Multi-File Programming

Twinge of Chapter 9

Pages 447-457

What we know

- Programs can be very large
  - All the code in a single file is unacceptable
- Programs can be broken up into sub problems known as functions
- The main() function is like highlander, there can only be one

Study Case

```cpp
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
  int x = 5;
   x = inc(x);
   return 0;
}
```

main.cpp

```cpp
int inc(int num)
{
   return num + 1;
}
```

source1.cpp

- Source files are compiled first independently, then functions are linked together with the linker
- Problem: the compiler is unaware of the existance of inc() in main.cpp

Study Case

```cpp
#include <iostream>
#include <cmath>
using namespace std;
int inc(int);
int main()
{
  int x = 5;
   x = inc(x);
   return 0;
}
```

main.cpp

```cpp
int inc(int num)
{
   return num + 1;
}
```

source1.cpp

- Easy solution: prototype the function in main.cpp

Study Case

```cpp
#include <iostream>
#include <cmath>
using namespace std;
int inc(int);
int main()
{
  int x = 5;
   x = inc(x);
   x = func(x);
   return 0;
}
```

main.cpp

```cpp
int inc(int num)
{
   return num + 1;
}
```

source1.cpp

```cpp
int func(int num)
{
   num = inc(num);
   num *= 2;
   return num;
}
```

source2.cpp

- Ok, now we need to prototype func() in main.cpp, and inc() in source2.cpp

Study Case

```cpp
#include <iostream>
#include <cmath>
using namespace std;
int inc(int);
int func(int);
int main()
{
  int x = 5;
   x = inc(x);
   x = func(x);
   return 0;
}
```

main.cpp

```cpp
int inc(int num)
{
   return num + 1;
}
```

source1.cpp

```cpp
int inc(int);
int func(int num)
{
   num = inc(num);
   num *= 2;
   return num;
}
```

source2.cpp

- Phew! Little bit more work but we got it.

Study Case

```cpp
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
    func1(…);
    func2(…);
    func3(…);
    …
    funcn(…);
    return 0;
}
```

main.cpp

```cpp
int func1(…)
{ … }
int func2(…)
{ … }
int func3(…)
{ … }
…
int funcn(…)
{ … }
```

source1.cpp

```cpp
int someFunc()
{
    func1(…);
    func2(…);
    func3(…);
    …
    funcn(…);
}
```

source2.cpp

- Got a situation here.

Study Case

```
int func1(…)
{ … }
int func2(…)
{ … }
int func3(…)
{ … }
…
int funcn(…)
{ … }
```

source1.cpp

```
int sFunc1()
{
    func1(…);
    func2(…);
    func3(…);
    …
    funcn(…);
}
```

source2.cpp

```
int sFunc2()
{
    func1(…);
    func2(…);
    func3(…);
    …
    funcn(…);
}
```

source2.cpp

…

```
int sFuncN()
{
    func1(…);
    func2(…);
    func3(…);
    …
    funcn(…);
}
```

sourceN.cpp

- Forget it. Pretty sure you can make more money as a business major anyways.

Header Files

Contains Function Defines

source1.cpp

Calls functions from source1.cpp

source2.cpp

Calls functions from source1.cpp

sourceN.cpp

```
int func1(…);
int func2(…);
int func3(…);
…
int funcn(…)
```

source1.h

- What if we centralize our prototypes?
  - We will only have to write them once
- But all the source files are still unaware of the prototypes in source1.h

Header Files

Contains
Function
Defines

source1.cpp

```
#include "source1.h"
…
```

source2.cpp

```
#include "source1.h"
…
```

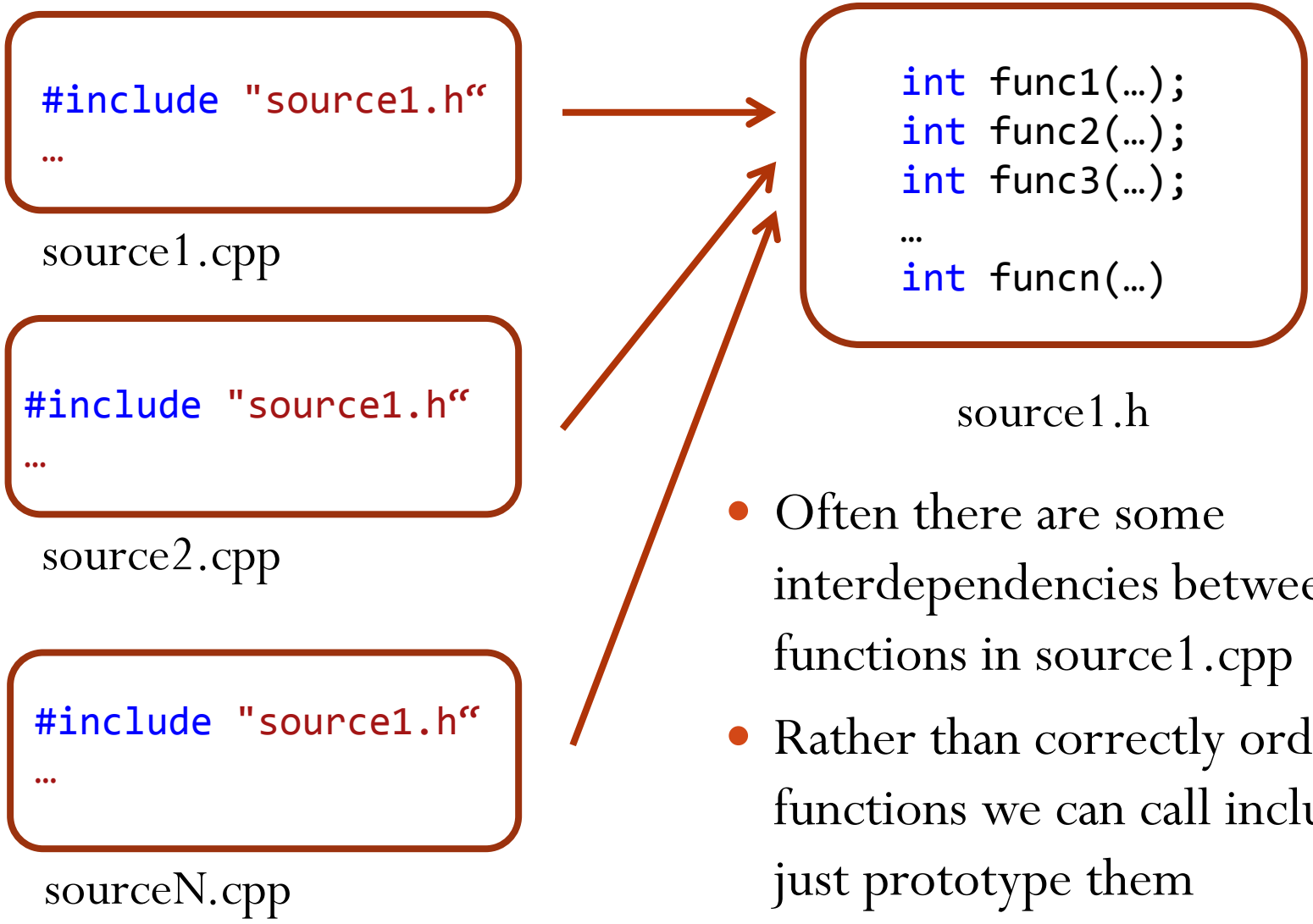
sourceN.cpp

```
int func1(…);
int func2(…);
int func3(…);
…
int funcn(…)
```

source1.h

- Using the include directive, the prototypes can be included into multiple files
- One line per file instead of dozens

Header Files



- Often there are some interdependencies between functions in source1.cpp
- Rather than correctly ordering the functions we can call include and just prototype them

Header Files

- Typically for each new source file (*.cpp) there will be an associated header file (*.h)
  - Common exception is the main.cpp

Typically Contain	Typically Don't Contain
• Function Prototypes • Class/Structure Defines • Enumerations • Templates	• Function Defines • Globals!! • Method Defines

- Header files are there to declare the existence of code to other cpp files

Example 13.1

- Create a source file named *geometry*
  - In this source file create two functions that calculate the area of a triangle and a rectangle
- Create a header file to prototype the functions
- Create a source file named *main*
  - Include geometry header file
  - Call the functions from the geometry file using data of your choice
  - Print the results

Classes and Multi-File Compilation

- Create a Rectangle Class with:
  - Four data members
    - X, Y, Width, Height
  - Default Constructor
  - Non-Default Constructor
  - Member function for calculating area
  - Member function for calculating perimeter

Header file → Class Definition

Source file → Member Function Definition

Rectangle Class

```
class Rectangle {
public:
    Rectangle();
    Rectangle(float xPos, float yPos, float w, float h);
    float area();
    float perimeter();
    float x, y, width, height;
};
```

Rectangle.h

Rectangle Class

```
Rectangle::Rectangle() {
x = 0;
y = 0;
width = 0;
height = 0;
}

Rectangle::Rectangle(float xPos, float yPos, float w, float h) {
x = xPos;
y = yPos;
width = w;
height = h;
}
…
```

Rectangle.cpp

Rectangle Class

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

using namespace std;

int main()
{
    Rectangle r(0, 0, 4, 8);
    cout << r.area() << endl;
    return 0;
}
```

main.cpp

Visiting the Family

- Consider the following

```
#include “grandfather.h"
#include “father.h"
int main()
{
  foo pa;
  return 0;
}
```

main.cpp

```
#include "grandfather.h"
```

father.h

```
struct foo {
   int member;
}
```

grandfather.h

- struct foo is included twice by in main()
- this will cause a redefinition compiler error

Solution 1: Remove Include

```
#include “grandfather.h"
//#include “father.h"
int main()
{
  foo pa;
  return 0;
}
```

main.cpp

```
#include "grandfather.h"
```

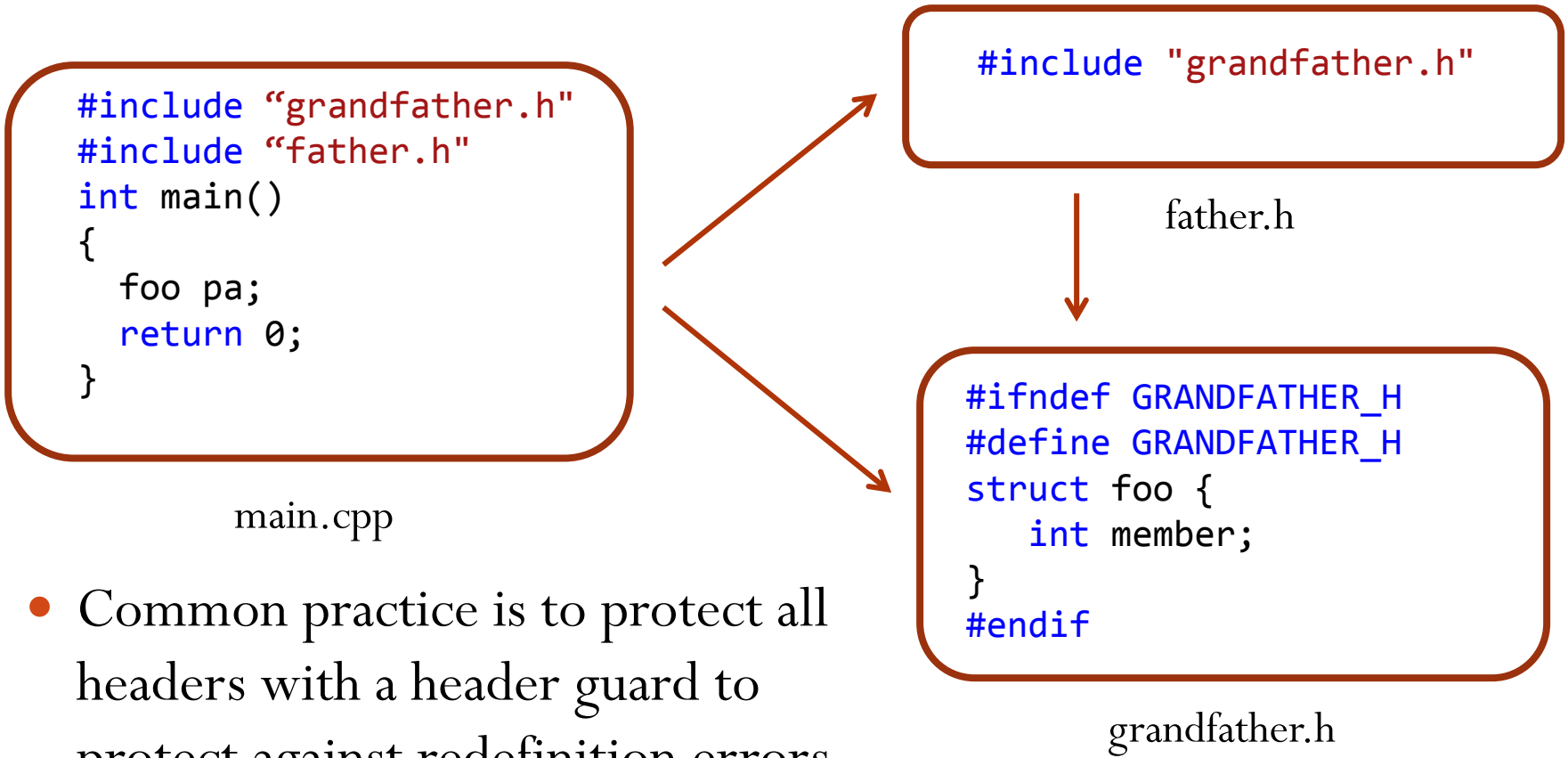
father.h

```
struct foo {
   int member;
}
```

grandfather.h

- It may not always be obvious as to which include is causing the problem
- Includes may be complex and difficult to untangle

Solution 2: Header Guards



- Common practice is to protect all headers with a header guard to protect against redefinition errors

Solution 2: Header Guards

```
#ifndef GRANDFATHER_H
#define GRANDFATHER_H

struct foo {
   int member;
}

#endif
```

- Remember any command with a # is a preprocessor directive
  - Not in final program
  - Executed before compilation commences

Solution 2: Header Guards

```
#ifndef GRANDFATHER_H
#define GRANDFATHER_H

struct foo {
    int member;
}

#endif
```

- ifndef: a if statement that will execute if the identifier (GRANDFATHER_H) isn't defined
- define: defines the identifier
- endif: defines the end of ifndef (or any other preprocessor conditional)

Solution 2: Header Guards

```
#ifndef GRANDFATHER_H
#define GRANDFATHER_H

struct foo {
   int member;
}

#endif
```

- Stepping through the preprocessor steps of main.cpp
  1. Grandfather.h is included first
  2. GRANDFATHER_H hasn't been defined so the condition is executed
  3. GRANDFATHER_H is defined
  4. struct foo is defined
  5. Father is included which includes Grandfather.h again
  6. GRANDFATHER_H so contents of the header are skipped

Strategy of Multi-File Programming

- Functions should be grouped into files based on commonality
- Each source file should have a associated header file of the same name
  - All functions in the source should be defined in the header file
- Some functions are only meant to be used locally in the *.cpp file
  - Keep the prototyping local to the *.cpp file
  - Reduces unnecessary function conflicts
- Often a header file will be used just for defines and structures
- For class programming
  - Each class should occupy a separate source and header file

Example 13.2

- Restructure the Bank Employee Example such that:
  - Only the main function is in the main.cpp
  - Bank Class is in its own set of files
  - Employee Class is in its own set of files

Example 13.3

- Create a time class with the following properties:
  - Member representing seconds
  - Member representing minutes
  - Default constructor initializing members to zero
  - Method which adds a second, when 60 seconds are reached, the seconds reset and minute increases
- Place the time class in its own set of files
- In the main.cpp, use initiate a time object and increase the seconds.