Compiling Programs

Follow us on Twitter for important news and updates:

@ACCREVandy
Outline

• Compiling process
• Linking libraries
• Common compiling options
• Automating the process
Program compilation

- Programmers usually write code in high-level programming languages (e.g. C, C++, Java, Fortran, Python, Perl, Ruby, etc.)
- Computers can only execute binary instructions
- Source code (the high-level program) needs to be converted to machine code (an executable binary)
- We call this process program **compilation**
- It actually includes multiple steps (collectively called “building”), namely:
  - compilation
  - linking
Compilation and linking

- **Compilation** (with a compiler) refers to the processing of source code files (.c, .cc, or .cpp) and the creation of an 'object' file (.o, .obj) – does not create anything you can run.

- Compiler produces the machine language instructions that correspond to the source code file that was compiled.

- **Linking** (with a linker) refers to the creation of a single executable file from multiple object files.

[Diagram of the C Compilation Model]

- Source code
- Preprocessor
- Compiler
- Assembly Code
- Assembler
- Object Code
- Link Editor
- Libraries
- Executable Code

C Compilation Model
Why separate steps?

• By keeping the functions separate, the complexity of the program is reduced
• Allows the creation of large programs without having to redo the compilation step every time a file is changed—using "conditional compilation", it is necessary to compile only those source files that have changed
• Makes it simple to implement libraries of pre-compiled code: just create object files and link them just like any other object file
• Easy to track bugs:
  – Compiler errors are usually syntactic in nature -- a missing semicolon, an extra parenthesis
  – Linking errors usually have to do with missing or multiple definitions
  – If you get an error that a function or variable is defined multiple times from the linker, that's a good indication that the error is that two of your source code files have the same function or variable
• Separate compilation model
Setting up the compiling environment

• Usage:
  – `pkginfo` with no options prints list of installed packages
  – `pkginfo -p package -i` prints detailed info on package
  – `setpkg` with no options prints help to screen (no man page)
  – `setpkg -a package_list` adds environment variables
  – `setpkg -e package_list` erases environment variables
  – `setpkg -r package_list` replaces all with packages listed
Today’s Examples

• Visit ACCRE’s Github repository:
  • https://github.com/accre/Compiler.git

• All examples can be downloaded by doing the following from a cluster gateway:
  • setpkgs –a git
  • git clone https://github.com/accre/Compiler.git
  • This will create a subdirectory in your current directory called Compiler
  • If you have Matlab in your environment you will need to remove it temporarily with “setpkgs –e matlab”
Compile a simple program in C

• hello.c

/*
 * File: hello.c
 * __________
 * This simple C program prints out the text "Hello world!".
 */

#include<stdio.h>
int main(void) {
    printf("Hello world!\n");
}

$ gcc hello.c
$ /a.out
Hello world!

$ gcc -o hello hello.c
$ ./hello
Hello world!
Compiling a simple C program

- Compiling and assembling
  - translates the C code into assembly language, which is a machine level code that contains instructions that manipulate the memory and processor directly

```
$ gcc -S hello.c
```

```
.file  "hello.c"
.section  .rodata
.LC0:
    .string "Hello world!\n"
.text
.globl main
    .type  main,@function
main:
    pushl  %ebp
    movl   %esp, %ebp
    subl   $8, %esp
    andl   $-16, %esp
    movl   $0, %eax
    subl   %eax, %esp
    subl   $12, %esp
    pushl  $.LC0
    call   printf
    addl   $16, %esp
    leave
    ret
```

```
hello.s
```

```
.Lfe1:
.size   main,.Lfe1-main
.section  .note.GNU-stack,"",@progbits
.ident  "GCC: (GNU) 3.2.3 20030502
        (Red Hat Linux 3.2.3-59)"
```
Compiling a simple C program

• Usually you don’t need .s instead you need .o (object file)
  
  $ gcc -c hello.c  \; (create\; hello.o) 

• Linking
  
  – use the linker to process your main function and any possible input arguments you might use, and link your program with other programs that contain functions that your program uses
  
  $ gcc -o hello hello.c  \; (create\; hello.o\; in\; tmp\; directory\; and\; does\; the\; linking)$
Compiling a simple C program

• Pre-processor directives
  – Selectively remove/add blocks of code before compiling
  – Directives begin with # character
  – Examples:

```c
#define PI 3.14159265359

#ifdef MY_ARG

#ifndef _FILE_NAME_H_
#define _FILE_NAME_H_
#define _FILE_NAME_H_
#endif
#endif
```
Compilers

• **C/C++**
  – GCC (GNU Compiler Collection)
  – Intel C/C++ compiler (known as icc or icl)

• **Fortran**
  – g77 from GCC
  – gfortran from GCC (for Fortran 95)
  – Intel Fortran compiler
  – Absoft

• **Java**
  – javac (from Sun)
  – GCJ (from GCC)
• Types of errors:
  – compiler warnings
  – compiler error
  – linker error
• Compiler warnings
  – an indication that something might go wrong at runtime
  – typical errors, e.g. using = instead of ==
  – variables not initialized
• Compiler errors
  – cannot complete the compilation process
  – restrict to single source file and “syntax error”
  – you’ve done something the compiler cannot understand
  – includes line number with the output
• Linker errors
  – nothing to do with “syntax error”
Dealing with errors

• Compiler errors
  – start from the top error message because later errors may caused by the earlier errors
  – error messages:
    ```
    foo.cc:7: error: semicolon missing after struct declaration
    ```
  – look earlier in the program
  – think about how the compiler is trying to interpret the file

• Linker errors
  – provide your linker with the correct path to the library that has the actual implementation to avoid "undefined function" error messages
  – include all of the necessary object files that you created to define the functions you need
  – more than one definition for a class, function, or variable
Examples

• Serial pi
• Parallel pi
Outline

• Compiling process
• Linking libraries
• Common compiling options
• Automating the process
Linking libraries

• Static vs. dynamic

• Static linking (lib**.a)
  – the linker copies all library routines used in the program into the executable image
  – require more disk space and memory than dynamic linking
  – more portable (does not require the presence of the library on the system where run)

• Dynamic linking (lib**.so)
  – placing the name of a sharable library in the executable image
  – actual linking with the library routines does not occur until the image is run, when both executable and library are placed in memory
  – multiple programs can share a single copy of the library
Linking libraries

• By default, gcc compiles programs using \texttt{.so} (if both \texttt{.so} and \texttt{.a} exist)

• Default search path for header files:
  
  \begin{itemize}
    \item [/usr/include]
    \item [/usr/local/include]
  \end{itemize}

for libraries:

\begin{itemize}
  \item [/usr/lib]
  \item [/usr/local/lib]
\end{itemize}

• Specify additional path using \texttt{-I} and \texttt{–L}

\begin{verbatim}
gcc -c myexec -I/path/to/myheader/ -L/path/to/mylib -lmylib
\end{verbatim}

\texttt{libmylib.so}
Linking libraries

• The search path can also be controlled by environmental variables
  • `C_INCLUDE_PATH, CPLUS_INCLUDE_PATH`
  • `LIBRARY_PATH`
  • `LD_LIBRARY_PATH`  (for loading shared libraries at runtime)
Example

```c
#include <stdio.h>
#include <gsl/gsl_sf_bessel.h>

int main(void)
{
    double x = 5.0;
    double y = gsl_sf_bessel_J0(x);
    printf("J0(%.18e) = %.18e\n", x, y);
    return 0;
}
```

Execute these commands before compiling and running:

```
setpkgs -a gcc_compiler
setpkgs -a gsl_gcc
```

This will ensure that your PATH and LD_LIBRARY_PATH are pointing to the correct versions of GCC and GSL.

```
gcc -L/usr/local/gsl/latest/x86_64/gcc46/nonet/lib
    -I/usr/local/gsl/latest/x86_64/gcc46/nonet/include
    -lgsl -lgslcblas bessel.c -Wall -O3 -o calc_bessel
```
Outline

• Compiling process
• Linking libraries
• **Common compiling options**
• Automating the process
Warning Messages

• Output during compile, usually warnings about run-time errors
• Instruct the compiler to give more warnings: use the
  -W{exp} flag
• -Wall
Compiling for debugging and profiling

• In order to debug a program effectively, you need to generate debugging information when you compile it.
• The debugging information is stored in the object file: the data type of each variable or function and the correspondence between source line numbers and addresses in the executable code.
• Use `-g` or `-ggdb` for use with GDB.
• Profiling allows you to learn where your program spent its time and which functions called which other functions while it was executing.
• Use `-pg` for use with `gprof`.
Debugging

• Debugger: gdb, DDT
• To use DDT on cluster:
  • setpkgs –a ddt
• Log in to the cluster gateway with X11 forwarding enabled
Compiling with optimization

- Want the program to run faster or take less space
- Turn on the $-O$ flag
- Compilation takes longer
  - Compiler applies various optimization algorithms
- Optimization is designed to be conservative
  - Ensures code will function the same as without optimization
- Different levels of optimization
  - Add number arguments to $-Ox$: $-O2, -O3, -O4$
  - The higher the number the greater the optimization and slower the compiler
• Compiling process
• Linking libraries
• Common compiling options
• Automating the process
Automating the build process

• **make** utility
  – Provides a way for separate compilation
  – Describe the dependencies among the project files
  – Default file to look for is *makefile* or *Makefile*
What’s in a Makefile

• Basic element is the rule.

```makefile
target : dependencies
  commands
#shell commands
```

Example:

```makefile
executable : project1.o project2.o
gcc -o executable project1.o project2.o

project1.o : project1.c common.h
gcc -c project1.c

project2.o : project2.c common.h
gcc -c project2.c
```
Dependency Rules

• Define under what conditions a given file (or a type of file) needs to be re-compiled
• For example:

```
main.o: main.c
<TAB> gcc -g -Wall -c main.c
```

• Here the target `main.o` must be recompiled whenever `main.c` is revised, recompiled with “`gcc -g -Wall -c main.c`”