Hints for Compiling Software

Howard Powell
howard.powell@gmail.com

Version: July 16, 2012

Contents

Chapter 1

What are these files?

What's a .a file
.a files are ar archives, similar to .tar files. ar archives typically hold libraries which provide subroutines for other programs to use.

What's a .o file
.o files are object files, which are code that has been assembled by the compiler but not yet linked together to form the final executable file.

What's a .so file
.so files are Linux shared object files. Shared objects are commonly used routines (for example the jpeg library) which are shared amongst many programs. Rather than create a copy of the jpeg routines in every file, the executables simply link to the .so file and obtain the routines necessary to run.

What's a .dylib file (mac)
.dylib files are dynamic libraries, and for the purposes of this document are the mac equivalent of .so files from Linux.

What's a .c file
.c files are usually ASCII text files containing C code.

What's a .cc file
.cc files are usually ASCII text files containing C++ code.

What's a .c++ file
.c++ is another extension for ASCII files containing C++ code.

What's a .C file
.C files could be ASCII files containing C code, C++ code, or it could be a unix compressed

file. This one's pretty ambiguous, so avoid using it.

What's a .h file
.h files are C header files. Typically these are ASCII text files containing accessory code and definitions necessary to compile a program.

What's a .f file
.f files are Fortran (typically understood to be Fortran 77) source code files.

What's a .f77 file
.f77 files contain Fortran 77 code. This extension is preferred over the .f extension.

What's a .f90 file
.f90 files contain Fortan 90 code.

What's a .f95 file
.f95 files contain Fortran 95 code.

1.1 What are these weird paths?

-L (path to shared objects directory)

-I (path to header include files)

Chapter 2

Basic Compiling Tricks

2.1 Getting Started

This guide is meant to help you get started compiling software with the UVa Astronomy Department. By the physical laws of computing, it cannot be all inclusive, but hopefully it contains enough tricks to get you started on your way to success.

Compiling software is an art unto itself, and requires some black magic and voodoo to get it all working. Just be patient and try to figure out what's happening when your compile fails.

2.2 32 bit vs 64 bit

The first question everyone asks is "should I compile this as a 32 bit binary or a 64 bit binary? The short answer is - *only create a 64 bit binary if you need to access more than 2GB of ram with that program.* More specifically, if any thread of your program needs access to more than 2GB of ram, you'll need to compile a 64 bit binary. Otherwise a 32 bit binary will be more useful (as it will run on both 32 bit and 64 bit machines) and much simpler to build and maintain.

Contrary to popular belief, 64 bit binaries will not necessarily run faster. On a same-clock-speed cpu, a 32 bit binary should run faster. However, 64 bit CPUs often have more on-chip cache (which greatly enhances performance) and sometimes are performance tuned to run 64 bit binaries faster. If performance is your goal, you'll want to benchmark both a 32 bit and 64bit binary and see which performs better.

The biggest problem with creating a 64 bit binary is you cannot link to a 32 bit dependency library, and vice-versa. When compiling a 64 bit binary, all 64 bit versions of dependent shared object files have to be present to create an working executable.

2.2.1 32 bit compiling

On a 32 bit machine, a 64 bit binary *will not run*.

On a 32 bit machine, the GNU compiler will automatically produce a 32 bit binary.

On a 32 bit machine, the GNU compiler can produce a 64 bit binary (assuming it can link any 64 bit dependency libraries) by using the -m64 flag.

On a 32 bit machine, the automake system (running ./configure; make; make install) will automatically create a 32 bit binary.

On a 32 bit machine, the 64 bit intel compiler will not run. It is not possible to create a 64 bit binary on a 32bit machine using an Intel compiler.

2.2.2 64 bit compiling

On a 64 bit machine, a 32 bit binary *will run*.

On a 64 bit machine, the GNU compiler will automatically produce a 64 bit binary.

On a 64 bit machine, the GNU compiler can produce a 32 bit binary (assuming it can link any 32 bit dependency libraries) by using the -m32 flag.

On a 64 bit machine, the automake system (running ./configure; make; make install) will automatically create a 64 bit binary.

On a 64 bit machine, there are separate 32 bit and 64 bit versions of the intel compilers. Please be sure to use the correct version to get the type of binary you expect.

2.3 Setting up your Environment

2.3.1 $PATH

The GNU compiler is part of the default $PATH (it's in /usr/bin/).

The 32 bit Intel compiler is in /astro/intel. The latest version of the compiler is always at /astro/intel/latest/bin.

The 64 bit Intel compiler is in /astro64/intel. The latest version of the compiler is always at /astro64/intel/latest/bin.

To modify your $PATH in tcsh, use setenv:

```
[hbp4c@Realos ~]$ setenv PATH /astro/intel/latest/bin:$PATH
```

Note: If you source the intel variables script (/astro/intel/latest/bin/iccvars.csh for 32 bit or /astro64/intel/latest/bin/iccvars.csh for 64 bit), your $PATH will automatically include the intel compiler bin path.

To modify your $PATH in bash, use export:

```
[hbp4c@Realos ~]$ export PATH=/astro/intel/latest/bin:${PATH}
```

Some examples of common $PATHs that you might consider:

- /astro/bin
- /astro64/bin (for 64 bit software only!)
- /astro/intel/latest/bin

Note: If you source the intel variables script (/astro/intel/latest/bin/iccvars.sh for 32 bit or /astro64/intel/latest/bin/iccvars.sh for 64 bit), your $PATH will automatically include the intel compiler bin path.

2.3.2 $LD_LIBRARY_PATH

Normally I try to encourage users to avoid setting an $LD_LIBRARY_PATH, but the exception is when compiling software. The $LD_LIBRARY_PATH variable lets your shell figure out where to look for shared object files (libraries). Modifying this variable sometimes leads to unintended consequences.

To modify your $LD_LIBRARY_PATH in tcsh, use setenv:

```
[hbp4c@Realos ~]$ setenv LD_LIBRARY_PATH /astro/intel/latest/lib:$LD_LIBRARY_PATH
```

If you get an error about an undefined $LD_LIBRARY_PATH, simplify the above to this:

```
[hbp4c@Realos ~]$ setenv LD_LIBRARY_PATH /astro/intel/latest/lib
```

Note: If you source the intel variables script (/astro/intel/latest/bin/iccvars.csh for 32 bit or /astro64/intel/latest/bin/iccvars.csh for 64 bit), your $LD_LIBRARY_PATH will automatically include the intel libraries.

To modify your $LD_LIBRARY_PATH in bash, use export:

```
[hbp4c@Realos ~]$ export LD_LIBRARY_PATH=/astro/intel/latest/lib:${LD_LIBRARY_PATH}
```

Some examples of common $LD_LIBRARY_PATHs that you might consider:

- /astro/lib
- /astro64/lib (for 64 bit software only!)
- /astro/intel/latest/lib

Note: If you source the intel variables script (/astro/intel/latest/bin/iccvars.sh for 32-bit or /astro64/intel/latest/bin/iccvars.sh for 64-bit), your $LD_LIBRARY_PATH will automatically include the intel libraries.

2.3.3 $LD_RUN_PATH

$LD_RUN_PATH is a special shell variable that is only used when compiling software. When running software, the shell usually checks the locations in $LD_LIBRARY_PATH plus a few system-defined locations for shared object files and dependencies for the binary. If you set $LD_RUN_PATH before you compile a binary object, any paths you add to $LD_RUN_PATH will be hard-coded into the resulting binary. By setting this variable, you will save yourself and others from having to remember to set a custom $LD_LIBRARY_PATH in the future.

I recommend setting $LD_RUN_PATH equal to $LD_LIBRARY_PATH if you use the latter variable. Otherwise, set $LD_RUN_PATH to whatever paths you need to link to shared objects.

To set your $LD_RUN_PATH in tcsh, use setenv:

```
[hbp4c@Realos ~]$ setenv LD_RUN_PATH /astro/intel/latest/lib:$LD_LIBRARY_PATH
```

If you get an error about an undefined $LD_LIBRARY_PATH, simplify the above to this:

```
[hbp4c@Realos ~]$ setenv LD_RUN_PATH /astro/intel/latest/lib
```

To set your $LD_RUN_PATH in bash, use export:

```
[hbp4c@Realos ~]$ export LD_RUN_PATH=/astro/intel/latest/lib:$LD_LIBRARY_PATH
```

If you get an error about an undefined $LD_LIBRARY_PATH, simplify the above to this:

```
[hbp4c@Realos ~]$ export LD_RUN_PATH=/astro/intel/latest/lib
```

Some examples of common $LD_RUN_PATHs that you might consider:

- /astro/lib
- /astro64/lib (for 64 bit software only!)

- /astro/intel/latest/lib

Note that the iccvars.csh and iccvars.sh scripts do not modify $LD_RUN_PATH.

Chapter 3

Using Autoconf Scripts

3.1 What's Autoconf?

Autoconf is a popular Unix tool that can probe your user environment, detect what is available and installed, and then set up a Makefile using that information all automatically. Many GNU projects you may download the source for from the web will use the autoconf tool to create the Makefile necessary to build your program.

Autoconf itself is often run by the builder or maintainer of the code base. You, the end user, will interact with the autoconf system by running a "configure" script which tests your computer and creates the makefile. After the configure step is done, you simply need to run make and optionally "make install" to compile and install the software you want to use.

3.2 Basic Configuration

3.2.1 configure –help

Possibly the most useful thing you'll want to do first is check the help script to see what options you can specify when configuring your package.

```
[hbp4c@Realos ~]$ ./configure --help
```

3.2.2 configure –prefix=

Often, your account will not have access to install to a system-wide directory, or else you will want to put your compiled software in a special, shared location. By default, most software configure scripts try to put the software in /usr or /usr/local. Using the –prefix option, you can specify a location to install your software to:

```
[hbp4c@Realos ~]$ ./configure --prefix=~/
```

3.2.3 configure –enable/–disable

Sometimes you can turn on or off optional components and parts of the program. The –help option usually lists all of the optional components you can select or deselect, and explains what is the default if unspecified.

3.2.4 configure –with/–without

Sometimes configure cannot detect software packages that your program needs to be able to compile and link to. If this is the case, you can usually override configure's automatic detection and specify an exact location to a shared library or tool.

3.3 Environmental Variables

The configure script also pays attention to your environmental variables to decide what compiler to use and what options to compile with. For example, you can specify using the intel compiler by setting the $CC variable to "icc" before running configure. The configure script will detect this, and create it's Makefile with icc as the compiler instead of gcc.

Some of the typical environment variables are listed below, others are always listed at the end of the –help output for your package.

3.3.1 $CC

Use $CC to select the preferred C compiler. Your options are typically gcc, or icc (the intel compiler).

3.3.2 $CFLAGS

Use $CFLAGS to pass options to the C compiler at build time. For example, you can optimize the code by passing the -O flag.

Keep in mind that whatever you set for $CFLAGS will get passed to every invocation of the C compiler.

3.3.3 $LDFLAGS

The $LDFLAGS variable is used to pass options to the linker, after the compiler has created output files. The linker is invoked to assemble the .o files into larger, complete binary executables. In some cases, you may need to add -L locations to specify paths to libraries installed in non-standard locations, such as /astro/lib. This is comparable to setting $LD_LIBRARY_PATH.

3.3.4 $CXX

The $CXX variable stores the preferred C++ compiler.

3.3.5 $CXXFLAGS

$CXXFLAGS is the equivalent of $CFLAGS, but for the C++ compiler.

3.3.6 $F77/$F90

The $F77 and/or $F90 variables specify the preferred Fortran compiler.

3.3.7 $FFLAGS

The $FFLAGS sets the options to pass to the fortran compiler at build time.

3.4 Make

Once your Makefile has been generated by the configure script, you simply need to run 'make' to compile your software. Depending on what you're building, this can be very quick or take a long time.

Make also accepts some flags to help control the program. In particular, you might want to compile multiple parts of your software at the same time by using multiple CPUs on modern computers. You can do this with the -j flag.

```
[hbp4c@Realos ~]$ make -j4
```

The above example starts 4 make threads all working on different input files. In theory this should speed up your compile time by a factor of 4, but in practice some parts of your program may need to wait until other parts are already compiled before they start. Also, starting more threads than you have available CPUs can simply overwhelm a computer, making the compile step far slower than it otherwise would be.

3.4.1 Make Install

Finally, most software includes a special rule in the makefile that handles the automatic installation of your package. 'make install' is the final step to copy the resulting binary to it's final home, wherever you specified in –prefix during the configure step, or in /usr/ or /usr/local.

3.4.2 Make Clean

Sometimes you screw up. Maybe you'll forget to specify a –prefix after all, or you'll forget to enable an option or specify a preferred compiler. 'make clean' deletes the Makefile that configure created and "cleans up" the source tree. Once cleaned, you should be able to start over at the configure step and begin again.

'make clean' also removes and .o binary objects or any other compiled parts of the code. If make crashes, sometimes you'll just need to clean the tree to get rid of the problem.

3.4.3 Make Distclean

The 'make distclean' rule (not always possible, it depends on the package) resets the source tree to the exact setup you downloaded and extracted when you first started the compile. There are absolutely no Makefiles, binary objects or anything that didn't come with the source tree when you downloaded it.

3.5 Tips and Tricks

First, make sure you set up your environment as you want before beginning the configure script. Make sure you set up a $LD_RUN_PATH and make sure that the compiler you want to use is in your $PATH.

Also, note that if you're on a 64 bit system, configure is going to produce a 64 bit binary unless you use a 32 bit only compiler. Keep in mind that not every system in the UVa Astronomy network is 64 bit capable, so if anyone else uses your software or if you move to another system, your binary may not work.

Below is a sample setup I use when compiling software using the automake system (note that I use the bash shell, adapt as you need if you use tcsh):

```
[hbp4c@Realos ~]$ export PATH=/astro/intel/latest/bin
[hbp4c@Realos ~]$ export LD_LIBRARY_PATH=/astro/intel/latest/lib
[hbp4c@Realos ~]$ export LD_RUN_PATH=${LD_LIBRARY_PATH}
[hbp4c@Realos ~]$ ./configure --help
...
[hbp4c@Realos ~]$ CC=icc CFLAGS="-O2" ./configure --prefix=~/
...
[hbp4c@Realos ~]$ make -j2
...
[hbp4c@Realos ~]$ make install
```

Chapter 4

The GNU Compilers

The GNU compilers are a set of free C, C++ and Fortran compilers available from the GNU software foundation. http://gcc.gnu.org/.

You certainly want to use the GNU online documentation for GCC or the gcc man pages for more details on the optimizations you can give the compiler when building your code. For quick reference, I've included the options that I've used as a sysadmin often to compile code.

Don't forget to set up your Shell[2] before compiling.

4.1 Optimizing gcc, g++, and gfortran

gcc is used to compile C software, g++ compiles C++ software, and gfortran compiles Fortran 77, 95 and parts of the later Fortran standards. The basic format of any of these compilers is

```
$ gcc input_filename -o output_filename
```

The GNU compilers can be optimized with a lot of options. The -O ["oh"] flags are simple, but effective ways to optimize your produced binary.

-O

-O1 tries to reduce code size and execution time.

-O2 performs nearly all supported optimizations that do not involve a space-speed tradeoff.

-O3 turns on optimizations that unroll loops and function inlining.

-O0 disables optimization - necessary for debugging purposes. If -O is omitted, this is default.

-Os optimizes the output for size.

-mtune options optimize the binary further for a specific processor. Tuning a binary for a later-generation processor breaks compatibility with older processors (so binaries for Pentium 4s will not run on pentium 3 or lower chips).

-mtune=generic optimizes code for any intel processor.

-mtune=native optimizes code for the processor you compile on.

-mtune=pentium4 optimizes code for the Intel Pentium4 with MMX, SSE and SSE2 instruction set support.

-mtune=prescott optimizes code for the Intel Pentium4 CPU with MMX, SSE, SSE2 and SSE3 instruction set support.

-mtune=core2 optimizes code for the Intel Core2 CPU with 64-bit extensions, MMX, SSE, SSE2, SSE3 and SSSE3 instruction set support.

-mtune=athlon optimizes code for the AMD Athlon CPU with MMX, 3dNOW!, enhanced 3dNOW! and SSE prefetch instructions support.

-mtune=opteron optimizes code for the AMD K8 core based CPUs with x86-64 instruction, MMX, SSE, SSE2, 3dNOW!, and enhanced 3dNOW! extensions support.

-mtune=barcelona optimizes code for the AMD Family 10h core based CPUs with x86-64 instruction, MMX, SSE, SSE2, SSE3, SSE4A, 3dNOW!, enhanced 3dNOW!, and ABM extension set support.

Also, it's possible to control what format (32bit or 64bit) the output binaries will be with -m32 or -m64.

-m32 produces 32bit binaries

-m64 produces 64bit binaries

Finally, if you need to debug your code with an external debugger (such as gdb) you'll need to enable the debug flag, -g:

```
$ gcc -g input_filename -o output_filename
```

Enabling -g while optimizing the binary using any of the above methods can cause problems. When debugging, it's best to leave all optimizations at their defaults.

Chapter 5

The Intel Compilers

Chapter 6

Perl

6.1 Modules

6.2 CPAN

Chapter 7

Python Modules

7.1 Python 2.4

Python 2.4 is included with Redhat Enterprise Linux 5.x systems.

7.1.1 32-bit machines

To compile a Python module on a 32-bit machine, you'll need to log into a 32 bit system (easiest way) or make sure that you compile using the 32-bit version of python included on 64 bit systems (much harder!!).

Any non-system python modules are kept in /astro/lib/python2.4/site-packages/. To make the python compiler install to this directory, use the –prefix option during setup:

```
[root@Realos ~]# python setup.py install --prefix=/astro
```

If all goes well, the new python module will be installed to /astro/lib/python2.4/site-packages/.

7.1.2 64-bit machines

To compile a Python module as a 64-bit program, you must be logged into a 64-bit system.

Any non-system python modules are kept in /astro64/lib64/python2.4/site-packages/. To make the python computer install to this directory, use the –prefix option during setup:

```
[root@Saguaro ~]# python setup.py install --prefix=/astro64
```

If all goes well, the new python module will be installed in /astro64/lib64/python2.4/site-packages/. Please be sure to build both a 32 bit and a 64 bit version of all python modules for maximum flexibility.

7.2 Python 2.5

Python 2.5 is not included by default with Redhat EL 5.x systems. I've created an astronomy-specific version of Python 2.5 which can be run from the shell via the command "python25" or "python2.5".

7.2.1 32-bit machines

To build a 32-bit Python 2.5 module, you should run these steps on a 32-bit system:

```
[root@Realos ~]# python2.5 setup.py install --prefix=/astro
```

If all goes well, the module will be installed in /astro/lib/python2.5/site-packages/.

7.2.2 64-bit machines

To build a 64-bit Python 2.5 module, you should run these steps on a 64-bit system:

```
[root@Saguaro ~]# python2.5 setup.py install --prefix=/astro64
```

If all goes well, the module will be installed in /astro64/lib64/python2.5/site-packages/.

7.3 Admin Notes on /astro and Python

As you know, most non-system software gets installed in /astro by default. Python can include /astro in it's search path if you define a .pth file in /usr/lib*/python2.*/site-packages/. In this text file, simply put the name of the director you want python to search for modules. Multiple paths can be appended to the same file if necessary.

For example:

```
[root@realos ~]# echo "/astro/lib/python2.4/site-packages/" >
     /usr/lib/python2.4/site-packages/astro.pth
[root@realos ~]# echo "/astro/lib/python2.4/site-packages/PIL" >>
     /usr/lib/python2.4/site-packages/astro.pth
```

From now on this system will know to check /astro for more python modules, regardless if the user has set their PYTHONPATH variable or not.