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 `.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 `-I` and `–L`

  ```
  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` (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 
  \texttt{-W\{exp\}} flag
• \texttt{-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
Outline

• 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.

```
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
• For example:

```bash
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`”