C++: Separate compilation

Pierre-Alain Fayolle

Table of contents

Some terminology

Introduction

Compilation of a single file

Header files

Separate compilation

Preprocessor directives

Some basic terminology

These terms will appear throughout the course.

- Class: a user defined data type, i.e. a set of states and operations to transition between these states
- Object: a region of storage with associated semantics
- Instantiating an object: is the process of creating an object
- Data member: data associated to a class. Each object, instance of a class, has its own data
- Function member or method: function associated to a class and used to modify the state of an object

Introduction

- The first thing we need to learn is how to compile C++ files to generate executable binaries (programs)
- Like in C, a compiler is used to compile each source file to a binary object
- The binary objects are then linked together to form an executable

Compilation of a single file

- Let us start by the case of a single C++ file:

```
// Date.cpp
#include <iostream>

class Date { public: int year; };

int main (void) {
 Date d1;
 d1.year = 2012;
 std::cout << d1.year << std::endl;
 return 0;
}
```

- This code defines one class named Date with one field named year
- It instantiates a Date object named d1, sets some value to its unique field and then prints its content

Compilation of a single file

- To compile Date.cpp we can either do:

  ```
  $ g++ -c Date.cpp
  $ g++ -o Date Date.o
  ```

  which first compiles Date.cpp into Date.o and then link Date.o with the C++ runtime.
- or in one step:

  ```
  $ g++ -o Date Date.cpp
  ```
- The two methods are equivalent and produce an executable named Date. To run it, type:

  ```
  $ ./Date
  ```

Header files

- The general case in C++ is to separate the class interface from its implementation
  - interface: public data and prototype of the public methods
  - implementation: code for methods (both public and internal) and internal data (data used by the implementation but not exported by the interface)
- Header files contain the exported interface
  - Headers are recognized by the filename extension, which can be: .h or .hpp or .hh or .hxx
- Implementation files contain the implementation
  - Implementation files are recognized by the filename extension, which can be: .cpp or .cc or .cxx or .C
- Interfaces are imported by including headers with the preprocessor command #include followed by an header filename

Example

```
// Date.h
class Date {
  public:
    void set(int m, int d, int y);
    void print();
  private:
    int month, day, year;
};
```

Example

```
// Date.cpp
#include <iostream>
#include "Date.h"

void Date::set(int m, int d, int y) {
  month = m; day = d; year = y;
}

void Date::print() {
  std::cout << month << " / " << day
             << " / " << year << std::endl;
}
```

Example

```
// main.cpp
#include "Date.h"

int main(void) {
  Date date;
  date.set(1,29,2012);
  date.print();
  return 0;
}
```

Compilation of multiple files

- In order to generate an executable from the previous files, type at the command prompt:

  ```
  $ g++ -c Date.cpp
  $ g++ -c main.cpp
  $ g++ -o Date main.o Date.o
  ```

- Lines 1 and 2 create object files (Date.o and main.o); the option -c of the compiler compiles source code to binary object files
- The object files need to be linked together in order to create an executable program (line 3); the option -o indicates the name of the executable. If this option is not used, the executable file gets a default name: a.out

Separate compilation

Besides for modularity reasons, why is it a good idea to have code separated in different files that are compiled separately ?

- Suppose that we perform some modifications in Date.cpp, e.g. we change the implementation of Date::print()
- To generate an executable, only two steps are needed:
  - Recompile Date.cpp
  - Link main.o and Date.o

  Note that we do not need to recompile the files that did not change (e.g. main.cpp)
- Suppose now that the project instead of containing three files contains several thousands. The actions needed are:
  - Recompile source files that were modified
  - Link all object files

  Which provides a significant speed improvement

Preprocessor directives

The C++ preprocessor is similar to the C preprocessor and provides the same facilities:

- #include
- #define
- Compile time conditional expressions
- Macros
- and other more advanced preprocessor tricks

File inclusion

- Directives like #include<iostream> or #include"Date.h" are used to import library into a program
- #include instructs the preprocessor to locate the specified file, open it and insert its content in place of the directive (same as a copy and paste)
- < > (angle brackets) is used for standard header files searched in the standard library directories
- " " is used for user-defined header files, sought in the current directory
- It is possible to specify the location where the preprocessor should search for header files. With the GNU C++ compiler, this is done with the option -I. Example:

  ```
  $ g++ test.cpp -I/home/students/user/include
  ```

  will tell the preprocessor to search in the specified directory in addition to the the current directory.

Compile time conditional expressions

Conditional directives are used to prevent multiple inclusion of an header file. Let us look at an example first to motivate the usage of conditional directives.

```
// Date.h
struct Date {
  int month, day, year;
};

// main.cpp
#include "Date.h"
#include "Date.h"

int main(void) {
  return 0;
}
```

Example

The previous code is transformed by the preprocessor to:

```
// main.cpp
struct Date {
  int month, day, year;
};
struct Date { // Error
  int month, day, year;
};

int main(void) {
  return 0;
}
```

Compile time conditional expressions

- The previous code does not compile: the symbol Date is defined several times
- While this example may seem artificial, it is not uncommon in practice to have several files that include each other, resulting in a file indirectly including the same file twice
- The solution for preventing multiple inclusion of an header file consists in using conditional directives

Example

Let us rewrite Date.h by using conditional directives:

```
#ifndef DATE_H
#define DATE_H
struct Date {
  int month, day, year;
};
#endif
```

Example

After preprocessing, main.cpp becomes

```
#ifndef DATE_H
#define DATE_H
struct Date {
  int month, day, year;
};
#endif
#ifndef DATE_H
#define DATE_H
struct Date {
  int month, day, year;
};
#endif

int main(void) {
  return 0;
}
```

Example

As the preprocessor evaluates the #include statements, the first ifndef block will be included, the constant DATE_H defined and the second ifndef block excluded as the constant DATE_H is already defined. The final code after preprocessing will be:

```
struct Date {
  int month, day, year;
};

int main(void) {
  return 0;
}
```