Object-Oriented Programming and Data Structures

COMP2012: Separate Compilation and Makefile

Brian Mak
Desmond Tsoi

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China

COMP 2011 Example: Bulbs and Lamps

- Recall that the example deals with 2 classes: Bulb and Lamp.
- A lamp has at least one light bulb.
- All bulbs of a lamp are the same in terms of price and wattage (power).
- The price of a lamp that is passed to the Lamp's constructor does not include the price of its bulbs which have to be bought separately.
- One installs bulb(s) onto a lamp by calling its member function install_bulbs.

COMP 2011 Example: lamp-test.cpp

```cpp
#include "lamp.h"       /* File: lamp-test.cpp */

int main()
{
    Lamp lamp1(4, 100.5); // lamp1 costs HKD100.5 itself; needs 4 bulbs
    Lamp lamp2(2, 200.6); // lamp2 costs HKD200.6 itself; needs 2 bulbs

    // Install 4 bulbs of 20 Watts, each costing HKD30.1 on lamp1
    lamp1.install_bulbs(20, 30.1);
    lamp1.print("lamp1");

    // Install 2 bulbs of 60 Watts, each costing HKD50.4 on lamp2
    lamp2.install_bulbs(60, 50.4);
    lamp2.print("lamp2");

    return 0;
}
/* To compile: g++ -o lamp-test lamp-test.cpp bulb.cpp lamp.cpp */
```

COMP 2011 Example: bulb.h

```
/* File: bulb.h */

class Bulb
{
  private:
    int wattage;            // A light bulb's power in watt (W)
    float price;            // A light bulb's price in dollars

  public:
    int get_power() const;
    float get_price() const;
    void set(int w, float p); // w = bulb's wattage; p = its price
};
```

COMP 2011 Example: bulb.cpp

```cpp
/* File: bulb.cpp */

#include "bulb.h"

int Bulb::get_power() const { return wattage; }

float Bulb::get_price() const { return price; }

void Bulb::set(int w, float p) { wattage = w; price = p; }
```

COMP 2011 Example: lamp.h

```
#include "bulb.h"       /* File: lamp.h */

class Lamp
{
  private:
    int  num_bulbs; // A lamp MUST have 1 or more light bulbs
    Bulb* bulbs;    // Dynamic array of bulbs installed onto a lamp
    float price;    // Price of a lamp, NOT including price of its bulbs

  public:
    Lamp(int n, float p); // n = number of bulbs; p = lamp's price
    ~Lamp();

    int total_power() const; // Total power/wattage of the light bulbs
    float total_price() const; // Price of a lamp PLUS its light bulbs

    // Print out a lamp's information; see outputs from our example
    void print(const char* prefix_message) const;

    // All light bulbs of a lamp have the same power/wattage and price:
    // w = a light bulb's wattage; p = a light bulb's price
    void install_bulbs(int w, float p);
};
```

COMP 2011 Example: lamp.cpp

```cpp
#include "lamp.h"       /* File: lamp.cpp */
#include <iostream>
using namespace std;

Lamp::Lamp(int n, float p) { num_bulbs = n; price = p; bulbs = new Bulb [n]; }

Lamp::~Lamp() { delete [] bulbs; }

int Lamp::total_power() const { return num_bulbs*bulbs[0].get_power(); }

float Lamp::total_price() const { return price + num_bulbs*bulbs->get_price(); }

void Lamp::print(const char* prefix_message) const
{
    cout << prefix_message << ": total power = " << total_power() << "W"
         << " , total price = $" << total_price() << endl;
}

void Lamp::install_bulbs(int w, float p)
{
    for (int j = 0; j < num_bulbs; ++j)
        bulbs[j].set(w, p);
}
```

Compilation of a Program with Several .cpp Files

- In the Bulbs and Lamps example, there are:
  - 2 header files: bulb.h and lamp.h
  - 2 class implementation files: bulb.cpp and lamp.cpp
  - 1 app program file: lamp-test.cpp
- On Linux/MacOS/Windows/VSCode, you may open a terminal and type in the following command to compile the app executable using the g++ compiler:

```
g++ -o lamp-test lamp-test.cpp bulb.cpp lamp.cpp
```

- g++ has many options; google it for details.

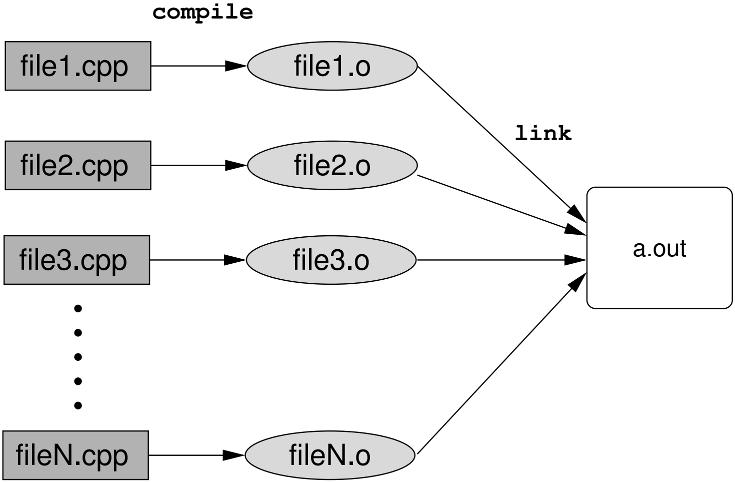
Separate Compilation

- One may also compile each .cpp source file separately as follows:

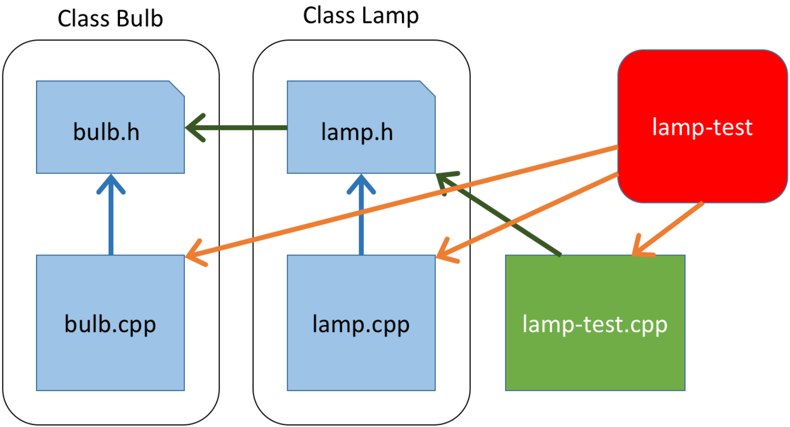
```
g++ -c bulb.cpp
g++ -c lamp.cpp
g++ -c lamp-test.cpp
g++ -o lamp-test bulb.o lamp.o lamp-test.o
```

- The first 3 lines that use g++ with the "-c" option create the object files "bulb.o", "lamp.o", "lamp-test.o".
- The .o object files can't run on their own.
- The last line creates the executable program called "lamp-test" (with the "-o" option) by linking the object files together.
- Linker: a program that combines separately compiled codes together.

Linking Object Files



Dependencies Among Files



Separate Compilation ..

- If only “bulb.cpp” is modified, separate compilation allows us to only re-compile as few files as possible:

  ```
  g++ -c bulb.cpp
  g++ -o lamp-test bulb.o lamp.o lamp-test.o
  ```

- Similarly, if only “lamp.h” is modified but other files are not:

  ```
  g++ -c lamp.cpp
  g++ -c lamp-test.cpp
  g++ -o lamp-test bulb.o lamp.o lamp-test.o
  ```

- Question: Which files need be re-compiled if “bulb.h” is modified?
- To do separate compilation efficiently, we need to find out the dependencies among all the sources .h and .cpp files.
- If you have tens or hundreds of source files in your program, finding out all the dependencies manually is not easy.
- Solution: automate with “make” using a “Makefile”.

A Simple Makefile

```
# Definition of variables
SRCS    = bulb.cpp lamp.cpp lamp-test.cpp
OBJS    = bulb.o lamp.o lamp-test.o

# Rules' Format
# TARGET: DEPENDENCIES
# [TAB]   COMMAND USED TO CREATE THE TARGET
lamp-test: $(OBJS)
        g++ -o lamp-test $(OBJS)

bulb.o: bulb.cpp bulb.h
        g++ -c bulb.cpp

lamp.o: lamp.cpp lamp.h bulb.h
        g++ -c lamp.cpp

lamp-test.o: lamp-test.cpp lamp.h bulb.h
        g++ -c lamp-test.cpp

clean:; /bin/rm lamp-test *.o
```

Get The .h Dependencies Using g++ Option -MM

- Since a header file may include other header file(s), it may not be easy to find out all the dependencies for a source file.
- It becomes non-trivial when a project consists of many files.
- Now g++ has an option -MM to print out the include file dependencies:
- However, -MM alone does not overwrite the Makefile. You will have to insert them by, e.g., copy-and-paste.

```
ras:mak [~/example/bulb-lamp] g++ -MM *.cpp

bulb.o: bulb.cpp bulb.h
lamp-test.o: lamp-test.cpp lamp.h bulb.h
lamp.o: lamp.cpp lamp.h bulb.h
```

A Smarter Makefile

```
# Definition of variables
SRCS = bulb.cpp lamp.cpp lamp-test.cpp
OBJS = $(SRCS:.cpp=.o)
DEPS = $(OBJS:.o=.d)
EXE = lamp-test
CXXFLAGS = -std=c++11

# Rules:
# target: dependencies
# [TAB] command used to create the target
$(EXE): $(OBJS)
        g++ $(CXXFLAGS) -o $@ $(OBJS)

# To include the .d dependency files
-include $(DEPS)

# -MMD -MP creates the .d dependency files
.cpp.o:; g++ $(CXXFLAGS) -MMD -MP -c $<

clean:; /bin/rm $(EXE) $(OBJS) $(DEPS)
```

Libraries

- If you use any functions declared in the standard C++ header files (iostream, string, etc.), to produce a working executable, the linker needs to include their codes, which can be found in the standard C++ libraries.
- A library is a collection of object codes.
- The linker selects object codes from the libraries that contain the definitions for functions used in the program files, and includes them in the executable.
- Some libraries, such as the standard C++ library, are searched automatically by the C++ linker.
- Other libraries have to be specified by the user during the linking process with the '-l" option.

  e.g., To link with a library called "libABC.a" in the local folder,

  ```
  g++ -o myprog myprog.o -lABC
  ```

Static and Dynamic Linking With a Library

Static linking: copy all relevant library functions that are used by a program into its executable.

- Pros: Run faster and is more portable since everything it needs are in the executable.
- Cons: larger file size

Dynamic linking: assume that the library functions are shared — and can be found on the target machines and only write down which shared libraries are required to use at runtime in the executable.

- Pros: smaller file size, and many programs can share a single copy of the shared libraries.
- Cons#1: Run more slowly as the actual linking with the libraries are done at runtime.
- Cons#2: Less portable as a machine may not have installed the required shared libraries.

Preprocessor Directives: #include

- Besides statements allowed in a programming language, useful program development features are added via directives.
- Directives are handled by a program called preprocessor before the source code is compiled.
- In C++, preprocessor directives begin with the # sign in the very first column.
- The #include directive reads in the contents of the named file.

```
#include <iostream>
#include "myfile.h"
```

- < > are used to include standard header files which are searched at the standard library directories.
- " " are used to include user-defined header files which are searched first at the current directory.
- "`g++ -I`" may be used to change the search path.

#ifndef, #define, #endif

```
/* program.h */      /* b.h */          /* c.h */
#include "b.h"       #include "a.h"     #include "a.h"
#include "c.h"       #include "d.h"     #include "e.h"
...                  ...                ...
```

Since #include directives may be nested, the same header file may be included twice!

- multiple processing ⇒ waste of time
- re-definition of global variables, constants, classes

Thus, the need of conditional directives

```
#ifndef LAMP_H
#define LAMP_H
// object declarations, class definitions, functions
#endif  // LAMP_H
```