C++ for Science and Engineering COSC3000/6000

2018 Spring Semester

Part IX
Separate Compilation and Makefiles

1 Separate Compilation

- C++ allows you to divide a program into parts
  - Each part can be stored in a separate file
  - Each part can be compiled separately
  - A class definition can be stored separately from a program.
    - This allows you to use the class in multiple programs

```
// sample
//
#include <iostream>

class TimeOfDay
{
        ////////////////////////
        /// CLASS DEFINITION ///
        ////////////////////////
};

////////////////////////////////////////////
/// CLASSS MENBER FUNCTION IMPLEMENTATION ///
////////////////////////////////////////////

////////////////////////
/// FRIEND FUNCTIONS ///
////////////////////////

////////////////////////
/// ORDINARY FUNCTIONS ///
////////////////////////

int main(int argc, char **argv)
{
        /// APPLICATION ROUTINE
}
```

See **sample36.cpp**

1.1 ADT Review

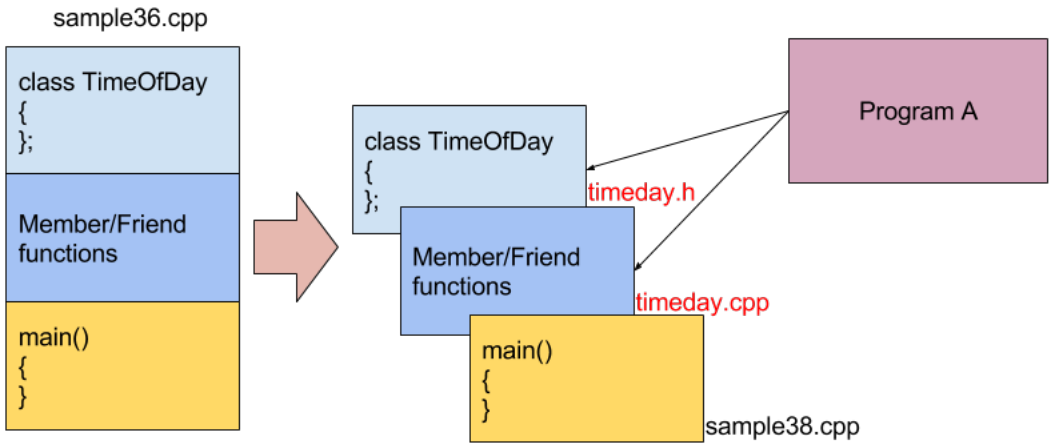
- An ADT is a class defined to separate the interface and the implementation
  - All member variables are private
  - The class definition along with the function and operator declarations are grouped together as the interface of the ADT
  - Group the implementation of the operations together and make them unavailable to the programmer using the ADT

1.1.1 The ADT Interface

- The interface of the ADT includes
  - The class definition
  - The declarations of the basic operations which can be one of the following
    - Public member functions
    - Friend functions
    - Ordinary functions
    - Overloaded operators
  - The function comments

1.1.2 The ADT Implementation

- The **implementation** of the ADT includes
  - The function definitions
    - The public member functions
    - The private member functions
    - Non-member functions
    - Private helper functions
  - Overloaded operator definitions
  - Member variables
  - Other items required by the definitions



1.2 Separate Files

- In C++ the ADT interface and implementation can be stored in separate files
  - **The interface file** stores the ADT interface
  - **The implementation file** stores the ADT implementation
- C++ does not allow splitting the public and private parts of the class definition across files
  - **The entire class definition** is usually in the interface file

2 Case Study : TimeOfDay

We have developed a class "**TimeOfDay**", which will be an abstract data type for time of day .

- The interface file of the "**TimeOfDay**" ADT class contains the class definition
  - The values of the class are:
  - Time of day, such as 9:30, in 24 hour notation
    - **hours** must be 0,1,2,...22,23 : $0 \leq$ **hours** ≤ 23
    - **minutes** must be 0,2,3...58,59 : $0 \leq$ **minutes** ≤ 59
      - If **minutes** < 0, add 60 to **minutes** and subtract 1 from **hours,** repeat those until **minutes** ≥ 0.
      - If **minutes** > 59, subtract 60 from **minutes** and add 1 to **hours** repeat those until **minutes** ≤ 59.

2.1 Naming The Interface File

- The "**TimeOfDay**" ADT interface is stored in a file named **timeday.h**
  - The **.h** suffix means this is a header file
  - Sometimes use **.hpp**
  - Interface files are always header files
- A program using **timeday.h** must include it using an include directive

```
#include "timeday.h"
```

2.1.1 #include " " or < > ?

- To include a predefined header file use < and >

```
#include <iostream>
```

  - < and > tells the compiler to look where the system stores predefined header files
- To include a header file you wrote, use " and "

```
#include "timeday.h"
```

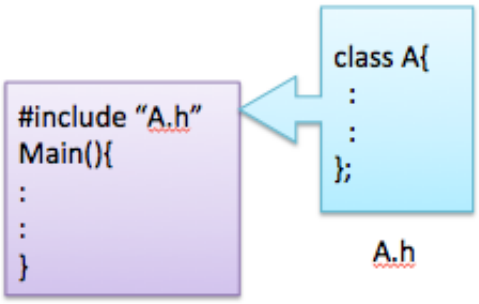
  - " and " usually cause the compiler to look in the current directory for the header file

2.2 The Implementation File

- Contains the definitions of the ADT functions
- Usually has the same name as the header file but a different suffix
  - Since our header file is named **timeday.h**, the implementation file is named **timeday.cpp**
  - You can use **.cxx** or **.C** instead.

2.2.1 #include "timeday.h"

- The implementation file requires an include directive to **include the interface file**:
```
#include "timeday.h"
```
- #**include** directive inserts a **header file** when compiled.



2.3 The Application File

- The Application file is the file that contains the program that uses the ADT
  - It is also called a driver file
  - Must use an include directive to include the interface file:
```
#include "timeday.h"
```

2.4 Running The Program

- Basic steps required to run a program:
  - Compile the implementation file
    - ```
      g++ -c timeday.cpp
      ```
  - Compile the application file
    - ```
      g++ -c sample38.cpp
      ```
  - Link the files to create an executable program using a utility called a **linker**
    - Linking is often done automatically depending on how you compile codes.
    - ```
      g++ sample38.o timeday.o
      ```
      - this will create **a.out**
    - ```
      g++ -o sp38 sample38.o timeday.o
      ```
      - this will create sp38
  - IDE (Xcode etc.) does these process automatically.

2.4.1 Compile timeday.h ?

- The interface file is not compiled separately
  - The preprocessor replaces any occurrence of #**include "timeday.h"** with the text of **timeday.h** before compiling
  - Both the implementation file and the application file contain #**include "timeday.h"**
    - The text of **timeday.h** is seen by the compiler in each of these files
    - There is no need to compile **timeday.h** separately

2.5 Why Three Files?

- Using separate files permits
  - The ADT to be used in other programs without rewriting the definition of the class for each
  - Implementation file to be compiled once even if multiple programs use the ADT
  - Changing the implementation file does not require changing the program using the ADT

2.6 Reusable Components

- An ADT coded in separate files can be used over and over
- The **reusability** of such an ADT class
  - Saves effort since it does not need to be
    - Redesigned
    - Recoded
    - Retested
  - Is likely to result in more reliable components

3 Create separate files with IDE

3.1 Xcode

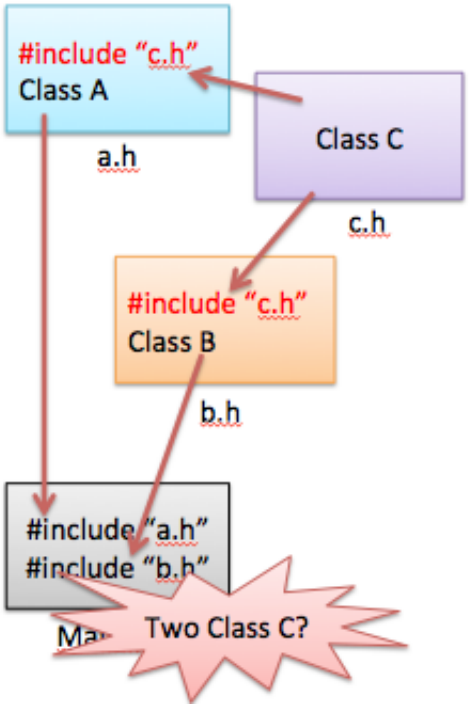
1. Open/Create a project
2. Select [File]→[New]→[File...]
3. Select "Source" under "iOS"
4. Click on [C++ File] then [Next]
5. Type in the name of file. Check [Also create a header file] on.
   (a) you will be asked the location to store.
6. Two files will be added. You can work with those.

3.2 Visual Studio

1. Open/Create a project
2. In Solution Explorer, right-click the project, click Add, and then click Class. This opens the Add Class Dialog Box dialog box.
3. Type a class name (not a file name) and OK.
4. Two files will be added. You can work with those.

4 Multiple Classes

- A program may use several classes
  - Each could be stored in its own interface and implementation files
    - Some files can "**include**" other files, that include still others
  - It is possible that the same interface file could be included in multiple files
  - C++ does not allow multiple declarations of a class
  - The #**ifndef** directive can be used to prevent multiple declarations of a class



4.1 Introduction to #ifndef

- To prevent multiple declarations of a class,we can use these directives:
  - **#define TIMEDAY_H**
    adds **TIMEDAY_H** to a list indicating **TIMEDAY_H** has been seen
  - **#ifndef TIMEDAY_H**
    checks to see if **TIMEDAY_H** has been defined
  - **#endif**
    If **TIMEDAY_H** has been defined, skip to **#endif**

4.1.1 Using #ifndef

- Consider this code in the interface file:

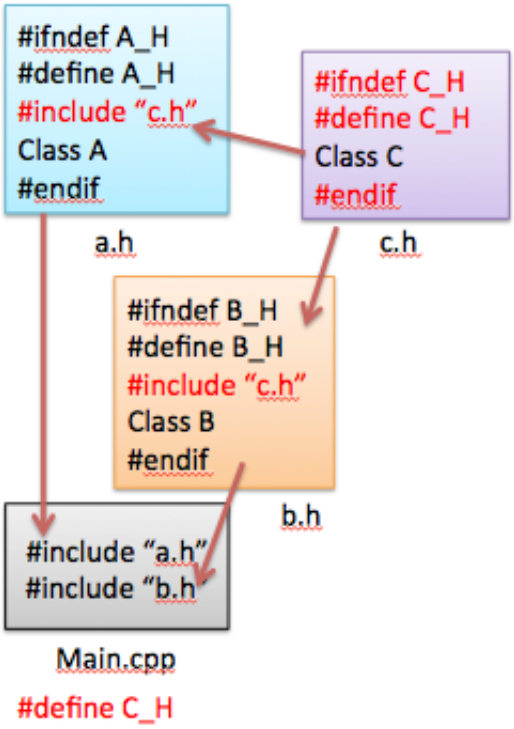
```
#ifndef TIMEDAY_H // if TIMEDAY_H has not defined yet get into #ifndef body.
// If TIMEDAY_H has defined already, skip to #endif
#define TIMEDAY_H // Now TIMEDAY_H is defined.
//////////////////////////////////////////
//  TimeOfDay class definition goes here //
//////////////////////////////////////////
#endif
```

ifndef=if not defined

- Why **TIMEDAY_H**?
  - **TIMEDAY_H** is the normal convention for creating an identifier to use with **ifndef**
    - It is the file name in all caps
    - Use ' _ ' instead of ' . '
  - You may use any other identifier, but will make your code more difficult to read

4.1.2 How #ifndef...#end works?

- "Main.cpp" includes "a.h". Since "a.h" includes "c.h", "c.h" comes with "a.h" into "Main.cpp".
- "Main.cpp" includes "b.h". Since "b.h" includes "c.h", "c.h" comes with "b.h" into "Main.cpp", but the class C definition is not included because C_H is defined already.



4.2 Defining Libraries

- You can create your own libraries of functions
  - You do not have to define a class to use separate files
  - If you have a collection of functions...
    - Declare them in a header file with their comments
    - Define them in an implementation file
    - Use the library files just as you use your class interface and implementation files

5 Makefile

Use command line tools on Mac `http://railsapps.github.io/xcode-command-line-tools.html`

- A common way of automating software builds and other complex tasks with dependencies.
- A **Makefile** is itself a program in a special language.
- The command **'make'** reads **'Makefile'** and do some tasks.
- Can specify file name '**make -f mymakefile**'.
- The default name is "**Makefile**"

```
###################################
#   C++ sample program
###################################
#
CXX     = g++
#
CXXFLAGS  = -g
#
SMPL38=sample38.o timeday.o
SMPL39=sample39.o timeday.o
#
#
TARGET= smpl38 smpl39
all:    $(TARGET)
#
smpl38  :$(SMPL38)
        $(CXX) $(CXXFLAGS) -o $@ $(SMPL38)

smpl39  :$(SMPL39)
        $(CXX) $(CXXFLAGS) -o $@ $(SMPL39)

timeday.o: timeday.cpp timeday.h
        $(CXX) $(CXXFLAGS) -c timeday.cpp

#####################################
.cpp.o:
        $(CXX) $(CXXFLAGS) -c $<

#####################################
clean:
        rm -f *.o $(TARGET) core
```

- To compile
  - make

5.1 Structure of Makefile

Typical element in the simple Makefile:

```
target: dependencies
[TAB] command_to_make_target
~~~~
```

It is important to use tab character, not spaces.

- command 'make target' means:
  - Make sure all the dependencies are up to date
  - If target is older than any dependency, recreate it using the specified commands.

5.2 Age of dependency

The **Makefile** above will do:

- sample38.cpp ⟶ sample38.o
- sample39.cpp ⟶ sample39.o

- timeday.cpp ⟶ timeday.o
- sample38.o & timeday.o ⟶ smpl38
- sample39.o & timeday.o ⟶ smpl39

Each process will be done if

- the product file (file in right side) didn't exist.
  or
- target file (file in left side) is newer than the product file (file in right side)
  - The last modification time of the file is used.

5.2.1 For example:

We have files like:

```
-rw-r--r--  1 fuji  staff   1159 Mar 16 11:06 sample38.cpp
-rw-r--r--  1 fuji  staff    423 Mar 18 10:16 sample39.cpp
-rw-r--r--  1 fuji  staff   2116 Mar 16 11:15 timeday.cpp
-rw-r--r--  1 fuji  staff   2504 Mar 16 11:33 timeday.h
```

```
make
```

This will create files:

```
g++ -g  -c sample38.cpp
g++ -g  -c timeday.cpp
g++ -g  -o smpl38 sample38.o timeday.o
g++ -g  -c sample39.cpp
g++ -g  -o smpl39 sample39.o timeday.o
```

Now We have:

```
-rw-r--r--  1 fuji  staff   1159 Mar 16 11:06 sample38.cpp
-rw-r--r--  1 fuji  staff  87340 Mar 18 10:22 sample38.o
-rw-r--r--  1 fuji  staff    423 Mar 18 10:16 sample39.cpp
-rw-r--r--  1 fuji  staff  84872 Mar 18 10:22 sample39.o
-rwxr-xr-x  1 fuji  staff  25688 Mar 18 10:22 smpl38
-rwxr-xr-x  1 fuji  staff  21532 Mar 18 10:22 smpl39
-rw-r--r--  1 fuji  staff   2116 Mar 16 11:15 timeday.cpp
-rw-r--r--  1 fuji  staff   2504 Mar 16 11:33 timeday.h
-rw-r--r--  1 fuji  staff  96552 Mar 18 10:22 timeday.o
```

Do make again

```
make
```

```
make: Nothing to be done for 'all'.
```

Here We just change the time stamp:

```
touch timeday.h
```

```
-rw-r--r--  1 fuji  staff   1159 Mar 16 11:06 sample38.cpp
-rw-r--r--  1 fuji  staff  87340 Mar 18 10:22 sample38.o
-rw-r--r--  1 fuji  staff    423 Mar 18 10:16 sample39.cpp
-rw-r--r--  1 fuji  staff  84872 Mar 18 10:22 sample39.o
-rwxr-xr-x  1 fuji  staff  25688 Mar 18 10:22 smpl38
-rwxr-xr-x  1 fuji  staff  21532 Mar 18 10:22 smpl39
-rw-r--r--  1 fuji  staff   2116 Mar 16 11:15 timeday.cpp
-rw-r--r--  1 fuji  staff   2504 Mar 18 10:22 timeday.h
-rw-r--r--  1 fuji  staff  96552 Mar 18 10:22 timeday.o
```

Do make again

```
make
```

```
g++ -g  -c timeday.cpp
g++ -g  -o smpl38 sample38.o timeday.o
g++ -g  -o smpl39 sample39.o timeday.o
```

What's happened?

- timeday.h is newer than timeday.o : timeday.cpp ⟶ timeday.o
- timeday.o is newer than smpl38 : sample38.o & timeday.o ⟶ smpl38
- timeday.o is newer than smpl39 : sample39.o & timeday.o ⟶ smpl39

make clean

```
make clean
```

This executes this part:

```
clean:
        rm -f *.o $(TARGET)       # This delete files
```

Therefore:

```
rm -f *.o ex27 ex28 ex29
```

This delete ex27.o, ex28.o, ex29.o, cg.o, ex27, ex28 and ex29.

5.3 Suffix Rule

For C++, object files (*.o files) are created *.cpp files.

```
.cpp.o:
        $(CXX) $(CXXFLAGS) -c $<
```

This line for compiling all ***.cpp** files involved in targets if needed. Note that **$<** is a dependent file. For example, when creating "**sample38.o**", **$<** has **"sample38.cpp**".

5.3.1 Automatic Variables

Here is a table of automatic variables:

- $@
  - The file name of the target of the rule. If the target is an archive member, then '$@' is the name of the archive file. In a pattern rule that has multiple targets '$@' is the name of whichever target caused the rule's recipe to be run.
- $<
  - The name of the first prerequisite. If the target got its recipe from an implicit rule, this will be the first prerequisite added by the implicit rule (see Implicit Rules).

There are some more: see `http://www.gnu.org/software/make/manual/make.html#Automatic-Variables`

5.4 Tools to generate Makefile

- autoconf `https://www.gnu.org/software/autoconf/autoconf.html`
- cmake `https://cmake.org/`