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 etc.)
• Machine can only execute binary instruction
• We need to transform the source code to machine code.
• We call this process program *compilation*.
• It actually includes two steps:
  – 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
• 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

- Preprocessor
  - remove comments from the source code and interpret preprocessor directives which are given by the statements that begin with #

- 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

```c
$ gcc -S hello.c

hello.s
```

```assembly
.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
addl $16, %esp
leave
ret
```

```assembly
.ident  "GCC: (GNU) 3.2.3 20030502 (Red Hat Linux 3.2.3-59)"
```
• 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)
Compilers

- C/C++
  - GCC (Gnu Compiler Collection)
  - Intel C/C++ compiler (known as icc or icl)
  - IBM XL C/C++ (unix only)
  - Borland C/C++ (windows only)

- Fortran
  - g77 from GCC
  - gfortran from GCC (for Fortran 95)
  - Intel Fortran compiler
  - Absoft

- Java
  - javac (from Sun)
  - GCJ (from GCC)
Setting up the compiling environment

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

- Serial pi
- Parallel pi
• 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
• convert.c
• Compiling process
• Linking libraries
• Common compiling options
• Automating the process
• 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
• By default, `gcc` compiles programs using `.so` (if both `.so` and `.a` exist)

• Default search path for header files:
  
  `/usr/include`
  `/usr/local/include`

for libraries:

  `/usr/lib`
  `/usr/local/lib`

• Specify additional path using `-l` and `–L`

  ```bash
  gcc -c myexec -I/path/to/myheader/ -L/path/to/mylib -lmylib
  ```

  `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`
• Compiling process
• Linking libraries
• **Common compiling options**
• Automating the process
• Output during compile, usually warnings about run-time errors
• Instruct the compiler to give more warnings: use the
  \texttt{-W\{exp\}} flag
• \texttt{-Wall}
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`
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 \( -0x \) : \(-02, -03, -04\)
  ▪ The higher the number the greater the optimization and slower the compiler
• Compiling process
• Linking libraries
• Common compiling options
• Automating the process
• **make** utility
  – Provide a way for separate compilation.
  – Describe the dependencies among the project files
  – Default file to look for is *makefile* or *Makefile*.
• Basic element is the rule.

```
target : dependencies
    TAB commands    #shell commands
```

Example:

```
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
• How to compile it. 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`”