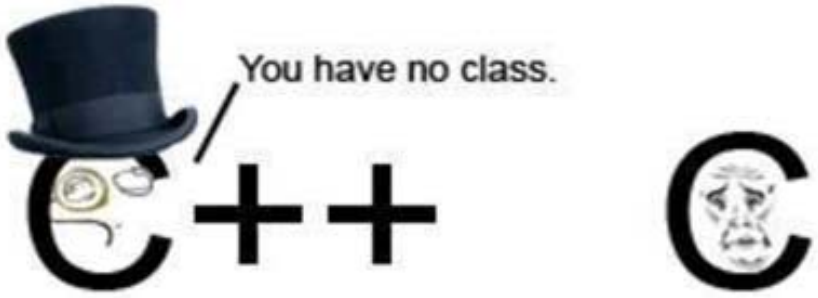
# Introduction to Class

Meng-Hsun Tsai
CSIE, NCKU



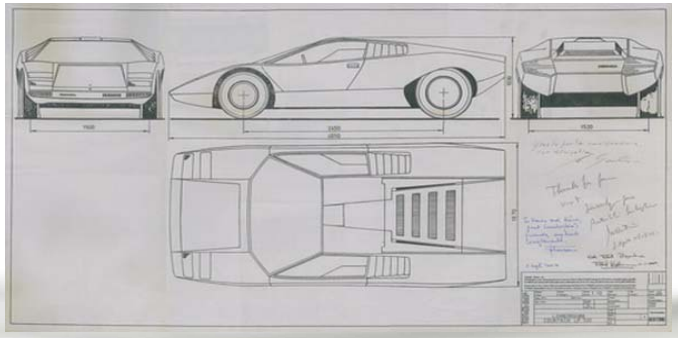
# Introduction

- In this lecture, you'll begin writing programs that employ the basic concepts of object-oriented programming.
- Typically, the programs you develop will consist of function `main` and one or more classes, each containing data members and member functions.

| Program | | |
|---|---|---|
| main ( )<br>function1 ( )<br>function2 ( )<br>function3 ( )<br>… | class1 | class2 |
| | int data1;<br>float data2;<br>function4 ( )<br>function5 ( ) | char data3;<br>double data4;<br>function6 ( )<br>function7 ( ) |

# Classes, Objects, Member Functions and Data Members

- Suppose you want to drive a car and make it go faster by pressing down on its accelerator pedal (gas pedal).
- Before you can drive a car, someone has to *design it and build it.*
- A car typically begins as engineering drawings that include the design for an accelerator pedal that makes the car go faster.
- The pedal "hides" the complex mechanisms that actually make the car go faster, just as the brake pedal "hides" the mechanisms that slow the car, the steering wheel "hides" the mechanisms that turn the car and so on.

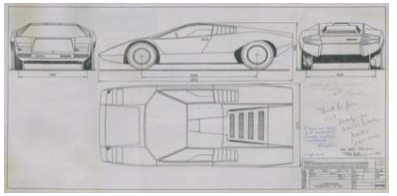


engineering drawings

# Classes, Objects, Member Functions and Data Members (cont.)

- A function belonging to a class is called a member function.
- In a class, you provide one or more member functions that are designed to perform the class's tasks.
- Just as you cannot drive an engineering drawing of a car, you cannot "drive" a class.
- *You must create an object of a class for your program to perform the tasks the class describes.*

```
class Car {
    accelerate();
    brake();
    turn();
};
```



class

object

# Classes, Objects, Member Functions and Data Members (cont.)

- You send messages to an object—each message is known as a member-function call and tells a member function of the object to perform its task.
- In addition to the capabilities a car provides, it also has many attributes, such as its color, the number of doors and its current speed.
- Every car maintains its own attributes. Similarly, an object has attributes that are carried with the object as it's used in a program.
- Attributes are specified by the class's data members.

```
class Car {
    accelerate();
    brake();
    turn();
};
Car NCKU;
```

```
NCKU.accelerate();
NCKU.brake();
```

```
class Car {
    string color;
};
Car NCKU;
NCKU.color = "red";
NCKU.color = "green";
```

# Defining a Class with a Member Function

- We begin with an example that consists of class `Timer` used to maintain elapsed time, and a `main` function which creates a `Timer` object.
- Function `main` uses this object and its member function to display a message on the screen.

# Class *Timer* with Member Function *hello()*

```
 1 #include <iostream>
 2 using namespace std;
 3
 4 class Timer {
 5    public:
 6       void hello()
 7       {
 8          cout << "Hello C++ !!!" << endl;
 9       }
10
11 };
12 int main()
13 {
14    Timer tmr;     // create object tmr|
15    tmr.hello();   // call member function hello()
16    return 0;
17 }
```

```
> g++ -o timer1 timer1.cpp
> ./timer1
Hello C++ !!!
>
```

# Outline of a Class Definition

- The `Timer` class definition (lines 4–10) begins with keyword `class` and contains a member function called `hello` (lines 6–9) that displays a message on the screen (line 8).
- Need to make an object of class `Timer` (line 14) and call its `hello` member function (line 15) to get line 8 to execute and display the welcome message.
- By convention, the name of a user-defined class begins with a capital letter, and for readability, each subsequent word in the class name begins with a capital letter.
- This capitalization style is often referred to as camel case.

```
class SuperMan {
        ...
};
```

# Outline of a Class Definition (cont.)

- Every class's body is enclosed in a pair of left and right braces (`{` and `}`), as in lines 4 and 10.
- The class definition terminates with a semicolon (line 10).
- The access-specifier label `public:` contains the keyword `public` as an access specifier.
  - Indicates that the function is "available to the public"—that is, it can be called by other functions in the program (such as `main`), and by member functions of other classes (if there are any).

```
 4 class Timer {
 5 public:
 6     void hello()
 7     {
 8         ...
 9     }
10 };
11
12 int main()
13 {
14     Timer tmr;
15     tmr.hello();
```

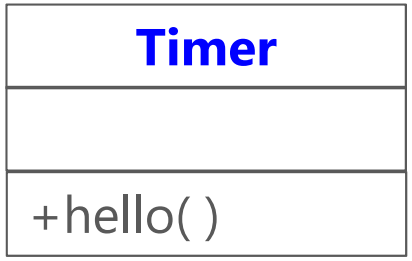
# Calling a Member Function

- Typically, you cannot call a member function of a class until you create an object of that class.
- First, create an object of class `Timer` called `tmr`.
- Call the member function `hello` by using variable `tmr` followed by the dot operator (`.`), the function name `hello` and an empty set of parentheses.

```
12  int main()
13  {
14      Timer tmr;
15      tmr.hello();
```

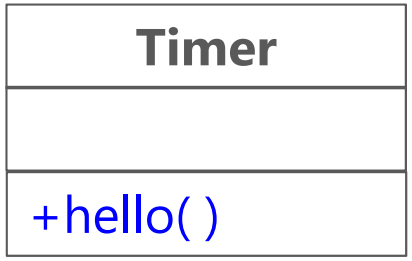
# UML Class Diagram

- In the UML, each class is modeled in a UML class diagram as a rectangle with three compartments.
- The top compartment contains the class's name centered horizontally and in boldface type.
- The middle compartment contains the class's attributes, which correspond to data members in C++.
  - Currently empty, because class `Timer` does not have any attribute.



# UML Class Diagram (cont.)

- The bottom compartment contains the class's operations, which correspond to member functions in C++.
- The UML models operations by listing the operation name followed by a set of parentheses.
- The plus sign (+) in front of the operation name indicates that `hello` is a public operation in the UML.



# Defining a Member Function with a Parameter

- Car analogy
  - Pressing a car's gas pedal sends a message to the car to perform a task—make the car go faster.
  - But how fast should the car accelerate? As you know, the farther down you press the pedal, the faster the car accelerates.
- Additional information that a function needs to perform its task is known as a parameter.
- A function call supplies values—called arguments—for each of the function's parameters.

# Defining *hello()* with a Parameter

```
 1  #include <iostream>
 2  #include <string>
 3  using namespace std;
 4  class Timer {
 5  public:
 6        void hello(string name)
 7        {
 8              cout << "Hello " << name << "!!!" << endl;
 9        }
10  };
11  int main()
12  {
13        Timer tmr;
14        string username;
15        cout << "Please enter your name: ";
16        getline(cin, username);
17        tmr.hello(username);
18        return 0;
19  }
```

```
> ./timer2
Please enter your name: Alan Turing
Hello Alan Turing!!!
```

# string Class

- The new version of `hello` requires a parameter (`name` in line 6) that represents the name to output.
- A string is actually an object of the C++ Standard Library class `string`.
- Defined in header file `<string>` and part of namespace `std`.

```
2  #include <string>
       ...
6          void hello(string name)
7          {
8                  cout << "Hello " << name << "!!!" << endl;
```

# *getline()* Function

- Library function `getline` reads a line of text into a `string`.
- The function call `getline(cin, username)` reads characters (including the space characters that separate the words in the input) from the standard input stream object `cin` (i.e., the keyboard) until the newline character is encountered, places the characters in the `string` variable `username` and discards the newline character.
- The `<string>` header file must be included in the program to use function `getline`.

# Difference Between *getline()* and *cin*

```
string str ;

getline(cin, str);
cout <<   “getline() get ”   << str
      << "\t length = " << str.length() << endl;

cin >> str;
cout <<   “cin get ”   << str
      << "\t length = " << str.length() << endl;
```

```
> ./getline_cin
NCKU is best!
getline() get NCKU is best !     length = 13
NCKU is best!
cin get NCKU     length = 4
```

# Calling *hello()* with a Parameter

- Line 17 calls `Timer`'s `hello` member function.
- The `username` variable in parentheses is the argument that is passed to member function `hello` so that it can perform its task.

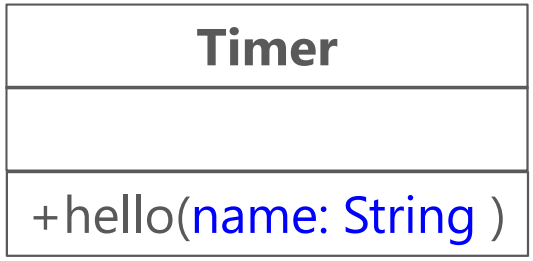
```
17          tmr.hello(username);
```

- The value of variable `username` in `main` becomes the value of member function `hello`'s parameter `name` in line 6.

```
6           void hello(string name)
7           {
8                   cout << "Hello " << name << "!!!" << endl;
```

# UML Class Diagram with a Parameter

- The UML models a parameter by listing the parameter name, followed by a colon and the parameter type in the parentheses following the operation name.
- The UML has its own data types similar to those of C++.
- The UML is language independent—it's used with many different programming languages—so its terminology does not exactly match that of C++.



# Data Members

- A timer records the starting timestamp.
- Every instance (i.e., object) of a class contains one copy of each of the class's data members.
- A benefit of making a variable a data member is that all the member functions of the class can manipulate any data members that appear in the class definition.

# Size of an Object

```
class Cls {
        int x1, x2, x3;
        int fun(int a, int b)
        {
                int y, z;
                return 1;
        }
};
int main()
{
        Cls obj;
        cout << sizeof (Cls) << endl;
        cout << sizeof (obj) << endl;
        return 0;
}
```

Output:
12
12

# Data Members and Member Functions

- The compiler creates only one copy of the class's member functions and shares that copy among all the class's objects.
- Each object, of course, needs its own copy of the class's data members, because their contents can vary among objects.
- The member function code, however, is not modifiable, so it can be shared among all objects of the class.
- Therefore, the size of an object depends on the amount of memory required to store the class's data members.

# Data Members, *set Functions and get Functions*

```
 1  #include <iostream>
 2  #include <ctime>
 3  #include <unistd.h>
 4  using namespace std;
 5  class Timer {
 6     public:
 7        void setStart(time_t ts) {
 8           start_ts = ts;
 9        }
10        time_t getStart() {
11           return start_ts;
12        }
13        int getElapsedTime() {
14           return time(NULL) – getStart();
15        }
16     private:
17        time_t start_ts;
18  };
```

```
19  int main() {
20     Timer tmr;
21     time_t t;
22
23     ts = time(NULL);
24     tmr.setStart(t);
25     sleep(2);
26
27     cout << " Start Time: " <<
         tmr.getStart() << endl;
28     cout << "Elapsed Time: " <<
         tmr.getElapsedTime() << endl;
29     return 0;
30  }
```

```
> ./timer3
Start Time: 1391356562
Elapsed Time: 2
```

# *private* Access Specifier

- Most data-member declarations appear after the access-specifier label `private:`
- Variables or functions declared after access specifier `private` (and before the next access specifier) are accessible only to member functions of the class for which they're declared.
- The default access for class members is `private` so all members after the class header and before the first access specifier are `private`.
- The access specifiers `public` and `private` may be repeated, but this is unnecessary and can be confusing.

```
class Timer {
    // private
public:
    // public
private:
    // private
private:
    // private
public:
    // public
};
```

# *private* Access Specifier (cont.)

- Generally, data members should be declared `private` and member functions should be declared `public`.
- Member functions might be declared `private` if they are to be accessed only by other member functions of the class.
- Despite the fact that the `public` and `private` access specifiers may be repeated and intermixed, list all the public members of a class first in one group then list all the private members in another group. This focuses the programmer's attention on the class' `public` interface rather than on the class's implementation.

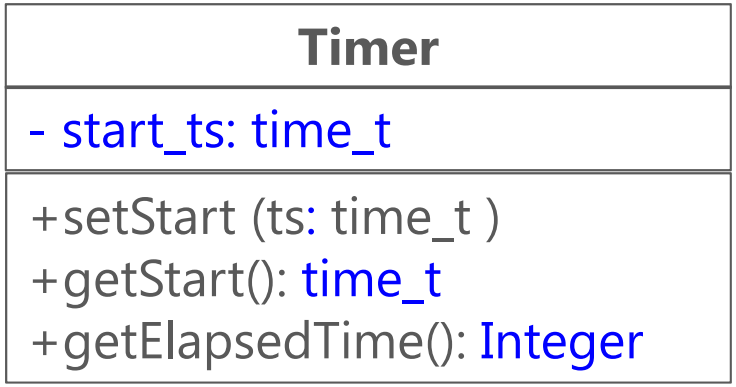
# *set* and *get* Functions

- Classes often provide `public` member functions to allow functions from outside the object to set (i.e., assign values to) or get (i.e., obtain the values of ) `private` data members.
- These member function names need not begin with `set` or `get`, but this naming convention is common.

```
class Point {
 public:
    void setX( int x_value) { ... }
    int getX() { ... }
    void setY( float  y_value) { ... }
    float getY() { ... }
 private:
    int x;
    float y;
};
```

# UML Class Diagram for Data Members

- The UML represents data members as attributes by listing the attribute name, followed by a colon and the attribute type.
- The minus sign in the UML is equivalent to the `private` access specifier in C++.



# Class *Timer* with Constructors

```
 1 #include <iostream>
 2 #include <ctime>
 3 using namespace std;
 4 class Timer {
 5    public:
 6       Timer() {
 7          start_ts = 0;
 8       }
 9       Timer(time_t s) {
10          setStart(s);
11       }
12       void start() {
13          start_ts = time(NULL);
14       }
15       void setStart(time_t ts) {
16          start_ts = ts;
17       }
18       time_t getStart() {
19          return start_ts;
20       }
21       int getElapsedTime() {
22          return time(NULL) - getStart();
23       }
24    private:
25       time_t start_ts;
26 };
27 int main() {
28    Timer tmr1;
29    Timer tmr2(time(NULL));
30
31    tmr1.start();
32    sleep(2);
33    cout << "tmr1.start=" << tmr1.getStart()
34         << ", elapsed time =" <<
                tmr1.getElapsedTime() << endl;
35    cout << "tmr2.start=" << tmr2.getStart()
36         << ", elapsed time =" <<
                tmr2.getElapsedTime() << endl;
37    return 0;
38 }
```

```
tmr1.start=1391358061,
elapsed time =2
tmr2.start=1391358061,
elapsed time =2
```

# Initializing Objects with Constructors

- Each class can provide a constructor that can be used to initialize an object of the class when the object is created.
- A constructor is a special member function that must be defined with the same name as the class.
- An important difference between constructors and other functions is that constructors cannot return values, so they cannot specify a return type (not even `void`).
- Normally, constructors are declared `public`.

# Initializing Objects with Constructors (cont.)

- C++ requires a constructor call for each object that is created, which helps ensure that each object is initialized before it's used in a program.
- The constructor call occurs implicitly when the object is created.
- If a class does not explicitly include a constructor, the compiler provides a default constructor—that is, a constructor with no parameters.

# Initializing Objects with Constructors (cont.)

- Line 10 in the constructor's body passes the constructor's parameter `s` to member functions `setStart`, which simply assigns the value of its parameter to data member `start_ts`.

```
 6       Timer() {
 7           start_ts = 0;
 8       }
 9       Timer(time_t s) {
10           setStart(s);
11       }
```

# Initializing Objects with Constructors (cont.)

- Line 28 creates and initializes a `Timer` object called `tmr1`.
- When this line executes, the `Timer` constructor (lines 6–8) is called (implicitly by C++).
- Line 29 initializes the `Timer` object called `tmr2`, and another constructor (lines 9-11) is called with argument `time(NULL)`, which are used to initialize `tmr2`'s data members.

```
 6      Timer() {
 7          start_ts = 0;
 8      }
 9      Timer(time_t s) {
10          setStart(s);
11      }
```

```
28      Timer tmr1;
29      Timer tmr2(time(NULL));
```

# Initializing Objects with Constructors (cont.)

- A class gets a default constructor in one of two ways:
  - The compiler implicitly creates a default constructor in a class that does not define a constructor.
  - You explicitly define a constructor that takes no arguments.
- If you define a constructor with arguments, C++ will not implicitly create a default constructor for that class.
- Data members can be initialized in a constructor, or their values may be set later after the object is created. However, it is better to ensure that an object is fully initialized before the client code invokes the object's member functions.

# *Error*: Class Without Default Constructor

> cat no_def_ctor.cpp

```
        ...
 5  class Cls {
 6  public:
 7          Cls(int i) { }
 8  };
 9  int main()
10  {
11          Cls obj1(3);
12          Cls obj2;
13          return 0;
14  }
```


```
$ g++ -o no_def_ctor no_def_ctor.cpp
no_def_ctor.cpp : In function `int main()':
no_def_ctor.cpp :12: error: no matching function for call to `Cls::Cls()'
```

# Sequence of Constructor Calls in Composition

$ cat compo_ctor.cpp

```
...
class Tire {
      Tire()  { cout <<   "Tire Constructor" << endl; }
};
class Car {
public:
      Car() { cout <<   "Car Constructor" << endl;  }
private:
      Tire tireB;
};
int main()
{
      Car objA;
      return 0;
}
```

Output:

Tire Constructor

Car Constructor

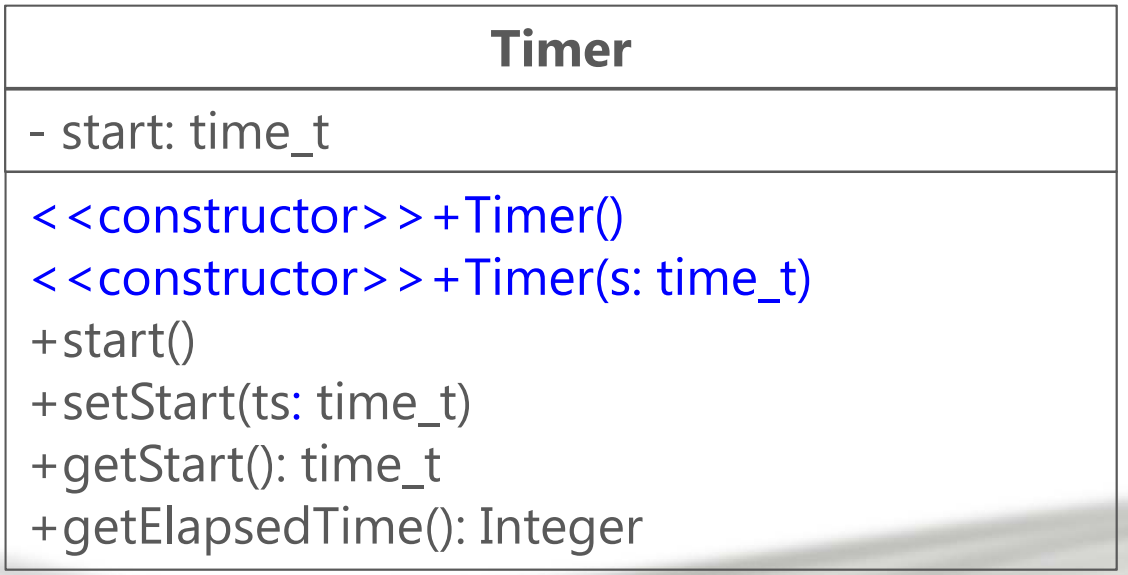
# *Error*: Illegal Member Initialization Outside Constructor

```
$ cat -n err_init.cpp
     ...
     4  class Cls {
     5  public:
     6          Cls() { x = 4; }
     7  private:
     8          int x = 3;
     9  };
    11  int main()
    12  {
    13          Cls obj;
    14          return 0;
    15  }
```


```
$ g++ -o err_init err_init.cpp
err_init.cpp:8: error: ISO C++ forbids initialization of member `x'
```

# UML Class Diagram for Constructor

- To distinguish a constructor from a class's operations, the UML places the word "constructor" between « and » before the constructor's name.
- It's customary to list the class's constructor before other operations in the third compartment.



# Placing a Class in a Separate File for Reusability

- One of the benefits of creating class definitions is that, when packaged properly, our classes can be reused by programmers—potentially worldwide.
- Programmers who wish to use our `Timer` class cannot simply include the files shown in previous slides in another program.

Reused.cpp

```
class Cls {
        ...
};

int main()
{

}
```

Reusing.cpp

```
#include "Reused.cpp"
int main()
{

}
```

Duplicate main() !!

# Placing a Class in a Separate File for Reusability (cont.)

- When building an object-oriented C++ program, it's customary to define reusable source code (such as a class) in a file that by convention has a `.h` filename extension—known as a header file.
- Programs use `#include` preprocessor directives to include header files and take advantage of reusable software components.

# Placing a Class in a Separate File for Reusability (cont.)

- Our next example separates the code into two files—`Timer5.h` and `main5.cpp`.
  - As you look at the header file in `Timer5.h`, notice that it contains only the `Timer` class definition (lines 4–26), the appropriate header files and a `using` declaration.
  - The `main` function that uses class `Timer` is defined in the source-code file `main5.cpp` in lines 8–18.

# timer5.h

```
 1  #include <iostream>
 2  #include <ctime>
 3  using namespace std;
 4  class Timer {
 5      public:
 6          Timer() {
 7              start_ts = 0;
 8          }
 9          Timer(time_t s) {
10              setStart(s);
11          }
12          void start() {
13              start_ts = time(NULL);
14          }
15          void setStart(time_t ts) {
16              start_ts = ts;
17          }
18          time_t getStart() {
19              return start_ts;
20          }
21          int getElapsedTime() {
22              return time(NULL) - getStart();
23          }
24      private:
25          time_t start_ts;
26  };
```

# main5.cpp

```
 1  #include <iostream>
 2  #include <ctime>
 3  #include "timer5.h"
 4  using namespace std;
 5  int main() {
 6      Timer tmr1;
 7      Timer tmr2(time(NULL));
 8
 9      tmr1.start();
10      sleep(2);
11      cout << "tmr1.start=" << tmr1.getStart()
12           << ", elapsed time =" << tmr1.getElapsedTime() << endl;
13      cout << "tmr2.start=" << tmr2.getStart()
14           << ", elapsed time =" << tmr2.getElapsedTime() << endl;
15      return 0;
16  }
```

# Placing a Class in a Separate File for Reusability (cont.)

- Placing the whole class definition in a header file reveals the entire implementation of the class to the class's clients.
- However, the client code needs to know only what member functions to call, what arguments to provide to each member function and what return type to expect from each member function.
- Hiding the class's implementation details makes it easier to change the class's implementation while minimizing, and hopefully eliminating, changes to client code.

# Separating Interface from Implementation

- Interfaces define and standardize the ways in which things such as people and systems interact with one another.
- The interface of a class describes what services a class's clients can use and how to request those services, but not how the class carries out the services.
- A class's `public` interface consists of the class's `public` member functions.

# Separating Interface from Implementation (cont.)

- In our prior examples, each class definition contained the complete definitions of the class's `public` member functions and the declarations of its `private` data members.
- It's better software engineering to define member functions outside the class definition, so that their implementation details can be hidden from the client code.
- By convention, member-function definitions are placed in a source-code file of the same base name (e.g., `Timer`) as the class's header file but with a `.cpp` filename extension.

# *timer6.h* and *timer6.cpp*

*timer6.h*

```
 1 #include <ctime>
 2 using namespace std;
 3 class Timer {
 4     public:
 5         Timer();
 6         Timer(time_t s);
 7         void start();
 8         void setStart(time_t ts);
 9         time_t getStart();
10         int getElapsedTime();
11     private:
12         time_t start_ts;
13 };
```

*timer6.cpp*

```
 1 #include "timer6.h"
 2 Timer::Timer() { start_ts = 0; }
 3 Timer::Timer(time_t s) {
 4     setStart(s);
 5 }
 6 void Timer::start() {
 7     start_ts = time(NULL);
 8 }
 9 void Timer::setStart(time_t ts) {
10     start_ts = ts;
11 }
12 time_t Timer::getStart() {
13     return start_ts;
14 }
15 int Timer::getElapsedTime() {
16     return time(NULL) - getStart();
17 }
```

# timer6.h

- Header file `Timer6.h` is similar to the one in `Timer5.h`, but the function definitions in `Timer5.h` are replaced here with function prototypes (lines 5–10) that describe the class's `public` interface without revealing the class's member-function implementations.
- A function prototype is a declaration of a function that tells the compiler the function's name, its return type and the types of its parameters.

# timer6.cpp

- Source-code file `timer6.cpp` defines class `Timer`'s member functions, which were declared in lines 5–10 of `timer6.h`.
- Notice that each member-function name in the function headers (lines 2, 3, 6, 9, 12 and 15) is preceded by the class name and `::`, which is known as the binary scope resolution operator.

# *timer6.cpp* (cont.)

- To indicate that the member functions in `timer6.cpp` are part of class `Timer`, we must first include the `timer6.h` header file (line 1).
- This allows us to access the class name `Timer` in the `timer6.cpp` file.
- When compiling `timer6.cpp`, the compiler uses the information in `timer6.h` to ensure that
  - the first line of each member function matches its prototype in the `timer6.h` file, and that
  - each member function knows about the class's data members and other member functions

# main6.cpp

```
 1 #include <iostream>
 2 #include <ctime>
 3 #include "timer6.h"
 4 using namespace std;
 5 int main() {
 6     Timer tmr1;
 7     Timer tmr2(time(NULL));
 8
 9     tmr1.start();
10     sleep(2);
11     cout << "tmr1.start=" << tmr1.getStart()
12          << ", elapsed time =" << tmr1.getElapsedTime() << endl;
13     cout << "tmr2.start=" << tmr2.getStart()
14          << ", elapsed time =" << tmr2.getElapsedTime() << endl;
15     return 0;
16 }
```

```
> g++ -c timer6.cpp
> g++ -c main6.cpp
> g++ -o timer6 main6.o timer6.o
```

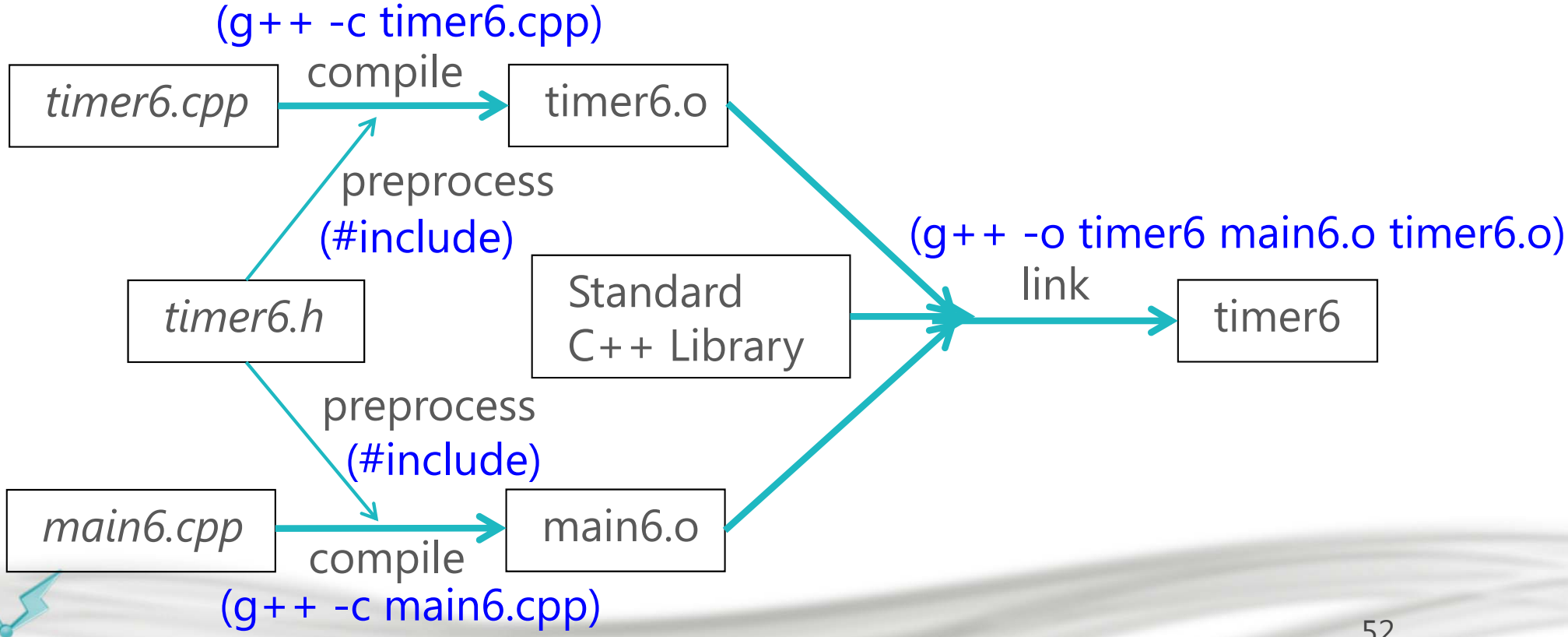
# *Error*: Mismatch of Member Function Declaration and Definition

```
> cat only_mem_function.cpp
int LaLa::func()
{
}
> g++ -c only_mem_function.cpp
only_mem_function.cpp:1: error: `LaLa' has not been declared
```


```
> cat non_declared_member_function.cpp
class LaLa{
};
int LaLa::func()
{
}
> g++ -c non_declared_member_function.cpp
non_declared_member_function.cpp:6: error: no `int LaLa::func()' member function declared in class `LaLa'
```

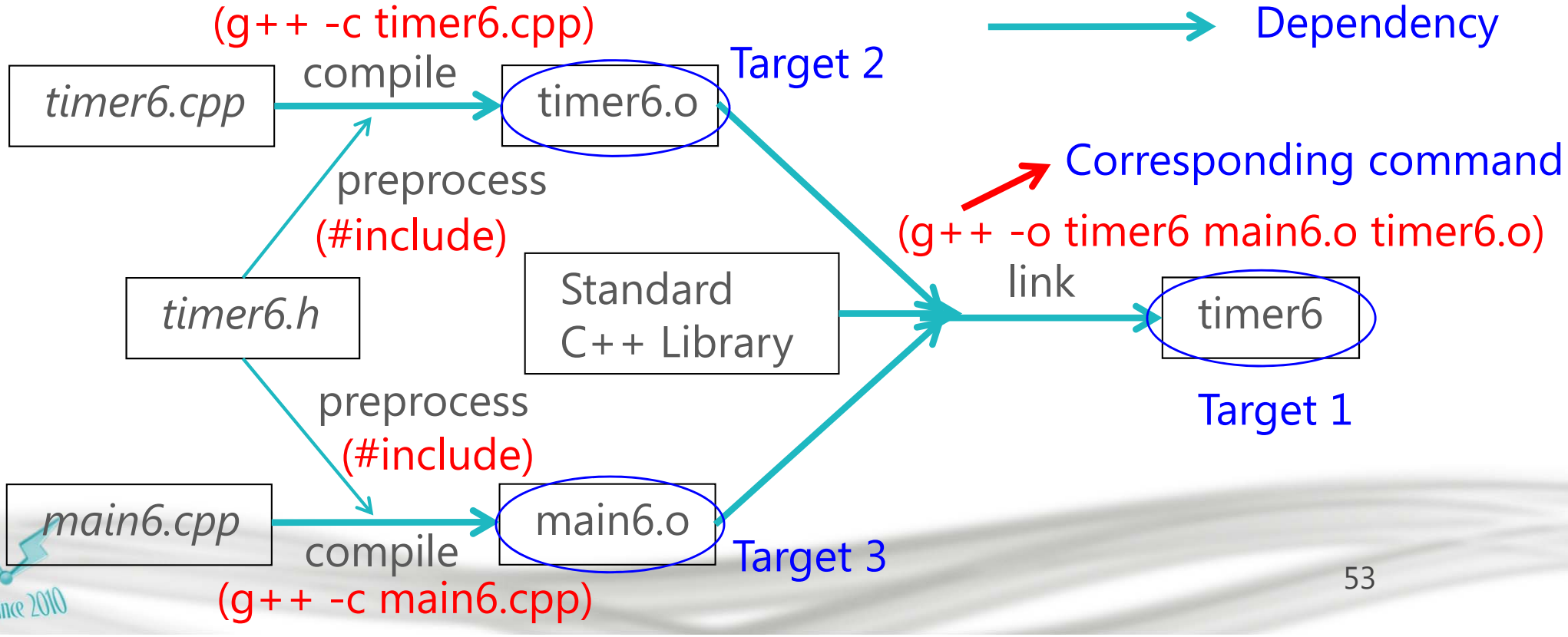
# Preprocess, Compile and Link

- Before executing this program, the source-code files in `timer6.cpp` and `main6.cpp` must both be compiled, then linked together.



# Writing a *Makefile* for Automation

- Step 1: Identify targets (executable program and .o files).
- Step 2: Identify dependencies of each target.
- Step 3: Write down corresponding commands (for compilation).



# Makefile

```
1  main6: timer6.o main6.o
2    g++ -o main6 main6.o timer6.o
3
4  timer6.o: timer6.cpp timer6.h
5    g++ -c timer6.cpp
6
7  main6.o: main6.cpp timer6.h
8    g++ -c main6.cpp
9
10 clean:
11   rm main6 *.o
```

```
<target1>: <dependence>
<tab><command>
<tab><command>

<target2>: <dependence>
<tab><command>
```

```
> make
g++ -c timer6.cpp
g++ -c main6.cpp
g++ -o main6 main6.o timer6.o
> touch timer6.cpp
> make
g++ -c timer6.cpp
g++ -o main6 main6.o timer6.o
> touch timer6.h
> make
g++ -c timer6.cpp
g++ -c main6.cpp
g++ -o main6 main6.o timer6.o
> make clean
rm main6 *.o
```

# *timer7.h* and *timer7.cpp*

*timer7.h*

```
 1  #include <ctime>
 2  using namespace std;
 3  class Timer {
 4     public:
 5        Timer();
 6        Timer(time_t s, time_t n);
 7        void setStart(time_t start_ts);
 8        void setNow(time_t now_ts);
 9        time_t getStart();
10        time_t getNow();
11        int getElapsedTime();
12     private:
13        time_t start, now;
14  };
```

*timer7.cpp*

```
 1  #include "timer7.h"
 2  Timer::Timer() { setStart(0); }
 3  Timer::Timer(time_t s) {
 4     setStart(s);
 5  }
 6  void Timer::start() {
 7     setStart(time(NULL));
 8  }
 9  void Timer::setStart(time_t ts) {
10     start_ts = (ts>0)?ts:time(NULL);
11  }
12  time_t Timer::getStart() {
13     return start_ts;
14  }
15  int Timer::getElapsedTime() {
16     return time(NULL) - getStart();
17  }
```

# main7.cpp

```
 1 #include <iostream>
 2 #include <ctime>
 3 #include "timer7.h"
 4 using namespace std;
 5 int main() {
 6     Timer tmr1;
 7     Timer tmr2(time(NULL));
 8
 9     tmr1.setStart(-3);
10     sleep(2);
11     cout << "tmr1.start=" << tmr1.getStart()
12          << ", elapsed time =" << tmr1.getElapsedTime() << endl;
13     cout << "tmr2.start=" << tmr2.getStart()
14          << ", elapsed time =" << tmr2.getElapsedTime() << endl;
15     return 0;
16 }
```

(retrieve timer7.o from the developer of Timer class)

```
> g++ -c main7.cpp
> g++ -o timer7 main7.o timer7.o
```

```
> ./timer7
tmr1.start=1391361369, elapsed time =2
tmr2.start=1391361369, elapsed time =2
```

# Validating Data with *set Functions*

- `Timer7.cpp` enhances class `Timer`'s member function `setStart` to perform validation (also known as validity checking).
- Since the interface of the class remains unchanged, clients of this class need not be changed when the definition of member function `setStart` is modified.
- The constructor simply calls `setStart`, rather than duplicating its validation code.