Physics 329: Using Fortran and C in the Unix Environment

- Introduction and overview
- Compiling and linking Fortran programs: **f77**
- Compiling and linking C programs: **cc**
- Using and creating libraries
- Debugging programs: **dbx**
- Organizing and automating program builds: **make**

INTRODUCTION AND OVERVIEW

The purpose of these notes is to provide you with the basic information required to produce executables (programs) from Fortran and C source files within a Unix environment. I recommend that you do all of your Fortran 77 (hereafter referred to as Fortran) and C programming on the SGI machines (preferably `einstein`) and the discussion below is *somewhat* SGI-specific. However, the same basic ideas and techniques should be applicable on most Unix systems.

Before proceeding to specific discussions of the Unix commands we will use for program development, let us consider the basic job of a compiler and work through some simple examples. A compiler translates (or compiles) high-level code (such as C or Fortran) into a form which the hardware can more or less run directly. In brief then, a compiler's job is to convert source code into executables. In the simplest case, the source code will reside in a single source file: by convention, Fortran source code is prepared in files whose names have a `.f` extension. Here is a simple example:

```
% ls *.f
first.f

% cat first.f
      write(*,*) 'Hello World!'
      stop
      end
```

Here the Fortran source file `first.f` contains a complete Fortran program. We can produce an executable using the **f77** command as follows:

```
% f77 -g -n32 first.f -o first
```

The Fortran compiler silently does its work, producing an executable called `first` which you can then execute simply by entering '`first`:

```
% first
 Hello World!
```

The **f77** command issued above requires a little explanation. In addition to the source file **first.f**, we supply as arguments the option **-g** (debug option) which tells the compiler to include information in the executable to facilitate debugging, the option **-n32** which tells the compiler to generate "new-32-bit code" (very SGI-specific), and the option **-o first** which tells **f77** to name the executable `first`. Thus, there are effectively 4 arguments to **f77** in the above example: (1) **-g**, (2) **-n32**, (3) **first.f**, and (4) **-o first**. These can appear in any order on the command line, so the following invocations (among others), are equivalent to our original form:

```
% f77 -o first -n32 -g first.f
% f77 first.f -o first -n32 -g
```

If you don’t specify a name for the executable using the **-o** option, **f77** will call your executable `a.out`:

```
% f77 -g -n32 first.f
% a.out
 Hello World!
```

However, I strongly recommend that you avoid using this default behaviour.

Let us consider a slightly more complex example, in which we introduce the concept of an "intermediate" level of code---known as *object code*---which is also compiled from source code, but which is not directly executable. In this example we have two Fortran source files, `greeting.f` which contains a Fortran *main* program and `sayhello.f` which contains a Fortran *subroutine* (or procedure) that the main program calls:

```
% cd ~phy329/f77/ex1
% ls *.f
greeting.f   sayhello.f
% cat greeting.f
c     This is the main program

      program  greeting

      call sayhello()

      stop
      end

% cat sayhello.f
c     This is the subroutine

      subroutine sayhello()
         write(*,*) ’Hello World!’
         return
      end
```

As in our previous example, we can generate an executable directly using the **f77** command; we simply pass *both* source files as arguments:

```
% f77 -g -n32 greeting.f sayhello.f -o greeting
greeting.f:
sayhello.f:
```

Note that this time, the **f77** command echos the name of each source file (followed by a colon) as it is processed. Also note that the name of the executable produced is `greeting`:

```
% greeting
 Hello World!
```

If we check the contents of the directory:

```
% ls
greeting*    greeting.f   greeting.o   sayhello.f   sayhello.o
```

we notice that in addition to the executable, `greeting`, the **f77** command created two files, `greeting.o` `sayhello.o`, both having **.o** extensions. These are *object* files which, as mentioned above, you can view as an "intermediate" level between source code and executable code. Loosely speaking then, the process of translating source code into executable code in Unix can be separated into two phases:

- The *compilaton* of source code into object code
- The *linking* of object code (including code in *library archives*) to produce an executable.

Without going into too much detail, the linking phase involves assembling the main routine and various subroutines (or procedures), which constitute a program, to produce an executable.

Although there is a separate linking command in Unix (usually called **ld**), you don't have to invoke it directly---the **f77** (or **cc**) command will do it for you, *provided you have issued a command which calls for the creation of an executable*. Such is the case in the two examples above where we generally used the **-o** option. However, we can also use **f77** to create an executable in two phases. First, by supplying the **-c** option to the compiler we request that **.f** files *only be compiled* into **.o** files:

```
% RM greeting *.o
% ls
greeting.f   sayhello.f

% f77 -n32 -c greeting.f sayhello.f
greeting.f:
sayhello.f:

% ls
greeting.f   greeting.o   sayhello.f   sayhello.o
```

Note that this last **f77** command did *not* create an executable. To make the executable, we supply **f77** with the names of the **.o** files which contain the object code which we wish to be "linked" to create an executable, and, as in our early examples, we use the **-o** option to give the executable a specific name:

```
% f77 -n32 greeting.o sayhello.o -o greeting

% ls
greeting*    greeting.f   greeting.o   sayhello.f   sayhello.o

% greeting
 Hello World!
```

Here, **f77** basically passes all of its arguments along to the actual loader command, **ld**, along with additional information for **ld** which is common to all Fortran programs. The loader then creates the executable.

Although the two phase process of first creating object files using the **-c** option, and then linking them together to create an object file may seem awkward, there are advantages to this technique. For example, if we are working with a program consisting of many thousands of lines of source code contained in many distinct source files, and make a change to *one* of the source files, then by using separate compilation and link phases, we need only recompile (using **-c**) the single source file which was modified, then relink *all* of the object files to produce a new executable. For large programs this can significantly decrease the development cycle-time and hence is recommended practice. *However, for short programs, and in particular for programs which are entirely contained in a single source file, the first approach above will suffice.*

We end this survey with a brief discussion of *libraries* (or *library archives*) in Unix. Libraries are closely related to object code: and you can think of them as collections of routines (procedures, functions) which have been converted into object code and which are ready to be included (linked to) by any program which wants to call them. A simple C example will illustrate the idea:

```
% cd ~phy329/cc/ex1
% cat twinopen.c
/* Simple illustration of use of libraries ... */
#include <stdio.h>
#include <gl.h>

void main(int argc, char **argv) {
   winopen("twinopen");
   fprintf(stderr,"twinopen: Opened a window.\n");
}
```

This simple routine calls an SGI-specific routine `winopen` which opens a new window on the current display. However, `winopen` *is not* a basic part of the C language. Thus, if we naively try to build an executable, we get an "Unresolved"" error message:

```
% cc -n32 twinopen.c -o twinopen
cc -n32 twinopen.c -o twinopen
ld32: ERROR 33: Unresolved text symbol "winopen" -- 1st referenced by twinopen.o.
ld32: INFO 60: Output file removed because of error.
cc ERROR:  /usr/lib64/cmplrs/ld32 returned non-zero status 1
```

which tells us that the loader, **ld**, (automatically invoked here by the **cc** command, as it was in the **f77** examples above) was unable to locate object code for *any* routine named `sqrt`. In this case, the way to fix the problem is to include the option **-lgl** (for "graphics library") on the command line:

```
% cc -n32 twinopen.c -lgl -o twinopen
```

This time the compilation and linking succeeds, so we can execute `twinopen`:

```
% twinopen
twinopen: Opened a window.
```

We will discuss libraries in a little more detail below. Here I will only point out that the rather cryptic option **-lgl** combined with the fact that a file with the name

```
libgl.so
```

exists in the directory

```
/usr/lib
```

is sufficient to make things work. More specifically, **-lgl** tells the loader that it should search for a file named

```
libgl.a    or    libgl.so
```

in one of the "standard" directories where libraries are maintained on the system: since `/usr/lib` is one such directory, the loader finds the library, inspects it, detects that it contains object code for a routine named `winopen` and links it into the executable. The search that the loader performs for a specific library archive is very much analogous to the resolution of names of commands using the path, and it is possible to extend the "search path" for libraries as is described below.

COMPILING AND LINKING FORTRAN PROGRAMS: f77

Use the **f77** command to compile and link Fortran programs.

USAGE EXAMPLES

The following command compiles and loads `pgm.f` with the debug option, creating the executable `pgm`.

```
% f77 -g -n32 pgm.f -o pgm
```

The first of the following commands compiles `main.f` and `subs.f` producing object files `main.o` and `subs.o`. The second command loads both object files creating the executable `main`.

```
% f77 -g -n32 -c main.f subs.f
% f77 -g -n32 main.o subs.o -o main
```

The following example is the same as the previous one, except that we now link to the library `/usr/localn32/lib/libp329f.a` using the **-L** and **-l** options

```
% f77 -g -n32 -c main.f subs.f
% f77 -g -n32 -L/usr/localn32/lib main.o subs.o -lp329f -o main
```

Note that `-L/usr/localn32/lib` adds the *directory* `/usr/localn32/lib` to the default search path the loader uses when searching for library archives. The option `-lp329f` tells the loader which specific archive it is seeking. Observe that the strings `lib` and `.a` are *always* pre- and post-pended, respectively, to create the actual filename of the archive (`libp329f.a` in this case).

USEFUL f77 OPTIONS

See **man f77** for additional information. Note that compiler options tend to be system-specific. Options similar to those described here should be available on most Unix Fortran implementations.

- **-n32** Generate "new 32-bit" code. SGI specfic. *Always use* this option when compiling on one of the SGIs.
- **-g** Debug option. Recommended if you want to use **dbx** for program debugging.
- **-O** Perform fairly agressive code optimization. Recommended for production work, after code has been thoroughly tested and debugged.
- **-C** Generate code for runtime subscript range checking (array bounds checking). Default is to

supress checking. Highly recommended for program development since violation of array bounds is among the most common error Fortran programmers make.

COMPILING AND LINKING C PROGRAMS: cc

Use the **cc** command to compile and link C programs

USAGE EXAMPLES

The following command compiles and loads `pgm.c` with the debug option, creating the executable `pgm`.

```
% cc -g -n32 pgm.c -o pgm
```

The first of the following commands compiles `main.c` and `routines.c` producing object files `main.o` and `routines.o`. The second command loads both object files creating the executable `main`.

```
% cc -g -n32 -c main.c routines.c
% cc -g -n32 main.o routines.o -o main
```

The following example is the same as the previous one, except that we now link to the libraries `/usr/localn32/lib/libp329f.a` and `/usr/lib/libm.a` using the **-L** and **-l** options

```
% cc -g -n32 -c main.c routines.c
% cc -g -n32 -L/usr/localn32/lib routines.o subs.o -lp329f -lm -o main
```

Note that when multiple libraries are specified in the load phase, as they are above, the loader searches each library *exactly once* for unresolved symbols and searches in the order specified on the command line. Thus, if the program above calls a routine in the `p329` library, and that routine calls a routine in the `m` library (standard C math support) then

```
% cc -g -n32 -L/usr/localn32/lib routines.o subs.o -lm -lp329f -o main
```

will result in a load error.

#include’ing FILES FROM NON-STANDARD LOCATIONS

Statements of the form

```
#include "mytypes.h"
#include <stdio.h>
```

are C pre-processor directives which effectively include the contents of a file in-place in the C source. In the first form, where the filename is enclosed in double quotes ("), the specified file must reside in the working directory. In the second case, where the filename is enclosed in angle-brackets (<>), the preprocessor searches for the file in the ‘‘standard include directory’’, `/usr/include`. Additional directories which are to be searched for `#include` files may be specified with the **`-I`** option. Thus, assuming that `mytypes.h` lives in `/usr/people/matt/include` and that the source code for `myinclude.c` contains the statement

```
#include <mytypes.h>
```

then the **cc** command

```
% cc -I/usr/people/matt/include myinclude.c -o myinclude
```

will ensure that the file is properly included.

USEFUL cc OPTIONS

See **man cc** for additional information. Note that compiler options tend to be system-specific. Options similar to those described here should be available on most Unix C implementations.

- **-n32** Generate "new 32-bit" code. SGI specfic. *Always use* this option when compiling on one of the SGIs.
- **-g** Debug option. Recommended if you want to use **dbx** for program debugging.
- **-O** Perform fairly agressive code optimization. Recommended for production work, after code has been thoroughly tested and debugged.

USING AND CREATING LIBRARIES

Libraries (library archives) in Unix have, by convention, names which begin with `lib` and end with `.a`:

```
% cd /usr/lib
% ls lib*.a
lib300.a                  libblas_mp.a              libmangle.a
lib300s.a                 libbsd.a                  libmdebug.a
                            .
                            .
                            .
```

Link to libraries located in standard locations (notably `/lib` and `/usr/lib`) using the **-l** option to either **f77** or **cc**:

```
% cc  -n32 cpgm.o  -lm -lX11 -o cpgm
% f77 -n32 f77pgm.o -lblas -o f77pgm
```

Use the **-L** option to prepend a directory to the default search path for library archives. Thus, assuming that I have a library named `/usr/people/matt/lib/vutil.a`, the following **f77** command will link (if necessary) to the archive:

```
% f77 -n32 -L/usr/people/matt/lib pgm.o -lvutil -o pgm
```

CREATING LIBRARIES

Create and maintain library archives using the **ar** (archive) command.

Typically one creates an archive file from one or more object files. Thus assuming that the following object files reside in the working directory:

```
% ls *.o
procs1.o procs2.o procsio.o
```

Then the following **ar** command will create or overwrite a library archive file **libmylib.a** containing all routines defined by the 3 object files and will ensure that the archive has a ‘‘table-of-contents’’ as required by the loader:

```
% ar r libmylib.a procs1.o procs2.o procsio.o
```

Note that the `r` immediately following `ar` in the above is an option (replace) to the **ar** command: i.e. **ar** options do *not* begin with a minus sign. Also note that on some systems, **ar** will not automatically add a table of contents. In such cases there is usually a command **ranlib** which will do the job.

See **man ar** for more information.

DEBUGGING PROGRAMS: dbx

This section is still under construction, but, for example, to debug a program named `pgm`, try

```
% dbx pgm
dbx version 3.19 Nov  3 1994 19:59:46
Executable /tmp/pgm
(dbx) help
(dbx) help list
(dbx) help run
(dbx) help print
(dbx) help stop
```

to get going. Use

```
(dbx) quit
```

to exit **dbx**.

ORGANIZING AND AUTOMATING PROGRAM BUILDS: make

As discussed in class, the **make** program (utility) is primarily used to organize and automate compilation and linking of programs. By convention, input for **make** is prepared in any given directory which contains source code in a file named `makefile` or `Makefile`. Following is an example (with additional in-text comments) for the source code in `~phy329/f77/ex3`on `einstein`. **Important:** Note that all command-lines which define how a particular target is made MUST begin with a TAB character: be careful if you cut and paste parts of this example into one of your own makefiles.

```
############################################################
# Lines beginning with a '#' are comments
############################################################

############################################################
# The .IGNORE: directive tells 'make' to keep going if
# one or more commands executed as a result of the 'make'
# do not complete successfully.  The default is to bail
# out.
############################################################
.IGNORE:

############################################################
# Define some macros for
#
#    F77:       the name of the Fortran 77 compiler
#    F77FLAGS:  generic Fortran flags (to be used in both
#               the compile and load phases)
#    F77CFLAGS: flags to be used in the compilation phase
#    F77LFLAGS: flags to be used in the load phase
#
# Note that the right hand side of a macro definition can
# contain multiple words with intervening white space
# without needing to be enclosed in quotes
############################################################
F77       = f77
F77FLAGS  = -g -n32
F77CFLAGS = -c
F77LFLAGS = -L/usr/localn32/lib

############################################################
# Define some macros for
#
#    F77_COMPILE:  The command which will be used to compile
#                  Fortran source
#    F77_LOAD:     The command which will be used to load
#                  Fortran object files, link to libraries
#                  and create executables.
#
# In this Makefile, macros are used almost precisely like shell
# or environment variables.  Note, however, that macros are
# evaluated with the $(MACRONAME) construct: the () are CRUCIAL.
# Also note that ALL environment variables (HOME, DISPLAY,
# etc.) are automatically available as macros with the
# same name.  Thus, for example, $(HOME) will evaluate to
# your home directory.  You can use this feature to create
# Makefiles which are portable across systems provided that
# the appropriate environment variables are set properly
# (tpyically in your '~/.cshrc') on the various systems.
############################################################
F77_COMPILE  = $(F77) $(F77FLAGS) $(F77CFLAGS)
F77_LOAD     = $(F77) $(F77FLAGS) $(F77LFLAGS)
```

```
############################################################
# The following defines a GENERIC target (rule)
# which tells ’make’ how to produce a ’.o’ file from
# a ’.f’ file.  ’Make’ will automatically use such a rule
# unless a specific target overrides it.
############################################################
.f.o:
        $(F77_COMPILE) $*.f

############################################################
# Define a macro for all the executables in the directory
############################################################
EXECUTABLES = fdemo2 mysum tdvfrom tdvto

############################################################
# Since this is the first SPECIFIC target in the makefile,
# if ’make’ is invoked with no arguments, this is the target
# which will be made.  Since $(EXECUTABLES) evaluates to
# ’fdemo2 mysum tdvfrom tdvto’, ’make’ will make each of
# ’fdemo2’, ’mysum’, ’tdvfrom’ and ’tdvto’ in turn
############################################################
all: $(EXECUTABLES)

############################################################
# The target ’fdemo2’ depends on the object file ’fdemo2.o’.
# When ’fdemo2’ is being made, ’make’ figures out that it
# first needs to make ’fdemo2.o’ from ’fdemo2.f’ using
# the generic rule above.  Once the dependencies of any
# given  target have been updated, the commands which
# follow the target are executed in turn.  In this case,
# the object file is simply loaded and the executable
# ’fdemo2’ is created.  Again note that each command line
# MUST BEGIN WITH A TAB.  Continue long lines with \
# (backslash, followed by carriage return, with no
# intervening spaces).
############################################################
fdemo2: fdemo2.o
        $(F77_LOAD) fdemo2.o -o fdemo2

mysum: mysum.o
        $(F77_LOAD) mysum.o -o mysum
############################################################
# A little more complicated example since there are 2
# dependencies (’tdvfrom.o’ and ’dvfrom.o’) and we
# link to the ’p329f’ library
############################################################
tdvfrom: tdvfrom.o dvfrom.o
        $(F77_LOAD) tdvfrom.o dvfrom.o -lp329f -o tdvfrom

tdvto: tdvto.o dvto.o
        $(F77_LOAD) tdvto.o dvto.o -lp329f -o tdvto
############################################################
# Makefiles often have a ’clean’ target which cleans
# up object files, executables and other files which
# tend to consume precious disk space, and which can
# always be reconstructed (via ’make’ of course!)
############################################################
clean:
        rm *.o
        rm $(EXECUTABLES)
```

Here’s the same example with the bulk of the comments removed:

```
.IGNORE:

F77       = f77
F77FLAGS  = -g -n32
F77CFLAGS = -c
F77LFLAGS = -L/usr/localn32/lib

F77_COMPILE  = $(F77) $(F77FLAGS) $(F77CFLAGS)
F77_LOAD     = $(F77) $(F77FLAGS) $(F77LFLAGS)

.f.o:
        $(F77_COMPILE) $*.f

EXECUTABLES = fdemo2 mysum tdvfrom tdvto

all: $(EXECUTABLES)

fdemo2: fdemo2.o
        $(F77_LOAD) fdemo2.o -o fdemo2

mysum: mysum.o
        $(F77_LOAD) mysum.o -o mysum

tdvfrom: tdvfrom.o dvfrom.o
        $(F77_LOAD) tdvfrom.o dvfrom.o -lp329f -o tdvfrom

tdvto: tdvto.o dvto.o
        $(F77_LOAD) tdvto.o dvto.o -lp329f -o tdvto

clean:
        rm *.o
        rm $(EXECUTABLES)
```

and here’s some output from **make** generated with our makefile:

```
% make
        f77 -g -n32 -c fdemo2.f
        f77 -g -n32 -L/usr/localn32/lib fdemo2.o -o fdemo2
        f77 -g -n32 -c mysum.f
        f77 -g -n32 -L/usr/localn32/lib mysum.o -o mysum
        f77 -g -n32 -c tdvfrom.f
        f77 -g -n32 -c dvfrom.f
        f77 -g -n32 -L/usr/localn32/lib tdvfrom.o dvfrom.o -lp329f -o tdvfrom
        f77 -g -n32 -c tdvto.f
        f77 -g -n32 -c dvto.f
        f77 -g -n32 -L/usr/local32/lib tdvto.o dvto.o -lp329f -o tdvto
```

Since the first specific target in the makefile is `all`, the above is equivalent to

```
% make all
```

The Unix **touch** command simulates modification of its file arguments by setting the last-modified time of its arguments to the current time. Thus having previously made everything,

```
% touch dvto.f
% make
        f77 -g -n32 -c dvto.f
        f77 -g -n32 -L/usr/localn32/lib tdvto.o dvto.o -lp329f -o tdvto
```

we see that **make** re-makes only those targets which depend on the single modified file. Note that we can easily make a single target by supplying the target as the sole argument to **make**:

```
% make clean
        rm *.o
        rm fdemo2 mysum tdvfrom tdvto
% make fdemo2
        f77 -g -n32 -c fdemo2.f
        f77 -g -n32 -L/usr/localn32/lib fdemo2.o -o fdemo2
```

Finally if **make** deduces that a target is up to date, it will generally tell you so:

```
% make fdemo2
‘fdemo2’ is up to date.
```

See **man make** or the suggested Unix references for more information.