Compilers, Make, and Debuggers on Sun Workstations

This document gives you basic information on using compilers for several languages on UNIX systems. It lists extensive resources for learning more about programming languages and compilers.

Table of Contents

Files Used in Programming 3

Compiler Options 4

Compiler Usage 4

Programming Utilities 5

The FORTRAN Compiler 5
- FORTRAN Utilities 5
- FORTRAN Resources 6

The C Compiler 6
- Utilities 6
- C Resources 7

The C++ Compiler 7
- C++ Resources 7

Using Make 8
- Elements of Make 8
- Syntax of the Make Command 10
- Multiple Programs in a Makefile 10
- Other Elements in Makefiles 11
  - Non-File Targets 11
  - Comments 11
  - Variables 11
- Make Resources 13

Debuggers 13
- Image files 13
- Available Debuggers 13
- Debugger Resources 14
- Storing Your Files 14

Problems or Questions? 14

Files Used in Programming

UNIX programming files have a naming convention that has two components, the file name, and the file suffix. The suffix is almost always a period followed by one letter, for example, (**.c**). Creating a program on a UNIX system involves several different classes of files and each has its own function in the process of creating the program. A file's class is conventionally indicated by the suffix. The list below summarizes the file classes and their suffixes:

source — A text file containing a program or subroutine written in a language such as C, C++, FORTRAN, or Pascal. The suffix is determined by the language used:

.c for C

.cc for C++

.f for FORTRAN

.p for Pascal

.s for Assembly Languages

object — Not a text file, not readable by humans. It's the machine-readable equivalent of a source code file and is produced by running the source file through a compiler. There is a 1-1 correspondence: 1 source file produces 1 object file. Object files have the suffix (**.o**).

header — A file that is usually included in its text form into another source file. Header files usually contain variable declarations or definitions. Header files have the suffix (**.h**).

library — A collection of related functions that are in machine readable format and can be included into a program when it is compiled. Library files have the suffix (**.a**).

executable — The completed program, capable of running, or being executed, by the computer. By default, executables are given the name **a.out**, but they can be given any name by using a compiler option described in *Compiler Options*.

Creating a program involves the following basic steps using the file classes described above.

1. A program is typed into one or more source files in a high level language such as C, C++, or FORTRAN.
2. The compiler is invoked on the source files and it incorporates the header files and then produces object code from the source file.
3. The linker portion of the compiler links the object code modules with libraries to make an executable program. Only the relevant portions of the library files are linked into the program.
4. Dynamically liked binaries are coupled with shared libraries at run time to create a working memory image for execution.

Because the compilation and linking process produces object code that a particular machine or CPU architecture can read, it generally cannot be read by a machine or CPU of a different or unrelated architecture. For example, a program compiled for a Motorola 68020 CPU can be read by a Motorola 68030 CPU, since they are related architectures, but the program will not run on a SPARC architecture CPU.

Compiler Options

There are several options common to all of the compilers discussed below. These common options are as follows:

-c	Suppress linking; compile only. Make an **.o** file for each source file.
-o *output*	Name the final executable file *output* instead of **a.out**.
-l *library*	Link in routines from the library archive. Library archives are usually located in a library directory such as **/lib** , **/usr/lib** , or **/usr/local/lib** . Archives have the naming convention **lib***library***.a**, such as **libimsl.a** or **libm.a**.
-p	Generate data for use with the prof utility described in the section *FORTRAN Utilities*.
-gp	Generate data for use with the gprof utility described in the section *FORTRAN Utilities*.

Compiler Usage

All of the compilers described below have similar syntax and are invoked the same way. The compiler can be invoked to produce an executable from the source code without generating any intermediate files in your directory, such as the following example, which uses the FORTRAN compiler, **f77**:

```
f77 -o whales tails.f
```

In this example, f77 compiles the FORTRAN source file **tails.f** into object code, passes it to the linker, ld, which links it with some default libraries and produces the executable **whales**.

You can also split up the compiling and linking into multiple commands:

```
f77 -c tails.f
f77 -o whales tails.o
```

The first line compiles **tails.f** to **tails.o** and puts **tails.o** into your directory. The second links **tails.o** with some default libraries and produces the executable, **whales**.

In addition to the default libraries, you can specify other libraries to be linked in with the **-l** flag, as follows:

```
f77 -c tails.f

f77 -o whales tails.o -lfish
```

The **fish** library would actually be in a file called **libfish.a** stored in one of the library directories.

You can also write your own libraries: compile the object files, aggregate them with **ar**, then use **ranlib** to order them for library use. To modify where the compiler looks for libraries, use the environment variable **LD_LIBRARY_PATH**. See UNIX1 for instructions on environment variables.

Programming Utilities

UNIX provides several utilities for programming that can be used with executables created by any of the compilers mentioned in this document. These utilities are:

ctags	Create a tags file for use with the **ex** and **vi** editors. A tags file gives the location of functions and type definitions in a group of files.
gprof	Display call-graph profile data. This command produces a graphical execution profile of a program.
prof	Display execution profile data.
tcov	Construct test coverage analysis and statement-by-statement profile.

The FORTRAN Compiler

On UNIX systems, FORTRAN programs are created using the FORTRAN compiler, **f77**. There are many command line options for the compiler in addition to the ones described in the section, *Compiler Options*. Refer to the compiler manual pages on-line and in the *SunOS Reference Manual* and to the *Sun FORTRAN User's Guide*.

FORTRAN Utilities

The following are some of the UNIX utilities and applications that will assist you in creating FORTRAN programs.

fpr	A FORTRAN output filter for printing files that have FORTRAN carriage-control characters in the first column. Allows printing of non-UNIX style FORTRAN files on UNIX line-printers.
fsplit	Splits one FORTRAN file with many subroutines into multiple files, each with one subroutine in it.
dbx	A interactive symbolic debugger that can be used with FORTRAN.

adb An interactive general-purpose low-level debugger.

FORTRAN Resources

The following Sun system documents in the labs will provide more information on programming in FORTRAN:

Getting Started
Beyond the Basics
Performance Profiling Tools
Numerical Computation Guide
Debugging a Program, SPARCworks 3.1
Programming Utilities Guide
Building Programs with Maketool
Linker and Libraries Guide
Source Compatability Guide
Standards Conformance Guide
System Interface Guide

In addition, Fondren Library has several books and publications on programming in FORTRAN.

The C Compiler

There are three C compilers generally available on the UNIX systems, **cc**, **gcc,** and **acc. cc** is the C compiler from the system vendor, **gcc** is the C compiler from the Free Software Foundation's GNU Project, and **acc** is an ansi-compliant C compiler from Sun. There are numerous options for the C compilers in addition to the ones described in the section, *Compiler Options*. They are detailed in the **cc**, **gcc,** and **acc** manual pages and in the documents *C Programmer's Guide* and *Using and Porting GNU CC.*

Utilities

The following SunOS utilities can be used when programming in C. Some are used with source code files and others are used in conjunction with the C compiler and the executable program.

cb A C program beautifier.

cflow Generate a flow graph of C source code.

cpp The C preprocessor.

csplit Split a file with respect to a given context.

ctrace Generate a C program execution trace.

cxref	Generate a C program cross reference.
indent	Indent and format a C program source file.
lint	Check a C program for common errors.
mkstr	Create an error message file by massaging C source files.
xstr	Extract strings from C programs to implement shared strings.
lex	Lexical analysis program generator.
yacc	Yet another compiler-compiler; parsing program generator.

C Resources

The following Sun system documents in the labs will provide more information on programming in C:

C User Guide
Numerical Computation Guide
Debugging a Program
Programming Utilities Guide
C 4.2 Quick Reference Card

New manuals are the same as listed under FORTRAN Resources. Other resources for C programming available in the labs:

Using and Porting GNU CC, Richard Stallman, Free Software Foundation, Inc., 1990.
The C Preprocessor, Richard Stallman, Free Software Foundation, Inc., 1990.
lex & yacc, Tony Mason and Doug Brown, O'Reilly Associates, Inc., 1990.
Checking C Programs with lint, Ian Darwin, O'Reilly Associates, Inc., 1988.

In addition, Fondren Library has several books and publications on programming in C.

The C++ Compiler

The C++ compiler is invoked on UNIX systems with either the command **CC** for the system vendor version or **g++** for the Free Software Foundation version. The C++ compilers have many command line options in addition to the ones described in the section, *Compiler Options*. Refer to the **CC** or **g++** manual pages on-line.

C++ Resources

The following documents located in the labs provide extensive information on programming in C++.

C++ User's Guide, Michael Tiemann, Free Software Foundation, Inc., 1990.
C++ Library Reference, Doug Lea, Free Software Foundation, Inc., 1990.

In addition, Fondren Library has several books and publications on programming in C++:

C++ Quick Reference Card
Tools.h++ Class Library Reference
Tools.h++ User's Guide

Using Make

For a simple program, with only a single source code file, and not requiring any special libraries, you could compile and link with a single command, e.g.:

```
cc -o ducks ducks.c
```

A more complicated program, though, might involve the source code for several subroutines, and require several commands:

```
cc -c ducks.c
cc -c hughey.c
cc -c dewey.c
cc -c louie.c
cc -c ducks ducks.o hughey.o dewey.o louie.o
```

which would produce the executable **ducks**. In addition, if the program required the IMSL library functions, the last line would have to look like this:

```
cc -o ducks ducks.o hughey.o dewey.o louie.o -limsl
```

During program development, the programmer would have to keep track of which source code files were changed, when they would have to be recompiled to make the object code reflect the changes, and which special libraries needed to be included. For a project with hundreds of source files and several libraries, this can become an unmanageable task. The **make** program exists to automate the program development process.

Elements of Make

The concept of **make** is fairly simple: you write an additional text file, called a description file, which tells **make** what to do to produce the executable from the sources (other uses are possible but this is the most common). Then you can just type "make" and watch it do the work for you.

Here are some important **make** terms (which will become clearer with examples):

makefile — Another name for the description file, a text file containing commands that produce the desired result from the starting materials. The description file is often called this because it is usually named **makefile** or **Makefile**.

target — Something **make** is supposed to produce. An executable file is a good example. Object files can also be **targets**. Anything you need but don't

start with is a **target**. In the command sequences above, the executable **ducks** is the eventual target.

dependency — Something associated with a given target that must exist and be up-to-date before that target can be made. In the above example, **ducks** (executable) depends on **ducks.o**, **hughey.o**, **dewey.o**, and **louie.o** because they are needed to create it. **ducks.o**, in turn, depends on **ducks.c**, but not on **hughey.c**, **dewey.c**, or **louie.c**. **hughey.o**, **dewey.o**, and **louie.o** each depend on their respective source files also. So **ducks** depends on each of **ducks.c**, **hughey.c**, **dewey.c**, and **louie.c**, but only indirectly, through **ducks.o**, **hughey.o**, **dewey.o**, and **louie.o**.

rule — The command(s) that **make** uses to create a **target**.

An entry in a makefile looks like this:

```
target:   dependencies
    rule
```

The format is rigid: target, colon, one space, dependencies separated by a space, return, tab, rule, blank line. Note that it *must* be a single tab, 8 spaces will cause **make** to fail. If the number of dependencies is quite large or the rule is very long, you can break the line, putting a backslash (\) at the end of each incomplete part.

The makefile is typically a list of targets, dependencies, and rules, as in the following example:

```
whosonfirst: whosonfirst.o abbott.o costello.o
    cc -o whosonfirst whosonfirst.o abbott.o costello.o

whosonfirst.o: whosonfirst.c
    cc -c whosonfirst.c

abbott.o: abbott.c comic.h
    cc -c abbott.c

costello.o: costello.c comic.h
    cc -c costello.c
```

When you tell it to "make whosonfirst," **make** ensures that **whosonfirst**'s dependencies are up-to-date first: it sees the dependency **whosonfirst.o**, and "makes" **whosonfirst.o**, which it does by compiling **whosonfirst.c**. Similarly, it sees dependencies on **abbott.o**, and **costello.o**, and makes those, which it does by including **comic.h** into **abbott.c** and compiling **abbott.c** to **abbott.o** and so on. Then, when all the dependencies are satisfied, it can make **whosonfirst**.

Why all the talk about "up-to-date"? One of **make**'s best features is that it checks to see what's been changed since the target was last made. For example, if told to make **abbott.o**, it checks the last modification time/date on both **abbott.o** and **abbott.c**. If **abbott.c** has been modified since the last time **abbott.o** was modified, then **abbott.o** is no longer up-to-date, and it compiles **abbott.c** to **abbott.o**. If, however, **abbott.o** is more recent than the last modification to **abbott.c**, **make** concludes that no changes have been made, that recompiling would not result in any change in **abbott.o**, and that recom-

piling is therefore unnecessary. Thus, **make** keeps its targets up-to-date efficiently by changing only what needs to be changed.

Syntax of the Make Command

The **make** command has some primary arguments that it can be invoked with. The brackets ("[ ]") below indicate that the argument enclosed within them is optional. In other words, it only has to be supplied if you want to use it. (The brackets are not typed when supplying the argument.)

make [-f *makefilename*] [*target*]

The following is a description of each of the options shown above.

-f *makefilenam*e — **make** requires a description file for operation. If no filename is specified, **make** looks in the current directory for makefiles under the following names, in this order: **makefile**, **Makefile**, **s.makefile**, **s.Makefile**. (*Note*: **s.makefile** and **s.Makefile** can only be found in the SCCS revision control system.) If no filename is specified, and none of the above files is found in the current directory, **make** exits with an error message. Using the -f option relieves you of having to call the description file one of the names listed above. This is important when you want to have more than one makefile in a directory.

target — You can instruct **make** to make any target in the makefile by calling its name. In the above example,

make whosonfirst.o

would only make sure **whosonfirst.o** was up-to-date (with respect to **whosonfirst.c**). It would not even look at **whosonfirst**, **abbott.o**, or **costello.o**. If no target is specified, the first target in the makefile is used. Thus, it makes sense to put the main target (usually the executable) first in the list of targets. You can then just type "make" and the main target will be made.

There are many other command line options for **make** but they are used less frequently, or only for really complicated programs. Refer to the **make** manual pages in the *SunOS Reference Manual* or on-line for a description of these options.

There is another version of **make** available on some systems (including Owlnet), **gnumake** from the Free Software Foundation's GNU Project. **gnumake** is invoked in a similar manner to **make**.

Multiple Programs in a Makefile

Note that one makefile can be set to produce any number of executables or other targets by default, using a construction like this:

```
all: rain song

rain: rain.o cats.o dogs.o
```

```
        cc -o rain rain.o cats.o dogs.o -limsl

song: song.o three.o blind.o mice.o
        cc -o song song.o three.o blind.o mice.o

rain.o: rain.c
        cc -c rain.c umbrella.h

[...]
```

When you type “make,” make will make ‘all,’ which will result in both ‘rain’ and ‘song’ being made.

Other Elements in Makefiles

Non-File Targets

Sometimes targets that are not used to compile or link can come in handy. The target may have no dependencies and only a rule. A frequently used one is:

```
clean:
        rm -f *.o
```

so when you “make clean,” all object files in the current directory are removed, cleaning up the directory listing and saving space. (They can be recreated from the source if you need them again.) This is clearly not a good idea, however, if the object files from more than one program are in the same directory. Then you would want to specify which few objects to remove:

```
clean:
        rm -f sub1.o sub2.o
```

Or you could get even more creative, and add:

```
done: clean
        rm -f prog
```

Then, when you had run the program to your satisfaction and wished to delete the executable, “make done” would remove the executable and all relevant object code.

Comments

Comments can be added anywhere if the first non-blank character on the line is a (“#”). All characters up to the new line will be ignored. Each comment line must start with a (“#”).

Variables

It is possible to define variables at the beginning of the makefile to be used later on. The syntax for defining the variable is:

```
VARIABLE=value
```

where VARIABLE is (usually) the capitalized variable name you want to use, and *value* is the character string assigned to it. The value is all characters up to a newline. If the value is longer than a line, use the backslash character before the non-terminal newline characters.

The variable substitution takes place by using the following construct in the target, dependency, or rule.

```
$(VARIABLE)
```

For example, one could make up a generic makefile thus:

```
PROG=
SUBS=

$(PROG): $(PROG).o
    f77 -o $(PROG) $(PROG).o: $(SUBS)

$(PROG).o: $(PROG).f
    f77 -c $(PROG).f
```

Then, to use the makefile for different programs, only the lines at the top need be changed. With

```
PROG=makemyday
SUBS=
```

make sees the makefile as:

```
makemyday: makemyday.o
    f77 -o makemyday makemyday.o

makemyday.o: makemyday.f
    f77 -c makemyday.f
```

Whereas, with

```
PROG=makemyday
SUBS=go.o ahead.o punk.o
```

it looks like

```
makemyday: makemyday.o
    f77 -o makemyday makemyday.o go.o ahead.o punk.o

makemyday.o: makemyday.f
    f77 -c makemyday.f
```

(Which is fine if you're not planning to change **go.f**, **ahead.f**, or **punk.f**, and you just have the **.o** files in your directory. If you want to include them as targets, you need to add them to the makefile.)

Also, a variable can be used to pass other information on to make. Having

```
SHELL=/bin/sh
```

at the beginning with the variable settings results in all the rules being run under the default Bourne shell, **/bin/sh**, rather than the C shell, **/bin/csh**. This is necessary, for example, when using the IBM RTs to compile FORTRAN.

Make Resources

The following documents, located in the labs or available from the CRC, will help you in using **make** to maintain programs.

Programming Utilities Guide, Chapter 5
Beyond the Basics

Debuggers

Image files

On UNIX systems, when you run a program, a copy of it is loaded into a portion of the system's memory and the CPU begins executing the instructions there. If an error occurs in the program (a "bug"), often the execution will halt abruptly. In these instances, the operating system quickly saves a copy of the relevant portion of its memory to a file on the disk. This is known as a *memory dump*, and the file is called an *image file*. The memory dump is often referred to as a *core dump* for historical reasons. (Computer memory used to be called "core memory" because before the advent of silicon microchips for memory, computer memory was constructed of small ferrous cylinders or rings wrapped in copper wire. The ferrous part was referred to as the "core.") Following this tradition, the image file dumped to disk is usually named *core*. Since this image file is a duplicate of what was in the computer's memory at the time of the error, it can be used to discover what the error was.

Available Debuggers

Some of the debuggers available on the various UNIX systems are:

adb	A general purpose interactive debugger.
dbx	A robust symbolic debugger.
xdbx	A more friendly, X Window System interface to **dbx**.
gdb	Free Software Foundation GNU Project symbolic debugger.

Debugger Resources

The following documents in the documentation racks in the labs will assist you in using the various debuggers available on the system.

Debugging a Programl
gdb Manual

Storing Your Files

While you are developing a program, you will usually generate a number of object files (**.o** files) in addition to the source, header, and executable files. When your program development is complete, you can save a lot of disk space by deleting your **.o** files. You can use **make** to clean up your directory as described in the section, *Other Elements in Makefiles*. All of the object files can be recreated from the source files by invoking **make** again.

Problem or Questions

If you have problems or questions, you can contact the Consulting Center (103 Mudd Lab, or at 713.348.4983). You can also submit it on the Web at http://problem.rice.edu or you can send e-mail to problem@rice.edu.