Compiling Programs
Outline

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

• Programmers usually writes 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.
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
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!
");
}

$ 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

```assembly
.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
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_
  
  #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)
Errors

• 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