

II.8. Working with Large Programs

Command line switches

Most UNIX commands allow parameters to be set using “switches”. These come on the command line after the command name. They are normally a single letter preceded by a dash (‘-’) and followed by the parameter if required. For example, the ‘-o’ switch is used to specify the output file name for the CC compile command. E.g.:

```
>CC -o cat.exe cat.cc
```

will compile cat.cc into an executable file named “cat.exe” instead of “a.out”.

Header files

Normally class declarations and function declarations are put in separate “header” files so that they can be included by other source files. E.g.:

```
#include <iostream>
```

When compiling, use the compiler switch -I<directory> to add a directory to the search path for include files. E.g.,

```
>CC -I/home/mefac/bettig/MEEM5408_Assignments cat.cc
```

If you do not do this, then only the default include path is searched. The default is: /usr/include and /opt/SUNWspro/WS6U2/include/CC/Cstd on Sun workstations.

If the header file is in the same directory as the source file, you can use quotation marks:

```
#include "my_header.h"
```

Header files usually have a “.h” or “.hpp” extension. Some newer header files do not have any extension.

Create a new file called “cat.h” to contain your class declarations. Add an include directive to read in the file at the beginning of your source file.

To avoid circular loops in which one header file includes another header file which in turn includes the first file, use a compiler variable and “if” directive to contain all the declarations in the header file. E.g.:

```
// comments section first, then:  
  
#if !defined(__cat_h)  
#define __cat_h
```

```
// rest of file here ...  
#endif
```

Object files

Normally source files are first compiled into object files (.o), which are then linked into the executable.

Put your main function into a file called “main.cc” and your Cat functions into a file called “cat.cc”.

To compile an object file, use the -c compiler switch. E.g.:

```
>CC -c cat.cc -o cat.o  
>CC -c main.cc -o main.o
```

To link the executable, use:

```
>CC main.o cat.o
```

static and shared (dynamic) libraries

Libraries are collections of functions. A static library (.a extension) simply contains the functions, which are linked to the executable when it is compiled. The syntax for archiving object files into a static library is:

```
ar <flags> <library name> <object file names>
```

E.g., to compile an archive use:

```
>ar cru libcat.a cat.o
```

A dynamic library (.so, .sl or .dll extension) is linked to the executable at run time, thus making the executable smaller and faster to compile. Compiling shared libraries is similar to compiling executables, but use switches -pic (position-independent code) and -G, when compiling all source code and linking the shared library:

```
>CC -o libcat.so -G -pic cat.o
```

Usually each library embodies one module of a software program. Libraries are included the same as object files when compiling the executable. E.g.,

```
>CC main.o libcat.a  
or  
>CC main.o libcat.sl
```

When linking, use -L<directory> to add a directory to the search path for library files.

E.g.:

```
>CC -L/home/megrad/ME5990 main.o libcat.a
```

If the library name starts with lib and ends with .a or .so, you can use the -l compiler switch to abbreviate the above line:

```
>CC -L/home/megrad/ME5990 main.o -lcat
```

Before running the executable, make sure that all shared libraries are on the search path, given by the UNIX variable: LD_LIBRARY_PATH.

Makefiles

Makefiles are files that automate the execution of compiling and linking commands by associating UNIX commands with file dependency information. If a file has not been compiled yet, or has changed since it was last compiled, the output file is considered to be “out-of-date”. The make command will check for out-of-date files and execute UNIX commands according to rules in the makefile given by the following syntax:

```
<target> : <dependencies>
    <command1>
    <command2>
    ...
```

where target is an output file or variable name, and dependencies is a list of files or variable names upon which it depends. If the target is out of date with respect to its dependencies then the commands will be executed. Each command line must start with a tab. E.g.:

```
cat.exe : cat.o main.o
    CC -o cat.exe cat.o main.o
```

“Implicit” rules can also be used for common patterns. E.g., to compile every .cc source file into an .o object file, use:

```
%o: %cc
    CC $(<F) -c -o $(@F)
```

where \$(<F) and \$(@F) refer to the dependency and target file names.

Additional makefile functionalities are:

- use '#' to start a comment line
- use '=' to assign a text string to a variable.

E.g.: `CCFLAGS = -G -pic -features=rtti`

- use `$()` to evaluate a variable. E.g.: `CC $(CCFLAGS) cat.cc`
- use '@' to suppress printing a command when it is executed. E.g.: `@CC cat.cc`
- use 'echo' to print text to the terminal. E.g.: `echo "Compiling main() "`

Commands may continue to the next line if the line is terminated with a '\ ' and the next line starts with a tab.

The name of a makefile must be "Makefile" or "makefile". To execute a makefile that is in the current directory, use the make command:

```
>make <target>
```

If <target> is not specified, the first one will be used.

Example Makefile

```
CCFLAGS = -features=rtti

cat.exe : cat.o main.o Makefile
        CC $(CCFLAGS) -o cat.exe cat.o main.o

cat.o : cat.cc cat.h
        CC $(CCFLAGS) -c -o cat.o cat.cc

main.o : main.cc cat.h
        CC $(CCFLAGS) -c -o main.o main.cc
```