# OOP/Computer Programming

Instructor: Dr. Danish Shehzad

# Review

**Class**

- A Class is a user defined data type.
- Template for creating objects.
- Consists of data members and member functions.

**Object**

- The instances of the class are called Objects.

### Data Members

- Data members are the data variables

### Member Functions

- Member functions are the functions that operate on the data encapsulated in the class
- Public member functions are the interface to the class

# Member Functions (contd.)

INLINE FUNCTIONS

- Define member function inside the class definition

OR

OUT_OF-LINE FUNCTIONS

- Define member function outside the class definition
  - But they must be declared inside class definition

# Example: inline

```
class Student{
  int rollNo;
public:
  void setRollNo(int aRollNo){
     rollNo = aRollNo;
  }
};
```

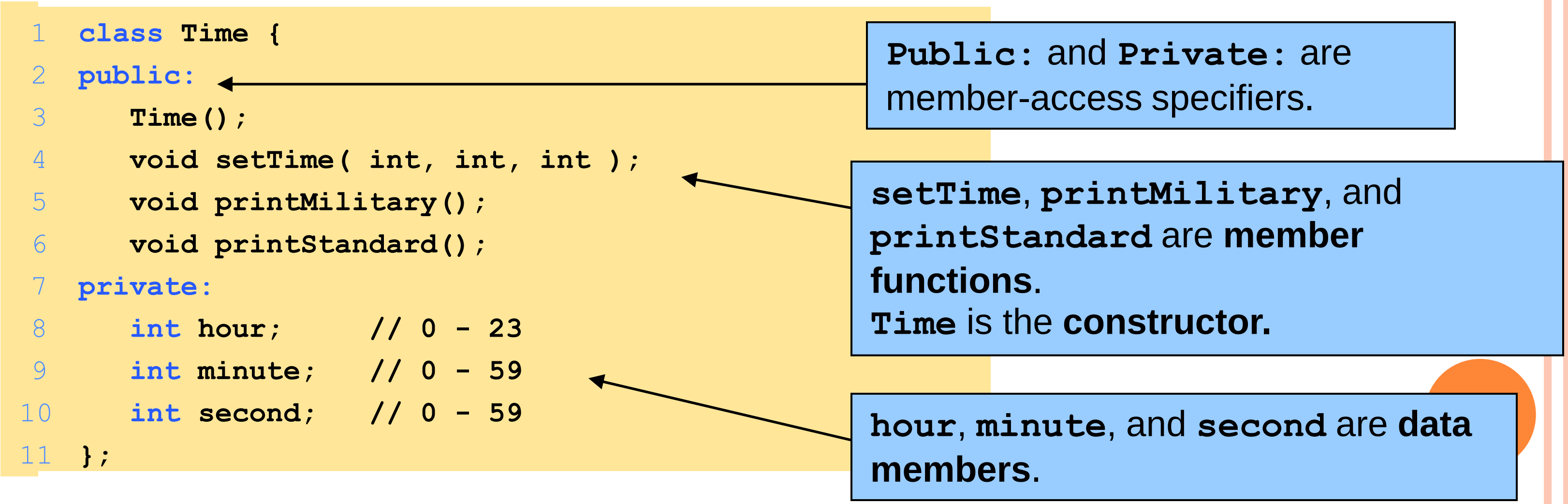
# Function Outside Class Body

```
class ClassName{
  …
  public:
  ReturnType FunctionName();
};
ReturnType ClassName::FunctionName()
{
  …
}
```

Scope resolution operator

# Review

- Classes
  - Model objects that have attributes (data members) and behaviors (member functions)
  - Defined using keyword **`class`**
  - Have a body delineated with braces (**`{`** and **`}`**)
  - Class definitions terminate with a semicolon



```
class Time {
public:
   Time();
   void setTime( int, int, int );
   void printMilitary();
   void printStandard();
private:
   int hour;     // 0 - 23
   int minute;   // 0 - 59
   int second;   // 0 - 59
};
```

**`Public:`** and **`Private:`** are member-access specifiers.

**`setTime`**, **`printMilitary`**, and **`printStandard`** are **member functions.**
**`Time`** is the **constructor.**

**`hour`**, **`minute`**, and **`second`** are **data members.**

# Review

- Class definition and declaration
  - Once a class has been defined, it can be used as a type in object, array and pointer declarations
  - Example:

```
Time sunset,                      // object of type Time
     arrayOfTimes[ 5 ],           // array of Time objects
     *pointerToTime,              // pointer to a Time object
     &dinnerTime = sunset;        // reference to a Time object
```

Note: The class name becomes the new type specifier.

```cpp

// Print Time in standard format
void Time::printStandard()
{
   cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
        << ":" << ( minute < 10 ? "0" : "" ) << minute
        << ":" << ( second < 10 ? "0" : "" ) << second
        << ( hour < 12 ? " AM" : " PM" );
}

// Driver to test simple class Time
int main()
{
   Time t;  // instantiate object t of class

   cout << "The initial military time is ";
   t.printMilitary();
   cout << "\nThe initial standard time is ";
   t.printStandard();

```

```
The initial military time is 00:00
The initial standard time is 12:00:00 AM
```

Notice how functions are called using the dot (.) operator.

```cpp
   t.setTime( 13, 27, 6 );
   cout << "\n\nMilitary time after setTime is ";
   t.printMilitary();
   cout << "\nStandard time after setTime is ";
   t.printStandard();

   t.setTime( 99, 99, 99 );  // attempt invalid settings
   cout << "\n\nAfter attempting invalid settings:"
        << "\nMilitary time: ";
   t.printMilitary();
   cout << "\nStandard time: ";
   t.printStandard();
   cout << endl;
   return 0;
}
```

```
Military time after setTime is 13:27
Standard time after setTime is 1:27:06 PM
```

```
After attempting invalid settings:
Military time: 00:00
Standard time: 12:00:00 AM
```

```
The initial military time is 00:00
The initial standard time is 12:00:00 AM

Military time after setTime is 13:27
Standard time after setTime is 1:27:06 PM

After attempting invalid settings:
Military time: 00:00
Standard time: 12:00:00 AM
```

# Interface vs Implementation

- Separating interface from implementation
  - Makes it easier to modify programs
  - Header files
    - Contains class definitions and function prototypes
  - Source-code files
    - Contains member function definitions

# Separation of interface and implementation

- Usually functions are defined in implementation files (.cpp) while the class definition is given in header file (.h)
- Some authors also consider this as separation of interface and implementation

# Student.h

```
class Student{
  int rollNo;
public:
  void setRollNo(int aRollNo);
  int getRollNo();
  …
};
```

# Student.cpp

```
#include “student.h”

void Student::setRollNo(int aNo){
  …
}
int Student::getRollNo(){
…
}
```

# Driver.cpp

```
#include “student.h”

int main(){
  Student aStudent;
}
```

```cpp
// Fig. 6.5: time1.h
// Declaration of the Time class.
// Member functions are defined in time1.cpp

// prevent multiple inclusions of header file
#ifndef TIME1_H
#define TIME1_H

// Time abstract data type definition
class Time {
public:
   Time();                        // constructor
   void setTime( int, int, int ); // set hour, minute, second
   void printMilitary();          // print military time format
   void printStandard();          // print standard time format
private:
   int hour;     // 0 - 23
   int minute;   // 0 - 59
   int second;   // 0 - 59
};

#endif
```

Dot ( **.** ) replaced with underscore ( _ ) in file name.

If **`time1.h`** (**`TIME1_H`**) is not defined (**`#ifndef`**) then it is loaded (**`#define TIME1_H`**). If **`TIME1_H`** *is* already defined, then everything up to **`#endif`** is ignored.
This prevents loading a header file multiple times.

```cpp
// Fig. 6.5: time1.cpp
// Member function definitions for Time class.
#include <iostream>

using std::cout;

#include "time1.h"

// Time constructor initializes each data member to zero.
// Ensures all Time objects start in a consistent state.
Time::Time() { hour = minute = second = 0; }

// Set a new Time value using military time. Perform validity
// checks on the data values. Set invalid values to zero.
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;
}

// Print Time in military format
void Time::printMilitary()
{
   cout << ( hour < 10 ? "0" : "" ) << hour << ":"
        << ( minute < 10 ? "0" : "" ) << minute;
}

// Print time in standard format
void Time::printStandard()
{
   cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
        << ":" << ( minute < 10 ? "0" : "" ) << minute
        << ":" << ( second < 10 ? "0" : "" ) << second
        << ( hour < 12 ? " AM" : " PM" );
}
```

Source file uses **`#include`** to load the header file

Source file contains function definitions

```
#include <iostream>

#include "time1.h"
```

```
// Driver to test simple class Time
int main()
{
   Time t;  // instantiate object t of class

   cout << "The initial military time is ";
   t.printMilitary();
   cout << "\nThe initial standard time is ";
   t.printStandard();

```

```
The initial military time is 00:00
The initial standard time is 12:00:00 AM
```

Notice how functions are called using the dot (.) operator.

# Initializing Objects

- Constructors are special class methods that are called when a class creates its object
  - they have the same name as their class
  - they don’t return anything

Example:

```
class Employee
{
        public:
                Employee ()
                {
                } // default constructor
}
```

# Constructors and Destructors

- Constructor is a function in every class which is called when class creates its object
  - Basically it helps in initializing data members of the class
  - A class may have multiple constructors

- Destructor is a function in every class which is called when the object of a class is destroyed
  - The main purpose of destructor is to remove all dynamic memories

# Initializing Objects

```
class Employee
{
        public:
                Employee ()
                {
                        cout << “Employee’s class object is created”;
                } // default constructor
                int var1;
}
void main (void)
{
        Employee emp;
}
```

Program Output:

```
        Employee’s class object is created
```

# When do Constructors Get Called?

- Class object creation time
- Dynamically allocated object
- Class argument passed by value
- Class object returned by value
- Array element

# What Constructors Do

- Have same name as the class
- Are called automatically when class object is declared
- Help in initializing (static) members
  - Employee() { id = 0; }
- Allocate memory for dynamic members
  - Employee() { char* nameptr = new char[20];}
- Allocate any needed resources

# Date Class in C++

```
class Date
{
  public: // services
    Date();
    Date (int, int , int );
    int getMonth();
    void incrDay();
    Int getDay();
    // more services...
  private: // state
    int year, day, month;
};
```

# Constructors for the Date Class

- default constructor

  **Date today;**

  we can’t initialize variable (day, month, year) values of object “today” with dot operator. Reason day, month and year variables are private.

- Another constructor (constructor overloading).

  **Date today(9,20,1999);**

  This constructor initializes values of variables (day, month, and year)

# Compiler-Generated Constructor

- What if we do not supply any constructor for a class?
- Compiler generates one with an empty body

```
Date() {}
```

- No data members initialized
- Do not rely on compiler-generated constructors

# Default Constructor

- Takes no arguments, or
- All arguments have default values

```
Date::Date() {…}


Date::Date(int m ,
          int d,
          int y) { … }
```

# Date::Date() Constructor

```
Date::Date () {
  //get current date from the system //and
  store it in date members.
  month = 0;
  day   = 0;
  year  = 0;
}
```

- Default Constructor is called when an object is declared without any arguments

```
Date d;
```

# Constructors for the Date Class

```
class Date
{
   public:
       Date(); // default constructor
       Date(int m, int d, int y); // explicitly specifying m,d,y
   private:
       int month, day, year;
};

Date::Date(int m, int d, int y)
{     month = m;
      day = d;
      year = y;
}
void main (void)
{
         Date dat(1,12,2012);
}
```

# Multiple Constructors

- What if you wanted your program to be able to create Date objects in a variety of formats?
  - Date today;
  - Date today("Sept.,20, 1999");
  - Date today(9,20,1999),
  - Date same_as_today(today);

- To do this we must have different versions of the Date constructor.

# Overloading Constructors

- Multiple ways to initialize a Date object
  - from Month, Day, Year
  - from a date string in a known format
  - from another Date object
  - …
- Overloaded constructors
  - different signatures (types and numbers of arguments)

# Calling Overloaded Constructors

```
Date today; // default constructor

// explicitly specify m, d, y
Date fdc(12, 31, 2000);

// initialize date from string
Date eoq(“12/12/1998”);

// from another date object
Date eoq2(eoq);
```

# Copy Constructor

- Initializes a new object from another, existing one
- Signature:

```
Class::Class(Class obj)
{


}
```

# Copy Constructor for Class Date

```
Date::Date(Date dat)
{
// no need to check passed date
  arg
   month = dat.month;
   day   = dat.day;
   year  = dat.year;
}
```

# Uses of the Copy Constructor

- Implicitly called in 3 situations
  - defining a new object from an existing object
  - passing an object by value
  - returning an object by value

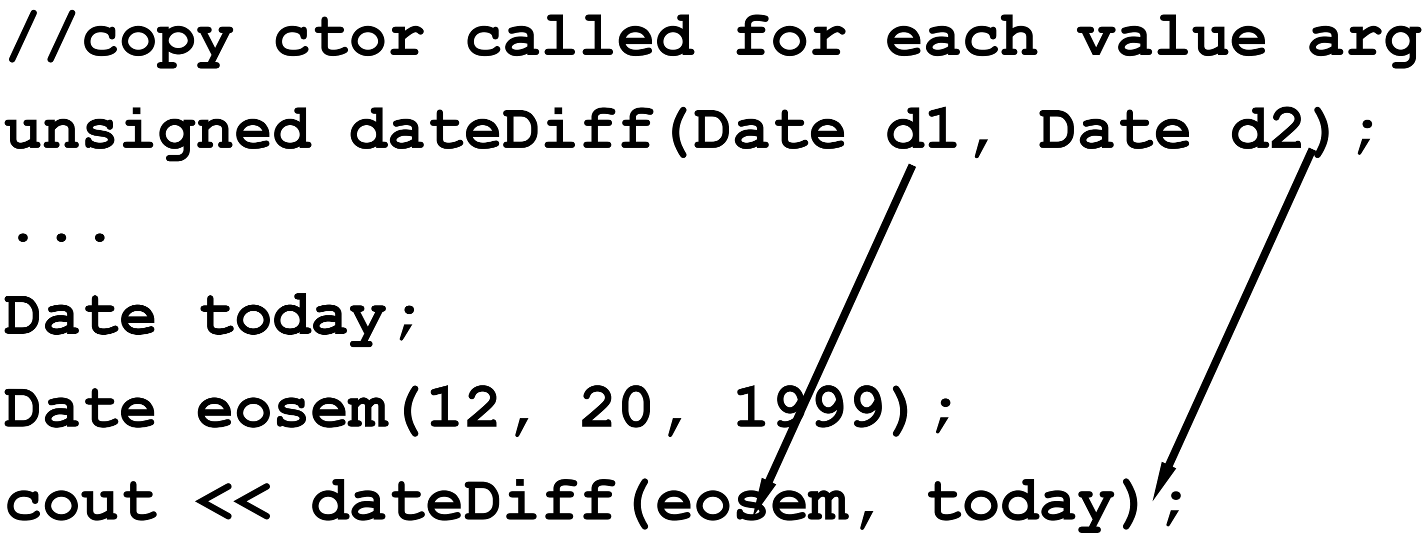
# Copy Constructor:
# Defining a New Object

```
Date eosem(“12/20/1999”);

// init 2 local objects from eosem
Date eosem2(eosem);// pass by value
Date eosem3 = eosem;// return value

// init a dynamic object from eosem
Date* pdate = new Date(eosem);
```

# Copy Constructor: Passing Objects by Value



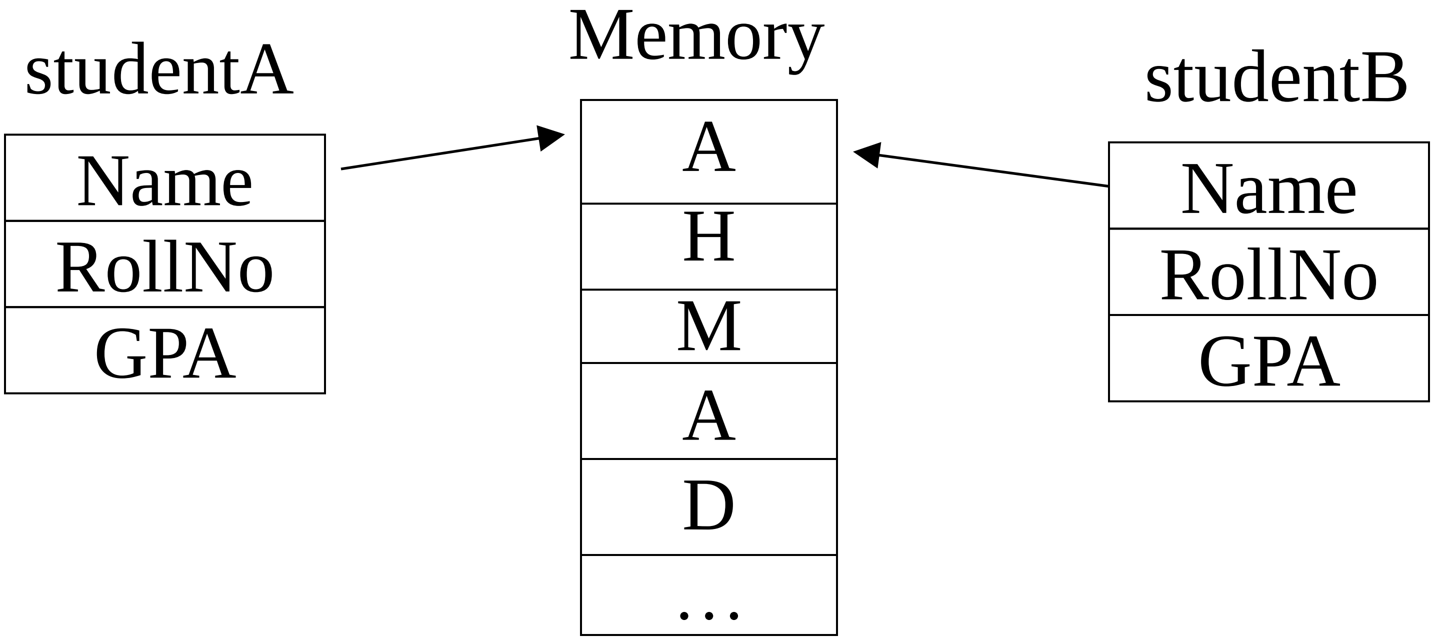
```
//copy ctor called for each value arg
unsigned dateDiff(Date d1, Date d2);
...
Date today;
Date eosem(12, 20, 1999);
cout << dateDiff(eosem, today);
```

# Shallow Copy

- When we initialize one object with another then the compiler copies state of one object to the other
- This kind of copying is called shallow copying

# EXAMPLE

```
Student studentA;
Student studentB = studentA;
```



studentA

Name

RollNo

GPA

Memory

A

H

M

A

D

…

studentB

Name

RollNo

GPA

# Deep Copy

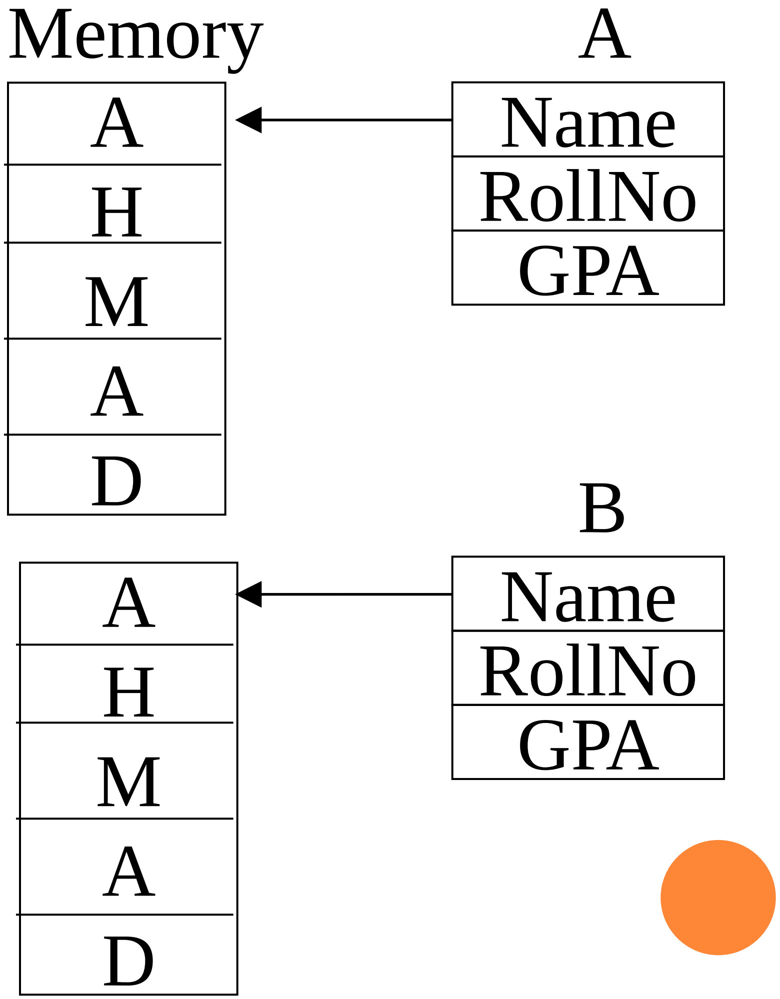
- Copy constructor is normally used to perform deep copy
- If we do not make a copy constructor then the compiler performs shallow copy

# Copy Constructor

```
Student::Student(
        const Student & obj){
 int len = strlen(obj.name);
 name = new char[len+1]
 strcpy(name, obj.name);
 …
 //copy rest of the data
 members
}
```

# EXAMPLE

```
Student studentA;
Student studentB = studentA;
```



Memory

A

H

M

A

D

A

Name

RollNo

GPA

A

H

M

A

D

B

Name

RollNo

GPA

# Destructors

- **What is destructor?**
  Destructor is a member function which destructs or deletes an object.
- **When is destructor called?**
  A destructor function is called automatically when the object goes out of scope:
  (1) the function ends
  (2) the program ends
  (3) a block containing local variables ends
  (4) a delete operator is called

# Local Object

```
{
  Date today; // constructor
             // implicitly called
   …

}  // destructor implicitly
  // called here
```

# Dynamic Object

```
{
   pstr = new string(“5113”);
   ...
   delete pstr; // destructor
  called
   ...
}
```

# CLASSNAME::~CLASSNAME(){}

- Cleanup is as important as initialization and is guaranteed through the use of destructors.

- Destructors never have any arguments, because it does not need any options.

# Destructor Example

```
class Employee
{
        public:
                ~Employee ()
                {
                        cout << “Employee’s class object is deleted”;
                }
                int var1;
}
void main (void)
{
        Employee emp;
} // destructor will call here
```

Program Output:

```
        Employee’s class object is created
```

```
class CreateAndDestroy {
public:
   CreateAndDestroy( int );  // constructor
   ~CreateAndDestroy();      // destructor
private:
   int data;
};

#endif
```

```

CreateAndDestroy::CreateAndDestroy( int value )
{
   data = value;
   cout << "Object " << data << "   constructor";
}

CreateAndDestroy::~CreateAndDestroy()
   { cout << "Object " << data << "   destructor " << endl; }
```

Constructor and Destructor
changed to print when they
called.

```

void create( void );   // prototy

CreateAndDestroy first( 1 );  //

int main()
{
   cout << "   (global created before main)" << endl;

   CreateAndDestroy second( 2 );        // local object
   cout << "   (local automatic in main)" << endl;

   static CreateAndDestroy third( 3 );  // local object
   cout << "   (local static in main)" << endl;

   create();  // call function to create objects

   CreateAndDestroy fourth( 4 );        // local object
   cout << "   (local automatic in main)" << endl;
   return 0;
}
```

```

// Function to create objects
void create( void )
{
   CreateAndDestroy fifth( 5 );
   cout << "   (local automatic in create)" << endl;

   static CreateAndDestroy sixth( 6 );
   cout << "   (local static in create)" << endl;

   CreateAndDestroy seventh( 7 );
   cout << "   (local automatic in create)" << endl;
}
```

```
OUTPUT
Object 1   constructor   (global created before main)
Object 2   constructor   (local automatic in main)
Object 3   constructor   (local static in main)
Object 5   constructor   (local automatic in create)
Object 6   constructor   (local static in create)
Object 7   constructor   (local automatic in create)
Object 7   destructor
Object 5   destructor
Object 4   constructor   (local automatic in main)
Object 4   destructor
Object 2   destructor
Object 6   destructor
Object 3   destructor
Object 1   destructor
```

Notice how the order of the constructor and destructor call depends on the types of variables (automatic, global and `static`) they are associated with.