

# g++ and make

Dan Wilson

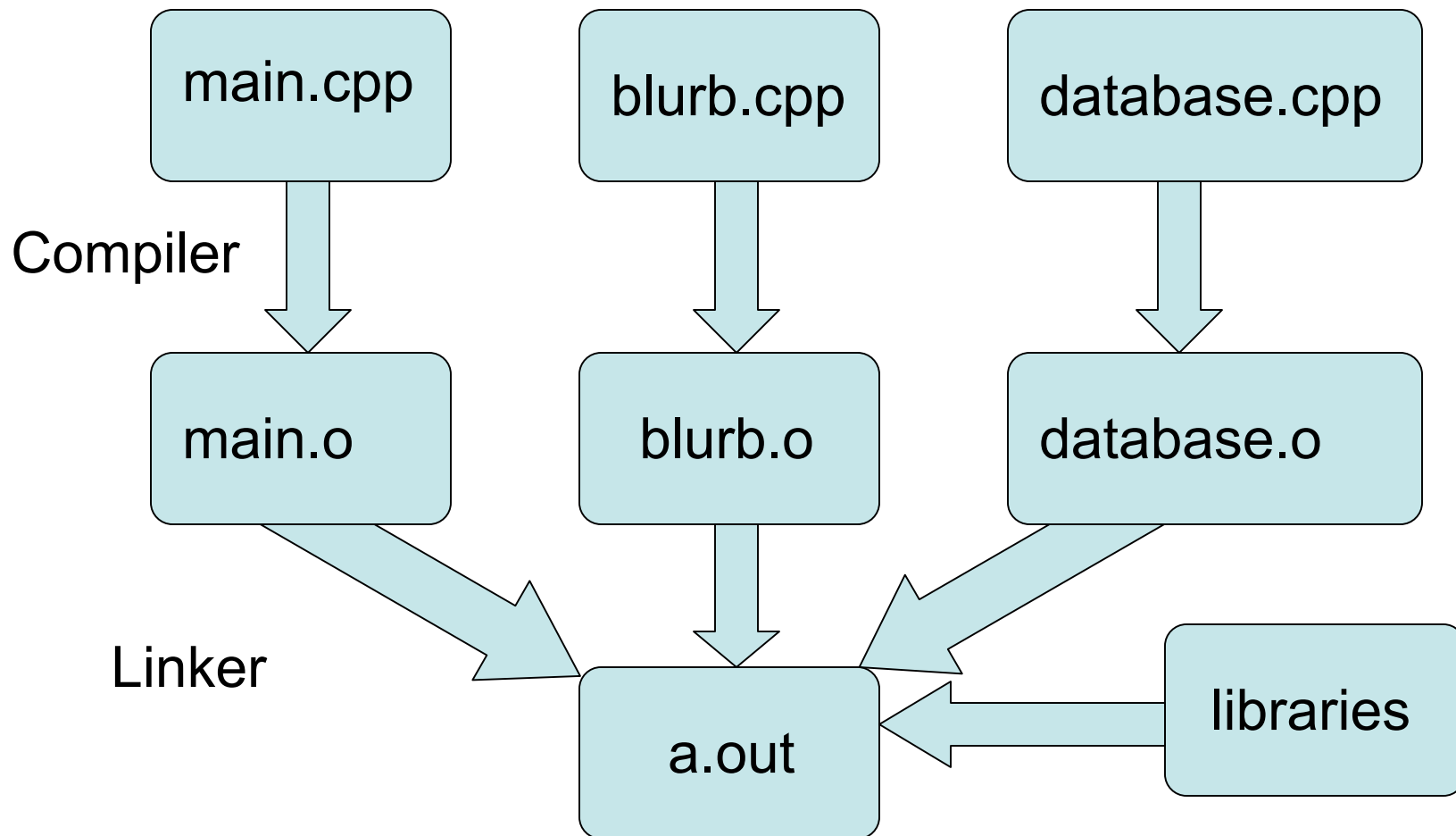
CS193

02/01/06

# The g++ Compiler

- What happens when you call g++ to build your program?
- Phase 1, Compilation: .cpp files are compiled into .o object modules
- Phase 2, Linking: .o modules are linked together to create a single executable.

# The Process



# How g++ Does All This

- Calling: “g++ <option flags> <file list>”
- When g++ is given .cpp files, it performs both compilation and linking. If given .o files, it performs linking only.
- Assignment 1: “g++ \*.cpp”
- Quick and easy, but not ideal. Who wants a program named “a.out?”

# Options For g++ Compilation

- `"-c"` : Compiles .cpp file arguments to .o but does not link (we'll need this for "make" later).
- `"-g"` : Generates debugging information that is used by gdb-based debuggers
- `"-I<dir>"` : Adds the directory <dir> to the list of directories searched for include files.

# Compilation Options, cont.

- “-Wall”: Directs g++ to give warnings about code that’s probably in error.
- For example, it will catch and report:

```
int number = GetRandomInteger()  
If (number == 6) std::cout <<  
    “Wow, number always seems to  
    equal 6!”;
```

# Options for g++ Linking

- “-o” : Specifies the name of the program to be linked together. Name textr “textr!”
- “-L<dir>” : Adds the directory <dir> to the list of directories searched for library files (string, STL Vector, examples of libraries.)
- “-l<libName>” : Makes compiler search the library <lib name> for unresolved names when linking.

# Sample g++ Commands: What Do They Do?

1) `g++ -o textr *.cpp`

2) `g++ -Wall -g -c main.cpp`

3) `g++ -o myProg`

`-L/usr/class/cs193/danLib -lDansFns  
*.o`

4) `g++ -I/usr/class/cs193/inc  
main.cpp src1.cpp src2.cpp`

## Now Onto “make”

- A GNU utility that determines which pieces of a large program need to be compiled or recompiled, and issues a commands to compile and link them in an automated fashion.
- Saves you from tedious, huge g++ commands! Just type in “make” in the directory containing the Makefile.

# The Makefile: Input for make

- Create a file called “Makefile” that contains a series of properly formatted commands that make will take as input.
- The file can be huge and complex, but when it’s done and is working, you just have to type in “make” and the utility uses your makefile to do everything.
- “make” command searches for “Makefile.”

# A Sample Makefile: “Huh?”

```
#Makefile for “textr” C++ application
#Created by Dan Wilson 1/29/06

PROG = textr
CC = g++
CPPFLAGS = -g -Wall -I/usr/class/cs193d/textrInc
LDFLAGS = -L/usr/class/cs193/lib -lCoolFns
OBJS = main.o blurb.o database.o

$(PROG) : $(OBJS)
    $(CC) $(LDFLAGS) -o $(PROG) $(OBJS)
main.o :
    $(CC) $(CPPFLAGS) -c main.cpp
blurb.o : blurb.h
    $(CC) $(CPPFLAGS) -c blurb.cpp
database.o : database.h
    $(CC) $(CPPFLAGS) -c database.cpp
clean:
    rm -f core $(PROG) $(OBJS)
```

# Makefile Lingo

- “target”: Usually the name of an executable or object file that is generated by g++, but it can also be the name of an action to carry out.
- “prerequisites”: A list of files needed to create the target. If one of these files have changed, then the make utility knows that it has to build the target file again. Also called “dependencies.”
- “command”: An action that make carries out, usually a g++ compilation or linking command. In make, commands are run and generate output just as if they were entered one by one into the UNIX prompt.

# Makefiles: Square One

- A simple Makefile consists of a series of “rules,” each of the form :

target ... : prerequisites ...

command

command

command

...

- The rule explains how and when to make or remake a target file.
- A simple “make” call executes first rule by default. For other rules type “make <target>”
- Make requires a <TAB> character before every command (silly, but causes errors).

# The “textr” App Makefile, Take 1:

```
textr : main.o blurb.o database.o
        g++ -o textr main.o blurb.o database.o
main.o : main.cpp
        g++ -c main.cpp
blurb.o : blurb.cpp blurb.h
        g++ -c blurb.cpp
database.o : database.cpp database.h
        g++ -c database.cpp
clean:
        rm -f core textr main.o blurb.o \
        database.o
```

# “clean” Target: Gotta Have One

- An important target that represents an action rather than a g++ operation.
- Has no dependencies, runs a command to remove all the compilation products from the directory, “cleaning” things up.
- Useful for porting code, getting rid of corrupted files, etc.
- Normally also removes the “core” file if it’s present from a past program meltdown.
- Call by typing “make clean” into prompt.

# Next Step: Add Variables

```
OBJS = main.o blurb.o database.o
```

```
textr : $(OBJS)
```

```
    g++ -o textr $(OBJS)
```

```
main.o : main.cpp
```

```
    g++ -c main.cpp
```

```
blurb.o : blurb.cpp blurb.h
```

```
    g++ -c blurb.cpp
```

```
database.o : database.cpp database.h
```

```
    g++ -c database.cpp
```

```
clean:
```

```
    rm -f core textr $(OBJS)
```

# Next Step: Omit the Obvious!

(make knows .cpp -> .o)

```
OBJS = main.o blurb.o database.o
```

```
textr : $(OBJS)
```

```
        g++ -o textr $(OBJS)
```

```
main.o :
```

```
        g++ -c main.cpp
```

```
blurb.o : blurb.h
```

```
        g++ -c blurb.cpp
```

```
database.o : database.h
```

```
        g++ -c database.cpp
```

```
clean:
```

```
        rm -f core textr $(OBJS)
```

# Adding Compiler Options

- Now assume that we need to specify some include files and libraries for our “textr” program (NOTE: THESE ARE IMAGINARY, DO NOT INCLUDE THEM IN YOUR MAKEFILE!)
- Also, we want to add the useful compile options to every g++ compilation command in the Makefile.
- More variables!

# Adding Compiler Options

```
CPPFLAGS = -g -Wall -I/usr/class/cs193d/include
LDFLAGS = -L/usr/class/cs193/lib -lCoolFns
OBJS = main.o blurb.o database.o
```

```
textr : $(OBJS)
        g++ $(LDFLAGS) -o textr $(OBJS)
main.o :
        g++ $(CPPFLAGS) -c main.cpp
blurb.o : blurb.h
        g++ $(CPPFLAGS) -c blurb.cpp
database.o : database.h
        g++ $(CPPFLAGS) -c database.cpp
clean:
        rm -f core textr $(OBJS)
```

# Adding the Finishing Touches

- Conform to standards by creating variables for the compiler and program name.
- Add comments.

# Our Final Draft

```
#Makefile for "textr" C++ application  
#Created by Dan Wilson 1/29/06
```

```
PROG = textr  
CC = g++  
CPPFLAGS = -g -Wall -I/usr/class/cs193d/include  
LDFLAGS = -L/usr/class/cs193/lib -lCoolFns  
OBJS = main.o blurb.o database.o
```

```
$(PROG) : $(OBJS)  
          $(CC) $(LDFLAGS) -o $(PROG) $(OBJS)  
main.o :  
          $(CC) $(CPPFLAGS) -c main.cpp  
blurb.o : blurb.h  
          $(CC) $(CPPFLAGS) -c blurb.cpp  
database.o : database.h  
            $(CC) $(CPPFLAGS) -c database.cpp  
clean:  
        rm -f core $(PROG) $(OBJS)
```

# Makefiles in Professional C++

- Used in Industry for UNIX/Linux platforms
- Fast to run: Only recompile what you need to recompile upon code change, with a simple command
- Great for code versioning and deployment; just version the source code and the Makefile and quickly deploy on any compatible system (no need to version binaries).