Header Files

Classes and Applications

- With several, elaborate classes, namespaces, functions, etc., it is **inconvenient to use a single file**
  - Hundreds of lines of code in a single place become unmanageable (or at the very least management becomes error prone)
  - Difficult and time consuming to navigate one file to make targeted modifications
  - Any change requires recompiling the whole code

Classes and Applications

- With several, elaborate classes, namespaces, functions, etc., it is **inconvenient to use a single file**
  - Hundreds of lines of code in a single place become unmanageable (or at the very least management becomes error prone)
  - Difficult and time consuming to navigate one file to make targeted modifications
  - Any change requires recompiling the whole code
- **Good practices**
  - Separate classes from applications
  - Use `#include` directive to add all classes needed by an application
  - Compile your classes separately
  - Include compiled classes (or libraries) when linking to the application

Naïve Attempt

Loosely based on `examples/04/Class7.cpp`

```
//NaiveDatum.cc
#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_;
};
```

```
//Putative application file
#include "NaiveDatum.cc"

int main() {

  Datum d1;
  d1.print();

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

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

  return 0;
}
```

- This separates the class from the applications, but the class is not compiled separately
- When the application is compiled, we are not including the compiled class by linking it to the application, but compiling it with the application
- We also exposed to the user the implementation of all methods!

Naïve Attempt

Loosely based on `examples/04/Class7.cpp`

```
//NaiveDatum.cc
#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_;
};
```

```
//Putative application file
#include "NaiveDatum.cc"

int main() {

  Datum d1;
  d1.print();

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

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

  return 0;
}
```

- This separates the class from the applications, but the class is not compiled separately
- When the application is compiled, we are not including the compiled class by linking it to the application, but compiling it with the application
- We also exposed to the user the implementation of all methods!

BAD PRACTICE – Name of class different from filename

DISCLAIMER – this is done in the `examples` and slides only for you to compare changes across files more easily

Separating Interface from Implementation

- Users only need to know and to rely on:

  1. the interface of classes

  2. the public members of classes

Separating Interface from Implementation

- Users only need to know and to rely on:

  1. the interface of classes
     - “users” includes co-developers of a big project – incremental development does not require knowledge of all details of the pre-established classes
  2. the public members of classes
     - internal data structure must be hidden and not needed in applications

Separating Interface from Implementation

- Users only need to know and to rely on:

  1. the interface of classes
     - “users” includes co-developers of a big project – incremental development does not require knowledge of all details of the pre-established classes
  2. the public members of classes
     - internal data structure must be hidden and not needed in applications

- Compiler only needs: the declaration of classes and their functions, and the signature of each function, i.e., the exact set of arguments passed to a function and its return type

Separating Interface from Implementation

- Users only need to know and to rely on:

  1. the interface of classes
     - “users” includes co-developers of a big project – incremental development does not require knowledge of all details of the pre-established classes

  2. the public members of classes
     - internal data structure must be hidden and not needed in applications

- Compiler only needs: the declaration of classes and their functions, and the signature of each function, i.e., the exact set of arguments passed to a function and its return type

- Linker only needs the compiled class code (definition): libraries are needed to link not to compile!

Header and Source Files

- Separate declarations of classes from their implementations
  - Declaration tells the compiler about data members (attributes) and member functions (functionalities) of a class
  - Declaration tells the user how many and what type of arguments a function has, without revealing how the function is implemented

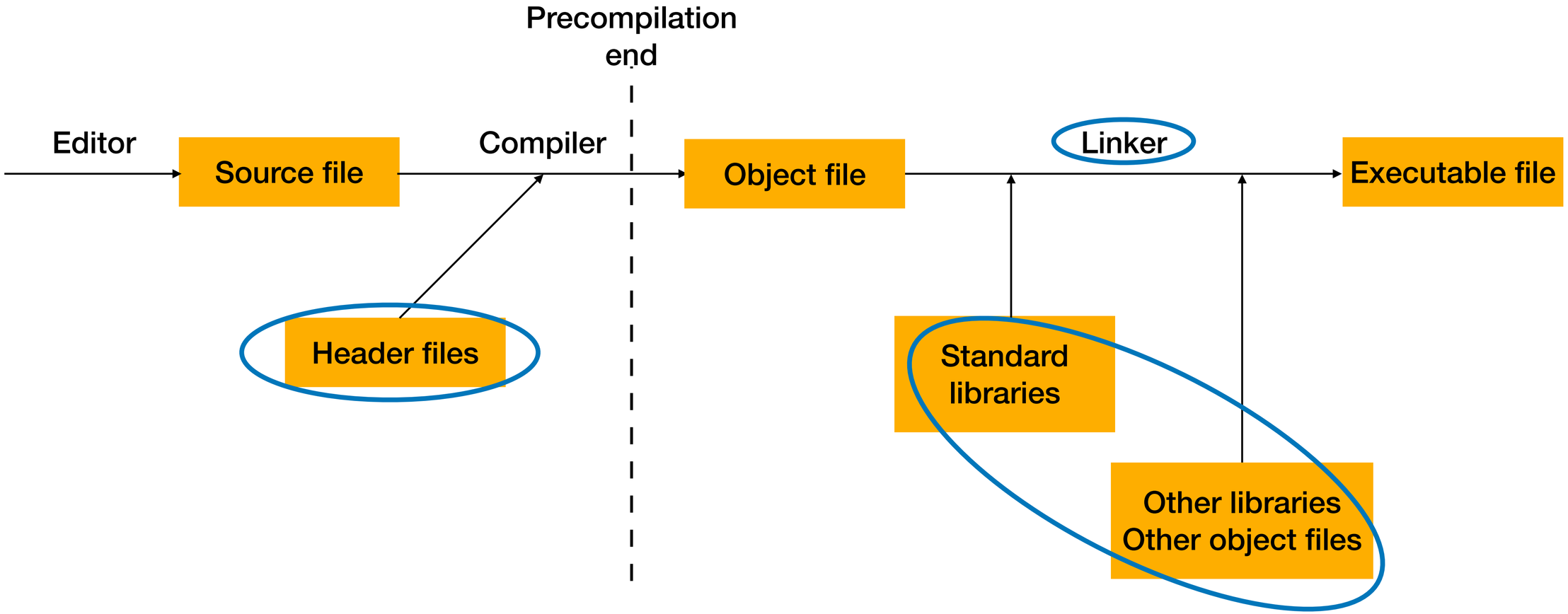
Header and Source Files

- Separate declarations of classes from their implementations
  - Declaration tells the compiler about data members (attributes) and member functions (functionalities) of a class
  - Declaration tells the user how many and what type of arguments a function has, without revealing how the function is implemented

- Common practice for an example class called `Datum`:
  - place declaration into a header file (`Datum.h` or `Datum.hh`)
  - place implementation of methods into a source file (`Datum.cc`)

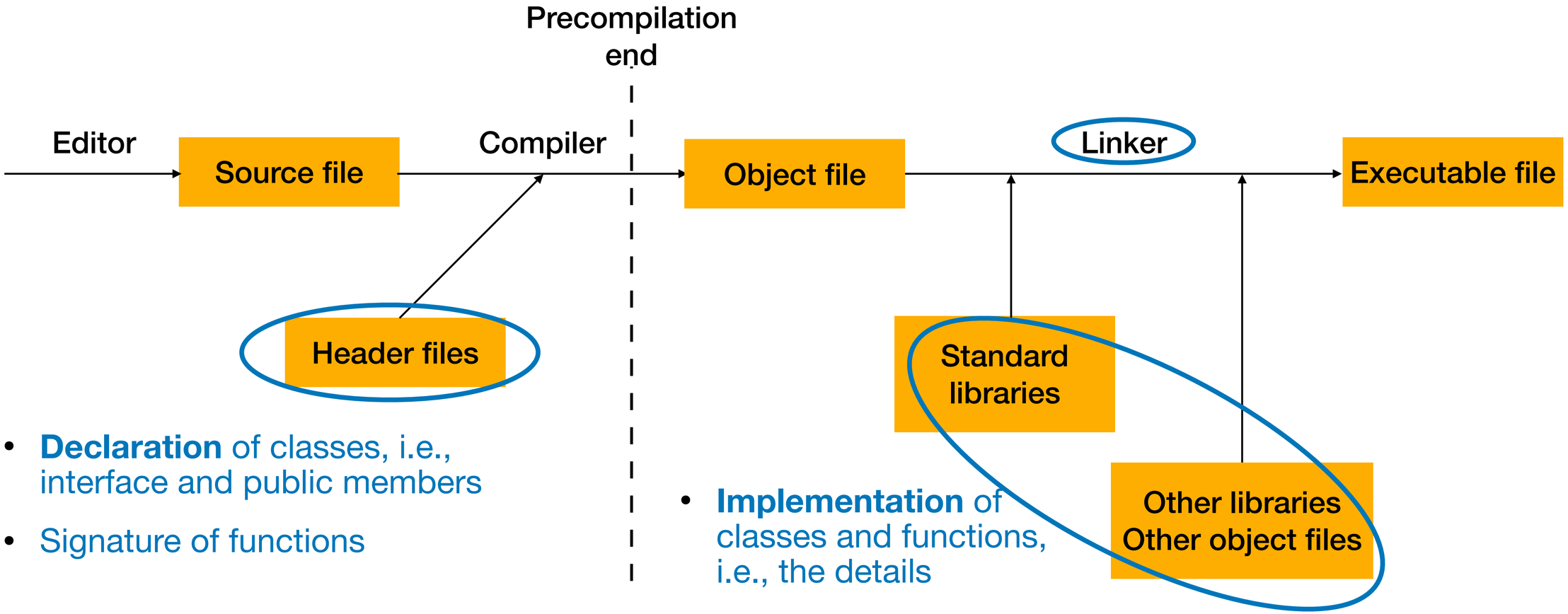
Remember this?

Part 1



Remember this?

Part 1

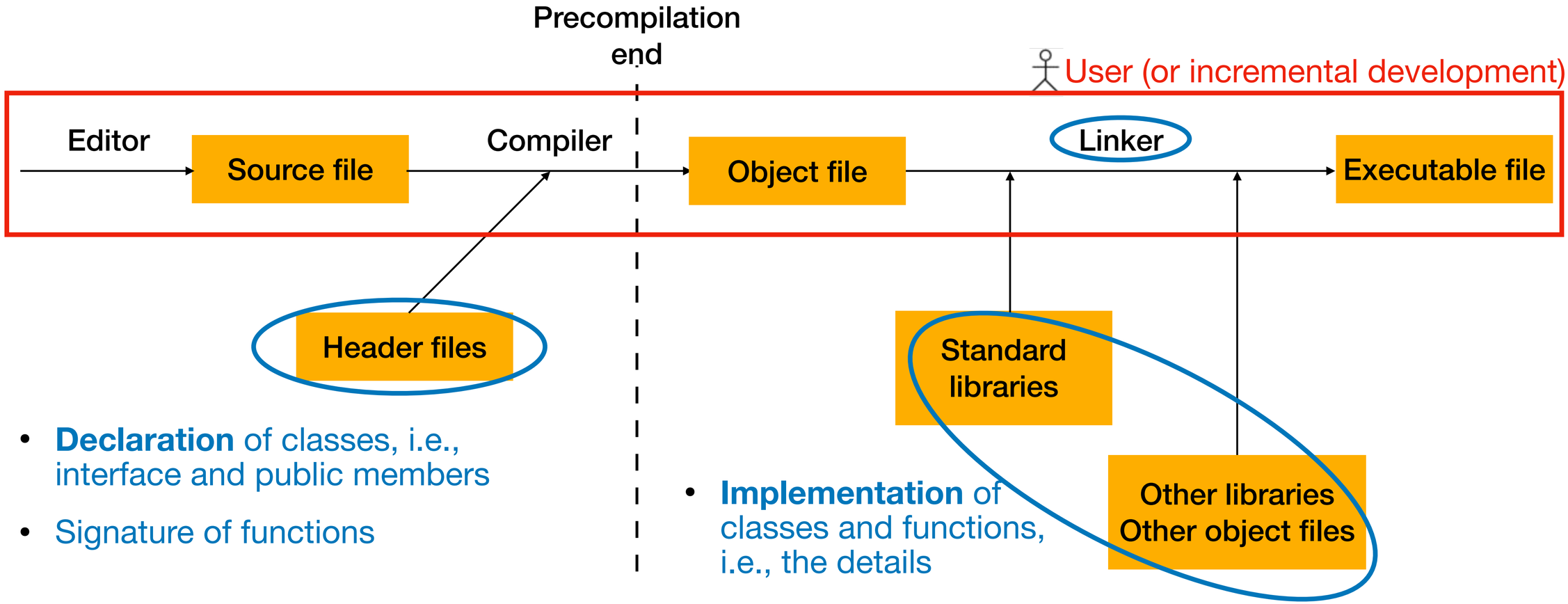


- **Declaration** of classes, i.e., interface and public members
- Signature of functions

- **Implementation** of classes and functions, i.e., the details
- Source/compiled code

Remember this?

Part 1



- **Declaration** of classes, i.e., interface and public members
- Signature of functions

- **Implementation** of classes and functions, i.e., the details
- Source/compiled code

Proper Attempt: Header File

Based on `examples/04/Datum.*`

```
// Datum.h
// header file of the Datum class

class Datum {
  public:
    // constructor
    Datum(double val, double error);

    // getters
    double value();
    double error();

    // setters
    void setValue(double value);
    void setError(double error);

    void print();

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

- 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
- When very simple methods are directly implemented in the header file, they are referred to as inline functions
  - Getter methods are a good candidate to become inline functions

Proper Attempt: Source File

Based on `examples/04/Datum.*`

```
// Datum.h
// header file of the Datum class

class Datum {
  public:
    // constructor
    Datum(double val, double error);

    // getters
    double value();
    double error();

    // setters
    void setValue(double value);
    void setError(double error);

    void print();

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

Proper Attempt: Source File

Based on `examples/04/Datum.*`

```
// Datum.h
// header file of the Datum class

class Datum {
  public:
    // constructor
    Datum(double val, double error);

    // getters
    double value();
    double error();

    // setters
    void setValue(double value);
    void setError(double error);

    void print();

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

```
// Datum.cc
// implementation of the Datum class

// include the class header file
#include "Datum.h"

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

// constructor
Datum::Datum(double val, double error) {
    value_ = val;
    error_ = error;
}

// getters
double Datum::value() { return value_; }
double Datum::error() { return error_; }

// setters
void Datum::setValue(double value) { value_ = value; }
void Datum::setError(double error) { error_ = error; }

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

The scope operator `::` is used to identify methods of a class

- Includes header file of the class being implemented
  - Compiler needs the prototype (declaration) of the methods
- Methods declared in header file
  - Scope operator `::` must be used to tell the compiler the implemented methods belong to a class
- Includes header files for all additional types used in the implementation but not needed in the header
  - Header files included in the header file of the class are automatically included in the source file

Proper Attempt: Application File

Based on `examples/04/Datum.*`

```
// Datum.h
// header file of the Datum class

class Datum {
  public:
    // constructor
    Datum(double val, double error);

    // getters
    double value();
    double error();

    // setters
    void setValue(double value);
    void setError(double error);

    void print();

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

```
// Datum.cc
// implementation of the Datum class

// include the class header file
#include "Datum.h"

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

// constructor
Datum::Datum(double val, double error) {
    value_ = val;
    error_ = error;
}

// getters
double Datum::value() { return value_; }
double Datum::error() { return error_; }

// setters
void Datum::setValue(double value) { value_ = value; }
void Datum::setError(double error) { error_ = error; }

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

The scope operator `::` is used to identify methods of a class

Proper Attempt: Application File

Based on `examples/04/Datum.*`

```
// Datum.h
// header file of the Datum class

class Datum {
  public:
    // constructor
    Datum(double val, double error);

    // getters
    double value();
    double error();

    // setters
    void setValue(double value);
    void setError(double error);

    void print();

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

```
// Datum.cc
// implementation of the Datum class

// include the class header file
#include "Datum.h"

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

// constructor
Datum::Datum(double val, double error) {
    value_ = val;
    error_ = error;
}

// getters
double Datum::value() { return value_; }
double Datum::error() { return error_; }

// setters
void Datum::setValue(double value) { value_ = value; }
void Datum::setError(double error) { error_ = error; }

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

The scope operator :: is used to identify methods of a class

```
// Datum.cpp
// example of an application
// of the Datum class

#include "Datum.h"

int main() {

  Datum d1(23.4,7.5);
  d1.print();

  d1.setValue( 8.563 );
  d1.setError( 0.45 );
  d1.print();

  return 0;
}
```

Includes the class header files

Working with the Proper Attempt

Theory

1. Compiling translates from code in high-level language to binary code that system can use

2. Linking puts together binary pieces corresponding to methods used in the `main` function to produce application, the end product

Working with the Proper Attempt

Theory

1. Compiling translates from code in high-level language to binary code that system can use

2. Linking puts together binary pieces corresponding to methods used in the `main` function to produce application, the end product

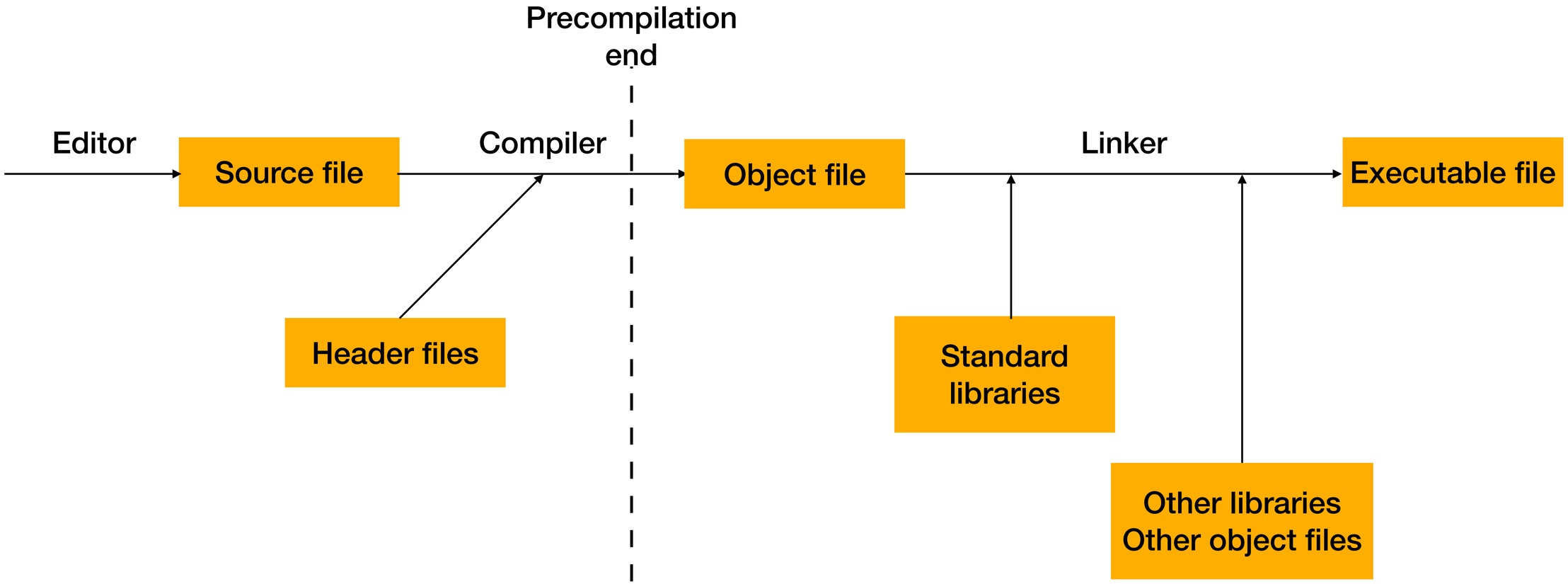
- When compiling class implementations, we do not have the `main` (we do not want to produce an application), so we must not do the linking

  1. Compile the class(es) with no linking

  2. Compile the main code, linking the compiled class(es)

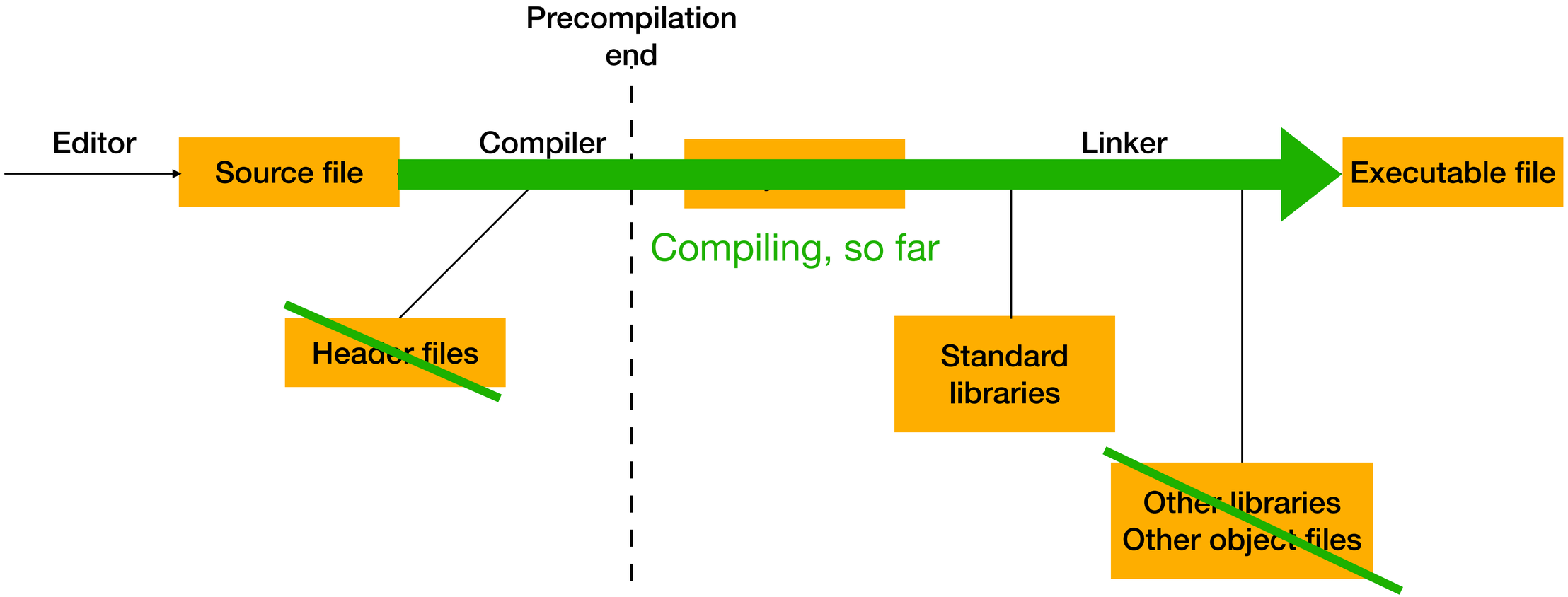
Remember this?

Part 2



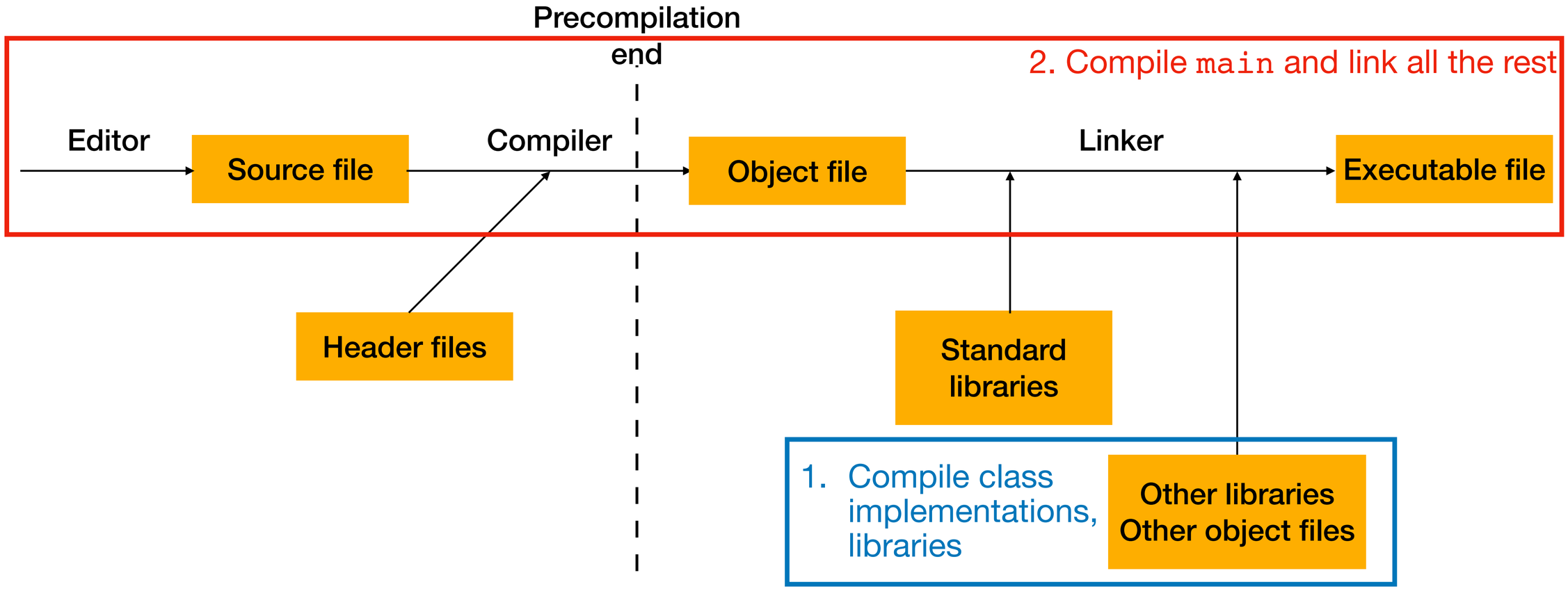
Remember this?

Part 2



Remember this?

Part 2



Working with the Proper Attempt

Instructions

- `g++` by default looks for a `main`: this becomes the program to run once the compiler is done linking the binary application
- `g++ Datum.cc` will therefore fail  Try this!

➡ Add the flag `-c` to require compilation only

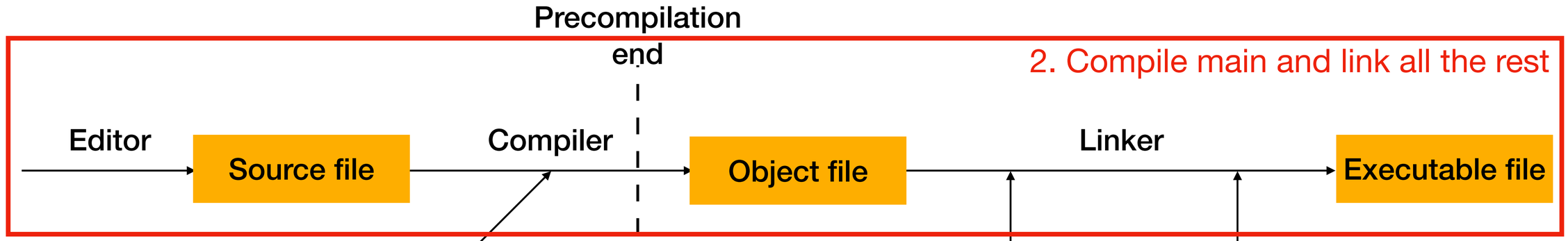
- When dealing with `Datum.cpp`, how do we tell `g++` to link `Datum.o` and avoid failures?

➡ List all needed object files along with the main source code

```
$ g++ -c Datum.cc
$ ls
Datum.cc  Datum.cpp  Datum.h  Datum.o
$ g++ -o Datum Datum.cpp Datum.o
$ ls
Datum  Datum.cc  Datum.cpp  Datum.h  Datum.o
$./Datum
datum: 23.4 +/- 7.5
datum: 8.563 +/- 0.45
```

Working with the Proper Attempt

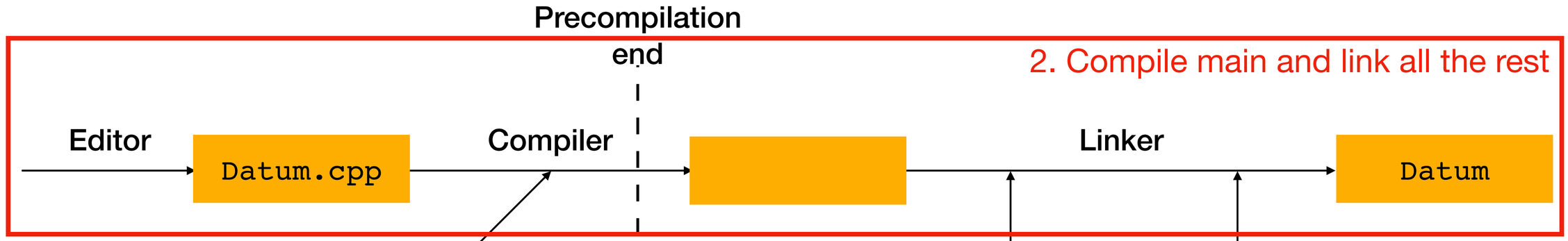
Summary




```
$ g++ -c Datum.cc
$ ls
Datum.cc   Datum.cpp   Datum.h   Datum.o
$ g++ -o Datum Datum.cpp Datum.o
$ ls
Datum   Datum.cc   Datum.cpp   Datum.h   Datum.o
```

Working with the Proper Attempt

Summary




```
$ g++ -c Datum.cc
$ ls
Datum.cc   Datum.cpp   Datum.h   Datum.o
$ g++ -o Datum Datum.cpp Datum.o
$ ls
Datum   Datum.cc   Datum.cpp   Datum.h   Datum.o
```

`#define`, `#ifndef`, and `#endif` Directives

Based on `examples/04/Datum.*`

- What if we include the same header file several times? E.g.,
  - `App.cpp` includes both `Foo.h` and `Bar.h`
  - `Foo.h` is included in `Bar.h` and `bar.cc`
  - Or simply:

```
// Datum.cpp

#include "Datum.h"
#include "Datum.h"

int main() {

  Datum d1(23.4,7.5);
  d1.print();

  d1.setValue( 8.563 );
  d1.setError( 0.45 );
  d1.print();

  return 0;
}
```

Watch `g++` complain with the double include

`#define`, `#ifndef`, and `#endif` Directives

Based on `examples/04/Datum.*`

- What if we include the same header file several times? E.g.,
  - `App.cpp` includes both `Foo.h` and `Bar.h`
  - `Foo.h` is included in `Bar.h` and `bar.cc`
  - Or simply:

```
// Datum.cpp

#include "Datum.h"
#include "Datum.h"

int main() {

  Datum d1(23.4,7.5);
  d1.print();

  d1.setValue( 8.563 );
  d1.setError( 0.45 );
  d1.print();

  return 0;
}
```

Watch g++ complain with the double include

```
// Datum.h
// header file of the Datum class

#ifndef Datum_h // if Datum_h is not defined, then...
#define Datum_h // ...define the new variable datum_h
class Datum {
  public:
    // constructor
    Datum(double val, double error);

    // getters
    double value();
    double error();

    // setters
    void setValue(double value);
    void setError(double error);

    void print();

  private:
    double value_;
    double error_;
};
#endif // end of ifndef block of code
```

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

Namespace of Classes

Based on `examples/04/DatumNamespace.*`

```
// DatumNamepace.h
// header file of the Datum class

#ifndef Datum_h // if Datum_h not def...
#define Datum_h // ...define datum_h

namespace rome{
  namespace teaching{
    class Datum {
      public:
        // constructor
        Datum(double val, double error);

        // getters
        double value();
        double error();

        // setters
        void setValue(double value);
        void setError(double error);

        void print();

      private:
        double value_;
        double error_;
    };
  } // namespace teaching
} //namespace rome
#endif // end of ifndef block of code
```

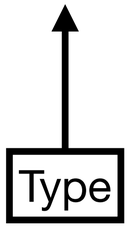
```
// DatumNamespace.cc
// implementation of the Datum class

// include the class header file
#include "DatumNamespace.h"

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

// constructor
Datum::Datum(double val, double error) {
    value_ = val;
    error_ = error;
}

// getters
double Datum::value() { return value_; }
double Datum::error() { return error_; }

// setters
void Datum::setValue(double value) { value_ = value; }
void Datum::setError(double error) { error_ = error; }

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

```
// DatumNamespace.cpp
// example of an application
// of the Datum class

#include "DatumNamespace.h"

int main() {

  rome::teaching::Datum d1(23.4,7.5);
  d1.print();

  d1.setValue( 8.563 );
  d1.setError( 0.45 );
  d1.print();

  return 0;
}
```

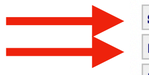
The `std::vector` Class

Interface of `std::vector<T>`

http://www.cplusplus.com/reference/vector/vector/
https://en.cppreference.com/w/cpp/container/vector



- Sequence container representing arrays that can change in size
- Consumes more memory in exchange for the ability to manage storage and grow dynamically in an efficient way
- Very efficient accessing its elements (just like arrays) and relatively efficient adding or removing elements from its end
- Operations that involve inserting/ removing elements at positions other than the end, perform worse



Member functions

(constructor)	Construct vector (public member function)
(destructor)	Vector destructor (public member function)
operator=	Assign content (public member function)

Iterators:

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function)
rend	Return reverse iterator to reverse end (public member function)
cbegin C++11	Return const_iterator to beginning (public member function)
cend C++11	Return const_iterator to end (public member function)
crbegin C++11	Return const_reverse_iterator to reverse beginning (public member function)
crend C++11	Return const_reverse_iterator to reverse end (public member function)

Capacity:

size	Return size (public member function)
max_size	Return maximum size (public member function)
resize	Change size (public member function)
capacity	Return size of allocated storage capacity (public member function)
empty	Test whether vector is empty (public member function)
reserve	Request a change in capacity (public member function)
shrink_to_fit C++11	Shrink to fit (public member function)

Element access:

operator[]	Access element (public member function)
at	Access element (public member function)
front	Access first element (public member function)
back	Access last element (public member function)
data C++11	Access data (public member function)

Modifiers:

assign	Assign vector content (public member function)
push_back	Add element at the end (public member function)
pop_back	Delete last element (public member function)
insert	Insert elements (public member function)
erase	Erase elements (public member function)
swap	Swap content (public member function)
clear	Clear content (public member function)
emplace C++11	Construct and insert element (public member function)
emplace_back C++11	Construct and insert element at the end (public member function)

Class `std::vector<T>`

Based on `examples/04/vector1.cpp`

- Interfacing classes: vector of Datum types!
- Using public members of Datum only, treating vector as an array [except for method `size()`]
- Fully exploiting both classes

```
#include <iostream>
#include <vector>
#include "Datum.h"

int main() {

  // create vectors with values and errors
  std::vector<double> vals, errs;
  vals.push_back(1.3);
  vals.push_back(-2.1);
  errs.push_back(0.2);
  errs.push_back(0.3);

  // or interface vector and Datum!
  std::vector<Datum> data;
  data.push_back( Datum(1.3, 0.2) );
  data.push_back( Datum(-2.1, 0.3) );

  std::cout << "# dati:: " << data.size() << std::endl;

  // print data using traditional loop on an array
  int i=0;
  std::cout << "Using [] operator on vector" << std::endl;
  for(i=0; i<data.size(); i++) {
    std::cout << "i: " << i+1
         << "\t data: " << data[i].value() << " +/- " << data[i].error()
         << std::endl;
  }

  // print data using vector iterator
  i=0;
  std::cout << "std::vector<T>::iterator " << std::endl;
  for(std::vector<Datum>::iterator d = data.begin(); d != data.end(); d++) {
    i++;
    std::cout << "i: " << i
         << "\t data: " << d->value() << " +/- " << d->error()
         << std::endl;
  }

  // print data using vector iterator
  i=0;
  std::cout << "C++11 extension feature " << std::endl;
  for(Datum data_itr : data) {
    i++;
    std::cout << "i: " << i
         << "\t data: " << data_itr.value() << " +/- " << data_itr.error()
         << std::endl;
  }

  return 0;

}
```

Class `std::vector<T>`

Based on `examples/04/vector1.cpp`

- Interfacing classes: vector of Datum types!
- Using public members of Datum only, treating vector as an array [except for method `size()`]
- Fully exploiting both classes

```
$ g++ -o vector1 vector1.cpp Datum.cc
vector1.cpp:43:22: warning: range-based for loop is a C++11 extension [-Wc++11-extensions]
  for(Datum data_itr : data) {
                     ^
1 warning generated.
$ ./vector1
# dati:: 2
Using [] operator on vector
i: 1              data: 1.3 +/- 0.2
i: 2              data: -2.1 +/- 0.3
std::vector<T>::iterator
i: 1              data: 1.3 +/- 0.2
i: 2              data: -2.1 +/- 0.3
C++11 extension feature
i: 1              data: 1.3 +/- 0.2
i: 2              data: -2.1 +/- 0.3
```

```
#include <iostream>
#include <vector>
#include "Datum.h"

int main() {

  // create vectors with values and errors
  std::vector<double> vals, errs;
  vals.push_back(1.3);
  vals.push_back(-2.1);
  errs.push_back(0.2);
  errs.push_back(0.3);

  // or interface vector and Datum!
  std::vector<Datum> data;
  data.push_back( Datum(1.3, 0.2) );
  data.push_back( Datum(-2.1, 0.3) );

  std::cout << "# dati:: " << data.size() << std::endl;

  // print data using traditional loop on an array
  int i=0;
  std::cout << "Using [] operator on vector" << std::endl;
  for(i=0; i<data.size(); i++) {
    std::cout << "i: " << i+1
         << "\t data: " << data[i].value() << " +/- " << data[i].error()
         << std::endl;
  }

  // print data using vector iterator
  i=0;
  std::cout << "std::vector<T>::iterator " << std::endl;
  for(std::vector<Datum>::iterator d = data.begin(); d != data.end(); d++) {
    i++;
    std::cout << "i: " << i
         << "\t data: " << d->value() << " +/- " << d->error()
         << std::endl;
  }

  // print data using vector iterator
  i=0;
  std::cout << "C++11 extension feature " << std::endl;
  for(Datum data_itr : data) {
    i++;
    std::cout << "i: " << i
         << "\t data: " << data_itr.value() << " +/- " << data_itr.error()
         << std::endl;
  }

  return 0;

}
```

Class `std::vector<T>`

Based on `examples/04/vector1.cpp`

- Interfacing classes: vector of Datum types!
- Using public members of Datum only, treating vector as an array [except for method `size()`]
- Fully exploiting both classes

```
$ g++ -o vector1 vector1.cpp Datum.cc
vector1.cpp:43:22: warning: range-based for loop is a C++11 extension [-Wc++11-extensions]
  for(Datum data_itr : data) {
                     ^
1 warning generated.
$ ./vector1
# dati:: 2
Using [] operator on vector
i: 1              data: 1.3 +/- 0.2
i: 2              data: -2.1 +/- 0.3
std::vector<T>::iterator
i: 1              data: 1.3 +/- 0.2
i: 2              data: -2.1 +/- 0.3
C++11 extension feature
i: 1              data: 1.3 +/- 0.2
i: 2              data: -2.1 +/- 0.3
```

Equivalent instructions

```
$ g++ -c Datum.cc
$ g++ -o vector1 vector1.cpp Datum.o
$ ./vector1
```

```
#include <iostream>
#include <vector>
#include "Datum.h"

int main() {

  // create vectors with values and errors
  std::vector<double> vals, errs;
  vals.push_back(1.3);
  vals.push_back(-2.1);
  errs.push_back(0.2);
  errs.push_back(0.3);

  // or interface vector and Datum!
  std::vector<Datum> data;
  data.push_back( Datum(1.3, 0.2) );
  data.push_back( Datum(-2.1, 0.3) );

  std::cout << "# dati:: " << data.size() << std::endl;

  // print data using traditional loop on an array
  int i=0;
  std::cout << "Using [] operator on vector" << std::endl;
  for(i=0; i<data.size(); i++) {
    std::cout << "i: " << i+1
         << "\t data: " << data[i].value() << " +/- " << data[i].error()
         << std::endl;
  }

  // print data using vector iterator
  i=0;
  std::cout << "std::vector<T>::iterator " << std::endl;
  for(std::vector<Datum>::iterator d = data.begin(); d != data.end(); d++) {
    i++;
    std::cout << "i: " << i
         << "\t data: " << d->value() << " +/- " << d->error()
         << std::endl;
  }

  // print data using vector iterator
  i=0;
  std::cout << "C++11 extension feature " << std::endl;
  for(Datum data_itr : data) {
    i++;
    std::cout << "i: " << i
         << "\t data: " << data_itr.value() << " +/- " << data_itr.error()
         << std::endl;
  }

  return 0;

}
```

Using `std::vector<T>` in functions

Based on `examples/04/vector2.cpp` (and `MeanStdDev.*`)

```
#include "Datum.h"
#include "MeanStdDev.h"

using std::vector;

int main(){

  std::vector<Datum> data;
  data.push_back( Datum(1.3, 0.2) );
  data.push_back( Datum(-2.1, 0.3) );

  Datum m_and_sd = mean_and_stdDev(data);
  m_and_sd.print();

  return 0;
}
```

```
#include <vector>

using std::vector;

// returns mean of values in data
// passing by constant reference to ensure
// data cannot be changed within the function
double mean(const vector<Datum>& data);

// returns a Datum containing
// value = mean(data)
// error = standard deviation of data values
// passing by constant reference to ensure
// data cannot be changed within the function
Datum mean_and_stdDev(const vector<Datum>& data);
```

- Combining a lot of what we have seen so far
- `vector` can be argument of functions (by value, pointer or reference)
- Declaration of functions with `vector<Datum>` arguments does not require `Datum.h`

Using `std::vector<T>` in functions

Based on `examples/04/vector2.cpp` (and `MeanStdDev.*`)

```
#include "Datum.h"
#include "MeanStdDev.h"

using std::vector;

int main(){

  std::vector<Datum> data;
  data.push_back( Datum(1.3, 0.2) );
  data.push_back( Datum(-2.1, 0.3) );

  Datum m_and_sd = mean_and_stdDev(data);
  m_and_sd.print();

  return 0;
}
```

```
#include <vector>

using std::vector;

// returns mean of values in data
// passing by constant reference to ensure
// data cannot be changed within the function
double mean(const vector<Datum>& data);

// returns a Datum containing
// value = mean(data)
// error = standard deviation of data values
// passing by constant reference to ensure
// data cannot be changed within the function
Datum mean_and_stdDev(const vector<Datum>& data);
```

```
#include <cmath>
#include "Datum.h"
#include "MeanStdDev.h"


double mean(const vector<Datum>& data) {

  double m(0.); // same as = 0.

  // loop over data
  for(Datum data_itr : data){
    m += data_itr.value();
  }
  // compute average: finds out data size!
  m /= data.size();

  return m;
}


Datum mean_and_stdDev(const vector<Datum>& data) {

  double m = mean(data), stdDev(0.);

  // loop over data
  for(Datum data_itr : data){
    stdDev += pow(data_itr.value()-m, 2.);
  }
  // divide by n-1
  stdDev /= (data.size()-1);
  // finally compute the stdDev
  stdDev = sqrt( stdDev );

  return Datum(m, stdDev);
}
```

- Combining a lot of what we have seen so far
- `vector` can be argument of functions (by value, pointer or reference)
- Declaration of functions with `vector<Datum>` arguments does not require `Datum.h`
- Function implementations exploit methods from both `vector` and `Datum`

Using `std::vector<T>` in functions

Based on `examples/04/vector2.cpp` (and `MeanStdDev.*`)

```
#include "Datum.h"
#include "MeanStdDev.h"

using std::vector;

int main(){

  std::vector<Datum> data;
  data.push_back( Datum(1.3, 0.2) );
  data.push_back( Datum(-2.1, 0.3) );

  Datum m_and_sd = mean_and_stdDev(data);
  m_and_sd.print();

  return 0;
}
```

```
#include <vector>

using std::vector;

// returns mean of values in data
// passing by constant reference to ensure
// data cannot be changed within the function
double mean(const vector<Datum>& data);

// returns a Datum containing
// value = mean(data)
// error = standard deviation of data values
// passing by constant reference to ensure
// data cannot be changed within the function
Datum mean_and_stdDev(const vector<Datum>& data);
```

```
#include <cmath>
#include "Datum.h"
#include "MeanStdDev.h"


double mean(const vector<Datum>& data) {

  double m(0.); // same as = 0.

  // loop over data
  for(Datum data_itr : data){
    m += data_itr.value();
  }
  // compute average: finds out data size!
  m /= data.size();

  return m;
}


Datum mean_and_stdDev(const vector<Datum>& data) {

  double m = mean(data), stdDev(0.);

  // loop over data
  for(Datum data_itr : data){
    stdDev += pow(data_itr.value()-m, 2.);
  }
  // divide by n-1
  stdDev /= (data.size()-1);
  // finally compute the stdDev
  stdDev = sqrt( stdDev );

  return Datum(m, stdDev);
}
```

- Combining a lot of what we have seen so far
- `vector` can be argument of functions (by value, pointer or reference)
- Declaration of functions with `vector<Datum>` arguments does not require `Datum.h`
- Function implementations exploit methods from both `vector` and `Datum`
- Suppressing warnings

```
$ g++ -w -o vector2 vector2.cpp Datum.cc MeanStdDev.cc
$ ./vector2
datum: -0.4 +/- 2.40416
```

Constant Member Functions, Default Values for Function Parameters

Constant Member Functions

Overview

- Enforce principle of least privilege, i.e., grant privilege iff needed
- They cannot
  1. modify data members
  2. be called on non-constant objects
- They tell user the function only “uses” input data or data members, but makes no changes to data members
- Pay attention to which functions can be called on which objects; objects can be constant, but:
  1. you cannot modify a constant object
  2. calling non-constant methods on constant objects does not make sense

Constant Member Functions

Based on `examples/04/DatumConst.*`

```
// DatumConst.h
#ifndef DatumConst_h
#define DatumConst_h

#include <iostream>
using namespace std;

class Datum {
  public:
    // constructors
    Datum();
    Datum(double val, double err);
    Datum(const Datum& datum);

    // getters
    double value() { return value_; }
    double error() { return error_; }

    // setters
    void setValue(double val) { value_ = val; }
    void setError(double err) { error_ = err; }

    double significance();
    void print(const std::string& comment);

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

```
//DatumConst.cc
#include "DatumConst.h"
#include <iostream>

Datum::Datum() {
  value_ = 0.;
  error_ = 0.;
}
Datum::Datum(double val, double err) {
  value_ = err;
  error_ = val;
}
Datum::Datum(const Datum& datum) {
  value_ = datum.value_;
  error_ = datum.error_;
}
double Datum::significance() {
  return value_/error_;
}
void Datum::print(const std::string&
comment) {
  using namespace std;
  cout << comment << ": " << value_
       << " +/- " << error_ << endl;
}
```

Which methods could become constant? **All methods that only return a value and do not change object attributes**

Constant Member Functions

Based on `examples/04/DatumConst.*`

```
// DatumConst.h
#ifndef DatumConst_h
#define DatumConst_h

#include <iostream>
using namespace std;

class Datum {
  public:
    // constructors
    Datum();
    Datum(double val, double err);
    Datum(const Datum& datum);

    // getters
    double value() { return value_; }
    double error() { return error_; }

    // setters
    void setValue(double val) { value_ = val; }
    void setError(double err) { error_ = err; }

    double significance();
    void print(const std::string& comment);

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

```
//DatumConst.cc
#include "DatumConst.h"
#include <iostream>

Datum::Datum() {
  value_ = 0.;
  error_ = 0.;
}
Datum::Datum(double val, double err) {
  value_ = err;
  error_ = val;
}
Datum::Datum(const Datum& datum) {
  value_ = datum.value_;
  error_ = datum.error_;
}
double Datum::significance() {
  return value_/error_;
}
void Datum::print(const std::string&
comment) {
  using namespace std;
  cout << comment << ": " << value_
       << " +/- " << error_ << endl;
}
```

Which methods could become constant? **All methods that only return a value and do not change object attributes**

➡ Therefore all getters

➡ Prints, as they must not change data

➡ Setters can never be constant, because they modify data members by definition

➡ Similarly constructors (and destructors, which we will soon see) cannot be constant

Constant Member Functions

Based on `examples/04/DatumConst.*`

```
// DatumConst.h
#ifndef DatumConst_h
#define DatumConst_h

#include <iostream>
using namespace std;

class Datum {
  public:
    // constructors
    Datum();
    Datum(double val, double err);
    Datum(const Datum& datum);

    // getters
    double value() const { return value_; }
    double error() const { return error_; }

    // setters
    void setValue(double val) { value_ = val; }
    void setError(double err) { error_ = err; }

    double significance() const;
    void print(const std::string& comment) const;

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

```
//DatumConst.cc
#include "DatumConst.h"
#include <iostream>

Datum::Datum() {
  value_ = 0.;
  error_ = 0.;
}
Datum::Datum(double val, double err) {
  value_ = err;
  error_ = val;
}
Datum::Datum(const Datum& datum) {
  value_ = datum.value_;
  error_ = datum.error_;
}
double Datum::significance() const {
  return value_/error_;
}
void Datum::print(const std::string&
comment) const {
  using namespace std;
  cout << comment << ": " << value_
       << " +/- " << error_ << endl;
}
```

Which methods could become constant? **All methods that only return a value and do not change object attributes**

➡ Therefore all getters

➡ Prints, as they must not change data

➡ Setters can never be constant, because they modify data members by definition

➡ Similarly constructors (and destructors, which we will soon see) cannot be constant

Constant Member Functions

Based on `examples/04/DatumConst.*`

```
// DatumConst.h
#ifndef DatumConst_h
#define DatumConst_h

#include <iostream>
using namespace std;

class Datum {
  public:
    // constructors
    Datum();
    Datum(double val, double err);
    Datum(const Datum& datum);

    // getters
    double value() const { return value_; }
    double error() const { return error_; }

    // setters
    void setValue(double val) { value_ = val; }
    void setError(double err) { error_ = err; }

    double significance() const;
    void print(const std::string& comment) const;

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

```
//DatumConst.cc
#include "DatumConst.h"
#include <iostream>

Datum::Datum() {
  value_ = 0.;
  error_ = 0.;
}
Datum::Datum(double val, double err) {
  value_ = err;
  error_ = val;
}
Datum::Datum(const Datum& datum) {
  value_ = datum.value_;
  error_ = datum.error_;
}
double Datum::significance() const {
  return value_/error_;
}
void Datum::print(const std::string&
comment) const {
  using namespace std;
  cout << comment << ": " << value_
       << " +/- " << error_ << endl;
}
```

```
//DatumConst.cpp

#include "DatumConst.h"

int main() {

  Datum d1(-67.03, 32.12 );
  const Datum d2(-67.03, 32.12 );

  d1.print("datum");

  d2.print("const datum");

  return 0;
}
```

```
$ g++ -o DatumConst
DatumConst.cpp DatumConst.cc
$ ./DatumConst
datum: 32.12 +/- -67.03
const datum: 32.12 +/- -67.03
```

Default Values for Function Parameters

Overview

- Functions (including member functions in classes) might be invoked with recurrent values for their arguments

- It is possible to provide default values for arguments of any function in C++
  - Default arguments must be provided the first time the name of the function occurs
    - In declaration if separate implementation
    - In definition if the function is declared and defined at the same time

- Default values can be specified only for the rightmost parameters in the list of arguments
  - If a parameter has a default value, all parameters to its right must have default values too

Default Values for Function Parameters

Based on `examples/04/Counter.*`

```
// Counter.h

#include <iostream>
using namespace std;

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

  private:
    int count_;
};
```

```
// Counter.cc
#include "Counter.h"

#include <iostream>

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;
}
```

Two `increment` methods with very similar functionality

- `increment()` is a special case of `increment(int step)` with `step=1`
- Why two different methods?

Default Values for Function Parameters

Based on `examples/04/Counter.*`

```
// Counter.h

#include <iostream>
using namespace std;

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

    void increment(int step=1);

  private:
    int count_;
};
```

```
// Counter.cc
#include "Counter.h"

#include <iostream>

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

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

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




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

Two `increment` methods with very similar functionality

- `increment()` is a special case of `increment(int step)` with `step=1`
- Why two different methods?

Default Values for Function Parameters

Based on `examples/04/Counter.*`

```
// Counter.h

#include <iostream>
using namespace std;

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

    void increment(int step=1);

  private:
    int count_;
};
```

```
// Counter.cc
#include "Counter.h"

#include <iostream>

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

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

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




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

```
// Counter.cpp
#include "Counter.h"

using namespace std;

int main() {
  Counter counter;
  cout << "counter: " << counter.value() << endl;

  // no argument
  counter.increment();
  cout << "counter: " << counter.value() << endl;

  // provide argument, same function
  counter.increment(14);
  cout << "counter: " << counter.value() << endl;

  return 0;
}
```

Two `increment` methods with very similar functionality

- `increment()` is a special case of `increment(int step)` with `step=1`
- Why two different methods?

```
$ g++ -o Counter Counter.cpp Counter.cc
$ ./Counter
counter: 0
counter: 1
counter: 15
```

Default Values for Function Parameters

Final remarks

```
void fun(int i, int b = 0, int x = 3){
  // some code
};

fun(3, 4);
// is the same as
fun(3, 4, 3);

fun(3);
// is the same as
fun(3, 0, 3);
```

- Upon calling a method, unspecified parameters are taken to be equal to their default values from right to left
- Do not abuse default values:
  - They must be used for functions with obvious default values
  - If default values are not intuitive for user, think twice before using them
  - Quite often different constructors correspond to DIFFERENT ways of creating an object, so default values could be misleading
  - If arguments are physical quantities ask yourself whether the default value is meaningful and useful for everyone