Separating Interface and Implementation of Classes Header and Source Files

Shahram Rahatlou

SAPIENZA
UNIVERSITÀ DI ROMA

Corso di Programmazione++

Roma, 24 March 2009

Pointers and References to Objects

```
// app2.cpp
#include <iostream>
using std::cout;  // use using only for specific
classes
using std::endl;  // not for entire namespace

class Counter {
  public:
    Counter() { count_ = 0; x_=0.0; };
    int value()  { return count_; }
    void reset() { count_ = 0; x_=0.0; }
    void increment() { count_++; }
    void increment(int step)
       { count_ = count_+step; }
    void print() {
      cout << "---- Counter::print() ----" << endl;
      cout << "my count_: " << count_ << endl;
      // this is special pointer
      cout << "my address: " << this << endl;
      cout << "&x_ : " << &x_ << "  sizeof(x_): "
           << sizeof(x_) << endl;
      cout << "&count_ : " << &count_
      << "  sizeof(count_): "
      << sizeof(count_) << endl;
      cout << "---- Counter::print()----" << endl;
    }

  private:
    int count_;
    double x_; // dummy variable
};
```

```
void printCounter(Counter& counter)  {
  cout << "counter value: " << counter.value() << endl;
}

void printByPtr(Counter* counter)  {
  cout << "counter value: " << counter->value() << endl;
}
```

```
int main() {
  Counter counter;
  counter.increment(7);

  // ptr is a pointer to a Counter Object
  Counter* ptr = &counter;
  cout << "ptr = &counter: " << &counter << endl;

  // use . to access member of objects
  cout << "counter.value(): " << counter.value() << endl;

  // use -> with pointer to objects
  cout << "ptr->value(): " << ptr->value() << endl;

  printCounter( counter );
  printByPtr( ptr );

  ptr->print();

  cout << "sizeof(ptr): " << sizeof(ptr) << "\t"
       << "sizeof(counter): " << sizeof(counter)
       << endl;

return 0;
}
```

-> instead of . When using pointers to objects

Size and Address of Objects

gcc 3.4.4 on cygwin

```
$ g++ -o app2 app2.cpp
$ ./app2
ptr = &counter: 0x22ccd0
counter.value(): 7
ptr->value(): 7
printCounter: counter value: 7
printByPtr: counter value: 7
---- Counter::print() : begin ----
my count_: 7
my address: 0x22ccd0
&count_ : 0x22ccd0  sizeof(count_): 4
&x_ : 0x22ccd8  sizeof(x_): 8
---- Counter::print() : end ----
&i: 0x22ccc8
sizeof(ptr): 4  sizeof(counter): 16
sizeof(int): 4  sizeof(double): 8
```

gcc 4.1.1 on fedora core 6

```
$ g++ -o app2 app2.cpp
$ ./app2
ptr = &counter: 0xbf841e20
counter.value(): 7
ptr->value(): 7
printCounter: counter value: 7
printByPtr: counter value: 7
---- Counter::print() : begin ----
my count_: 7
my address: 0xbf841e20
&count_ : 0xbf841e20  sizeof(count_): 4
&x_ : 0xbf841e24  sizeof(x_): 8
---- Counter::print() : end ----
&i: 0xbf841e1c
sizeof(ptr): 4  sizeof(counter): 12
sizeof(int): 4  sizeof(double): 8
```

- Different size of objects on different platform!
  - Different configuration of compiler
  - Optimization for access to memory

- Address of object is address of first data member in the object

Classes and Applications

- So far we have always included the definition of classes together with the main application in one file
- The advantage is that we have only one file to modify
- Disadvantage are many
  - There is always ONE file to modify no matter what kind of modification you want to make
  - This file becomes VERY long after a very short time
  - Hard to maintain everything in only one place
  - We compile everything even after very simple changes

Example of Typical Application So Far

```
// app2.cpp
#include <iostream>
using std::cout;
using std::endl;

class Counter {
  public:
    Counter() { count_ = 0; };
    int value()  { return count_; }
    void reset() { count_ = 0; }
    void increment() { count_++; }
    void increment(int step) { count_ = count_+step; }

  private:
    int count_;
};

Counter makeCounter() {
  Counter c;
  return c;
}

void printCounter(Counter& counter)  {
  cout << "counter value: " << counter.value() << endl;
}

void printByPtr(Counter* counter)  {
  cout << "counter value: " << counter->value() << endl;
}
```

```
int main() {
  Counter counter;
  counter.increment(7);

  Counter* ptr = &counter;

  cout << "counter.value(): " << counter.value()
       << endl;
  cout << "ptr = &counter: " << &counter << endl;
  cout << "ptr->value(): " << ptr->value() << endl;

  Counter c2 = makeCounter();
  c2.increment();

  printCounter( c2 );

  cout << "sizeof(ptr): " << sizeof(ptr)
       << " sizeof(c2): " << sizeof(c2)
       << endl;

  return 0;

}
```

Separating Classes and Applications

- It’s good practice to separate classes from applications

- Create one file with only your application
  - Use #include directive to add all classes needed in your application
  - Keep a separate file for each class

- Compile your classes separately

- Include compiled classes (or libraries) when linking your application

First Attempt at Improving Code Management

```cpp
// Datum1.cc
// include all header files needed
#include <iostream>
using namespace std;

class Datum {
  public:
    Datum() { }

    Datum(double x, double y) {
      value_ = x;
      error_ = y;
    }

    Datum(const Datum& datum) {
      value_ = datum.value_;
      error_ = datum.error_;
    }

    void print() {
      cout << "datum: " << value_
           << " +/- " << error_
           << endl;
    }

  private:
    double value_;
    double error_;
};
```

```cpp
// app1.cpp
#include "Datum1.cc"

int main() {

  Datum d1;
  d1.print();

  Datum d2(0.23,0.212);
  d2.print();

  Datum d3( d2 );
  d3.print();

  return 0;
}
```

```
$ g++ -o app1 app1.cpp
$ ./app1
datum: NaN +/- 8.48798e-314
datum: 0.23 +/- 0.212
datum: 0.23 +/- 0.212
```

Problems with Previous Example

- Although we have two files it is basically if we had just one!

- Datum1.cc includes not only the declaration but also the definition of class Datum
  - Implementation of all methods exposed to user

- When compiling app1.cpp we also compile class Datum every time!
  - We do not need any library because app1.cpp includes all source code!
  - When compiling and linking app1.cpp we also create compiled code for Datum to be used ain out application
  - Remember what #include does!

Pre-Compiled version of **`Datum1.cc`**

```
$ wc -l Datum1.cc
30 Datum1.cc

$ wc -l app1.cpp
16 app1.cpp

$ g++ -E -c Datum1.cc > Datum1.cc-precomoiled

$ wc -l Datum1.cc-precompiled
23740 Datum1.cc-precompiled
```

- Our source file is only a few lines long

- The precompiled version is almost 24000 lines!
  - This is all code included in and by iostream

```
$ grep "#include"  /usr/lib/gcc/i686-pc-cygwin//3.4.4/include/c++/iostream
 *  This is a Standard C++ Library header.  You should @c #include this header
#include <bits/c++config.h>
#include <ostream>
#include <istream>
```

iostream

```
#ifndef _GLIBCXX_IOSTREAM
#define _GLIBCXX_IOSTREAM 1

#pragma GCC system_header

#include <bits/c++config.h>
#include <ostream>
#include <istream>

namespace std
{
  /**
   *  @name Standard Stream Objects
   *
*/
  //@{
  extern istream cin;          ///< Linked to standard input
  extern ostream cout;         ///< Linked to standard output
  extern ostream cerr;         ///< Linked to standard error (unbuffered)
  extern ostream clog;         ///< Linked to standard error (buffered)

#ifdef _GLIBCXX_USE_WCHAR_T
  extern wistream wcin;        ///< Linked to standard input
  extern wostream wcout;       ///< Linked to standard output
  extern wostream wcerr;       ///< Linked to standard error (unbuffered)
  extern wostream wclog;       ///< Linked to standard error (buffered)
#endif
  //@}

  // For construction of filebuffers for cout, cin, cerr, clog et. al.
  static ios_base::Init __ioinit;
} // namespace std

#endif /* _GLIBCXX_IOSTREAM */
```

I have removed all comments from the file to make it fit in this slide

Additional code included by the header files in this file

Separating Interface from Implementation

- Clients of your classes only need to know the interface of your classes

- Remember:
  - Users should only rely on public members of your class
  - Internal data structure must be hidden and not needed in applications

- Compiler needs only the declaration of your classes, its functions and their signature to compile the application
  - Signature of a function is the exact set of arguments passed to a function and it return type

- The compiled class code (definition) is needed only at link time
  - Libraries are needed to link not to compile!

Header and Source Files

- We can separate the declaration of a class from its implementation
  - Declaration tells the compiler about data members and member functions of a class
  - We know how many and what type of arguments a function has by looking at the declaration but we don’t know how the function is implemented
- Declaration of a class Counter goes into a file usually called Counter.h or Counter.hh suffix
- Implementation of methods goes into the source file usually called Counter.cc

Counter.h and Counter.cc

```
// Counter.h
// Counter Class: simple counter class.
// Allows simple or step
// increments and also a reset function

// include header files for types
// and classes used in the declaration

class Counter {
  public:
    Counter();
    int value();
    void reset();
    void increment();
    void increment(int step);

  private:
    int count_;
};
```

```
// Counter.cc
// include class header files
#include "Counter.h"

// include any additional header files
//  needed in the class
// definition
#include <iostream>
using std::cout;
using std::endl;

Counter::Counter() {
  count_ = 0;
};

int Counter::value()  {
  return count_;
}

void Counter::reset() {
  count_ = 0;
}

void Counter::increment() {
  count_++;
}

void Counter::increment(int step) {
  count_ = count_+step;
}
```

Scope operator :: is used to tell methods belong to Class Counter

What is included in header files?

- Declaration of the class
  - Public and data members

- All header files for types and classes used in the header
  - data members, arguments or return types of member functions

- Sometimes when we have very simple methods these are directly implemented in the header file

- Methods implemented in the header file are referred to as inline functions
  - For example getter methods are a good candidate to become inline functions

What is included in source file?

- Header file of the class being implemented
  - Compiler needs the prototype (declaration) of the methods

- Implementation of methods declared in the header file
  - Scope operator :: must be used to tell the compiler methods belong to a class

- Header files for all additional types used in the implementation but not needed in the header!
  - Nota bene: header files include in the header file of the class are automatically included in the source file

Compiling Source Files of a Class

WinXP+ cygwin

```
$ g++ Counter.cc
/usr/lib/gcc/i686-pc-cygwin/3.4.4/../../../libcygwin.a(libcmain.o)::
undefined reference to `_WinMain@16'
collect2: ld returned 1 exit status
```

Linux

```
$ g++ Counter.cc
/usr/lib/gcc/i386-redhat-linux/4.0.2/../../../crt1.o(.text+0x18):
In function `_start':: undefined reference to `main'
collect2: ld returned 1 exit status
```

- Do you understand the error?
- What does undefined symbol usually mean?
- Why we did not encounter this error earlier?

Reminder about g++

- g++ by default looks for a main function in the file being compiled unless differently instructed

- The main function becomes the program to run when the compiler is finished linking the binary application
  - Compiling: translate user code in high level language into binary code that system can use
  - Linking: put together binary pieces corresponding to methods used in the main function
  - Application: product of the linking process

- Source files of classes do not have any main method

- We need to tell g++ (and other compilers) no linking is needed

Compiling without Linking

- g++ has a **–c** option that allows to specify only compilation is needed

- User code is translated into binary but no attempt to look for main method and creating an application

```
$ ls -l Counter.*
-rw-r--r--  1 rahatlou users 449 May 15 00:55 Counter.cc
-rw-r--r--  1 rahatlou users 349 May 15 00:55 Counter.h

$ g++ -c Counter.cc

$ ls -l Counter.*
-rw-r--r--  1 rahatlou users  449 May 15 00:55 Counter.cc
-rw-r--r--  1 rahatlou users  349 May 15 00:55 Counter.h
-rw-r--r--  1 rahatlou users 1884 May 15 01:23 Counter.o
```

By default g++ creates a .o (object file) for the .cc file

Using Header Files in Applications

```
// app2.cpp
#include <iostream>
using namespace std;

#include "Counter.h"


Counter makeCounter() {
  Counter c;
  return c;
}

void printCounter(Counter& counter)  {
  cout << "counter value: "
       << counter.value() << endl;
}

void printByPtr(Counter* counter)  {
  cout << "counter value: "
       << counter->value() << endl;
}
```

```
int main() {
  Counter counter;
  counter.increment(7);

  Counter* ptr = &counter;

  cout << "counter.value(): "
       <<counter.value() << endl;
  cout << "ptr = &counter: "
       << &counter << endl;
  cout << "ptr->value(): "
       << ptr->value() << endl;

  Counter c2 = makeCounter();
  c2.increment();

  printCounter( c2 );

  return 0;
}
```

```
$ g++ -o app2 app2.cpp
/tmp/ccJuugJc.o:app2.cpp:(.text+0x10d): undefined reference to `Counter::Counter()'
/tmp/ccJuugJc.o:app2.cpp:(.text+0x124): undefined reference to `Counter::value()'
/tmp/ccJuugJc.o:app2.cpp:(.text+0x16e): undefined reference to `Counter::value()'
/tmp/ccJuugJc.o:app2.cpp:(.text+0x1dc): undefined reference to `Counter::Counter()'
/tmp/ccJuugJc.o:app2.cpp:(.text+0x1ef): undefined reference to `Counter::increment(int)'
/tmp/ccJuugJc.o:app2.cpp:(.text+0x200): undefined reference to `Counter::value()'
/tmp/ccJuugJc.o:app2.cpp:(.text+0x272): undefined reference to `Counter::value()'
/tmp/ccJuugJc.o:app2.cpp:(.text+0x2b7): undefined reference to `Counter::increment()'
collect2: ld returned 1 exit status
```

Providing compiled Class Code at Link Time

- Including the header file is not sufficient!
  - It tells the compiler only about arguments and return type
  - But it does not tell him what to execute
  - Compiler doesn’t have the binary code to use to create the application!

- We must use the compiled object file at link time
  - g++ is told to make an application called app2 from source code in app2.cpp and using also the binary file Counter.o to find any symbol needed in app2.cpp

```
$ g++ -o app2 app2.cpp Counter.o
$ ./app2
counter.value(): 7
ptr = &counter: 0x23ef10
ptr->value(): 7
counter value: 1
```

Problem: Multiple Inclusion of Header Files!

- What if we include the same header file several times?

- This can happen in many ways

- Some pretty common ways are
  - **`App.cpp`** includes both **`Foo.h`** and **`Bar.h`**
  - **`Foo.h`** is included in **`Bar.h`** and **`Bar.cc`**

```
// Bar.h

#include “Foo.h”

class Bar {

  // class goes here
  Bar(const Foo& afoo, double x);


}`
```

```
// App.cpp

#include “Foo.h”
#include “Bar.h”

int main() {

  // program goes here
  Foo f1;
  Bar b1(f1, 0.3);

  return 0;
}
```

Example of Multiple Inclusion

```
// app3.cpp
#include <iostream>
using namespace std;
#include "Counter.h"

Counter makeCounter() {
  Counter c;
  return c;
}

void printCounter(Counter& counter)  {
  cout << "counter value: " << counter.value() << endl;
}

void printByPtr(Counter* counter)  {
  cout << "counter value: " << counter->value() << endl;
}

#include "Counter.h"
int main() {
  Counter counter;
  counter.increment(7);

  Counter c2 = makeCounter();
  c2.increment();

  printCounter( counter );
  printCounter( c2 );

  return 0;
}
```

Line 19

```
// Counter.h
// Counter Class: simple counter class. All
// increments and also a reset function

// include header files for types and classe
// used in the declaration

class Counter {
  public:
    Counter();
    int value();
    void reset();
    void increment();
    void increment(int step);

  private:
    int count_;
}
```

Line 8

```
$ g++ -o app3 app3.cpp Counter.o
In file included from app3.cpp:19:
Counter.h:8: error: redefinition of `class Counter'
Counter.h:8: error: previous definition of `class Counter'
```

#define, #ifndef and #endif directives

- Problem of multiple inclusion can be solved at pre-compiler level

1: if Datum_h is not defined follow the instruction until #endif

2: define a new variable called Datum_h

3: end of ifndef block

```
#ifndef Datum_h
#define Datum_h
// Datum.h

class Datum {
  public:
    Datum();
    Datum(double x, double y);
    Datum(const Datum& datum);
    double value() { return value_; }
    double error() { return error_; }

  private:
    double value_;
    double error_;
};
#endif
```

Example: application using Datum

```cpp
// app4.cpp
#include "Datum.h"
#include <iostream>

void print(Datum& input) {
  using namespace std;
  cout << "input: " << input.value()
       << " +/- " << input.error()
       << endl;
}

#include "Datum.h"

int main() {
  Datum d1(-1.4,0.3);
  print(d1);

  return 0;
}
```

```
$ g++ -c Datum.cc
$ g++ -o app4 app4.cpp Datum.o
$ ./app4
input: -1.4 +/- 0.3
```

Typical Errors

- Forget to use the scope operator :: in .cc files

```
#ifndef FooDatum_h
#define FooDatum_h
// FooDatum.h

class FooDatum {
 public:
   FooDatum();
   FooDatum(double x, double y);
   FooDatum(const FooDatum& datum);
   double value() { return value_; }
   double error() { return error_; }
   double significance();

 private:
   double value_;
   double error_;
};
#endif
```

```
#include "FooDatum.h"

FooDatum::FooDatum() { }

FooDatum::FooDatum(double x, double y) {
  value_ = x;
  error_ = y;
}

FooDatum::FooDatum(const FooDatum& datum) {
  value_ = datum.value_;
  error_ = datum.error_;
}

double
significance() {
  return value_/error_;
}
```

```
$ g++ -c FooDatum.cc
FooDatum.cc: In function `double significance()':
FooDatum.cc:17: error: `value_' undeclared (first use this function)
FooDatum.cc:17: error: (Each undeclared identifier is reported only once
 for each function it appears in.)
FooDatum.cc:17: error: `error_' undeclared (first use this function)
```

- Functions implemented as global
- error when applying function as a member function to objects
- No error compiling the classes but error when compiling the application

Reminder: Namespace of Classes

- C++ uses namespace as integral part of a class, function, data member

- Any quantity declared within a namespace can be accessed ONLY by using the scope operator :: and by specifying its namespace

- When using a new class, you must look into its header file to find out which namespace it belongs to
  - There are no shortcuts!

- When implementing a class you must specify its namespace
  - Unless you use the using directive

Another Example of Namespace

```cpp
#ifndef CounterNS_h_
#define CounterNS_h_
#include <string>

namespace rome {
  namespace didattica {

    class Counter {
      public:
        Counter(const std::string& name);
        ~Counter();
        int value();
        void reset();
        void increment(int step =1);
        void print();

      private:
        int count_;
        std::string name_;
    }; // class counter

  } // namespace didattica
} //namespace rome
#endif
```

```cpp
#include "CounterNS.h"

int main() {
  rome::didattica::Counter c1("c1");
  c1.print();
  return 0;
}
```

```cpp
// CounterNS.cc
#include "CounterNS.h"

// include any additional heade files needed in the class
// definition
#include <iostream> // needed for input/output
using std::cout;
using std::endl;
using namespace rome::didattica;

Counter::Counter(const std::string& name) {
  count_ = 0;
  name_ = name;
  cout << "Counter::Counter() called for Counter "<<
name_ << endl;
};

Counter::~Counter() {
  cout << "Counter::~Counter() called for Counter "<<
name_ << endl;
};

int Counter::value()  {
  return count_;
}

void Counter::reset() {
  count_ = 0;
}

void Counter::increment(int step) {
  count_ = count_+step;
}

void Counter::print() {
  cout << "Counter::print(): name: " << name_ << "
value: " << count_ << endl;
}
```