` and `$INCSUFFIX` construction variables to the beginning and end of each directory in `$CPPPATH`. Any command lines you define that need the `CPPPATH` directory list should include `$_CPPINCFLAGS`:

```
env = Environment(CCCOM="my_compiler $_CPPINCFLAGS -c -o $TARGET $SOURCE")
```

CPPSUFFIXES

The list of suffixes of files that will be scanned for C preprocessor implicit dependencies (#include lines). The default list is:

```
[ ".c", ".C", ".cxx", ".cpp", ".c++", ".cc",  
  ".h", ".H", ".hxx", ".hpp", ".hh",  
  ".F", ".fpp", ".FPP",  
  ".m", ".mm",  
  ".S", ".spp", ".SPP" ]
```

CVS

The CVS executable.

CVSCOFLAGS

Options that are passed to the CVS checkout subcommand.

CVSCOM

The command line used to fetch source files from a CVS repository.

CVSCOMSTR

The string displayed when fetching a source file from a CVS repository. If this is not set, then \$CVSCOM (the command line) is displayed.

CVSFLAGS

General options that are passed to CVS. By default, this is set to -d \$CVSREPOSITORY to specify from where the files must be fetched.

CVSREPOSITORY

The path to the CVS repository. This is referenced in the default \$CVSFLAGS value.

CXX

The C++ compiler.

CXXCOM

The command line used to compile a C++ source file to an object file. Any options specified in the \$CXXFLAGS and \$CPPFLAGS construction variables are included on this command line.

CXXCOMSTR

The string displayed when a C++ source file is compiled to a (static) object file. If this is not set, then \$CXXCOM (the command line) is displayed.

```
env = Environment(CXXCOMSTR = "Compiling static object $TARGET")
```

CXXFILESUFFIX

The suffix for C++ source files. This is used by the internal CXXFile builder when generating C++ files from Lex (.ll) or YACC (.yy) input files. The default suffix is .cc. SCons also treats files with the suffixes .cpp, .cxx, .c++, and .C++ as C++ files, and files with .mm suffixes as Objective C++ files. On case-sensitive systems

(Linux, UNIX, and other POSIX-alikes), SCons also treats .c (upper case) files as C++ files.

CXXFLAGS

General options that are passed to the C++ compiler. By default, this includes the value of \$CCFLAGS, so that setting \$CXXFLAGS affects both C and C++ compilation. If you want to add C++-specific flags, you must set or override the value of \$CXXFLAGS.

CXXVERSION

The version number of the C++ compiler. This may or may not be set, depending on the specific C++ compiler being used.

Dir

A function that converts a string into a Dir instance relative to the target being built.

Dirs

A function that converts a list of strings into a list of Dir instances relative to the target being built.

DSUFFIXES

The list of suffixes of files that will be scanned for imported D package files. The default list is:

```
[ '.d' ]
```

DVIPDF

The TeX DVI file to PDF file converter.

DVIPDFCOM

The command line used to convert TeX DVI files into a PDF file.

DVIPDFCOMSTR

The string displayed when a TeX DVI file is converted into a PDF file. If this is not set, then \$DVIPDFCOM (the command line) is displayed.

DVIPDFFLAGS

General options passed to the TeX DVI file to PDF file converter.

DVIPS

The TeX DVI file to PostScript converter.

DVIPSFLAGS

General options passed to the TeX DVI file to PostScript converter.

ENV

A dictionary of environment variables to use when invoking commands. When \$ENV is used in a command all list values will be joined using the path separator and any other non-string values will simply be coerced to a string. Note that, by default, *scons* does *not* propagate the environment in force when you execute *scons* to the commands used to build target files. This is so that builds will be guaranteed repeatable regardless of the environment variables set at the time *scons* is invoked.

If you want to propagate your environment variables to the commands executed to build target files, you must do so explicitly:

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

Note that you can choose only to propagate certain environment variables. A common example is the system PATH environment variable, so that `scons` uses the same utilities as the invoking shell (or other process):

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

ESCAPE

A function that will be called to escape shell special characters in command lines. The function should take one argument: the command line string to escape; and should return the escaped command line.

F77

The Fortran 77 compiler. You should normally set the `$FORTRAN` variable, which specifies the default Fortran compiler for all Fortran versions. You only need to set `$F77` if you need to use a specific compiler or compiler version for Fortran 77 files.

F77COM

The command line used to compile a Fortran 77 source file to an object file. You only need to set `$F77COM` if you need to use a specific command line for Fortran 77 files. You should normally set the `$FORTRANCOM` variable, which specifies the default command line for all Fortran versions.

F77COMSTR

The string displayed when a Fortran 77 source file is compiled to an object file. If this is not set, then `$F77COM` or `$FORTRANCOM` (the command line) is displayed.

F77FLAGS

General user-specified options that are passed to the Fortran 77 compiler. Note that this variable does *not* contain `-I` (or similar) include search path options that `scons` generates automatically from `$F77PATH`. See `$_F77INCFLAGS` below, for the variable that expands to those options. You only need to set `$F77FLAGS` if you need to define specific user options for Fortran 77 files. You should normally set the `$FORTRANFLAGS` variable, which specifies the user-specified options passed to the default Fortran compiler for all Fortran versions.

\$_F77INCFLAGS

An automatically-generated construction variable containing the Fortran 77 compiler command-line options for specifying directories to be searched for include files. The value of `$_F77INCFLAGS` is created by appending `$INCPREFIX` and `$INCSUFFIX` to the beginning and end of each directory in `$F77PATH`.

F77PATH

The list of directories that the Fortran 77 compiler will search for include directories. The implicit dependency scanner will search these directories for include files. Don't explicitly put include directory arguments in `$F77FLAGS` because the result will be non-portable and the directories will not be searched by the dependency scanner. Note: directory names in `$F77PATH` will be looked-up relative to the SConscript directory when they are used in a command. To force `scons` to

look-up a directory relative to the root of the source tree use #: You only need to set \$F77PATH if you need to define a specific include path for Fortran 77 files. You should normally set the \$FORTRANPATH variable, which specifies the include path for the default Fortran compiler for all Fortran versions.

```
env = Environment(F77PATH='#/include')
```

The directory look-up can also be forced using the `Dir()` function:

```
include = Dir('include')
env = Environment(F77PATH=include)
```

The directory list will be added to command lines through the automatically-generated `$_F77INCFLAGS` construction variable, which is constructed by appending the values of the `$INCPREFIX` and `$INCSUFFIX` construction variables to the beginning and end of each directory in `$F77PATH`. Any command lines you define that need the `F77PATH` directory list should include `$_F77INCFLAGS`:

```
env = Environment(F77COM="my_compiler $_F77INCFLAGS -c -o $TARGET $SOURCE")
```

F77PPCOM

The command line used to compile a Fortran 77 source file to an object file after first running the file through the C preprocessor. Any options specified in the `$F77FLAGS` and `$CPPFLAGS` construction variables are included on this command line. You only need to set `$F77PPCOM` if you need to use a specific C-preprocessor command line for Fortran 77 files. You should normally set the `$FORTRANPPCOM` variable, which specifies the default C-preprocessor command line for all Fortran versions.

F90

The Fortran 90 compiler. You should normally set the `$FORTRAN` variable, which specifies the default Fortran compiler for all Fortran versions. You only need to set `$F90` if you need to use a specific compiler or compiler version for Fortran 90 files.

F90COM

The command line used to compile a Fortran 90 source file to an object file. You only need to set `$F90COM` if you need to use a specific command line for Fortran 90 files. You should normally set the `$FORTRANCOM` variable, which specifies the default command line for all Fortran versions.

F90COMSTR

The string displayed when a Fortran 90 source file is compiled to an object file. If this is not set, then `$F90COM` or `$FORTRANCOM` (the command line) is displayed.

F90FLAGS

General user-specified options that are passed to the Fortran 90 compiler. Note that this variable does *not* contain `-I` (or similar) include search path options that `scons` generates automatically from `$F90PATH`. See `$_F90INCFLAGS` below, for the variable that expands to those options. You only need to set `$F90FLAGS` if you need to define specific user options for Fortran 90 files. You should normally set the `$FORTRANFLAGS` variable, which specifies the user-specified options passed to the default Fortran compiler for all Fortran versions.

\$_F90INCFLAGS

An automatically-generated construction variable containing the Fortran 90 compiler command-line options for specifying directories to be searched

for include files. The value of `$_F90INCFLAGS` is created by appending `$INCPREFIX` and `$INCSUFFIX` to the beginning and end of each directory in `$F90PATH`.

F90PATH

The list of directories that the Fortran 90 compiler will search for include directories. The implicit dependency scanner will search these directories for include files. Don't explicitly put include directory arguments in `$F90FLAGS` because the result will be non-portable and the directories will not be searched by the dependency scanner. Note: directory names in `$F90PATH` will be looked-up relative to the SConscript directory when they are used in a command. To force `scons` to look-up a directory relative to the root of the source tree use `#`: You only need to set `$F90PATH` if you need to define a specific include path for Fortran 90 files. You should normally set the `$FORTRANPATH` variable, which specifies the include path for the default Fortran compiler for all Fortran versions.

```
env = Environment(F90PATH='#/include')
```

The directory look-up can also be forced using the `Dir()` function:

```
include = Dir('include')
env = Environment(F90PATH=include)
```

The directory list will be added to command lines through the automatically-generated `$_F90INCFLAGS` construction variable, which is constructed by appending the values of the `$INCPREFIX` and `$INCSUFFIX` construction variables to the beginning and end of each directory in `$F90PATH`. Any command lines you define that need the `F90PATH` directory list should include `$_F90INCFLAGS`:

```
env = Environment(F90COM="my_compiler $_F90INCFLAGS -c -o $TARGET $SOURCE")
```

F90PPCOM

The command line used to compile a Fortran 90 source file to an object file after first running the file through the C preprocessor. Any options specified in the `$F90FLAGS` and `$CPPFLAGS` construction variables are included on this command line. You only need to set `$F90PPCOM` if you need to use a specific C-preprocessor command line for Fortran 90 files. You should normally set the `$FORTRANPPCOM` variable, which specifies the default C-preprocessor command line for all Fortran versions.

F95

The Fortran 95 compiler. You should normally set the `$FORTRAN` variable, which specifies the default Fortran compiler for all Fortran versions. You only need to set `$F95` if you need to use a specific compiler or compiler version for Fortran 95 files.

F95COM

The command line used to compile a Fortran 95 source file to an object file. You only need to set `$F95COM` if you need to use a specific command line for Fortran 95 files. You should normally set the `$FORTRANCOM` variable, which specifies the default command line for all Fortran versions.

F95COMSTR

The string displayed when a Fortran 95 source file is compiled to an object file. If this is not set, then `$F95COM` or `$FORTRANCOM` (the command line) is displayed.

F95FLAGS

General user-specified options that are passed to the Fortran 95 compiler. Note that this variable does *not* contain `-I` (or similar) include search path options that `scons` generates automatically from `$F95PATH`. See `$_F95INCFLAGS` below, for the variable that expands to those options. You only need to set `$F95FLAGS` if you need to define specific user options for Fortran 95 files. You should normally set the `$FORTRANFLAGS` variable, which specifies the user-specified options passed to the default Fortran compiler for all Fortran versions.

\$_F95INCFLAGS

An automatically-generated construction variable containing the Fortran 95 compiler command-line options for specifying directories to be searched for include files. The value of `$_F95INCFLAGS` is created by appending `$INCPREFIX` and `$INCSUFFIX` to the beginning and end of each directory in `$F95PATH`.

F95PATH

The list of directories that the Fortran 95 compiler will search for include directories. The implicit dependency scanner will search these directories for include files. Don't explicitly put include directory arguments in `$F95FLAGS` because the result will be non-portable and the directories will not be searched by the dependency scanner. Note: directory names in `$F95PATH` will be looked-up relative to the `SConscript` directory when they are used in a command. To force `scons` to look-up a directory relative to the root of the source tree use `#`: You only need to set `$F95PATH` if you need to define a specific include path for Fortran 95 files. You should normally set the `$FORTRANPATH` variable, which specifies the include path for the default Fortran compiler for all Fortran versions.

```
env = Environment(F95PATH='#/include')
```

The directory look-up can also be forced using the `Dir()` function:

```
include = Dir('include')
env = Environment(F95PATH=include)
```

The directory list will be added to command lines through the automatically-generated `$_F95INCFLAGS` construction variable, which is constructed by appending the values of the `$INCPREFIX` and `$INCSUFFIX` construction variables to the beginning and end of each directory in `$F95PATH`. Any command lines you define that need the `F95PATH` directory list should include `$_F95INCFLAGS`:

```
env = Environment(F95COM="my_compiler $_F95INCFLAGS -c -o $TARGET $SOURCE")
```

F95PPCOM

The command line used to compile a Fortran 95 source file to an object file after first running the file through the C preprocessor. Any options specified in the `$F95FLAGS` and `$CPPFLAGS` construction variables are included on this command line. You only need to set `$F95PPCOM` if you need to use a specific C-preprocessor command line for Fortran 95 files. You should normally set the `$FORTRANPPCOM` variable, which specifies the default C-preprocessor command line for all Fortran versions.

File

A function that converts a string into a `File` instance relative to the target being built.

FORTTRAN

The default Fortran compiler for all versions of Fortran.

`FORTRANCOM`

The command line used to compile a Fortran source file to an object file. By default, any options specified in the `$FORTRANFLAGS`, `$CPPFLAGS`, `$_CPPDEFFLAGS`, `$_FORTRANMODFLAG`, and `$_FORTRANINCFLAGS` construction variables are included on this command line.

`FORTRANCOMSTR`

The string displayed when a Fortran source file is compiled to an object file. If this is not set, then `$FORTRANCOM` (the command line) is displayed.

`FORTRANFLAGS`

General user-specified options that are passed to the Fortran compiler. Note that this variable does *not* contain `-I` (or similar) include or module search path options that `scons` generates automatically from `$FORTRANPATH`. See `$_FORTRANINCFLAGS` and `$_FORTRANMODFLAG`, below, for the variables that expand those options.

`$_FORTRANINCFLAGS`

An automatically-generated construction variable containing the Fortran compiler command-line options for specifying directories to be searched for include files and module files. The value of `$_FORTRANINCFLAGS` is created by prepending/appending `$INCPREFIX` and `$INCSUFFIX` to the beginning and end of each directory in `$FORTRANPATH`.

`FORTRANMODDIR`

Directory location where the Fortran compiler should place any module files it generates. This variable is empty, by default. Some Fortran compilers will internally append this directory in the search path for module files, as well.

`FORTRANMODDIRPREFIX`

The prefix used to specify a module directory on the Fortran compiler command line. This will be appended to the beginning of the directory in the `$FORTRANMODDIR` construction variables when the `$_FORTRANMODFLAG` variables is automatically generated.

`FORTRANMODDIRSUFFIX`

The suffix used to specify a module directory on the Fortran compiler command line. This will be appended to the beginning of the directory in the `$FORTRANMODDIR` construction variables when the `$_FORTRANMODFLAG` variables is automatically generated.

`$_FORTRANMODFLAG`

An automatically-generated construction variable containing the Fortran compiler command-line option for specifying the directory location where the Fortran compiler should place any module files that happen to get generated during compilation. The value of `$_FORTRANMODFLAG` is created by prepending/appending `$FORTRANMODDIRPREFIX` and `$FORTRANMODDIRSUFFIX` to the beginning and end of the directory in `$FORTRANMODDIR`.

`FORTRANMODPREFIX`

The module file prefix used by the Fortran compiler. `SCons` assumes that the Fortran compiler follows the quasi-standard naming convention for module files of `module_name.mod`. As a result, this variable is left empty, by default. For situations in which the compiler does not necessarily follow the normal convention,

the user may use this variable. Its value will be appended to every module file name as `scons` attempts to resolve dependencies.

FORTRANMODSUFFIX

The module file suffix used by the Fortran compiler. `SCons` assumes that the Fortran compiler follows the quasi-standard naming convention for module files of `module_name.mod`. As a result, this variable is set to `".mod"`, by default. For situations in which the compiler does not necessarily follow the normal convention, the user may use this variable. Its value will be appended to every module file name as `scons` attempts to resolve dependencies.

FORTRANPATH

The list of directories that the Fortran compiler will search for include files and (for some compilers) module files. The Fortran implicit dependency scanner will search these directories for include files (but not module files since they are autogenerated and, as such, may not actually exist at the time the scan takes place). Don't explicitly put include directory arguments in `FORTRANFLAGS` because the result will be non-portable and the directories will not be searched by the dependency scanner. Note: directory names in `FORTRANPATH` will be looked-up relative to the `SConscript` directory when they are used in a command. To force `scons` to look-up a directory relative to the root of the source tree use #:

```
env = Environment(FORTRANPATH='#/include')
```

The directory look-up can also be forced using the `Dir()` function:

```
include = Dir('include')
env = Environment(FORTRANPATH=include)
```

The directory list will be added to command lines through the automatically-generated `$_FORTRANINCFLAGS` construction variable, which is constructed by appending the values of the `$INCPREFIX` and `$INCSUFFIX` construction variables to the beginning and end of each directory in `$FORTRANPATH`. Any command lines you define that need the `FORTRANPATH` directory list should include `$_FORTRANINCFLAGS`:

```
env = Environment(FORTRANCOM="my_compiler $_FORTRANINCFLAGS -c -o $TARGET $SOURCE")
```

FORTRANPPCOM

The command line used to compile a Fortran source file to an object file after first running the file through the C preprocessor. By default, any options specified in the `$FORTRANFLAGS`, `$CPPFLAGS`, `_CPPDEFFLAGS`, `$_FORTRANMODFLAG`, and `$_FORTRANINCFLAGS` construction variables are included on this command line.

FORTRANSUFFIXES

The list of suffixes of files that will be scanned for Fortran implicit dependencies (`INCLUDE` lines and `USE` statements). The default list is:

```
[".f", ".F", ".for", ".FOR", ".ftn", ".FTN", ".fpp", ".FPP",
".f77", ".F77", ".f90", ".F90", ".f95", ".F95"]
```

FRAMEWORKPATH

On Mac OS X with `gcc`, a list containing the paths to search for frameworks. Used by the compiler to find framework-style includes like `#include <Fmwk/Header.h>`. Used by the linker to find user-specified frameworks when linking (see `$FRAMEWORKS`). For example:

```
env.AppendUnique(FRAMEWORKPATH='#myframeworkdir')
```

will add

```
... -Fmyframeworkdir
```

to the compiler and linker command lines.

`_FRAMEWORKPATH`

On Mac OS X with gcc, an automatically-generated construction variable containing the linker command-line options corresponding to `$FRAMEWORKPATH`.

`FRAMEWORKPATHPREFIX`

On Mac OS X with gcc, the prefix to be used for the `FRAMEWORKPATH` entries. (see `$FRAMEWORKPATH`). The default value is `-F`.

`FRAMEWORKPREFIX`

On Mac OS X with gcc, the prefix to be used for linking in frameworks (see `$FRAMEWORKS`). The default value is `-framework`.

`_FRAMEWORKS`

On Mac OS X with gcc, an automatically-generated construction variable containing the linker command-line options for linking with `FRAMEWORKS`.

`FRAMEWORKS`

On Mac OS X with gcc, a list of the framework names to be linked into a program or shared library or bundle. The default value is the empty list. For example:

```
env.AppendUnique(FRAMEWORKS=Split('System Cocoa SystemConfiguration'))
```

`FRAMEWORKSFLAGS`

On Mac OS X with gcc, general user-supplied frameworks options to be added at the end of a command line building a loadable module. (This has been largely superseded by the `$FRAMEWORKPATH`, `$FRAMEWORKPATHPREFIX`, `$FRAMEWORKPREFIX` and `$FRAMEWORKS` variables described above.)

`GS`

The Ghostscript program used to convert PostScript to PDF files.

`GSCOM`

The Ghostscript command line used to convert PostScript to PDF files.

`GSCOMSTR`

The string displayed when Ghostscript is used to convert a PostScript file to a PDF file. If this is not set, then `$GSCOM` (the command line) is displayed.

`GSFLAGS`

General options passed to the Ghostscript program when converting PostScript to PDF files.

`IDLSUFFIXES`

The list of suffixes of files that will be scanned for IDL implicit dependencies (`#include` or `import` lines). The default list is:

```
[".idl", ".IDL"]
```

INCPREFIX

The prefix used to specify an include directory on the C compiler command line. This will be appended to the beginning of each directory in the \$CPPPATH and \$FORTRANPATH construction variables when the \$_CPPINCFLAGS and \$_FORTRANINCFLAGS variables are automatically generated.

INCSUFFIX

The suffix used to specify an include directory on the C compiler command line. This will be appended to the end of each directory in the \$CPPPATH and \$FORTRANPATH construction variables when the \$_CPPINCFLAGS and \$_FORTRANINCFLAGS variables are automatically generated.

INSTALL

A function to be called to install a file into a destination file name. The default function copies the file into the destination (and sets the destination file's mode and permission bits to match the source file's). The function takes the following arguments:

```
def install(dest, source, env):
```

`dest` is the path name of the destination file. `source` is the path name of the source file. `env` is the construction environment (a dictionary of construction values) in force for this file installation.

INSTALLSTR

The string displayed when a file is installed into a destination file name. The default is:

```
Install file: "$SOURCE" as "$TARGET"
```

INTEL_C_COMPILER_VERSION

Set by the "intelc" Tool to the major version number of the Intel C compiler selected for use.

JAR

The Java archive tool.

JARCHDIR

The directory to which the Java archive tool should change (using the `-C` option).

JARCOM

The command line used to call the Java archive tool.

JARCOMSTR

The string displayed when the Java archive tool is called. If this is not set, then \$JARCOM (the command line) is displayed.

```
env = Environment(JARCOMSTR = "JARchiving $SOURCES into $TARGET")
```

JARFLAGS

General options passed to the Java archive tool. By default this is set to `cf` to create the necessary **jar** file.

JARSUFFIX

The suffix for Java archives: `.jar` by default.

JAVAC

The Java compiler.

JAVACCOM

The command line used to compile a directory tree containing Java source files to corresponding Java class files. Any options specified in the `$JAVACFLAGS` construction variable are included on this command line.

JAVACCOMSTR

The string displayed when compiling a directory tree of Java source files to corresponding Java class files. If this is not set, then `$JAVACCOM` (the command line) is displayed.

```
env = Environment(JAVACCOMSTR = "Compiling class files $TARGETS from $SOURCES")
```

JAVACFLAGS

General options that are passed to the Java compiler.

JAVACCLASSDIR

The directory in which Java class files may be found. This is stripped from the beginning of any Java `.class` file names supplied to the `JavaH` builder.

JAVACCLASSSUFFIX

The suffix for Java class files; `.class` by default.

JAVAH

The Java generator for C header and stub files.

JAVAHCOM

The command line used to generate C header and stub files from Java classes. Any options specified in the `$JAVAHFLAGS` construction variable are included on this command line.

JAVAHCOMSTR

The string displayed when C header and stub files are generated from Java classes. If this is not set, then `$JAVAHCOM` (the command line) is displayed.

```
env = Environment(JAVAHCOMSTR = "Generating header/stub file(s) $TARGETS from $SOUR
```

JAVAHFLAGS

General options passed to the C header and stub file generator for Java classes.

JAVASUFFIX

The suffix for Java files; `.java` by default.

LATEX

The LaTeX structured formatter and typesetter.

LATEXCOM

The command line used to call the LaTeX structured formatter and typesetter.

LATEXCOMSTR

The string displayed when calling the LaTeX structured formatter and typesetter. If this is not set, then \$LATEXCOM (the command line) is displayed.

```
env = Environment(LATEXCOMSTR = "Building $TARGET from LaTeX input $SOURCES")
```

LATEXFLAGS

General options passed to the LaTeX structured formatter and typesetter.

LATEXRETRIES

The maximum number of times that LaTeX will be re-run if the .log generated by the \$LATEXCOM command indicates that there are undefined references. The default is to try to resolve undefined references by re-running LaTeX up to three times.

LATEXSUFFIXES

The list of suffixes of files that will be scanned for LaTeX implicit dependencies (\include or \import files). The default list is:

```
[".tex", ".ltx", ".latex"]
```

LDMODULE

The linker for building loadable modules. By default, this is the same as \$SHLINK.

LDMODULECOM

The command line for building loadable modules. On Mac OS X, this uses the \$LDMODULE, \$LDMODULEFLAGS and \$FRAMEWORKSFLAGS variables. On other systems, this is the same as \$SHLINK.

LDMODULECOMSTR

The string displayed when building loadable modules. If this is not set, then \$LDMODULECOM (the command line) is displayed.

LDMODULEFLAGS

General user options passed to the linker for building loadable modules.

LDMODULEPREFIX

The prefix used for loadable module file names. On Mac OS X, this is null; on other systems, this is the same as \$SHLIBPREFIX.

LDMODULESUFFIX

The suffix used for loadable module file names. On Mac OS X, this is null; on other systems, this is the same as \$SHLIBSUFFIX.

LEX

The lexical analyzer generator.

LEXCOM

The command line used to call the lexical analyzer generator to generate a source file.

LEXCOMSTR

The string displayed when generating a source file using the lexical analyzer generator. If this is not set, then \$LEXCOM (the command line) is displayed.

```
env = Environment(LEXCOMSTR = "Lex'ing $TARGET from $SOURCES")
```

LEXFLAGS

General options passed to the lexical analyzer generator.

_LIBDIRFLAGS

An automatically-generated construction variable containing the linker command-line options for specifying directories to be searched for library. The value of \$_LIBDIRFLAGS is created by appending \$LIBDIRPREFIX and \$LIBDIRSUFFIX to the beginning and end of each directory in \$LIBPATH.

LIBDIRPREFIX

The prefix used to specify a library directory on the linker command line. This will be appended to the beginning of each directory in the \$LIBPATH construction variable when the \$_LIBDIRFLAGS variable is automatically generated.

LIBDIRSUFFIX

The suffix used to specify a library directory on the linker command line. This will be appended to the end of each directory in the \$LIBPATH construction variable when the \$_LIBDIRFLAGS variable is automatically generated.

_LIBFLAGS

An automatically-generated construction variable containing the linker command-line options for specifying libraries to be linked with the resulting target. The value of \$_LIBFLAGS is created by appending \$LIBLINKPREFIX and \$LIBLINKSUFFIX to the beginning and end of each filename in \$LIBS.

LIBLINKPREFIX

The prefix used to specify a library to link on the linker command line. This will be appended to the beginning of each library in the \$LIBS construction variable when the \$_LIBFLAGS variable is automatically generated.

LIBLINKSUFFIX

The suffix used to specify a library to link on the linker command line. This will be appended to the end of each library in the \$LIBS construction variable when the \$_LIBFLAGS variable is automatically generated.

LIBPATH

The list of directories that will be searched for libraries. The implicit dependency scanner will search these directories for include files. Don't explicitly put include directory arguments in \$LINKFLAGS or \$SHLINKFLAGS because the result will be non-portable and the directories will not be searched by the dependency scanner. Note: directory names in LIBPATH will be looked-up relative to the SConscript directory when they are used in a command. To force scon to look-up a directory relative to the root of the source tree use #:

```
env = Environment(LIBPATH='#/libs')
```

The directory look-up can also be forced using the Dir() function:

```
libs = Dir('libs')  
env = Environment(LIBPATH=libs)
```

The directory list will be added to command lines through the automatically-generated `$_LIBDIRFLAGS` construction variable, which is constructed by appending the values of the `$LIBDIRPREFIX` and `$LIBDIRSUFFIX` construction variables to the beginning and end of each directory in `$LIBPATH`. Any command lines you define that need the `LIBPATH` directory list should include `$_LIBDIRFLAGS`:

```
env = Environment(LINKCOM="my_linker $_LIBDIRFLAGS $_LIBFLAGS -o $TARGET $SOURCE")
```

LIBPREFIX

The prefix used for (static) library file names. A default value is set for each platform (posix, win32, os2, etc.), but the value is overridden by individual tools (ar, mslib, sgjar, sunar, tlib, etc.) to reflect the names of the libraries they create.

LIBPREFIXES

A list of all legal prefixes for library file names. When searching for library dependencies, SCons will look for files with these prefixes, the base library name, and suffixes in the `$LIBSUFFIXES` list.

LIBS

A list of one or more libraries that will be linked with any executable programs created by this environment.

The library list will be added to command lines through the automatically-generated `$_LIBFLAGS` construction variable, which is constructed by appending the values of the `$LIBLINKPREFIX` and `$LIBLINKSUFFIX` construction variables to the beginning and end of each filename in `$LIBS`. Any command lines you define that need the `LIBS` library list should include `$_LIBFLAGS`:

```
env = Environment(LINKCOM="my_linker $_LIBDIRFLAGS $_LIBFLAGS -o $TARGET $SOURCE")
```

If you add a File object to the `$LIBS` list, the name of that file will be added to `$_LIBFLAGS`, and thus the link line, as is, without `$LIBLINKPREFIX` or `$LIBLINKSUFFIX`. For example:

```
env.Append(LIBS=File('/tmp/mylib.so'))
```

In all cases, scons will add dependencies from the executable program to all the libraries in this list.

LIBSUFFIX

The suffix used for (static) library file names. A default value is set for each platform (posix, win32, os2, etc.), but the value is overridden by individual tools (ar, mslib, sgjar, sunar, tlib, etc.) to reflect the names of the libraries they create.

LIBSUFFIXES

A list of all legal suffixes for library file names. When searching for library dependencies, SCons will look for files with prefixes, in the `$LIBPREFIXES` list, the base library name, and these suffixes.

LINK

The linker.

LINKCOM

The command line used to link object files into an executable.

LINKCOMSTR

The string displayed when object files are linked into an executable. If this is not set, then \$LINKCOM (the command line) is displayed.

```
env = Environment(LINKCOMSTR = "Linking $TARGET")
```

LINKFLAGS

General user options passed to the linker. Note that this variable should *not* contain `-l` (or similar) options for linking with the libraries listed in \$LIBS, nor `-L` (or similar) library search path options that scons generates automatically from \$LIBPATH. See \$LIBFLAGS above, for the variable that expands to library-link options, and \$LIBDIRFLAGS above, for the variable that expands to library search path options.

M4

The M4 macro preprocessor.

M4COM

The command line used to pass files through the M4 macro preprocessor.

M4COMSTR

The string displayed when a file is passed through the M4 macro preprocessor. If this is not set, then \$M4COM (the command line) is displayed.

M4FLAGS

General options passed to the M4 macro preprocessor.

MAKEINDEX

The makeindex generator for the TeX formatter and typesetter and the LaTeX structured formatter and typesetter.

MAKEINDEXCOM

The command line used to call the makeindex generator for the TeX formatter and typesetter and the LaTeX structured formatter and typesetter.

MAKEINDEXCOMSTR

The string displayed when calling the makeindex generator for the TeX formatter and typesetter and the LaTeX structured formatter and typesetter. If this is not set, then \$MAKEINDEXCOM (the command line) is displayed.

MAKEINDEXFLAGS

General options passed to the makeindex generator for the TeX formatter and typesetter and the LaTeX structured formatter and typesetter.

MAXLINELENGTH

The maximum number of characters allowed on an external command line. On Win32 systems, link lines longer than this many characters are linked via a temporary file name.

MSVS

When the Microsoft Visual Studio tools are initialized, they set up this dictionary with the following keys:

VERSION: the version of MSVS being used (can be set via MSVS_VERSION)

VERSIONS: the available versions of MSVS installed

VCINSTALLDIR: installed directory of Visual C++

VSINSTALLDIR: installed directory of Visual Studio

FRAMEWORKDIR: installed directory of the .NET framework

FRAMEWORKVERSIONS: list of installed versions of the .NET framework, sorted latest to oldest.

FRAMEWORKVERSION: latest installed version of the .NET framework

FRAMEWORKSDKDIR: installed location of the .NET SDK.

PLATFORMSDKDIR: installed location of the Platform SDK.

PLATFORMSDK_MODULES: dictionary of installed Platform SDK modules, where the dictionary keys are keywords for the various modules, and the values are 2-tuples where the first is the release date, and the second is the version number.

If a value isn't set, it wasn't available in the registry.

MSVS_IGNORE_IDE_PATHS

Tells the MS Visual Studio tools to use minimal INCLUDE, LIB, and PATH settings, instead of the settings from the IDE.

For Visual Studio, SCons will (by default) automatically determine where MSVS is installed, and use the LIB, INCLUDE, and PATH variables set by the IDE. You can override this behavior by setting these variables after Environment initialization, or by setting MSVS_IGNORE_IDE_PATHS = 1 in the Environment initialization. Specifying this will not leave these unset, but will set them to a minimal set of paths needed to run the tools successfully.

For VS6, the mininimal set is:

```
INCLUDE: '<VSDir>\VC98\ATL\include;<VSDir>\VC98\MFC\include;<VSDir>\VC98\include'
LIB: '<VSDir>\VC98\MFC\lib;<VSDir>\VC98\lib'
PATH: '<VSDir>\Common\MSDev98\bin;<VSDir>\VC98\bin'
```

For VS7, it is:

```
INCLUDE: '<VSDir>\Vc7\atlmfc\include;<VSDir>\Vc7\include'
LIB: '<VSDir>\Vc7\atlmfc\lib;<VSDir>\Vc7\lib'
PATH: '<VSDir>\Common7\Tools\bin;<VSDir>\Common7\Tools;<VSDir>\Vc7\bin'
```

Where '<VSDir>' is the installed location of Visual Studio.

MSVS_PROJECT_BASE_PATH

The string placed in a generated Microsoft Visual Studio solution file as the value of the SccProjectFilePathRelativizedFromConnection0 and SccProjectFilePathRelativizedFromConnection1 attributes of the GlobalSection(SourceCodeControl) section. There is no default value.

MSVS_PROJECT_GUID

The string placed in a generated Microsoft Visual Studio project file as the value of the ProjectGUID attribute. The string is also placed in the SolutionUniqueID attribute of the GlobalSection(SourceCodeControl) section of the Microsoft Visual Studio solution file. There is no default value.

MSVS_SCC_AUX_PATH

The path name placed in a generated Microsoft Visual Studio project file as the value of the `SccAuxPath` attribute if the `MSVS_SCC_PROVIDER` construction variable is also set. There is no default value.

MSVS_SCC_LOCAL_PATH

The path name placed in a generated Microsoft Visual Studio project file as the value of the `SccLocalPath` attribute if the `MSVS_SCC_PROVIDER` construction variable is also set. The path name is also placed in the `SccLocalPath0` and `SccLocalPath1` attributes of the `GlobalSection(SourceCodeControl)` section of the Microsoft Visual Studio solution file. There is no default value.

MSVS_SCC_PROJECT_NAME

The project name placed in a generated Microsoft Visual Studio project file as the value of the `SccProjectName` attribute. There is no default value.

MSVS_SCC_PROVIDER

The string placed in a generated Microsoft Visual Studio project file as the value of the `SccProvider` attribute. The string is also placed in the `SccProvider1` attribute of the `GlobalSection(SourceCodeControl)` section of the Microsoft Visual Studio solution file. There is no default value.

MSVS_USE_MFC_DIRS

Tells the MS Visual Studio tool(s) to use the MFC directories in its default paths for compiling and linking. Under MSVS version 6, setting `MSVS_USE_MFC_DIRS` to a non-zero value adds the `ATL\include` and `MFC\include` directories to the default `INCLUDE` external environment variable, and adds the `MFC\lib` directory to the default `LIB` external environment variable. Under MSVS version 7, setting `MSVS_USE_MFC_DIRS` to a non-zero value adds the `atlmfc\include` directory to the default `INCLUDE` external environment variable, and adds the `atlmfc\lib` directory to the default `LIB` external environment variable. The current default value is 1, which means these directories are added to the paths by default. This default value is likely to change in a future release, so users who want the ATL and MFC values included in their paths are encouraged to enable the `MSVS_USE_MFC_DIRS` value explicitly to avoid future incompatibility. This variable has no effect if the `INCLUDE` or `LIB` environment variables are set explicitly.

MSVS_VERSION

Sets the preferred version of MSVS to use.

SCons will (by default) select the latest version of MSVS installed on your machine. So, if you have version 6 and version 7 (MSVS .NET) installed, it will prefer version 7. You can override this by specifying the `MSVS_VERSION` variable in the Environment initialization, setting it to the appropriate version ('6.0' or '7.0', for example). If the given version isn't installed, tool initialization will fail.

MSVSBUILDCOM

The build command line placed in a generated Microsoft Visual Studio project file. The default is to have Visual Studio invoke SCons with any specified build targets.

MSVSCLEANCOM

The clean command line placed in a generated Microsoft Visual Studio project file. The default is to have Visual Studio invoke SCons with the `-c` option to

remove any specified targets.

MSVSENCODING

The encoding string placed in a generated Microsoft Visual Studio project file. The default is encoding `Windows-1252`.

MSVSPROJECTCOM

The action used to generate Microsoft Visual Studio project files.

MSVSPROJECTSUFFIX

The suffix used for Microsoft Visual Studio project (DSP) files. The default value is `.vcproj` when using Visual Studio version 7.x (.NET), and `.dsp` when using earlier versions of Visual Studio.

MSVSREBUILDCOM

The rebuild command line placed in a generated Microsoft Visual Studio project file. The default is to have Visual Studio invoke SCons with any specified rebuild targets.

MSVSSCONS

The SCons used in generated Microsoft Visual Studio project files. The default is the version of SCons being used to generate the project file.

MSVSSCONSCOM

The default SCons command used in generated Microsoft Visual Studio project files.

MSVSSCONSCRIPT

The sconsript file (that is, `SConstruct` or `SConscript` file) that will be invoked by Visual Studio project files (through the `$MSVSSCONSCOM` variable). The default is the same sconsript file that contains the call to `MSVSProject` to build the project file.

MSVSSCONSFLAGS

The SCons flags used in generated Microsoft Visual Studio project files.

MSVSSOLUTIONCOM

The action used to generate Microsoft Visual Studio solution files.

MSVSSOLUTIONSUFFIX

The suffix used for Microsoft Visual Studio solution (DSW) files. The default value is `.sln` when using Visual Studio version 7.x (.NET), and `.dsw` when using earlier versions of Visual Studio.

MWCW_VERSION

The version number of the MetroWerks CodeWarrior C compiler to be used.

MWCW_VERSIONS

A list of installed versions of the MetroWerks CodeWarrior C compiler on this system.

no_import_lib

When set to non-zero, suppresses creation of a corresponding Windows static import lib by the `SharedLibrary` builder when used with MinGW, Microsoft

Visual Studio or Metrowerks. This also suppresses creation of an export (.exp) file when using Microsoft Visual Studio.

OBJPREFIX

The prefix used for (static) object file names.

OBJSUFFIX

The suffix used for (static) object file names.

P4

The Perforce executable.

P4COM

The command line used to fetch source files from Perforce.

P4COMSTR

The string displayed when fetching a source file from Perforce. If this is not set, then \$P4COM (the command line) is displayed.

P4FLAGS

General options that are passed to Perforce.

PCH

The Microsoft Visual C++ precompiled header that will be used when compiling object files. This variable is ignored by tools other than Microsoft Visual C++. When this variable is defined SCons will add options to the compiler command line to cause it to use the precompiled header, and will also set up the dependencies for the PCH file. Example:

```
env[ 'PCH' ] = 'StdAfx.pch'
```

PCHCOM

The command line used by the PCH builder to generate a precompiled header.

PCHCOMSTR

The string displayed when generating a precompiled header. If this is not set, then \$PCHCOM (the command line) is displayed.

PCHSTOP

This variable specifies how much of a source file is precompiled. This variable is ignored by tools other than Microsoft Visual C++, or when the PCH variable is not being used. When this variable is defined it must be a string that is the name of the header that is included at the end of the precompiled portion of the source files, or the empty string if the "#pragma hrdstop" construct is being used:

```
env[ 'PCHSTOP' ] = 'StdAfx.h'
```

PDB

The Microsoft Visual C++ PDB file that will store debugging information for object files, shared libraries, and programs. This variable is ignored by tools other than Microsoft Visual C++. When this variable is defined SCons will add options to the compiler and linker command line to cause them to generate external debugging information, and will also set up the dependencies for the PDB file. Example:


```
env['PDB'] = 'hello.pdb'
```

The Visual C++ compiler switch that SCons uses by default to generate PDB information is `/Z7`. This works correctly with parallel (`-j`) builds because it embeds the debug information in the intermediate object files, as opposed to sharing a single PDB file between multiple object files. This is also the only way to get debug information embedded into a static library. Using the `/Zi` instead may yield improved link-time performance, although parallel builds will no longer work. You can generate PDB files with the `/Zi` switch by overriding the default `$CCPDBFLAGS` variable; see the entry for that variable for specific examples.

PDFCOM

A deprecated synonym for `$DVIPDFCOM`.

PDFPREFIX

The prefix used for PDF file names.

PDFSUFFIX

The suffix used for PDF file names.

PKGCHK

On Solaris systems, the package-checking program that will be used (along with `$PKGINFO`) to look for installed versions of the Sun PRO C++ compiler. The default is `/usr/sbin/pgkchk`.

PKGINFO

On Solaris systems, the package information program that will be used (along with `$PKGCHK`) to look for installed versions of the Sun PRO C++ compiler. The default is `pkginfo`.

PLATFORM

The name of the platform used to create the Environment. If no platform is specified when the Environment is created, `scons` autodetects the platform.

```
env = Environment(tools = [])
if env['PLATFORM'] == 'cygwin':
    Tool('mingw')(env)
else:
    Tool('msvc')(env)
```

PRINT_CMD_LINE_FUNC

A Python function used to print the command lines as they are executed (assuming command printing is not disabled by the `-q` or `-s` options or their equivalents). The function should take four arguments: `s`, the command being executed (a string), `target`, the target being built (file node, list, or string name(s)), `source`, the source(s) used (file node, list, or string name(s)), and `env`, the environment being used.

The function must do the printing itself. The default implementation, used if this variable is not set or is `None`, is:

```
def print_cmd_line(s, target, source, env):
    sys.stdout.write(s + "\n")
```

Here's an example of a more interesting function:

```
def print_cmd_line(s, target, source, env):
    sys.stdout.write("Building %s -> %s...\n" %
```

```
( ' and '.join([str(x) for x in source]),  
  ' and '.join([str(x) for x in target]))  
env=Environment(PRINT_CMD_LINE_FUNC=print_cmd_line)  
env.Program('foo', 'foo.c')
```

This just prints "Building targetname from sourcename..." instead of the actual commands. Such a function could also log the actual commands to a log file, for example.

PROGPREFIX

The prefix used for executable file names.

PROGSUFFIX

The suffix used for executable file names.

PSCOM

The command line used to convert TeX DVI files into a PostScript file.

PSCOMSTR

The string displayed when a TeX DVI file is converted into a PostScript file. If this is not set, then \$PSCOM (the command line) is displayed.

PSPREFIX

The prefix used for PostScript file names.

PSSUFFIX

The prefix used for PostScript file names.

QT_AUTOSCAN

Turn off scanning for mocable files. Use the Moc Builder to explicitly specify files to run moc on.

QT_BINPATH

The path where the qt binaries are installed. The default value is '\$QTDIR/bin'.

QT_CPPPATH

The path where the qt header files are installed. The default value is '\$QTDIR/include'. Note: If you set this variable to None, the tool won't change the \$CPPPATH construction variable.

QT_DEBUG

Prints lots of debugging information while scanning for moc files.

QT_LIB

Default value is 'qt'. You may want to set this to 'qt-mt'. Note: If you set this variable to None, the tool won't change the \$LIBS variable.

QT_LIBPATH

The path where the qt libraries are installed. The default value is '\$QTDIR/lib'. Note: If you set this variable to None, the tool won't change the \$LIBPATH construction variable.

QT_MOC

Default value is '\$QT_BINPATH/moc'.

QT_MOCCXXPREFIX

Default value is `""`. Prefix for moc output files, when source is a cxx file.

QT_MOCCXXSUFFIX

Default value is `'moc'`. Suffix for moc output files, when source is a cxx file.

QT_MOCFROMCPPFLAGS

Default value is `'-i'`. These flags are passed to moc, when mocking a cpp file.

QT_MOCFROMCXXCOM

Command to generate a moc file from a cpp file.

QT_MOCFROMCXXCOMSTR

The string displayed when generating a moc file from a cpp file. If this is not set, then `$QT_MOCFROMCXXCOM` (the command line) is displayed.

QT_MOCFROMHCOM

Command to generate a moc file from a header.

QT_MOCFROMHCOMSTR

The string displayed when generating a moc file from a cpp file. If this is not set, then `$QT_MOCFROMHCOM` (the command line) is displayed.

QT_MOCFROMHFLAGS

Default value is `""`. These flags are passed to moc, when mocking a header file.

QT_MOCHPREFIX

Default value is `'moc_'`. Prefix for moc output files, when source is a header.

QT_MOCHSUFFIX

Default value is `'$CXXFILESUFFIX'`. Suffix for moc output files, when source is a header.

QT_UIC

Default value is `'$QT_BINPATH/uic'`.

QT_UICCOM

Command to generate header files from .ui files.

QT_UICCOMSTR

The string displayed when generating header files from .ui files. If this is not set, then `$QT_UICCOM` (the command line) is displayed.

QT_UICDECLFLAGS

Default value is `""`. These flags are passed to uic, when creating a h file from a .ui file.

QT_UICDECLPREFIX

Default value is `""`. Prefix for uic generated header files.

QT_UICDECLSUFFIX

Default value is `'h'`. Suffix for uic generated header files.

QT_UICIMPLFLAGS

Default value is `""`. These flags are passed to `uic`, when creating a `cxx` file from a `.ui` file.

QT_UICIMPLPREFIX

Default value is `'uic_'`. Prefix for `uic` generated implementation files.

QT_UICIMPLSUFFIX

Default value is `'$CXXFILESUFFIX'`. Suffix for `uic` generated implementation files.

QT_UISUFFIX

Default value is `'ui'`. Suffix of designer input files.

QTDIR

The `qt` tool tries to take this from `os.environ`. It also initializes all `QT_*` construction variables listed below. (Note that all paths are constructed with `python's os.path.join()` method, but are listed here with the `'/'` separator for easier reading.) In addition, the construction environment variables `$CPPPATH`, `$LIBPATH` and `$LIBS` may be modified and the variables `PROGEMITTER`, `SHLIBEMITTER` and `LIBEMITTER` are modified. Because the build-performance is affected when using this tool, you have to explicitly specify it at Environment creation:

```
Environment(tools=['default', 'qt'])
```

The `qt` tool supports the following operations:

Automatic moc file generation from header files. You do not have to specify moc files explicitly, the tool does it for you. However, there are a few preconditions to do so: Your header file must have the same filebase as your implementation file and must stay in the same directory. It must have one of the suffixes `.h`, `.hpp`, `.H`, `.hxx`, `.hh`. You can turn off automatic moc file generation by setting `QT_AUTOSCAN` to 0. See also the corresponding builder method `.B Moc()`

Automatic moc file generation from cxx files. As stated in the `qt` documentation, include the moc file at the end of the `cxx` file. Note that you have to include the file, which is generated by the transformation `${QT_MOCCXXPREFIX}<basename>${QT_MOCCXXSUFFIX}`, by default `<basename>.moc`. A warning is generated after building the moc file, if you do not include the correct file. If you are using `BuildDir`, you may need to specify `duplicate=1`. You can turn off automatic moc file generation by setting `QT_AUTOSCAN` to 0. See also the corresponding `Moc` builder method.

Automatic handling of .ui files. The implementation files generated from `.ui` files are handled much the same as `yacc` or `lex` files. Each `.ui` file given as a source of Program, Library or SharedLibrary will generate three files, the declaration file, the implementation file and a moc file. Because there are also generated headers, you may need to specify `duplicate=1` in calls to `BuildDir`. See also the corresponding `Uic` builder method.

RANLIB

The archive indexer.

RANLIBCOM

The command line used to index a static library archive.

RANLIBCOMSTR

The string displayed when a static library archive is indexed. If this is not set, then \$RANLIBCOM (the command line) is displayed.

```
env = Environment(RANLIBCOMSTR = "Indexing $TARGET")
```

RANLIBFLAGS

General options passed to the archive indexer.

RC

The resource compiler used to build a Microsoft Visual C++ resource file.

RCCOM

The command line used to build a Microsoft Visual C++ resource file.

RCCOMSTR

The string displayed when invoking the resource compiler to build a Microsoft Visual C++ resource file. If this is not set, then \$RCCOM (the command line) is displayed.

RCFLAGS

The flags passed to the resource compiler by the RES builder.

RCS

The RCS executable. Note that this variable is not actually used for the command to fetch source files from RCS; see the \$RCS_CO construction variable, below.

RCS_CO

The RCS "checkout" executable, used to fetch source files from RCS.

RCS_COCOM

The command line used to fetch (checkout) source files from RCS.

RCS_COCOMSTR

The string displayed when fetching a source file from RCS. If this is not set, then \$RCS_COCOM (the command line) is displayed.

RCS_COFLAGS

Options that are passed to the \$RCS_CO command.

RDirs

A function that converts a string into a list of Dir instances by searching the repositories.

REGSVR

The program used on Windows systems to register a newly-built DLL library whenever the SharedLibrary builder is passed a keyword argument of register=1.

REGSVRCOM

The command line used on Windows systems to register a newly-built DLL library whenever the SharedLibrary builder is passed a keyword argument of register=1.

REGSVRCOMSTR

The string displayed when registering a newly-built DLL file. If this is not set, then \$REGSVRCOM (the command line) is displayed.

REGSVRFLAGS

Flags passed to the DLL registration program on Windows systems when a newly-built DLL library is registered. By default, this includes the /s that prevents dialog boxes from popping up and requiring user attention.

RMIC

The Java RMI stub compiler.

RMICCOM

The command line used to compile stub and skeleton class files from Java classes that contain RMI implementations. Any options specified in the \$RMICFLAGS construction variable are included on this command line.

RMICCOMSTR

The string displayed when compiling stub and skeleton class files from Java classes that contain RMI implementations. If this is not set, then \$RMICCOM (the command line) is displayed.

```
env = Environment(RMICCOMSTR = "Generating stub/skeleton class files $TARGETS from
```

RMICFLAGS

General options passed to the Java RMI stub compiler.

_RPATH

An automatically-generated construction variable containing the rpath flags to be used when linking a program with shared libraries. The value of \$_RPATH is created by appending \$RPATHPREFIX and \$RPATHSUFFIX to the beginning and end of each directory in \$RPATH.

RPATH

A list of paths to search for shared libraries when running programs. Currently only used in the GNU (gnulink), IRIX (sgilink) and Sun (sunlink) linkers. Ignored on platforms and toolchains that don't support it. Note that the paths added to RPATH are not transformed by scons in any way: if you want an absolute path, you must make it absolute yourself.

RPATHPREFIX

The prefix used to specify a directory to be searched for shared libraries when running programs. This will be appended to the beginning of each directory in the \$RPATH construction variable when the \$_RPATH variable is automatically generated.

RPATHSUFFIX

The suffix used to specify a directory to be searched for shared libraries when running programs. This will be appended to the end of each directory in the \$RPATH construction variable when the \$_RPATH variable is automatically generated.

RPCGEN

The RPC protocol compiler.

RPCGENCLIENTFLAGS

Options passed to the RPC protocol compiler when generating client side stubs. These are in addition to any flags specified in the \$RPCGENFLAGS construction variable.

RPCGENFLAGS

General options passed to the RPC protocol compiler.

RPCGENHEADERFLAGS

Options passed to the RPC protocol compiler when generating a header file. These are in addition to any flags specified in the \$RPCGENFLAGS construction variable.

RPCGENSERVICEFLAGS

Options passed to the RPC protocol compiler when generating server side stubs. These are in addition to any flags specified in the \$RPCGENFLAGS construction variable.

RPCGENXDRFLAGS

Options passed to the RPC protocol compiler when generating XDR routines. These are in addition to any flags specified in the \$RPCGENFLAGS construction variable.

SCANNERS

A list of the available implicit dependency scanners. New file scanners may be added by appending to this list, although the more flexible approach is to associate scanners with a specific Builder. See the sections "Builder Objects" and "Scanner Objects," below, for more information.

SCCS

The SCCS executable.

SCCSCOM

The command line used to fetch source files from SCCS.

SCCSCOMSTR

The string displayed when fetching a source file from a CVS repository. If this is not set, then \$SCCSCOM (the command line) is displayed.

SCCSFLAGS

General options that are passed to SCCS.

SCCSGETFLAGS

Options that are passed specifically to the SCCS "get" subcommand. This can be set, for example, to -e to check out editable files from SCCS.

SCONS_HOME

The (optional) path to the SCons library directory, initialized from the external environment. If set, this is used to construct a shorter and more efficient search path in the \$MSVSSCONS command line executed from Microsoft Visual Studio project files.

SHCC

The C compiler used for generating shared-library objects.

SHCCCOM

The command line used to compile a C source file to a shared-library object file. Any options specified in the \$SHCCFLAGS, \$SHCCFLAGS and \$CPPFLAGS construction variables are included on this command line.

SHCCCOMSTR

The string displayed when a C source file is compiled to a shared object file. If this is not set, then \$SHCCCOM (the command line) is displayed.

```
env = Environment(SHCCCOMSTR = "Compiling shared object $TARGET")
```

SHCCFLAGS

Options that are passed to the C and C++ compilers to generate shared-library objects.

SHCFLAGS

Options that are passed to the C compiler (only; not C++) to generate shared-library objects.

SHCXX

The C++ compiler used for generating shared-library objects.

SHCXXCOM

The command line used to compile a C++ source file to a shared-library object file. Any options specified in the \$SHCXXFLAGS and \$CPPFLAGS construction variables are included on this command line.

SHCXXCOMSTR

The string displayed when a C++ source file is compiled to a shared object file. If this is not set, then \$SHCXXCOM (the command line) is displayed.

```
env = Environment(SHCXXCOMSTR = "Compiling shared object $TARGET")
```

SHCXXFLAGS

Options that are passed to the C++ compiler to generate shared-library objects.

SHELL

A string naming the shell program that will be passed to the \$SPAWN function. See the \$SPAWN construction variable for more information.

SHF77

The Fortran 77 compiler used for generating shared-library objects. You should normally set the \$SHFORTRAN variable, which specifies the default Fortran compiler for all Fortran versions. You only need to set \$SHF77 if you need to use a specific compiler or compiler version for Fortran 77 files.

SHF77COM

The command line used to compile a Fortran 77 source file to a shared-library object file. You only need to set \$SHF77COM if you need to use a specific command line for Fortran 77 files. You should normally set the \$SHFORTRANCOM variable, which specifies the default command line for all Fortran versions.

SHF77COMSTR

The string displayed when a Fortran 77 source file is compiled to a shared-library object file. If this is not set, then \$SHF77COM or \$SHF77FORTRANCOM (the command line) is displayed.

SHF77FLAGS

Options that are passed to the Fortran 77 compiler to generate shared-library objects. You only need to set \$SHF77FLAGS if you need to define specific user options for Fortran 77 files. You should normally set the \$SHF77FORTRANFLAGS variable, which specifies the user-specified options passed to the default Fortran compiler for all Fortran versions.

SHF77PPCOM

The command line used to compile a Fortran 77 source file to a shared-library object file after first running the file through the C preprocessor. Any options specified in the \$SHF77FLAGS and \$CPPFLAGS construction variables are included on this command line. You only need to set \$SHF77PPCOM if you need to use a specific C-preprocessor command line for Fortran 77 files. You should normally set the \$SHF77FORTRANPPCOM variable, which specifies the default C-preprocessor command line for all Fortran versions.

SHF90

The Fortran 90 compiler used for generating shared-library objects. You should normally set the \$SHF90FORTRAN variable, which specifies the default Fortran compiler for all Fortran versions. You only need to set \$SHF90 if you need to use a specific compiler or compiler version for Fortran 90 files.

SHF90COM

The command line used to compile a Fortran 90 source file to a shared-library object file. You only need to set \$SHF90COM if you need to use a specific command line for Fortran 90 files. You should normally set the \$SHF90FORTRANCOM variable, which specifies the default command line for all Fortran versions.

SHF90COMSTR

The string displayed when a Fortran 90 source file is compiled to a shared-library object file. If this is not set, then \$SHF90COM or \$SHF90FORTRANCOM (the command line) is displayed.

SHF90FLAGS

Options that are passed to the Fortran 90 compiler to generate shared-library objects. You only need to set \$SHF90FLAGS if you need to define specific user options for Fortran 90 files. You should normally set the \$SHF90FORTRANFLAGS variable, which specifies the user-specified options passed to the default Fortran compiler for all Fortran versions.

SHF90PPCOM

The command line used to compile a Fortran 90 source file to a shared-library object file after first running the file through the C preprocessor. Any options specified in the \$SHF90FLAGS and \$CPPFLAGS construction variables are included on this command line. You only need to set \$SHF90PPCOM if you need to use a specific C-preprocessor command line for Fortran 90 files. You should normally set the \$SHF90FORTRANPPCOM variable, which specifies the default C-preprocessor command line for all Fortran versions.

SHF95

The Fortran 95 compiler used for generating shared-library objects. You should normally set the `$SHFORTRAN` variable, which specifies the default Fortran compiler for all Fortran versions. You only need to set `$SHF95` if you need to use a specific compiler or compiler version for Fortran 95 files.

SHF95COM

The command line used to compile a Fortran 95 source file to a shared-library object file. You only need to set `$SHF95COM` if you need to use a specific command line for Fortran 95 files. You should normally set the `$SHFORTRANCOM` variable, which specifies the default command line for all Fortran versions.

SHF95COMSTR

The string displayed when a Fortran 95 source file is compiled to a shared-library object file. If this is not set, then `$SHF95COM` or `$SHFORTRANCOM` (the command line) is displayed.

SHF95FLAGS

Options that are passed to the Fortran 95 compiler to generate shared-library objects. You only need to set `$SHF95FLAGS` if you need to define specific user options for Fortran 95 files. You should normally set the `$SHFORTRANFLAGS` variable, which specifies the user-specified options passed to the default Fortran compiler for all Fortran versions.

SHF95PPCOM

The command line used to compile a Fortran 95 source file to a shared-library object file after first running the file through the C preprocessor. Any options specified in the `$SHF95FLAGS` and `$CPPFLAGS` construction variables are included on this command line. You only need to set `$SHF95PPCOM` if you need to use a specific C-preprocessor command line for Fortran 95 files. You should normally set the `$SHFORTRANPPCOM` variable, which specifies the default C-preprocessor command line for all Fortran versions.

SHFORTRAN

The default Fortran compiler used for generating shared-library objects.

SHFORTRANCOM

The command line used to compile a Fortran source file to a shared-library object file.

SHFORTRANCOMSTR

The string displayed when a Fortran source file is compiled to a shared-library object file. If this is not set, then `$SHFORTRANCOM` (the command line) is displayed.

SHFORTRANFLAGS

Options that are passed to the Fortran compiler to generate shared-library objects.

SHFORTRANPPCOM

The command line used to compile a Fortran source file to a shared-library object file after first running the file through the C preprocessor. Any options specified in the `$SHFORTRANFLAGS` and `$CPPFLAGS` construction variables are included on this command line.

SHLIBPREFIX

The prefix used for shared library file names.

SHLIBSUFFIX

The suffix used for shared library file names.

SHLINK

The linker for programs that use shared libraries.

SHLINKCOM

The command line used to link programs using shared libraries.

SHLINKCOMSTR

The string displayed when programs using shared libraries are linked. If this is not set, then \$SHLINKCOM (the command line) is displayed.

```
env = Environment(SHLINKCOMSTR = "Linking shared $TARGET")
```

SHLINKFLAGS

General user options passed to the linker for programs using shared libraries. Note that this variable should *not* contain `-l` (or similar) options for linking with the libraries listed in \$LIBS, nor `-L` (or similar) include search path options that scons generates automatically from \$LIBPATH. See `$_LIBFLAGS` above, for the variable that expands to library-link options, and `$_LIBDIRFLAGS` above, for the variable that expands to library search path options.

SHOBJPREFIX

The prefix used for shared object file names.

SHOBSUFFIX

The suffix used for shared object file names.

SOURCE

A reserved variable name that may not be set or used in a construction environment. (See "Variable Substitution," below.)

SOURCES

A reserved variable name that may not be set or used in a construction environment. (See "Variable Substitution," below.)

SPAWN

A command interpreter function that will be called to execute command line strings. The function must expect the following arguments:

```
def spawn(shell, escape, cmd, args, env):
```

`sh` is a string naming the shell program to use. `escape` is a function that can be called to escape shell special characters in the command line. `cmd` is the path to the command to be executed. `args` is the arguments to the command. `env` is a dictionary of the environment variables in which the command should be executed.

SWIG

The scripting language wrapper and interface generator.

SWIGFILESUFFIX

The suffix that will be used for intermediate C source files generated by the scripting language wrapper and interface generator. The default value is `_wrap$CFILESUFFIX`. By default, this value is used whenever the `-c++` option is *not* specified as part of the `$SWIGFLAGS` construction variable.

SWIGCOM

The command line used to call the scripting language wrapper and interface generator.

SWIGCOMSTR

The string displayed when calling the scripting language wrapper and interface generator. If this is not set, then `$SWIGCOM` (the command line) is displayed.

SWIGCXXFILESUFFIX

The suffix that will be used for intermediate C++ source files generated by the scripting language wrapper and interface generator. The default value is `_wrap$CFILESUFFIX`. By default, this value is used whenever the `-c++` option is specified as part of the `$SWIGFLAGS` construction variable.

SWIGFLAGS

General options passed to the scripting language wrapper and interface generator. This is where you should set `-python`, `-perl5`, `-tcl`, or whatever other options you want to specify to SWIG. If you set the `-c++` option in this variable, `scons` will, by default, generate a C++ intermediate source file with the extension that is specified as the `$CXXFILESUFFIX` variable.

TAR

The tar archiver.

TARCOM

The command line used to call the tar archiver.

TARCOMSTR

The string displayed when archiving files using the tar archiver. If this is not set, then `$TARCOM` (the command line) is displayed.

```
env = Environment(TARCOMSTR = "Archiving $TARGET")
```

TARFLAGS

General options passed to the tar archiver.

TARGET

A reserved variable name that may not be set or used in a construction environment. (See "Variable Substitution," below.)

TARGETS

A reserved variable name that may not be set or used in a construction environment. (See "Variable Substitution," below.)

TARSUFFIX

The suffix used for tar file names.

TEMPFILEPREFIX

The prefix for a temporary file used to execute lines longer than \$MAXLINELENGTH. The default is '@'. This may be set for toolchains that use other values, such as '-@' for the diab compiler or '-via' for ARM toolchain.

TEX

The TeX formatter and typesetter.

TEXCOM

The command line used to call the TeX formatter and typesetter.

TEXCOMSTR

The string displayed when calling the TeX formatter and typesetter. If this is not set, then \$TEXCOM (the command line) is displayed.

```
env = Environment(TEXCOMSTR = "Building $TARGET from TeX input $SOURCES")
```

TEXFLAGS

General options passed to the TeX formatter and typesetter.

TEXINPUTS

List of directories that the LaTeX program will search for include directories. The LaTeX implicit dependency scanner will search these directories for \include and \import files.

TOOLS

A list of the names of the Tool specifications that are part of this construction environment.

WIN32_INSERT_DEF

A deprecated synonym for \$WINDOWS_INSERT_DEF.

WIN32DEFPREFIX

A deprecated synonym for \$WINDOWSDEFPREFIX.

WIN32DEFSUFFIX

A deprecated synonym for \$WINDOWSDEFSUFFIX.

WIN32EXPPREFIX

A deprecated synonym for \$WINDOWSEXPSUFFIX.

WIN32EXPSUFFIX

A deprecated synonym for \$WINDOWSEXPSUFFIX.

WINDOWS_INSERT_DEF

When this is set to true, a library build of a Windows shared library (.dllfile) will also build a corresponding .def file at the same time, if a .def file is not already listed as a build target. The default is 0 (do not build a .def file).

WINDOWS_INSERT_MANIFEST

When this is set to true, scons will be aware of the .manifest files generated by Microsoft Visual C/C++ 8.

WINDOWSDEFPREFIX

The prefix used for Windows `.def` file names.

WINDOWSDEFSUFFIX

The suffix used for Windows `.def` file names.

WINDOWSEXPREFIX

The prefix used for Windows `.exp` file names.

WINDOWSEXP_SUFFIX

The suffix used for Windows `.exp` file names.

WINDOWSPROGMANIFESTPREFIX

The prefix used for executable program `.manifest` files generated by Microsoft Visual C/C++.

WINDOWSPROGMANIFEST_SUFFIX

The suffix used for executable program `.manifest` files generated by Microsoft Visual C/C++.

WINDOWSSHLIBMANIFESTPREFIX

The prefix used for shared library `.manifest` files generated by Microsoft Visual C/C++.

WINDOWSSHLIBMANIFEST_SUFFIX

The suffix used for shared library `.manifest` files generated by Microsoft Visual C/C++.

YACC

The parser generator.

YACCCOM

The command line used to call the parser generator to generate a source file.

YACCCOMSTR

The string displayed when generating a source file using the parser generator. If this is not set, then `$YACCCOM` (the command line) is displayed.

```
env = Environment(YACCCOMSTR = "Yacc'ing $TARGET from $SOURCES")
```

YACCFLAGS

General options passed to the parser generator. If `$YACCFLAGS` contains a `-d` option, `SCons` assumes that the call will also create a `.h` file (if the yacc source file ends in a `.y` suffix) or a `.hpp` file (if the yacc source file ends in a `.yy` suffix)

YACCHFILESUFFIX

The suffix of the C header file generated by the parser generator when the `-d` option is used. Note that setting this variable does not cause the parser generator to generate a header file with the specified suffix, it exists to allow you to specify what suffix the parser generator will use of its own accord. The default value is `.h`.

YACCHXXFILESUFFIX

The suffix of the C++ header file generated by the parser generator when the `-d` option is used. Note that setting this variable does not cause the parser generator to generate a header file with the specified suffix, it exists to allow you to specify what suffix the parser generator will use of its own accord. The default value is `.hpp`.

YACCVCGFILESUFFIX

The suffix of the file containing the VCG grammar automaton definition when the `--graph=` option is used. Note that setting this variable does not cause the parser generator to generate a VCG file with the specified suffix, it exists to allow you to specify what suffix the parser generator will use of its own accord. The default value is `.vcg`.

ZIP

The zip compression and file packaging utility.

ZIPCOM

The command line used to call the zip utility, or the internal Python function used to create a zip archive.

ZIPCOMPRESSION

The `compression` flag from the Python `zipfile` module used by the internal Python function to control whether the zip archive is compressed or not. The default value is `zipfile.ZIP_DEFLATED`, which creates a compressed zip archive. This value has no effect when using Python 1.5.2 or if the `zipfile` module is otherwise unavailable.

ZIPCOMSTR

The string displayed when archiving files using the zip utility. If this is not set, then `$ZIPCOM` (the command line or internal Python function) is displayed.

```
env = Environment(ZIPCOMSTR = "Zipping $TARGET")
```

ZIPFLAGS

General options passed to the zip utility.

Appendix B. Builders

This appendix contains descriptions of all of the Builders that are *potentially* available "out of the box" in this version of SCons.

CFile()
env.CFile()

Builds a C source file given a lex (.l) or yacc (.y) input file. The suffix specified by the \$CFILESUFFIX construction variable (.c by default) is automatically added to the target if it is not already present. Example:

```
# builds foo.c
env.CFile(target = 'foo.c', source = 'foo.l')
# builds bar.c
env.CFile(target = 'bar', source = 'bar.y')
```

CXXFile()
env.CXXFile()

Builds a C++ source file given a lex (.ll) or yacc (.yy) input file. The suffix specified by the \$CXXFILESUFFIX construction variable (.cc by default) is automatically added to the target if it is not already present. Example:

```
# builds foo.cc
env.CXXFile(target = 'foo.cc', source = 'foo.ll')
# builds bar.cc
env.CXXFile(target = 'bar', source = 'bar.yy')
```

DVI()
env.DVI()

Builds a .dvi file from a .tex, .ltx or .latex input file. If the source file suffix is .tex, scons will examine the contents of the file; if the string \documentclass or \documentstyle is found, the file is assumed to be a LaTeX file and the target is built by invoking the \$LATEXCOM command line; otherwise, the \$TEXCOM command line is used. If the file is a LaTeX file, the DVI builder method will also examine the contents of the .aux file and invoke the \$BIBTEX command line if the string bibdata is found, start \$MAKEINDEX to generate an index if a .ind file is found and will examine the contents .log file and re-run the \$LATEXCOM command if the log file says it is necessary.

The suffix .dvi (hard-coded within TeX itself) is automatically added to the target if it is not already present. Examples:

```
# builds from aaa.tex
env.DVI(target = 'aaa.dvi', source = 'aaa.tex')
# builds bbb.dvi
env.DVI(target = 'bbb', source = 'bbb.ltx')
# builds from ccc.latex
env.DVI(target = 'ccc.dvi', source = 'ccc.latex')
```

Jar()
env.Jar()

Builds a Java archive (.jar) file from a source tree of .class files. If the \$JARCHDIR value is set, the jar command will change to the specified directory using the -C option. If the contents any of the source files begin with the string Manifest-Version, the file is assumed to be a manifest and is passed to the jar command with the m option set.

```
env.Jar(target = 'foo.jar', source = 'classes')
```

```
Java()  
env.Java()
```

Builds one or more Java class files from one or more source trees of .java files. The class files will be placed underneath the specified target directory. SCons will parse each source .java file to find the classes (including inner classes) defined within that file, and from that figure out the target .class files that will be created. SCons will also search each Java file for the Java package name, which it assumes can be found on a line beginning with the string `package` in the first column; the resulting .class files will be placed in a directory reflecting the specified package name. For example, the file `Foo.java` defining a single public `Foo` class and containing a package name of `sub.dir` will generate a corresponding `sub/dir/Foo.class` class file.

Example:

```
env.Java(target = 'classes', source = 'src')  
env.Java(target = 'classes', source = ['src1', 'src2'])
```

```
JavaH()  
env.JavaH()
```

Builds C header and source files for implementing Java native methods. The target can be either a directory in which the header files will be written, or a header file name which will contain all of the definitions. The source can be either the names of .class files, or the objects returned from the `Java` builder method.

If the construction variable `$JAVACLASSDIR` is set, either in the environment or in the call to the `JavaH` builder method itself, then the value of the variable will be stripped from the beginning of any .class file names.

Examples:

```
# builds java_native.h  
classes = env.Java(target = 'classdir', source = 'src')  
env.JavaH(target = 'java_native.h', source = classes)  
  
# builds include/package_foo.h and include/package_bar.h  
env.JavaH(target = 'include',  
          source = ['package/foo.class', 'package/bar.class'])  
  
# builds export/foo.h and export/bar.h  
env.JavaH(target = 'export',  
          source = ['classes/foo.class', 'classes/bar.class'],  
          JAVACLASSDIR = 'classes')
```

```
Library()  
env.Library()
```

A synonym for the `StaticLibrary` builder method.

```
LoadableModule()  
env.LoadableModule()
```

On most systems, this is the same as `SharedLibrary`. On Mac OS X (Darwin) platforms, this creates a loadable module bundle.

```
M4( )
env.M4( )
```

Builds an output file from an M4 input file. This uses a default \$M4FLAGS value of `-E`, which considers all warnings to be fatal and stops on the first warning when using the GNU version of m4. Example:

```
env.M4(target = 'foo.c', source = 'foo.c.m4')
```

```
Moc( )
env.Moc( )
```

Builds an output file from a moc input file. Moc input files are either header files or `cxx` files. This builder is only available after using the tool 'qt'. See the \$QTDIR variable for more information. Example:

```
env.Moc('foo.h') # generates moc_foo.cc
env.Moc('foo.cpp') # generates foo.moc
```

```
MSVSProject( )
env.MSVSProject( )
```

Builds a Microsoft Visual Studio project file, and by default builds a solution file as well.

This builds a Visual Studio project file, based on the version of Visual Studio that is configured (either the latest installed version, or the version specified by \$MSVS_VERSION in the Environment constructor). For Visual Studio 6, it will generate a `.dsp` file. For Visual Studio 7 (.NET), it will generate a `.dsp` file.

By default, this also generates a solution file for the specified project, a `.dsw` file for Visual Studio 6 or a `.sln` file for Visual Studio 7 (.NET). This behavior may be disabled by specifying `auto_build_solution=0` when you call `MSVSProject`, in which case you presumably want to build the solution file(s) by calling the `MSVSSolution Builder` (see below).

It takes several lists of filenames to be placed into the project file. These are currently limited to `srcs`, `incs`, `localincs`, `resources`, and `misc`. These are pretty self-explanatory, but it should be noted that these lists are added to the \$SOURCES construction variable as strings, NOT as SCons File Nodes. This is because they represent file names to be added to the project file, not the source files used to build the project file.

The above filename lists are all optional, although at least one must be specified for the resulting project file to be non-empty.

In addition to the above lists of values, the following values may be specified:

target: The name of the target `.dsp` or `.vcproj` file. The correct suffix for the version of Visual Studio must be used, but the \$MSVSPROJECTSUFFIX construction variable will be defined to the correct value (see example below).

variant: The name of this particular variant. For Visual Studio 7 projects, this can also be a list of variant names. These are typically things like "Debug" or "Release", but really can be anything you want. For Visual Studio 7 projects, they may also specify a target platform separated from the variant name by a | (vertical pipe) character: `Debug|Xbox`. The default target platform is Win32. Multiple calls to `MSVSProject` with different variants are allowed; all variants will be added to the project file with their appropriate build targets and sources.

buildtarget: An optional string, node, or list of strings or nodes (one per build variant), to tell the Visual Studio debugger what output target to use in what build variant. The number of `buildtarget` entries must match the number of `variant` entries.

runfile: The name of the file that Visual Studio 7 and later will run and debug. This appears as the value of the `Output` field in the resulting Visual Studio project file. If this is not specified, the default is the same as the specified `buildtarget` value.

Example usage:

```
barsrcs = ['bar.cpp'],
barincs = ['bar.h'],
barlocalincs = ['StdAfx.h']
barresources = ['bar.rc', 'resource.h']
barmisc = ['bar_readme.txt']

dll = env.SharedLibrary(target = 'bar.dll',
                        source = barsrcs)

env.MSVSProject(target = 'Bar' + env['MSVSPROJECTSUFFIX'],
                srcs = barsrcs,
                incs = barincs,
                localincs = barlocalincs,
                resources = barresources,
                misc = barmisc,
                buildtarget = dll,
                variant = 'Release')
```

```
MSVSSolution()
env.MSVSSolution()
```

Builds a Microsoft Visual Studio solution file.

This builds a Visual Studio solution file, based on the version of Visual Studio that is configured (either the latest installed version, or the version specified by `$MSVS_VERSION` in the construction environment). For Visual Studio 6, it will generate a `.dsw` file. For Visual Studio 7 (.NET), it will generate a `.sln` file.

The following values must be specified:

target: The name of the target `.dsw` or `.sln` file. The correct suffix for the version of Visual Studio must be used, but the value `$MSVSSOLUTIONSUFFIX` will be defined to the correct value (see example below).

variant: The name of this particular variant, or a list of variant names (the latter is only supported for MSVS 7 solutions). These are typically things like "Debug" or "Release", but really can be anything you want. For MSVS 7 they may also specify target platform, like this "Debug|Xbox". Default platform is Win32.

projects: A list of project file names, or Project nodes returned by calls to the `MSVSProject` Builder, to be placed into the solution file. (NOTE: Currently only one project is supported per solution.) It should be noted that these file names are NOT added to the `$SOURCES` environment variable in form of files, but rather as strings. This is because they represent file names to be added to the solution file, not the source files used to build the solution file.

Example Usage:

```
env.MSVSSolution(target = 'Bar' + env['MSVSSOLUTIONSUFFIX'],
                projects = ['bar' + env['MSVSPROJECTSUFFIX']],
                variant = 'Release')
```

```
Object()
env.Object()
```

A synonym for the `StaticObject` builder method.

```
PCH()
env.PCH()
```

Builds a Microsoft Visual C++ precompiled header. Calling this builder method returns a list of two targets: the PCH as the first element, and the object file as the second element. Normally the object file is ignored. This builder method is only provided when Microsoft Visual C++ is being used as the compiler. The PCH builder method is generally used in conjunction with the PCH construction variable to force object files to use the precompiled header:

```
env['PCH'] = env.PCH('StdAfx.cpp')[0]
```

```
PDF()
env.PDF()
```

Builds a .pdf file from a .dvi input file (or, by extension, a .tex, .ltx, or .latex input file). The suffix specified by the \$PDFSUFFIX construction variable (.pdf by default) is added automatically to the target if it is not already present. Example:

```
# builds from aaa.tex
env.PDF(target = 'aaa.pdf', source = 'aaa.tex')
# builds bbb.pdf from bbb.dvi
env.PDF(target = 'bbb', source = 'bbb.dvi')
```

```
PostScript()
env.PostScript()
```

Builds a .ps file from a .dvi input file (or, by extension, a .tex, .ltx, or .latex input file). The suffix specified by the \$PSSUFFIX construction variable (.ps by default) is added automatically to the target if it is not already present. Example:

```
# builds from aaa.tex
env.PostScript(target = 'aaa.ps', source = 'aaa.tex')
# builds bbb.ps from bbb.dvi
env.PostScript(target = 'bbb', source = 'bbb.dvi')
```

```
Program()
env.Program()
```

Builds an executable given one or more object files or C, C++, D, or Fortran source files. If any C, C++, D or Fortran source files are specified, then they will be automatically compiled to object files using the `Object` builder method; see that builder method's description for a list of legal source file suffixes and how they are interpreted. The target executable file prefix (specified by the \$PROG-PREFIX construction variable; nothing by default) and suffix (specified by the \$PROGSUFFIX construction variable; by default, .exe on Windows systems, nothing on POSIX systems) are automatically added to the target if not already present. Example:

```
env.Program(target = 'foo', source = ['foo.o', 'bar.c', 'baz.f'])
```

```
RES()
env.RES()
```

Builds a Microsoft Visual C++ resource file. This builder method is only provided when Microsoft Visual C++ or MinGW is being used as the compiler. The .res (or .o for MinGW) suffix is added to the target name if no other suffix is given. The source file is scanned for implicit dependencies as though it were a C file. Example:

```
env.RES('resource.rc')
```

```
RMIC()
env.RMIC()
```

Builds stub and skeleton class files for remote objects from Java `.class` files. The target is a directory relative to which the stub and skeleton class files will be written. The source can be the names of `.class` files, or the objects return from the Java builder method.

If the construction variable `$JAVACLASSDIR` is set, either in the environment or in the call to the `RMIC` builder method itself, then the value of the variable will be stripped from the beginning of any `.class` file names.

```
classes = env.Java(target = 'classdir', source = 'src')
env.RMIC(target = 'outdir1', source = classes)

env.RMIC(target = 'outdir2',
         source = ['package/foo.class', 'package/bar.class'])

env.RMIC(target = 'outdir3',
         source = ['classes/foo.class', 'classes/bar.class'],
         JAVACLASSDIR = 'classes')
```

```
RPCGenClient()
env.RPCGenClient()
```

Generates an RPC client stub (`_clnt.c`) file from a specified RPC (`.x`) source file. Because `rpcgen` only builds output files in the local directory, the command will be executed in the source file's directory by default.

```
# Builds src/rpcif_clnt.c
env.RPCGenClient('src/rpcif.x')
```

```
RPCGenHeader()
env.RPCGenHeader()
```

Generates an RPC header (`.h`) file from a specified RPC (`.x`) source file. Because `rpcgen` only builds output files in the local directory, the command will be executed in the source file's directory by default.

```
# Builds src/rpcif.h
env.RPCGenHeader('src/rpcif.x')
```

```
RPCGenService()
env.RPCGenService()
```

Generates an RPC server-skeleton (`_svc.c`) file from a specified RPC (`.x`) source file. Because `rpcgen` only builds output files in the local directory, the command will be executed in the source file's directory by default.

```
# Builds src/rpcif_svc.c
env.RPCGenClient('src/rpcif.x')
```

```
RPCGenXDR()
env.RPCGenXDR()
```

Generates an RPC XDR routine (`_xdr.c`) file from a specified RPC (`.x`) source file. Because `rpcgen` only builds output files in the local directory, the command will be executed in the source file's directory by default.

```
# Builds src/rpcif_xdr.c
env.RPCGenClient('src/rpcif.x')
```

```
SharedLibrary()
env.SharedLibrary()
```

Builds a shared library (.so on a POSIX system, .dll on Windows) given one or more object files or C, C++, D or Fortran source files. If any source files are given, then they will be automatically compiled to object files. The static library prefix and suffix (if any) are automatically added to the target. The target library file prefix (specified by the \$SHLIBPREFIX construction variable; by default, lib on POSIX systems, nothing on Windows systems) and suffix (specified by the \$SHLIBSUFFIX construction variable; by default, .dll on Windows systems, .so on POSIX systems) are automatically added to the target if not already present. Example:

```
env.SharedLibrary(target = 'bar', source = ['bar.c', 'foo.o'])
```

On Windows systems, the SharedLibrary builder method will always build an import (.lib) library in addition to the shared (.dll) library, adding a .lib library with the same basename if there is not already a .lib file explicitly listed in the targets.

Any object files listed in the source must have been built for a shared library (that is, using the SharedObject builder method). scons will raise an error if there is any mismatch.

On Windows systems, specifying register=1 will cause the .dll to be registered after it is built using REGSVR32. The command that is run ("regsvr32" by default) is determined by \$REGSVR construction variable, and the flags passed are determined by \$REGSVRFLAGS. By default, \$REGSVRFLAGS includes the /s option, to prevent dialogs from popping up and requiring user attention when it is run. If you change \$REGSVRFLAGS, be sure to include the /s option. For example,

```
env.SharedLibrary(target = 'bar',
                  source = ['bar.cxx', 'foo.obj'],
                  register=1)
```

will register bar.dll as a COM object when it is done linking it.

```
SharedObject()
env.SharedObject()
```

Builds an object file for inclusion in a shared library. Source files must have one of the same set of extensions specified above for the StaticObject builder method. On some platforms building a shared object requires additional compiler option (e.g. -fPIC for gcc) in addition to those needed to build a normal (static) object, but on some platforms there is no difference between a shared object and a normal (static) one. When there is a difference, SCons will only allow shared objects to be linked into a shared library, and will use a different suffix for shared objects. On platforms where there is no difference, SCons will allow both normal (static) and shared objects to be linked into a shared library, and will use the same suffix for shared and normal (static) objects. The target object file prefix (specified by the \$SHOBJPREFIX construction variable; by default, the same as \$OBJPREFIX) and suffix (specified by the \$SHOBSUFFIX construction variable) are automatically added to the target if not already present. Examples:

```
env.SharedObject(target = 'ddd', source = 'ddd.c')
env.SharedObject(target = 'eee.o', source = 'eee.cpp')
env.SharedObject(target = 'fff.obj', source = 'fff.for')
```

Note that the source files will be scanned according to the suffix mappings in the `SourceFileScanner` object. See the section "Scanner Objects," below, for a more information.

```
StaticLibrary()
env.StaticLibrary()
```

Builds a static library given one or more object files or C, C++, D or Fortran source files. If any source files are given, then they will be automatically compiled to object files. The static library prefix and suffix (if any) are automatically added to the target. The target library file prefix (specified by the `$LIBPREFIX` construction variable; by default, `lib` on POSIX systems, nothing on Windows systems) and suffix (specified by the `$LIBSUFFIX` construction variable; by default, `.lib` on Windows systems, `.a` on POSIX systems) are automatically added to the target if not already present. Example:

```
env.StaticLibrary(target = 'bar', source = ['bar.c', 'foo.o'])
```

Any object files listed in the `source` must have been built for a static library (that is, using the `StaticObject` builder method). `scons` will raise an error if there is any mismatch.

```
StaticObject()
env.StaticObject()
```

Builds a static object file from one or more C, C++, D, or Fortran source files. Source files must have one of the following extensions:

<code>.asm</code>	assembly language file
<code>.ASM</code>	assembly language file
<code>.c</code>	C file
<code>.C</code>	Windows: C file
	POSIX: C++ file
<code>.cc</code>	C++ file
<code>.cpp</code>	C++ file
<code>.cxx</code>	C++ file
<code>.cxx</code>	C++ file
<code>.c++</code>	C++ file
<code>.C++</code>	C++ file
<code>.d</code>	D file
<code>.f</code>	Fortran file
<code>.F</code>	Windows: Fortran file
	POSIX: Fortran file + C pre-processor
<code>.for</code>	Fortran file
<code>.FOR</code>	Fortran file
<code>.fpp</code>	Fortran file + C pre-processor
<code>.FPP</code>	Fortran file + C pre-processor
<code>.m</code>	Object C file
<code>.mm</code>	Object C++ file
<code>.s</code>	assembly language file
<code>.S</code>	Windows: assembly language file
	POSIX: assembly language file + C pre-processor
<code>.spp</code>	assembly language file + C pre-processor
<code>.SPP</code>	assembly language file + C pre-processor

The target object file prefix (specified by the `$OBJPREFIX` construction variable; nothing by default) and suffix (specified by the `$OBSUFFIX` construction variable; `.obj` on Windows systems, `.o` on POSIX systems) are automatically added to the target if not already present. Examples:

```
env.StaticObject(target = 'aaa', source = 'aaa.c')
env.StaticObject(target = 'bbb.o', source = 'bbb.c++')
env.StaticObject(target = 'ccc.obj', source = 'ccc.f')
```


Note that the source files will be scanned according to the suffix mappings in `SourceFileScanner` object. See the section "Scanner Objects," below, for a more information.

```
Tar()
env.Tar()
```

Builds a tar archive of the specified files and/or directories. Unlike most builder methods, the `Tar` builder method may be called multiple times for a given target; each additional call adds to the list of entries that will be built into the archive. Any source directories will be scanned for changes to any on-disk files, regardless of whether or not `scons` knows about them from other Builder or function calls.

```
env.Tar('src.tar', 'src')

# Create the stuff.tar file.
env.Tar('stuff', ['subdir1', 'subdir2'])
# Also add "another" to the stuff.tar file.
env.Tar('stuff', 'another')

# Set TARFLAGS to create a gzip-filtered archive.
env = Environment(TARFLAGS = '-c -z')
env.Tar('foo.tar.gz', 'foo')

# Also set the suffix to .tgz.
env = Environment(TARFLAGS = '-c -z',
                  TARSUFFIX = '.tgz')
env.Tar('foo')
```

```
TypeLibrary()
env.TypeLibrary()
```

Builds a Windows type library (`.tlb`) file from an input IDL file (`.idl`). In addition, it will build the associated interface stub and proxy source files, naming them according to the base name of the `.idl` file. For example,

```
env.TypeLibrary(source="foo.idl")
```

Will create `foo.tlb`, `foo.h`, `foo_i.c`, `foo_p.c` and `foo_data.c` files.

```
Uic()
env.Uic()
```

Builds a header file, an implementation file and a moc file from an ui file. and returns the corresponding nodes in the above order. This builder is only available after using the tool 'qt'. Note: you can specify `.ui` files directly as source files to the `Program`, `Library` and `SharedLibrary` builders without using this builder. Using this builder lets you override the standard naming conventions (be careful: prefixes are always prepended to names of built files; if you don't want prefixes, you may set them to ""). See the `$QTDIR` variable for more information. Example:

```
env.Uic('foo.ui') # -> ['foo.h', 'uic_foo.cc', 'moc_foo.cc']
env.Uic(target = Split('include/foo.h gen/uicfoo.cc gen/mocfoo.cc'),
        source = 'foo.ui') # -> ['include/foo.h', 'gen/uicfoo.cc', 'gen/mocfoo.cc']
```

```
Zip()  
env.Zip()
```

Builds a zip archive of the specified files and/or directories. Unlike most builder methods, the `zip` builder method may be called multiple times for a given target; each additional call adds to the list of entries that will be built into the archive. Any source directories will be scanned for changes to any on-disk files, regardless of whether or not `scons` knows about them from other Builder or function calls.

```
env.Zip('src.zip', 'src')  
  
# Create the stuff.zip file.  
env.Zip('stuff', ['subdir1', 'subdir2'])  
# Also add "another" to the stuff.tar file.  
env.Zip('stuff', 'another')
```

Appendix C. Tools

This appendix contains descriptions of all of the Tools that are available "out of the box" in this version of SCons.

386asm
XXX

aixc++
XXX

aixcc
XXX

aixf77
XXX

aixlink
XXX

ar
XXX

as
XXX

bcc32
XXX

BitKeeper
XXX

c++
XXX

cc
XXX

cvf
XXX

CVS
XXX

default
XXX

dmd
XXX

dvipdf
XXX

Appendix C. Tools

dvips
XXX

f77
XXX

f90
XXX

f95
XXX

fortran
XXX

g++
XXX

g77
XXX

gas
XXX

gcc
XXX

gnulink
XXX

gs
XXX

hpc++
XXX

hpcc
XXX

hplink
XXX

icc
XXX

icl
XXX

ifl
XXX

ifort
XXX

ilink
XXX

ilink32
XXX

intelc
XXX

jar
XXX

javac
XXX

javah
XXX

latex
XXX

lex
XXX

link
XXX

linkloc
XXX

m4
XXX

masm
XXX

midl
XXX

mingw
XXX

mslib
XXX

mslink
XXX

msvc
XXX

msvs
XXX

Appendix C. Tools

mwcc	XXX
mwld	XXX
nasm	XXX
pdflatex	XXX
pdftex	XXX
Perforce	XXX
qt	XXX
RCS	XXX
rmic	XXX
rpcgen	XXX
SCCS	XXX
sgiar	XXX
sgic++	XXX
sgicc	XXX
sgilink	XXX
Subversion	XXX
sunar	XXX
sunc++	XXX

suncc
 XXX

sunlink
 XXX

swig
 XXX

tar
 XXX

tex
 XXX

tlib
 XXX

yacc
 XXX

zip
 XXX

Appendix D. 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 D-1. Wildcard globbing to create a list of filenames

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

Example D-2. Filename extension substitution

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

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

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

