Object-Oriented Programming
(in C++)

Classes and Objects (Part I)

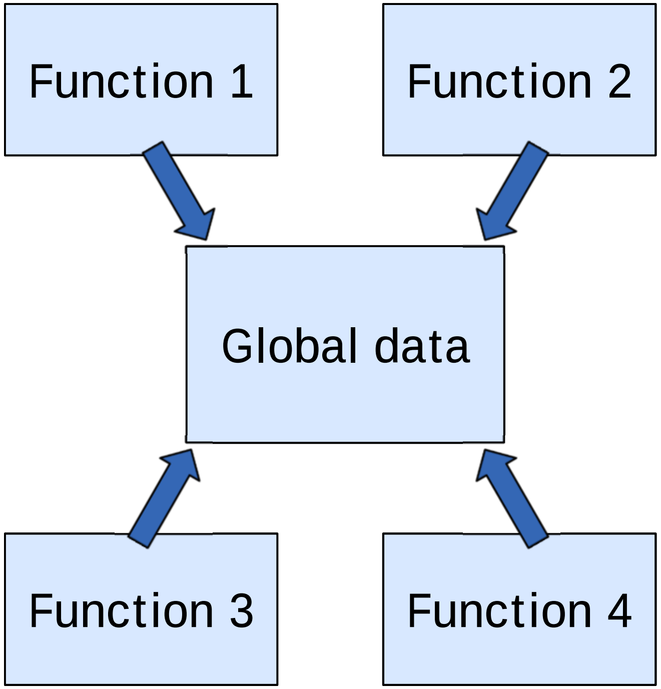
Professor Yi-Ping You (游逸平)
Department of Computer Science
http://www.cs.nctu.edu.tw/~ypyou/

Outline

- Introduction to OOP
  - What Are Objects and Classes?
  - Review of OOP
- Classes (Abstract Data Types)
  - Overview of Classes
  - Designing A `Time` Class
  - Designing Member Functions
  - Memberwise Assignment Operation
  - Passing and Returning Objects

Procedural vs Object-Oriented Programming

- Procedural program
  - Passive data



Function 1

Function 2

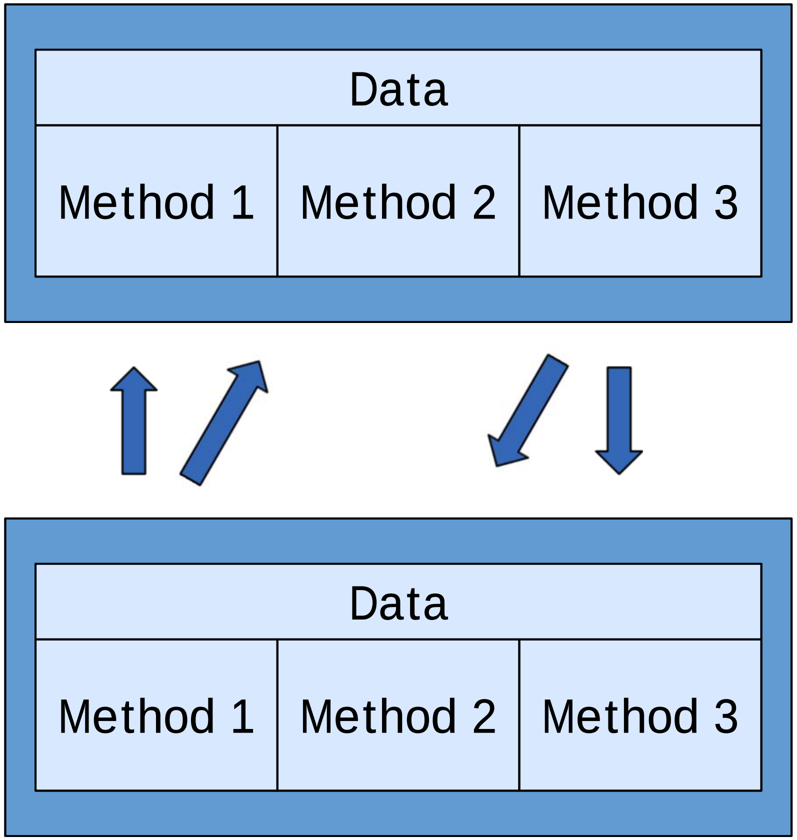
Global data

Function 3

Function 4

- Object-oriented program
  - Active data

An object



Data

Method 1

Method 2

Method 3

Data

Method 1

Method 2

Method 3

Another object

What Is An Object & A Class?

- If you look around you, you will see objects everywhere
  - Any physical entity is an object
  - E.g., Nick, Joanne, your chair, his desk, her pen, my computer, etc.
- Each object has its own attribute values
  - E.g., each student has his/her own name, age, ID, gender, and major
- Objects that share the same attributes form a class
  - Nick and Joanne are all students

Real World vs Programming World

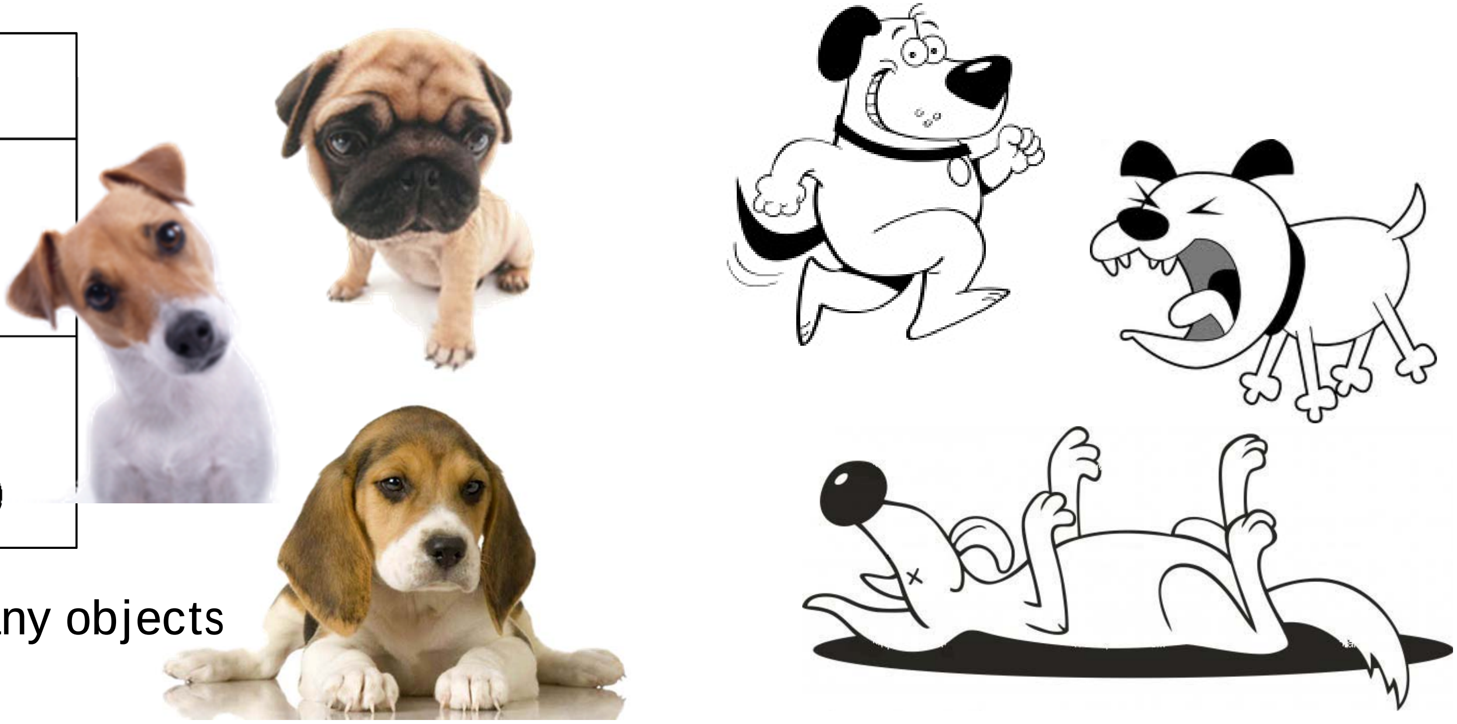
- Real world
  - Objects that share the same attributes form a class
- Programming world (programmers are God)
  - Programmers define classes that they need and create objects from the classes
  - Programmers design computer programs to *describe/represent* real-world objects or virtual objects

Objects & Classes

- An object is instantiated from a class
  - The object is called an instance of the class
- A class is a blueprint that is used to construct objects

DOG
breed size name

run()
bark()
playdead()



One class, many objects

Designing A Class

- When you design a class, think about the objects that will be created from that class
  - Things the object **knows** about itself
    - Data members
      - Represent the object's *state* or *attributes*
      - Also called fields or instance variables in OO languages
  - Things the object **does**
    - Member functions
      - Represents the object's *behavior*
      - Also called methods in OO languages

Designing A Class: Attributes & Behaviors

- Attributes = data members
- Behaviors = member functions
- Dogs
  - Attributes: name, color, breed, hungry, age
  - Behaviors: barking, fetching, wagging tail
- Students
  - Attributes: name, age, ID, gender, class, major
  - Behaviors: taking courses, doing homework, taking exams

struct vs class

- A class is a user-defined data type
  - An abstract data type
- struct in C is also a user-defined data type, but struct has no operations

```
struct myStruct {
  int field1;
  int field2;
};

int main() {
  struct myStruct var;
  var.field1 = 10;
  var.field2 = 20;
}
```

```
class myClass {
  int member1;
  int member2;
  int func() { ... }
  void myClass() { ... }
};

int main() {
  myClass var;
  var.member1 = 10;
  var.member2 = 20;
  var.func();
}
```

An object

Why We Call Classes Abstract Data Types?

- Data members or member functions in a class can be “hidden” to
  - Other classes/objects or
  - Programmers who use the class

Student (Lisa)

```
public:
  name: Lisa
  gender: female
  major: CS
private:
  age: 18
```

```
public:
bool isTeenager() {
  return (age >= 13
&& age <= 19);
}
```

You’ll never know.

How old is Lisa?

```
cout << Lisa.age;
```

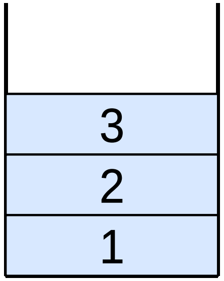
Compilation error!

```
cout << boolalpha <<
Lisa.isTeenager();
```

```
true
```

Designing A Class for Stack

- Stack: a last-in, first-out data structure



Stack

```
class Stack {
  private:
    int max, top;
    int *array;
  public:
    Stack() {
      max = 10;
      top = -1;
      array = new int[max];
    }
    ~Stack() {
      delete [] array;
    }
    void push(int e) { ... }
    int pop() { ... }
    bool isEmpty() { ... }
    bool isFull() { ... }
    void clear() {top = -1;}
};
```

```
int main() {
  Stack s;

  s.push(1);
  s.push(2);
  s.push(3);

  int e;
  e = s.pop();

  s.array[0] = 0;
 // compilation error

  s.top = -1;
 // compilation error

  s.clear();
}
```

Outline

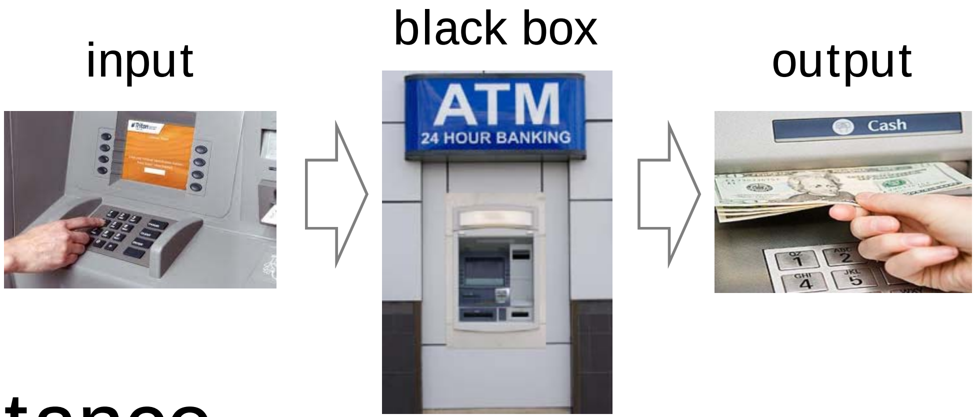
- Introduction to OOP
  - What Are Objects and Classes?
  - Review of OOP
- Classes (Abstract Data Types)
  - Overview of Classes
  - Designing A `Time` Class
  - Designing Member Functions
  - Memberwise Assignment Operation
  - Passing and Returning Objects

Object-Orient Programming

- Object-oriented programming
  - Programs may be seen as a collection of cooperating objects
- Procedural programming
  - Programs may be seen as a list of instructions to the computer
- In OOP, we **send messages to an object** and tells a member function of the object to perform its task
  - Each message is a member-function call

Important OO Concepts: P.I.E.

- Encapsulation (abstract data type)
  - “Black box” – information hiding



- Inheritance
  - Related classes share implementation and/or interface, allowing reuse of codes
- Polymorphism
  - Ability to use a class without knowing its type

Outline

- Introduction to OOP
  - What Are Objects and Classes?
  - Review of OOP
- Classes (Abstract Data Types)
  - Overview of Classes
  - Designing A `Time` Class
  - Designing Member Functions
  - Memberwise Assignment Operation
  - Passing and Returning Objects

Classes

- A class defines the abstract characteristics of a group of similar **`objects`**
  - **Attributes** (data members)
  - **Behaviors** (member functions)
- A class is a user-defined type
  - Can be used to create objects
    - Variables of the class type
  - C++ is an extensible language
- Syntax

```
class class_name {
  [access_specifier_label:]
    // declarations for data members
    // declarations/definitions for member functions
};
```

Access-Specifier Labels

- private
  - The data members or member functions can be used only in the class for which they are declared
  - The default accesses for class members
- public
  - The data members or member functions are available to public
- protected
  - Will be discussed later in Chapter 12 (Inheritance)

- Each data member of a class should have **private** visibility unless it can be proven that the data member needs **public** visibility
  - The principle of least privilege

Class Scope

- Class scope contains
  - Data members
    - Variables declared in the class definition
  - Member functions
    - Functions declared in the class definition
- Within a class's scope
  - Class members are accessible by all member functions
- Outside a class's scope
  - **`public`** class members are referenced through a handle (an object name, a reference/pointer to an object)
    - E.g., `object.member`
  - **`private`** class members could not be accessed by anybody

Class Scope (Cont'd)

- Variables declared in a member function
  - Have block scope
  - Known only to that function
- Hiding a class-scope variable
  - In a member function, define a variable with the same name as a variable with class scope
  - Such a hidden variable can be accessed by preceding the name with the class name followed by the scope resolution operator (::)

```
class Foo {
  int x;
  void bar() {
     int x;
     x = x + ::x;
  }
}
```

Declaring or Defining Member Functions

- The definition (implementation) of a member function is usually defined outside the class body
  - Preceded by the class name and `::`
- The compiler will attempt to **inline** a member function if the member function is defined completely in the class body

Outline

- Introduction to OOP
  - What Are Objects and Classes?
  - Review of OOP
- Classes (Abstract Data Types)
  - Overview of Classes
  - Designing A Time Class
  - Designing Member Functions
  - Memberwise Assignment Operation
  - Passing and Returning Objects

Designing and Using A Class Time

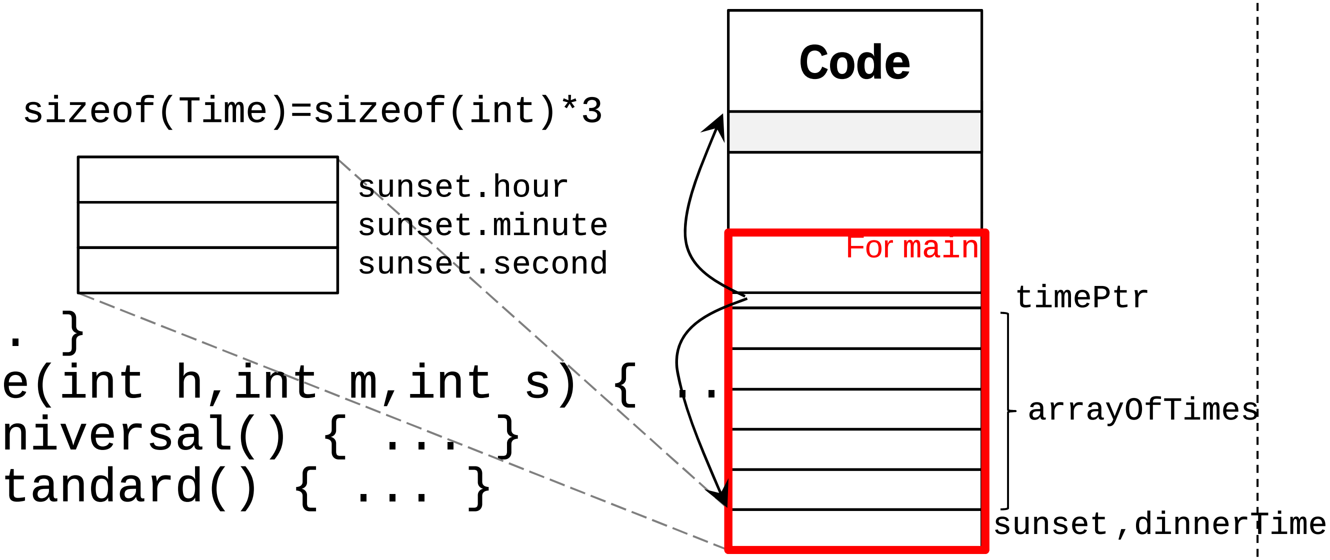
```
class Time {
public:
  Time(); // constructor
  void setTime(int, int, int); // set hour, minute, second
  void printUniversal(); // print in universal-time format
  void printStandard();  // print in standard-time format
private:
  int hour;   // 0-23
  int minute; // 0-59
  int second; // 0-59
};
Time::Time() {
  hour = minute = second = 0;
}
void Time::setTime(int h, int m, int s) {
  hour = (h >= 0 && h < 24) ? h : 0;
  minute = (m >= 0 && m < 60) ? m : 0;
  second = (s >= 0 && s < 60) ? s : 0;
}
void Time::printUniversal() {
  cout << setfill('0') << setw(2) << hour << ":"
    << setw(2) << minute << ":" << setw(2) << second << endl;
}
void Time::printStandard() {
  cout << ( (hour == 0 || hour == 12) ? 12 : hour % 12 )
    << ":" << setfill('0') << setw(2) << minute << ":"
    << setw(2) << second << (hour < 12 ? ″ AM" : " PM")
    << endl;
}
```

```
int main() {
  Time t;

  t.setTime(13, 27, 6);

  t.printUniversal();
  t.printStandard();

  t.hour = 10; // ERROR ✖

  return 0;
}
```

```
13:27:06
1:27:06 PM
```

Using the Time Class (Creating Time Objects)

```
class Time {
   ...
   int hour;
   int minute;
   int second;
};
Time::Time() { ... }
void Time::setTime(int h,int m,int s) { ..
void Time::printUniversal() { ... }
void Time::printStandard() { ... }

int main() {
   Time sunset; // object of type Time
   Time arrayOfTimes[ 5 ]; // array of 5 Time objects
   Time &dinnerTime = sunset; // reference to a Time object
   Time *timePtr = &dinnerTime; // pointer to a Time object
   timePtr = new Time; // equivalent to timePtr = new Time()

   return 0;
}
```



Using Member Functions (or Data Members)

- Member functions are within the class's scope
  - Known only to other members of the class unless referred to via
    - Dot member selection operator (.)
      - Object of the class

```
sunset.printUniversal();
```

      - Pointer to an object of the class

```
(*timePtr).printStandard();
```

    - Arrow member selection operator (->) for pointer

```
timePtr->printStandard();
```

Designing A Time Class (Cont’d)

The interface of Time class

```
class Time {
public:
  Time(); // constructor
  void setTime(int, int, int); // set hour, minute, second
  void printUniversal(); // print in universal-time format
  void printStandard();  // print in standard-time format
private:
  int hour;   // 0-23
  int minute; // 0-59
  int second; // 0-59
};
```

The implementation of Time class

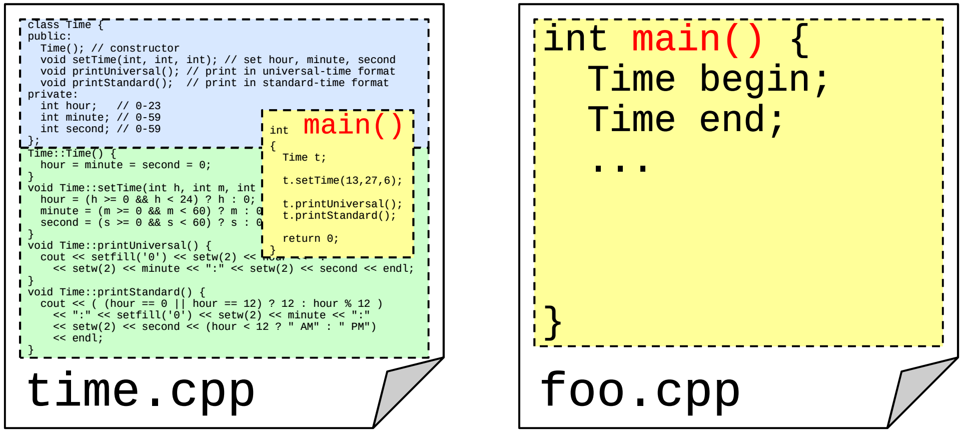
```
Time::Time() {
  hour = minute = second = 0;
}
void Time::setTime(int h, int m, int s) {
  hour = (h >= 0 && h < 24) ? h : 0;
  minute = (m >= 0 && m < 60) ? m : 0;
  second = (s >= 0 && s < 60) ? s : 0;
}
void Time::printUniversal() {
  cout << setfill('0') << setw(2) << hour << ":"
    << setw(2) << minute << ":" << setw(2) << second << endl;
}
void Time::printStandard() {
  cout << ( (hour == 0 || hour == 12) ? 12 : hour % 12 )
    << ":" << setfill('0') << setw(2) << minute << ":"
    << setw(2) << second << (hour < 12 ? " AM" : " PM")
    << endl;
}
```

```
int main() {
  Time t;

  t.setTime(13, 27, 6);

  t.printUniversal();
  t.printStandard();

  t.hour = 10; // ERROR

  return 0;
}
```

A client of Time class

Putting All Things in a File is Bad for Reusability

- We could put the interface and implementation of class `Time` and the client into a file, say `time.cpp`
  - But the `Time` class in `time.cpp` cannot be reused by other programs



```
class Time {
public:
  Time(); // constructor
  void setTime(int, int, int); // set hour, minute, second
  void printUniversal(); // print in universal-time format
  void printStandard();  // print in standard-time format
private:
  int hour;   // 0-23
  int minute; // 0-59
  int second; // 0-59
};
Time::Time() {
  hour = minute = second = 0;
}
void Time::setTime(int h, int m, int
  hour = (h >= 0 && h < 24) ? h : 0;
  minute = (m >= 0 && m < 60) ? m : 0
  second = (s >= 0 && s < 60) ? s : 0
}
void Time::printUniversal() {
  cout << setfill('0') << setw(2) <<
    << setw(2) << minute << ":" << setw(2) << second << endl;
}
void Time::printStandard() {
  cout << ( (hour == 0 || hour == 12) ? 12 : hour % 12 )
    << ":" << setfill('0') << setw(2) << minute << ":"
    << setw(2) << second << (hour < 12 ? " AM" : " PM")
    << endl;
}
```

```
int main()
{
  Time t;

  t.setTime(13,27,6);

  t.printUniversal();
  t.printStandard();

  return 0;
}
```

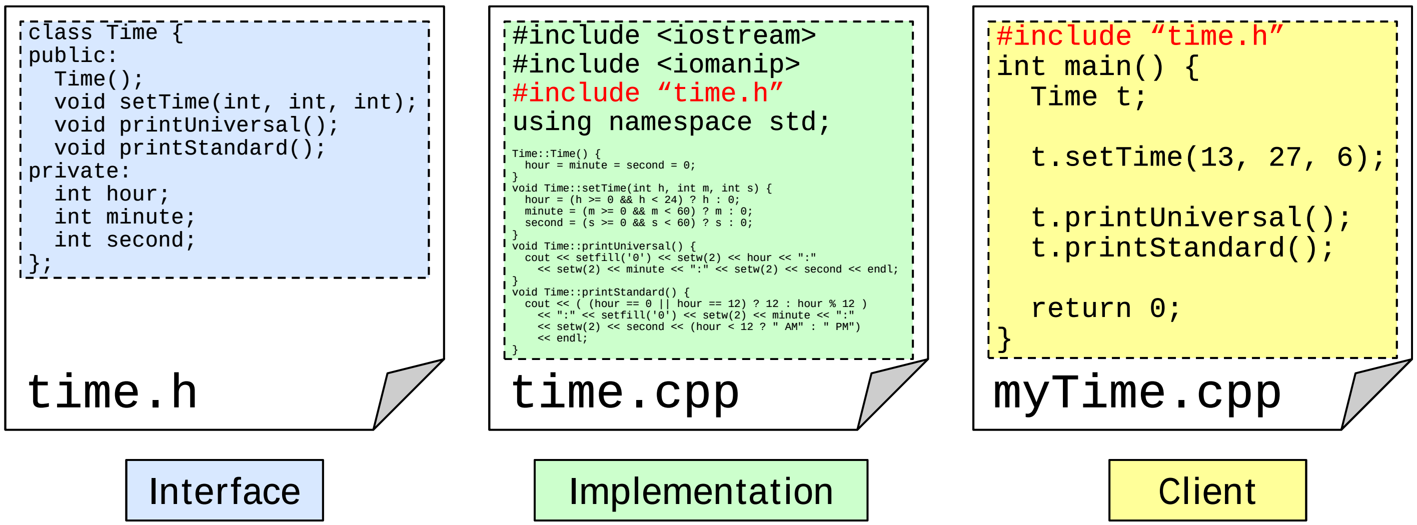
time.cpp

```
int main() {
  Time begin;
  Time end;
  ...
}
```

foo.cpp

```
> g++ time.cpp foo.cpp
redefinition of ’int main()'
```

Placing a Class in a Separate File for Reusability



```
class Time {
public:
   Time();
   void setTime(int, int, int);
   void printUniversal();
   void printStandard();
private:
   int hour;
   int minute;
   int second;
};
```

time.h

```
#include <iostream>
#include <iomanip>
#include “time.h”
using namespace std;

Time::Time() {
  hour = minute = second = 0;
}
void Time::setTime(int h, int m, int s) {
  hour = (h >= 0 && h < 24) ? h : 0;
  minute = (m >= 0 && m < 60) ? m : 0;
  second = (s >= 0 && s < 60) ? s : 0;
}
void Time::printUniversal() {
  cout << setfill('0') << setw(2) << hour << ":"
    << setw(2) << minute << ":" << setw(2) << second << endl;
}
void Time::printStandard() {
  cout << ( (hour == 0 || hour == 12) ? 12 : hour % 12 )
    << ":" << setfill('0') << setw(2) << minute << ":"
    << setw(2) << second << (hour < 12 ? " AM" : " PM")
    << endl;
}
```

time.cpp

```
#include “time.h”
int main() {
   Time t;

   t.setTime(13, 27, 6);

   t.printUniversal();
   t.printStandard();

   return 0;
}
```

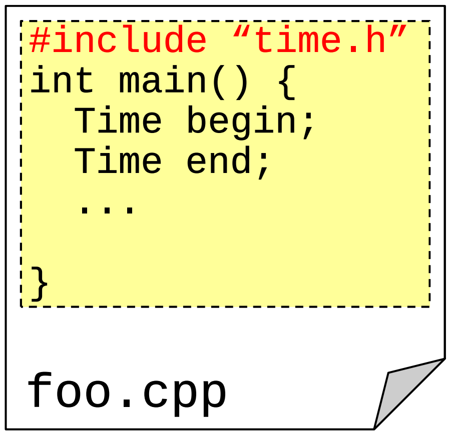
myTime.cpp

Interface

Implementation

Client

```
> g++ time.cpp myTime.cpp –o myTime.out
> ./myTime.out
13:27:06
1:27:06 PM
> g++ -c time.cpp
> g++ time.o foo.cpp –o foo.out
```



```
#include “time.h”
int main() {
   Time begin;
   Time end;
   ...

}
```

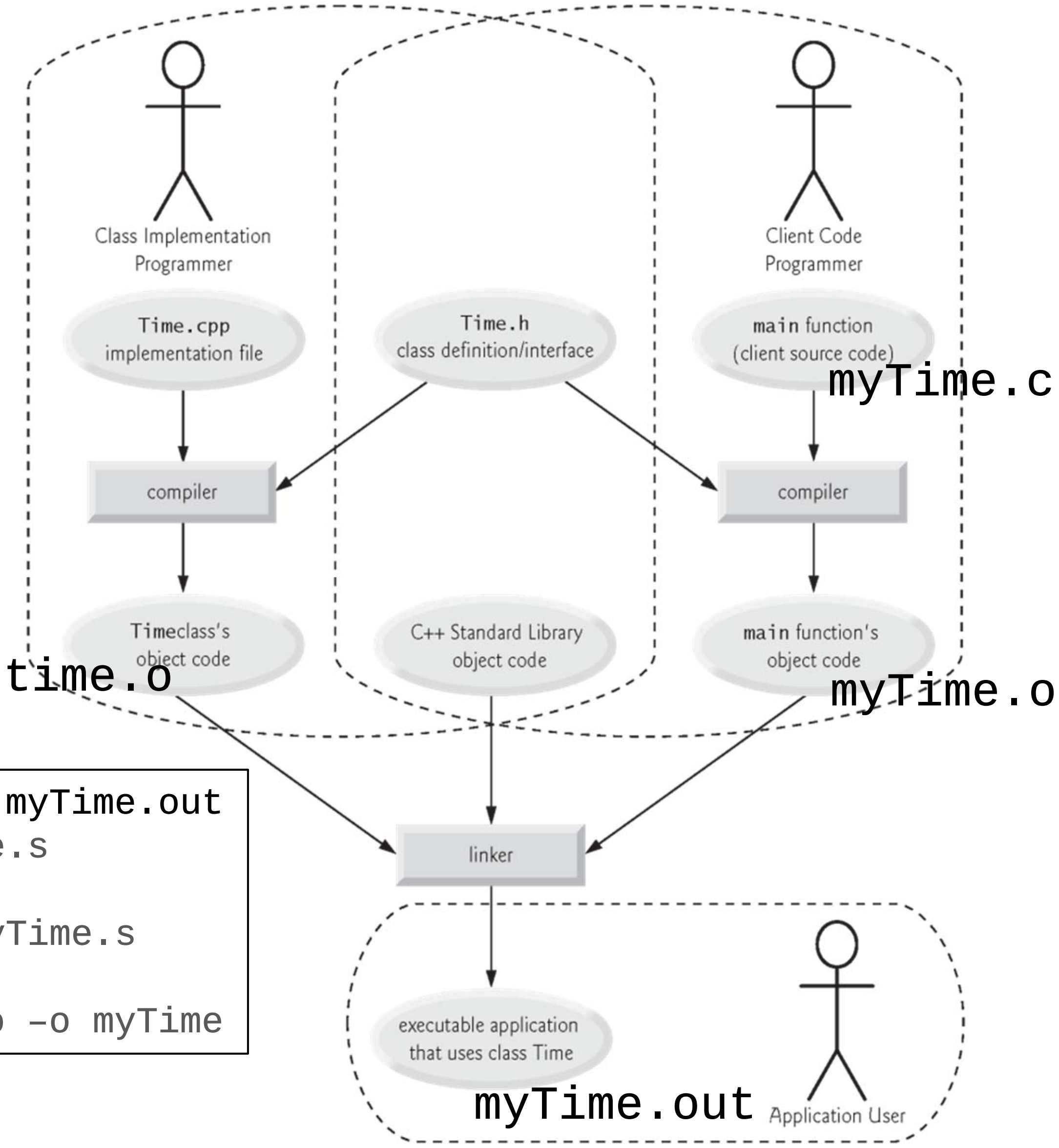
foo.cpp

- The implementation of class Time is encapsulated (hidden) to the clients (users)

Compilation and Linking Processes

- g++ performs a series of tasks
- You may use “-v” option to see the commands executed

```
> g++ -v time.cpp myTime.cpp –o myTime.out
.../cc1plus ... time.cpp –o time.s
.../as ... time.s –o time.o
.../cc1plus ... myTime.cpp –o myTime.s
.../as ...myTime.s –o myTime.o
.../collect2 ... time.o myTime.o –o myTime
```


Class Implementation Programmer
Client Code Programmer
Time.cpp implementation file
Time.h class definition/interface
main function (client source code)
compiler
compiler
Timeclass's object code
C++ Standard Library object code
main function's object code
linker
executable application that uses class Time
Application User

myTime.c

time.o

myTime.o

myTime.out

`#include` Preprocessor Directive

- Used to include header files
  - Instructs C++ preprocessor to replace directive with a copy of the contents of the specified file
- Quotes (" ") indicate user-defined header files
  - Preprocessor first looks in current directory
    - If the file is not found, looks in C++ Standard Library directory
- Angle brackets (< >) indicate C++ Standard Library
  - Preprocessor looks only in C++ Standard Library directory
- "`g++ -E myTime.cpp`” outputs the result after preprocessing

Preprocessor Wrappers

- What happens here?

```
#include “time.h”

// other header file
```

other.h

```
#include “time.h”
#include “other.h”

int main() {
   ...
}
```

myTime.cpp

- Use preprocessor wrappers for `time.h` to prevent multiple inclusions of header file

```
#ifndef TIME_H
#define TIME_H

class Time {
  ...
};

#endif
```

time.h

Outline

- Introduction to OOP
  - What Are Objects and Classes?
  - Review of OOP
- Classes (Abstract Data Types)
  - Overview of Classes
  - Designing A `Time` Class
  - Designing Member Functions
  - Memberwise Assignment Operation
  - Passing and Returning Objects

Designing Member Functions

- Recall: when designing a class, we have to define
  - Data members (attributes of objects)
  - Member functions (behaviors of objects)
- Categories of member functions
  - Constructors/destructors
  - Access functions
  - Utility functions (helper functions)
  - Set/get functions

Constructors (1/2)

- Constructor
  - A special member function that is written to
    - initialize data members of an object or/and
    - allocate additional memory for the object
  - Has the same name with the class
  - Returns no value
  - Implicitly called when an object is created
    - Treat an object like a variable and recall when a variable is mapped to the memory
      - Mapped to static data section,
      - Mapped to stack, or
      - Mapped to heap
  - Constructor itself does not actually allocate the object's memory

Constructors (2/2)

- Constructor
  - Compiler provides a default constructor (with no parameters) if none included
    - With “empty” body
  - Once you explicitly declare absolutely any constructor for a class, the compiler stops providing the implicit default constructor
    - If you still need the default constructor, you have to explicitly declare and define it yourself

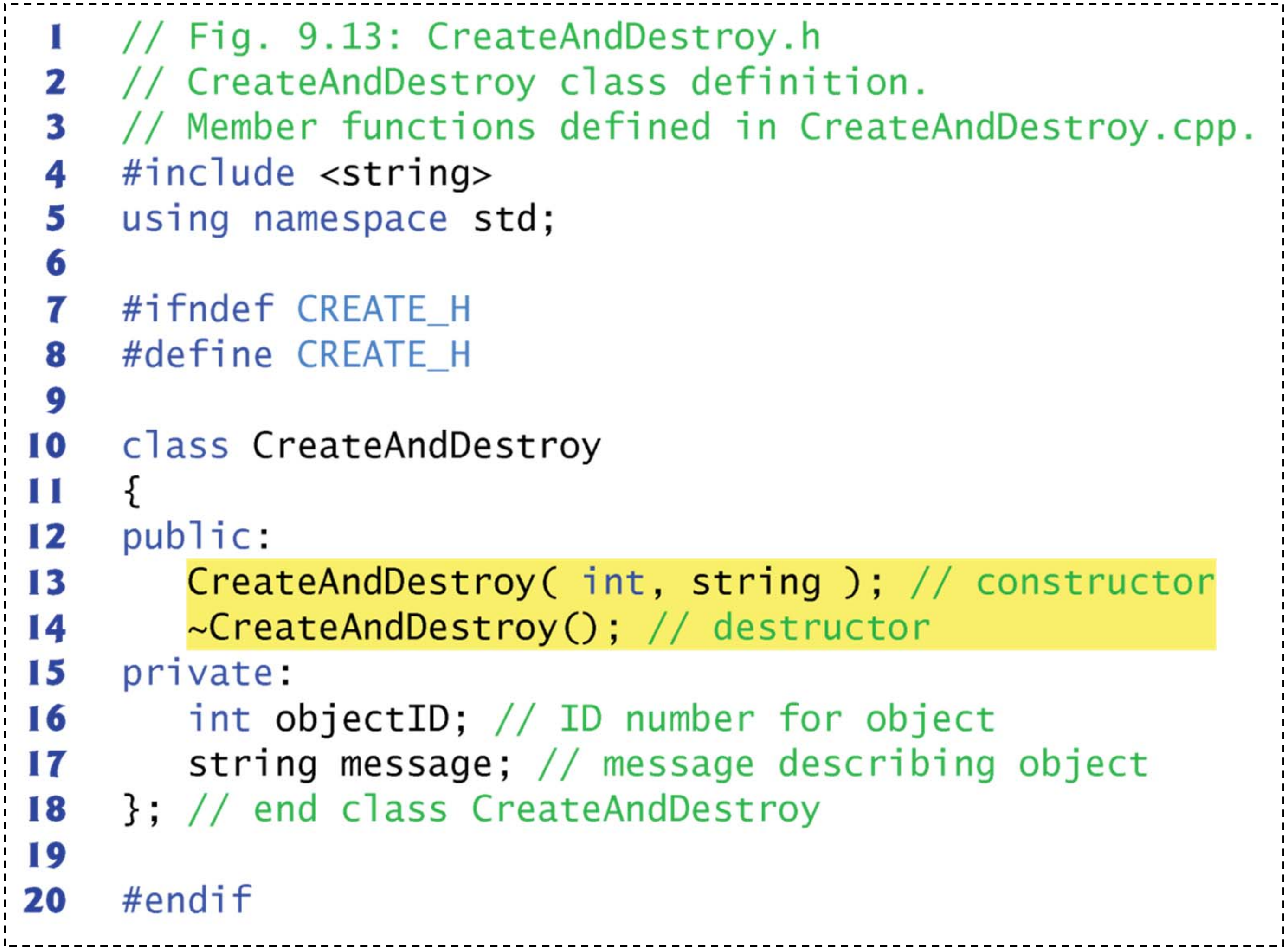
Destructors (2/2)

- Destructor
  - A special member function that is written to
    - perform termination housekeep or/and
    - reclaim the memory allocated by constructors
      - To avoid memory leak
  - Its name is the tilde character (~) followed by the class name
  - Receives no parameters and returns no value
  - Implicitly called when an object is destroyed

Destructors (2/2)

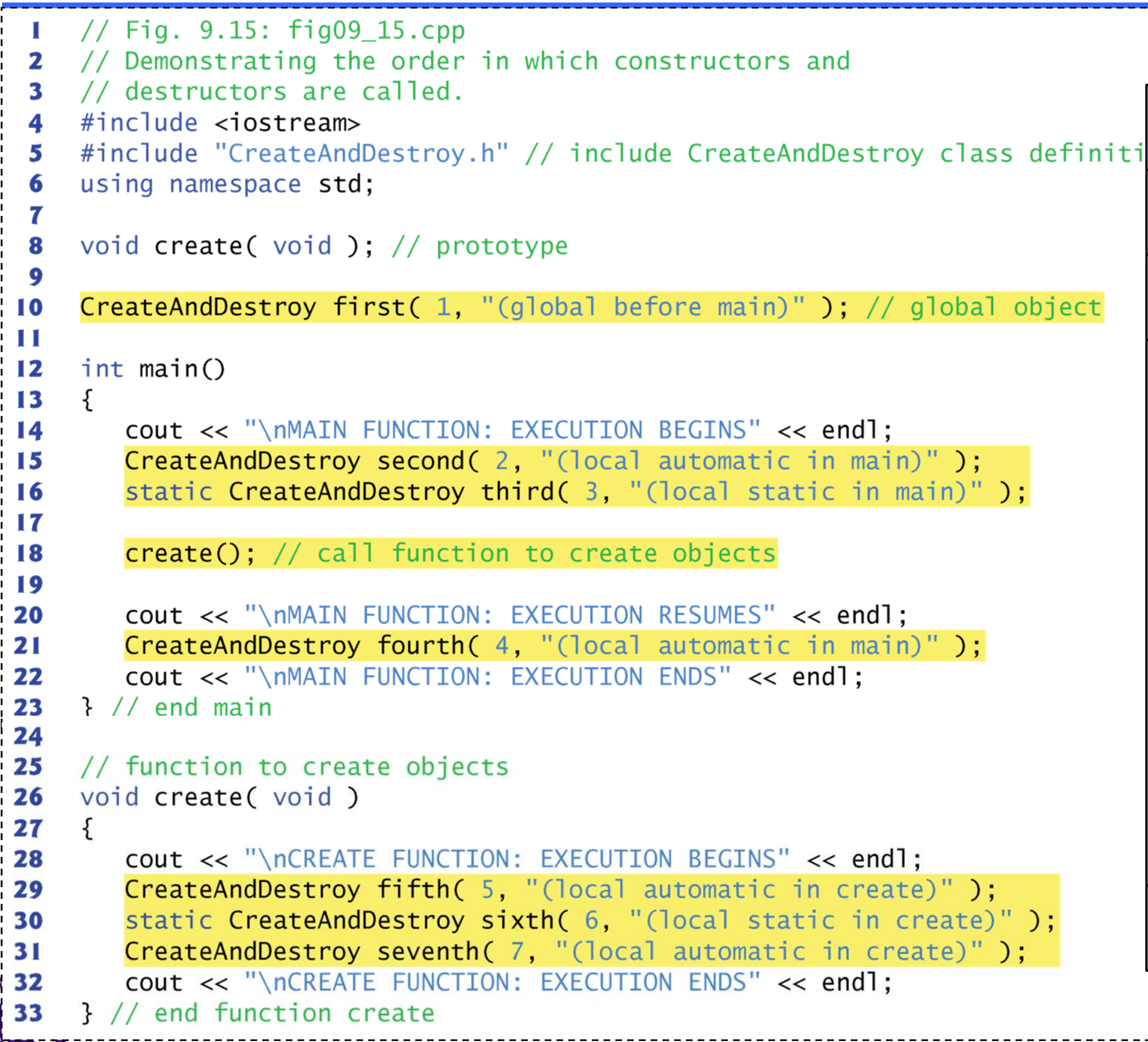
- Destructor
  - Destructor itself does not actually release the object's memory
    - Objects on stack (automatic objects) are destroyed when leaving the scope
    - Objects on static data section or heap are destroyed when the program terminates
    - Objects on heap can also be destroyed when an explicit deallocation is made
  - Compiler provides an "empty" destructor if none included
  - Generally, destructor calls are made in the reverse order of the corresponding constructor calls

An Example of Constructors/Destructors Being Called

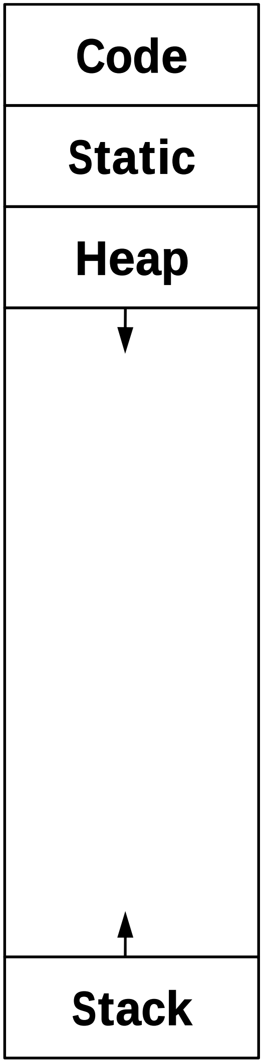


```
// Fig. 9.13: CreateAndDestroy.h
// CreateAndDestroy class definition.
// Member functions defined in CreateAndDestroy.cpp.
#include <string>
using namespace std;

#ifndef CREATE_H
#define CREATE_H

class CreateAndDestroy
{
public:
   CreateAndDestroy( int, string ); // constructor
   ~CreateAndDestroy(); // destructor
private:
   int objectID; // ID number for object
   string message; // message describing object
}; // end class CreateAndDestroy

#endif
```

An Example of Constructors/Destructors Being Called

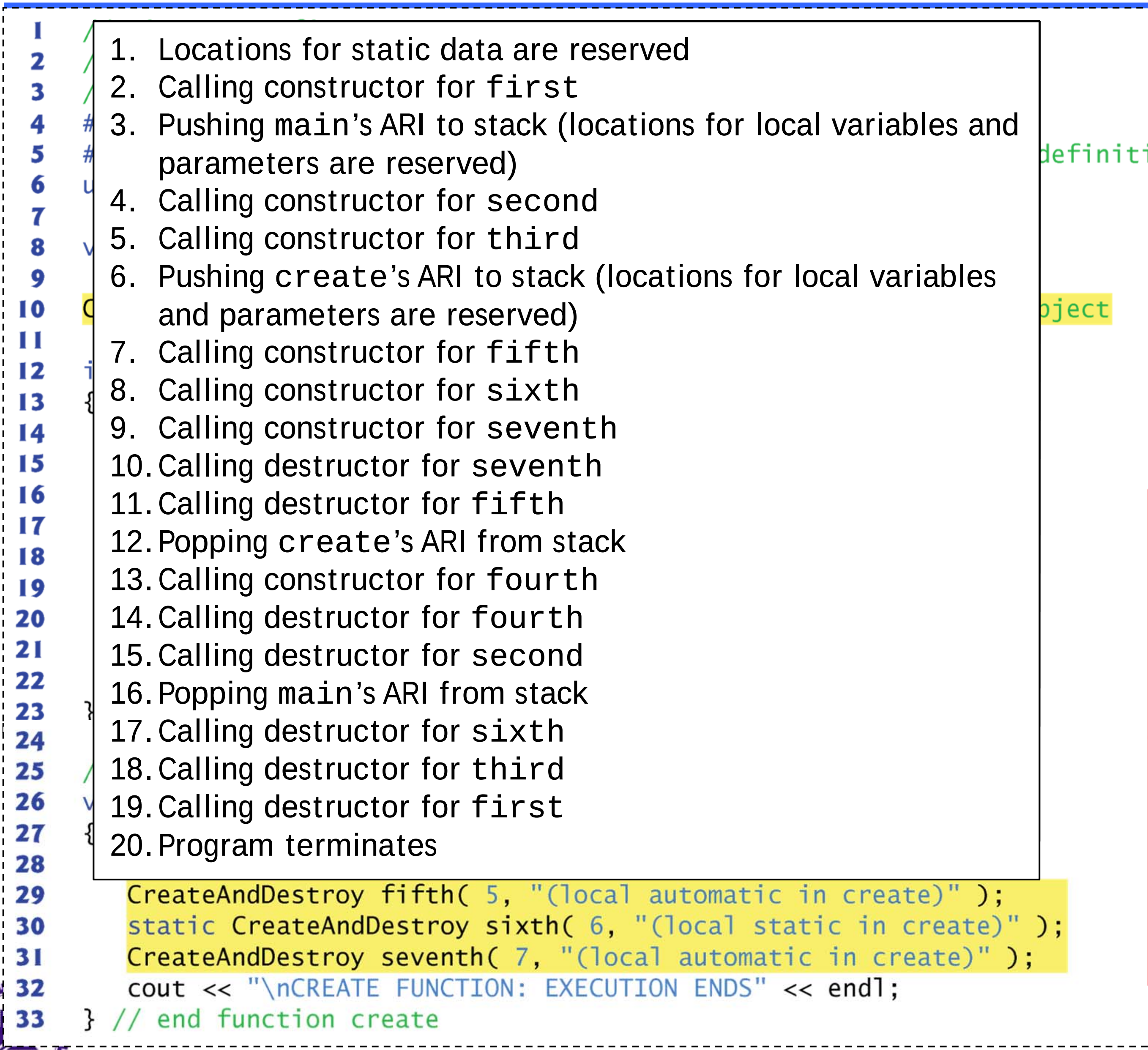


```cpp
// Fig. 9.15: fig09_15.cpp
// Demonstrating the order in which constructors and
// destructors are called.
#include <iostream>
#include "CreateAndDestroy.h" // include CreateAndDestroy class definiti
using namespace std;

void create( void ); // prototype

CreateAndDestroy first( 1, "(global before main)" ); // global object

int main()
{
   cout << "\nMAIN FUNCTION: EXECUTION BEGINS" << endl;
   CreateAndDestroy second( 2, "(local automatic in main)" );
   static CreateAndDestroy third( 3, "(local static in main)" );

   create(); // call function to create objects

   cout << "\nMAIN FUNCTION: EXECUTION RESUMES" << endl;
   CreateAndDestroy fourth( 4, "(local automatic in main)" );
   cout << "\nMAIN FUNCTION: EXECUTION ENDS" << endl;
} // end main

// function to create objects
void create( void )
{
   cout << "\nCREATE FUNCTION: EXECUTION BEGINS" << endl;
   CreateAndDestroy fifth( 5, "(local automatic in create)" );
   static CreateAndDestroy sixth( 6, "(local static in create)" );
   CreateAndDestroy seventh( 7, "(local automatic in create)" );
   cout << "\nCREATE FUNCTION: EXECUTION ENDS" << endl;
} // end function create
```

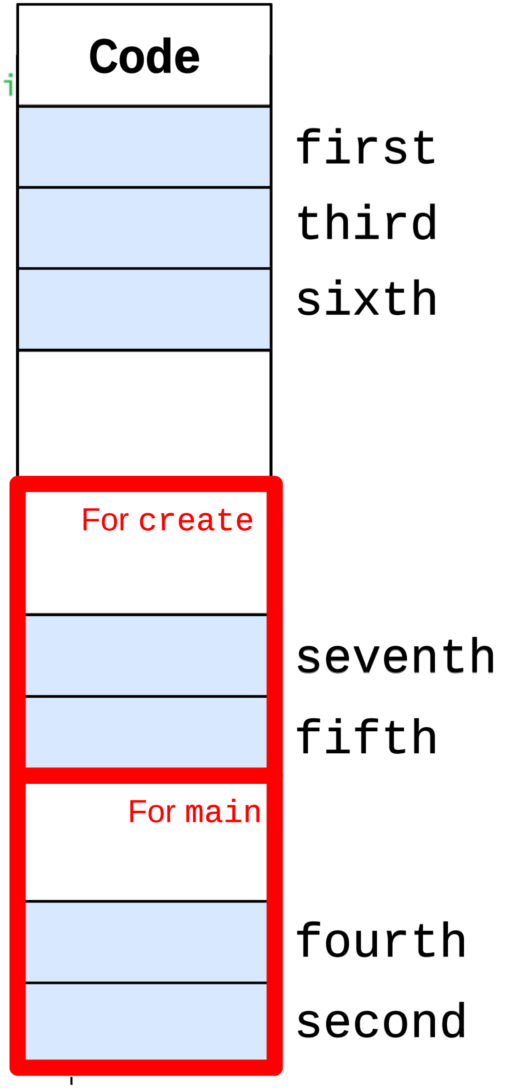


An Example of Constructors/Destructors Being Called

1. Locations for static data are reserved
2. Calling constructor for `first`
3. Pushing `main`'s ARI to stack (locations for local variables and parameters are reserved)
4. Calling constructor for `second`
5. Calling constructor for `third`
6. Pushing `create`'s ARI to stack (locations for local variables and parameters are reserved)
7. Calling constructor for `fifth`
8. Calling constructor for `sixth`
9. Calling constructor for `seventh`
10. Calling destructor for `seventh`
11. Calling destructor for `fifth`
12. Popping `create`'s ARI from stack
13. Calling constructor for `fourth`
14. Calling destructor for `fourth`
15. Calling destructor for `second`
16. Popping `main`'s ARI from stack
17. Calling destructor for `sixth`
18. Calling destructor for `third`
19. Calling destructor for `first`
20. Program terminates



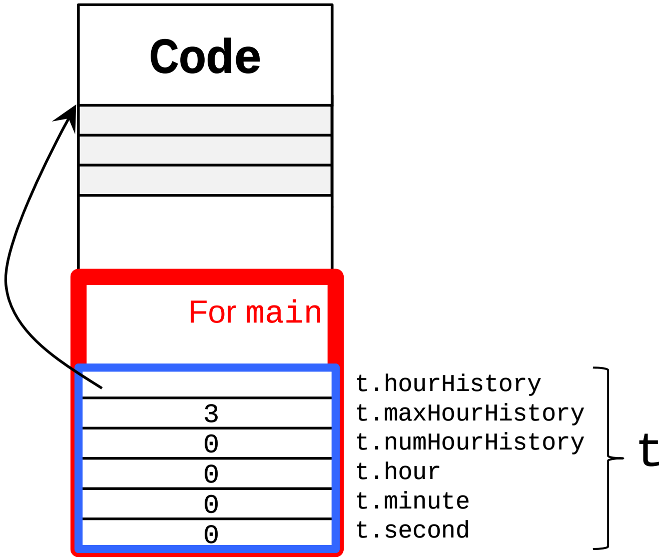
```
   CreateAndDestroy fifth( 5, "(local automatic in create)" );
   static CreateAndDestroy sixth( 6, "(local static in create)" );
   CreateAndDestroy seventh( 7, "(local automatic in create)" );
   cout << "\nCREATE FUNCTION: EXECUTION ENDS" << endl;
} // end function create
```



Rewriting the Time Class

```
class Time {
public:
  Time(int=10, int=0, int=0, int=0); // constructor
  ~Time();                           // destructor
  ...
private:
  int *hourHistory;   // a pointer to array of hour histories
  int maxHourHistory; // max number of hour histories
  int numHourHistory; // number of hour histories
  ...
};
Time::Time(int size, int h, int m, int s) {
  hourHistory = new int[size];
  maxHourHistory = size;
  numHourHistory = 0;
  setTime(h, m, s);
}
Time::~Time() {
  delete [] hourHistory;
}
void Time::setTime(int h, int m, int s) {
  hour = (h >= 0 && h < 24) ? h : 0;
  minute = (m >= 0 && m < 60) ? m : 0;
  second = (s >= 0 && s < 60) ? s : 0;
  hourHistory[numHourHistory++ % maxHourHistory] = hour;
}
```

```
int main() {
  Time t(3);

  return 0;
}
```



Program Termination with exit and abort

- Both functions often are used to terminate a program when an error is detected
  - Require `<cstdlib>`

-
  ```
  exit(1);
  ```
  - The destructors of objects on static data section or heap are called before termination
  - The destructors of automatic objects are not called

-
  ```
  abort();
  ```
  - No destructors are called

Access Functions

- Usually designed for the public to
  - Read or display data

```
void Time::printStandard() { ... }
void Time::printUniversal() { ... }
```

  - Test the truth or falsity of conditions
    - Such functions are often called predicate functions

```
bool Time::isAM() {
  return (hour < 12);
}
```

Utility Functions

- Also called helper functions
- Usually designed for the class (not the public) to support the operation of public member functions
  - They are private member functions

```
void Time::printStandard() {
  cout << convertHour()
    << ":" << setfill('0') << setw(2) << minute << ":"
    << setw(2) << second << (hour < 12 ? " AM" : " PM")
    << endl;
}

int Time::convertHour() {
  return ( (hour == 0 || hour == 12) ? 12 : hour % 12 );
}
```

Set and *Get* Member Functions

- Recall: Each data member of a class should have **`private`** visibility unless it can be proven that the data member needs **`public`** visibility

- `public` member functions
- Allow the client code to *set* and *get* the value of the `private` data members in a constrained manner
  - Set functions also called mutators
  - Get functions also called accessors

Enhancing the Time Class

```
class Time {
public:
   void setHour(int);
   int getHour();
   ...
private:
   int hour;
   int minute;
   int second;
   ...
};
void Time::setHour(int h) {
   hour = (h >= 0 && h < 24) ? h : 0;
   hourHistory[numHourHistory++ % maxHourHistory] = hour;
}
int Time::getHour() {
   return hour;
}
```

```
class Time {
public:
   int hour;
   int minute;
   int second;
   ...
};
```

```
int main() {
   Time t(3);

   t.setHour(30);
   // invalid value is detected

   t.hour = 30;
   // invalid value is set

   return 0;
}
```

Return a Reference/Pointer to a `private` Member

- DON’T DO THIS!!
  - This is a subtle trap
  - This enables the client code to access the class’s `private` members at will
    - Breaks the encapsulation of the class
    - Private members are not private any more

```
int &Time::badSetHour(int h) {
  hour = (h >= 0 && H < 24) ? h : 0;
  return hour;
}
int main() {
  Time t;
  int &hourRef = t.badSetHour(20);
  hourRef = 30; // invalid value is set
  return 0;
}
```

Outline

- Introduction to OOP
  - What Are Objects and Classes?
  - Review of OOP
- Classes (Abstract Data Types)
  - Overview of Classes
  - Designing A `Time` Class
  - Designing Member Functions
  - Memberwise Assignment Operation
  - Passing and Returning Objects

Default Memberwise Assignment

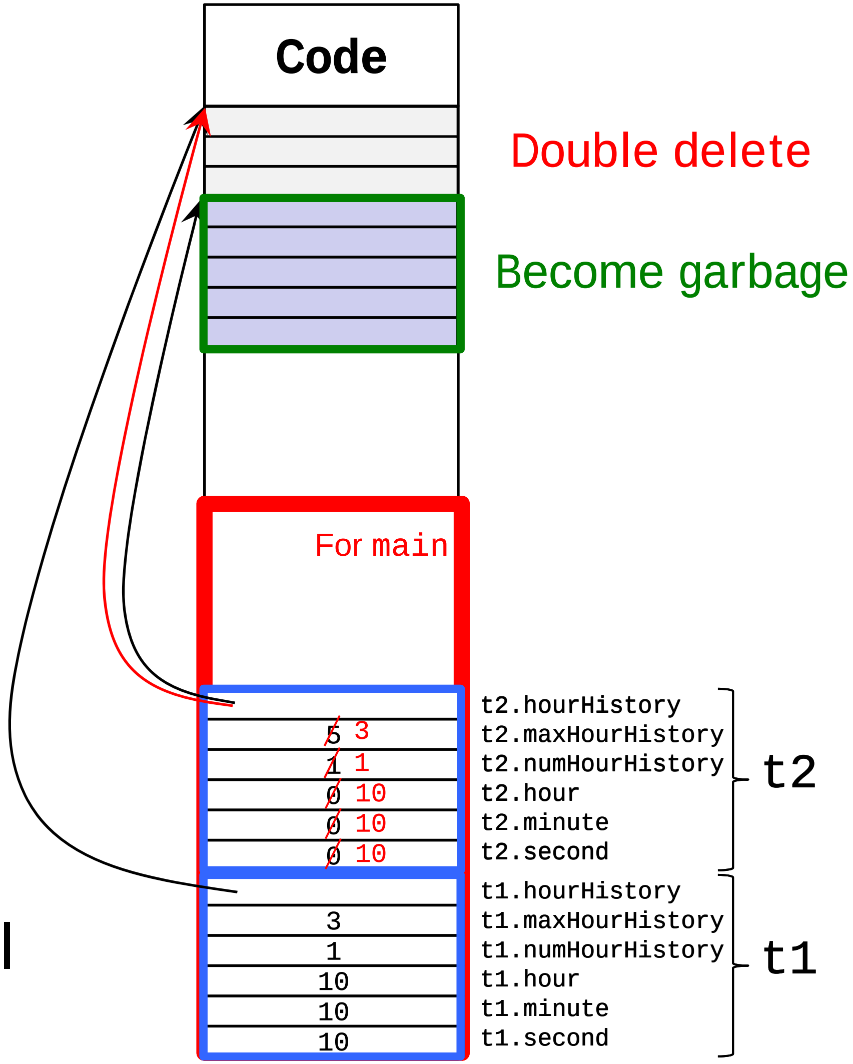
- Assignment operator (=)
  - Can be used to assign an object to another object of the same type
    - Each data member of the right object is assigned to the same data member in the left object
  - Can cause serious problems when data members contain pointers to dynamically allocated memory

Memberwise Assignment: An Example

```
#include “time.h”

int main() {
   Time t1(3,10,10,10);
   Time t2(5);
   t1 = t2;

   return 0;
}
```

time.h

```
Time::~Time() {
   delete [] hourHistory;
}
```



- Solutions to this problem will be discussed in Chapter 11 (Operator Overloading)

Outline

- Introduction to OOP
  - What Are Objects and Classes?
  - Review of OOP
- Classes (Abstract Data Types)
  - Overview of Classes
  - Designing A `Time` Class
  - Designing Member Functions
  - Memberwise Assignment Operation
  - Passing and Returning Objects

Passing and Returning Objects

- Similar to variables, objects may be passed as function arguments and may be returned from functions
  - Using pass-by-value by default (a copy of the object is passed or returned)
  - The copy constructor of the class that the object derived from will be called to create the new object

Default Copy Constructor

- For each class, the compiler provides a default copy constructor
  - Copies each member of the original object into the corresponding member of the new object
    - Can cause serious problems when data members contain pointers to dynamically allocated memory
    - More discussions along with the discussions about the problem on memberwise assignment will be made in Chapter 11 (Operator Overloading)

Default Copy Constructor for Class `Time`

time.h

```
class Time {
   ...
   Time(const Time &); // default copy constructor
   ...
   int hour;
   int minute;
   int second;
   ...
};
```

Must receive a reference to prevent infinite recursion, calling each object’s copy constructor again and again

time.cpp

```
Time::Time(const Time &t) {
   hour = t.hour;
   minute = t.minute;
   second = t.second;
} // copies all data members
```

Copy Constructor

- A *copy constructor* is called whenever a new variable is created from an object
  - An object is passed or returned by value
  - An object is declared and *initialized from another object*

```
Time sunset;      // constructor is used to build
                  // sunset
Time t1(sunset);  // copy constructor is used to
                  // build t1
Time t2 = t1;     // copy constructor is used to
                  // initialize t2 in declaration
t2 = t1; // Assignment operator, no constructor
         // or copy constructor is used
```