System Programming

Software Development: g++ and make
Software Development Process

- Creation of source files (.c, .h, .cpp)
- Compilation (e.g. *.c → *.o) and linking
- Running and testing programs
Development Tools

- Creation of source files (*.c, *.h, *.cpp)
  - Text editors
    - vi, emacs
  - Revision (version) control systems
    - rcs, cvs

- Compilation (*.o) and linking
  - Compilers
    - gcc, g++
  - Automatic building tools
    - make

- Running and testing (xdb, gdb)
Compilation Process

- **Preprocessor**
  - `#include`, `#define`, `#ifdef`

- **Compiler**
  - Translates code into assembly language

- **Optimizer**
  - Optional: Language-neutral code optimization

- **Assembler**
  - Translates assembly code into machine language code kept in object files

- **Linker-Loader**
  - Links object files and libraries to form one executable file
Basic g++ Examples

- g++ hello.cpp
  - compile hello.cpp
  - produce executable a.out
- g++ -o hello hello.cpp
  - compile hello.cpp
  - produce executable hello
- g++ -o hello hello.cpp util.cpp
  - compile hello.cpp and util.cpp
  - produce executable hello
Separate Compilation

- From any source file you can produce an object file to be linked in later to make an executable

```
g++ -c hello.cpp

g++ -c util.cpp

g++ -o hello hello.o util.o
```
g++ Options

- **-c**
  - compile source files, but do not link
  - output is an object file corresponding to source file

- **-o <file>**
  - puts output in file called `<file>`

- **-g**
  - include debugging symbols in the output
  - to be used later by debugging program (gdb)

- **-Wall**
  - display all warnings – program may still compile
g++ Options

- **-D<macro>**
  - defines macro with the string ‘1’
- **-l<name>**
  - include library called **lib<name>.a**
- **-I<path>**
  - look for include files in the directory provided
- **-L<path>**
  - look for libraries in the directory provided

There are default directories in which g++ looks for include files and libraries
Defines in g++

- Often programs contain conditional parts based on defines:
  ```c
  #ifdef DEBUG
  printf(“value of var is %d”, var);
  #endif
  ```

- You can set preprocessor defines on the command line
  ```bash
  g++ -DDEBUG -o prog prog.c
  ```
Separate Compilation Scenario

- You build a personal database and an interface program called `mydb` that has a user interface and database backend
  - **Source files:** `mydb.cpp`, `user.cpp`, `database.cpp`
  - **Common header file:** `mydb.h`
  - **Executable:** `mydb`
Build Dependencies

mydb.h

mydb.cpp

user.cpp

database.cpp

$g++ -c mydb.cpp$

mydb.o

$g++ -c user.cpp$

user.o

$g++ -c database.cpp$

database.o

$g++ -o mydb$

mydb

Chapter Ten  g++ and make  11
Using **make** in Compile

- With medium to large software projects containing many files, it is difficult to:
  - Type commands to compile all the files correctly each time
  - Keep track of which files have been changed
  - Keep track of dependencies among files
- **make** automates this process
Basic Operation of `make`

- Reads a file called `Makefile` that contains rules for building a program
  - if program is dependent on another file, then that file is built
  - all dependencies are built, working backward through the chain of dependencies
  - programs are only built if they are older than the files they depend upon
Basic Makefile Example

# Makefile for mydb
mydb: mydb.o user.o database.o
    g++ -o mydb mydb.o user.o database.o
mydb.o : mydb.cpp mydb.h
    g++ -c mydb.cpp
user.o : user.cpp mydb.h
    g++ -c user.cpp
database.o : database.cpp mydb.h
    g++ -c database.cpp
Build Dependencies

```
g++ -c mydb.cpp
  mydb.o

mydb.h

user.cpp
user.o

database.cpp
database.o

g++ -c mydb.cpp
  mydb.o

mydb.h

user.cpp
user.o

database.cpp
database.o

g++ -o mydb
  mydb
```
Parts of a Makefile

- **Dependency Lines**
  - contain target names and dependencies (optional)
  - dependencies
    - files
    - targets

- **Commands**
  - below dependency line
  - **always** begin with a tab
  - commands to satisfy the dependency
Parts of a Makefile (cont)

**target**

```
mydb:
```

**dependencies**

```
mydb.o  user.o  database.o
```

```
g++  -o mydb  mydb.o  user.o  database.o
```

```
mydb.o:  mydb.cpp
```

```
g++  -c  mydb.cpp
```

```
tab  command
```
Macros and Special Variables

- Use macros to represent text in Makefile
  - saves typing
  - allows easy modification of Makefile
- Assignment
  - MACRONAME = macro value
- Usage: ${MACRONAME}

- Special variables are used in commands
  - $@ represents the target
  - $? represents the dependencies
Simplifying the Example

```
OBJS = mydb.o user.o database.o
CC = /usr/bin/g++

mydb: ${OBJS}
   ${CC} -o $@ $? 
mydb.o: mydb.cpp mydb.h
   ${CC} -c $? 
user.o: user.cpp mydb.h
   ${CC} -c $? 
database.o: database.cpp mydb.h
   ${CC} -c $? 
```
Invoking `make`

- Be sure that the description file
  - is called `makefile` or `Makefile`
  - is in the directory where the source files are
- `make`
  - builds the first target in the file
- `make target(s)`
  - builds target(s)
- Other options
  - `-n`: don’t run the commands, just list them
  - `-f <file>`: `use <file>` instead of `[Mm]akefile`
Other Makefile Notes

- Comments begin with a ‘#’
- Can be placed at the beginning of a line or after a non-comment line
- Lines that are too long can be continued on the next line by placing a ‘\ ’ at the end of the first line
Suffix Rules

- Still tedious to specifically tell `make` how to build each `.o` file from source file
- Suffix rules can be used to generalize such situations
- A default suffix rule turns source files into `.o` files by running the command:
  ```
  ${CC} ${CFLAGS} -c $<
  ``
  - `$<` refers to the prerequisite (file.cpp)
Simplest Makefile Example

OBJS = mydb.cpp user.cpp database.cpp
CC = /usr/bin/g++

mydb: ${OBJS}
    ${CC} -o $@ $?
Other Useful Makefile Tips

- Include a way to remove intermediate files
  ```
  clean:
    rm -f mydb
    rm -f *.o
  ```
- Include a target to build multiple programs
  ```
  all:mydb mycalendar myhomework
  ```