C++ Programming on Linux
Multi-file development

CS 2308
Spring 2019

Jill Seaman

Programs with Multiple Files

- How the code is usually split up
  - Put main in its own file, with helper functions
    - acts like a driver
  - Put each class declaration in a separate *.h file (called a header file)
  - Put the implementation of each class (the member function definitions) in its own *.cpp file
  - Each *.cpp file (even the driver) must #include (directly or indirectly) the **header** file (*.h) of each class that it uses or implements.
  - **NEVER #include *.cpp files!!!**

Time class, separate files

Time.h

```
#include <string>
using namespace std;

// models a 12 hour clock
class Time {

private:
    int hour;
    int minute;
    void addHour();

public:
    void setHour(int);
    void setMinute(int);
    int getHour() const;
    int getMinute() const;

    string display() const;
    void addMinute();
};
```

Driver.cpp

```
//Example using Time class
#include<iostream>
#include "Time.h"
using namespace std;

int main() {
    Time t;
    t.setHour(12);
    t.setMinute(58);
    cout << t.display() <<endl;
    t.addMinute();
    cout << t.display() << endl;
    t.addMinute();
    cout << t.display() << endl;
    return 0;
}
```


Time class, separate files

Time.cpp

```
#include "Time.h"

void Time::setHour(int hr) {
  hour = hr;
}

void Time::setMinute(int min) {
  minute = min;
}

int Time::getHour() const {
  return hour;
}

int Time::getMinute() const {
  return minute;
}
```

```
void Time::addHour() {
  if (hour == 12)
     hour = 1;
  else
     hour++;
}
void Time::addMinute() {
  if (minute == 59) {
     minute = 0;
     addHour();
  } else
     minute++;
}
string Time::display() const {
  string hourStr = to_string(hour);
  string minuteStr = to_string(minute);
  if (minuteStr.length()==1)
     minuteStr = "0" + minuteStr;
  return hourStr + ":" + minuteStr;
}
```

How to compile a multiple file program

- From the command line (files in either order):

```
[...]$g++ Time.cpp Driver.cpp
```

  - The header file should **not** be listed. (it is #included in *.cpp files)
  - one (and only one) file must have the main function

- a.out is (by default) the executable file for the entire program.

```
[...]$ ./a.out
12:58
12:59
1:00
```


Separate Compilation

- If we make a change to Driver.cpp, we have to recompile it
  - but perhaps we would rather not have to recompile Time.cpp as well.
- We can compile one file at a time, and **link** the results together later to make the executable.
- Compiling without linking (use -c option):

```
[...]$g++ -c Time.cpp
[...]$g++ -c Driver.cpp
```

  - -c option produces object files, with a .o extension (Time.o, Driver.o)

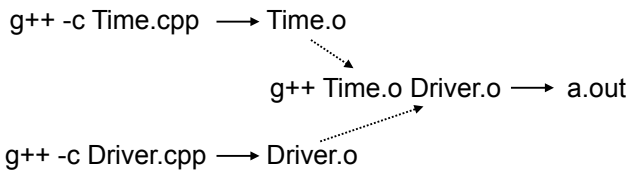
Separate Compilation

- The .o files must be **linked** together to produce the executable file (a.out):

```
[...]$ g++ Time.o Driver.o
[...]$ ./a.out
```

Note there is no -c option used here

- Graphic representation:



Separate Compilation

- Now if we change only Time.cpp, we can recompile just Time.cpp, and link the **new** Time.o file to the **original** Driver.o file:

```
[...]$g++ -c Time.cpp
[...]$g++ Time.o Driver.o
[...]$./a.out
```

Produces new Time.o

Links new Time.o to old Driver.o, making a new a.out

Make

- Make is a utility that manages (separate) compilation of large groups of source files.
- After the first time a project is compiled, make re-compiles **only the changed files** (and the files depending on the changed files).
- These dependencies are defined by rules contained in a makefile.
- The rules are defined and managed by humans (programmers).

Make

- Rule format:

```
target: [prerequisite files]
<tab>[linux command to execute]
```

- target is a filename (or an action/goal name)
- In order to produce the target file, the prerequisite files must exist and be up to date (if not, make finds a rule to produce them).
- An example rule:

```
Time.o: Time.cpp Time.h
   g++ -c Time.cpp
```

If Time.o does not exist, OR if Time.cpp or Time.h is **newer** than Time.o, (re)produce Time.o using this command

Makefile

- The makefile is a text file named “makefile”:

```
#makefile

a.out: Driver.o Time.o
   g++ Driver.o Time.o

Driver.o: Driver.cpp Time.h
   g++ -c Driver.cpp

Time.o: Time.cpp Time.h
   g++ -c Time.cpp
```

You can use nano to create this file

Do **not** copy/paste this to your makefile,

Don’t forget the tabs

Don’t call it makefile.txt

Make

- running make from the linux/unix prompt with no arguments executes first rule in the makefile.
  - This may trigger execution of other rules.

```
[...]$ make
```

- executing the make command followed by a target executes the rule for that target.

```
[...]$ make Time.o
```

Compile class + driver using make

- Make:

```
[...]$ make
g++ -c Driver.cpp
g++ -c Time.cpp
g++ Driver.o Time.o
```

This creates files Driver.o, Time.o, and a.out

- Execute:

```
[...]$ ./a.out
12:58
12:59
1:00
```

- Modify Driver.cpp in nano, make again:

```
[...]$ make
g++ -c Driver.cpp
g++ Driver.o Time.o
```

It knows the timestamp of Driver.cpp is newer than Driver.o, so it fires the rule to make Driver.o again

- Execute again:

```
[...]$ ./a.out
```