INHERITANCE

5.1 Introduction

Inheritance is a useful feature in object-oriented programming (OOP) supported by C++.

It allows a new class to include the members (data and functions) of an existing class.

The new class can also have its own members. This is called **inheritance**.

- The existing class is called the **base class** or **parent class** or **superclass**.
- The new class is called the **derived class** or **child class** or **subclass**.

Syntax:

```
class DerivedClass : accessSpecifier BaseClass
{
  // definition of derived class
};
```

Example:

```
class A { };

class B : public A
{
  // new features of class B
};
```

In this, `public` is the access specifier. Other access specifiers will be explained later.

5.1.1 Effects of Inheritance

Inheritance affects the derived class in two ways:

1. The derived class object contains data members of both base and derived classes, so it is larger in size.

    a. Exception: If both classes have no data members, both objects are 1 byte each.
2. You can call public functions of both the base and derived classes using a derived class object.
    a. (Exceptions exist, to be discussed later.)

Example (Listing 5.1):

A.h

```
class A
{
  int x;
public:
  void setX(const int = 0);
  int getX() const;
};
```

A.cpp

```
#include "A.h"
void A::setX(const int pX) { x = pX; }
int A::getX() const { return x; }
```

B.h

```
#include "A.h"
class B : public A
{
  int y;
public:
  void setY(const int = 0);
  int getY() const;
};
```

B.cpp

```
#include "B.h"
void B::setY(const int pY) { y = pY; }
int B::getY() const { return y; }
```

inherit.cpp

```
#include <iostream.h>
#include "B.h"

void main()
{
  cout << sizeof(A) << endl << sizeof(B) << endl;
  B B1;
  B1.setX(1);     // calling base class function
  B1.setY(3);     // calling derived class function
  cout << B1.getX() << endl;  // base class function
  cout << B1.getY() << endl;  // derived class function
}
```

Output:

```
4
8
1
3
```

Explanation:

Object of B has 2 integers (1 from A, 1 from B), so size is 8.

Functions from both A and B can be used with B object.

Inheritance means derived class **"is-a"** base class.

Example: Aircraft is a type of Vehicle.

But containership means **"has-a"** relationship.

Example: Aircraft **has** engines.

Another example:

```
class employee
{
  String name;
  double basic;
  Date doj;
```

```
  // rest of class
};

class manager : public employee
{
  employee* list;  // manager has list of employees
  // rest of class
};
```

So, derived class = specialized base class.

Inheritance is also called **specialization**.

5.1.2 Benefits of Inheritance

Derived class adds only the extra members.

Common data/functions go in base class.

So, inheritance helps in **code reusability**.

5.1.3 Inheritance in Actual Practice

A library programmer makes a class.

Another programmer can inherit from it and add new members.

No need to modify base class.

5.1.4 Base Class and Derived Class Objects

Some think derived class object comes from a base class object — that's **wrong**.

Each object is separate in memory.

5.1.5 Accessing Members of the Base Class in the Derived Class

Only **public** (and **protected**, explained later) members of base class can be accessed in derived class.

Private members **cannot** be accessed directly in derived class.

Example (wrong way):

```
void B::setY()
{
  x = y; // ERROR: x is private in base class A
```

```
}
```

Example (correct way):

```
void B::setY(const int q)
{
  y = q;
  setX(y);  // calls base class function
}
```

This protects private data. Only base class functions can access its private members.

If some base class functions are missing, inheritance **should not** be used to fix it.

Instead, base class should be corrected.

Inheritance adds **extra** features only, not fixes.

Friendship is not inherited.

Example (Listing 5.2):

```
class B;
class A
{
  friend class B;
  int x;
};

class B
{
  void fB(A * p)
  {
    p->x = 0; // OK: B is a friend of A
  }
};

class C : public B
{
  void fC(A * p)
  {
    p->x = 0; // ERROR: C is NOT a friend of A
```

```
    }
};
```

Even though class C is derived from B (which is a friend of A),

C is not a friend of A.

Friendship does not pass through inheritance.

5.2 Base Class and Derived Class Pointers

- A **base class pointer** can point to a **derived class object** — **this is allowed and safe**.
- A **derived class pointer cannot** point to a **base class object** — unless you use **typecasting**, but that's **unsafe** and can lead to **runtime errors**.

☑ Why base class pointer → derived object is safe

Let's look at these simple classes:

```
// A.h
class A {
public:
    int x;
};

// B.h
#include "A.h"
class B : public A {
public:
    int y;
};
```

And the main program:

```
// BasePtr01.cpp
#include "B.h"

void main() {
    A* APtr;
    B B1;
    APtr = &B1;     // OK: base class pointer points to derived
object
    APtr->x = 10;   // OK: 'x' is in base class A
    APtr->y = 20;   // ❌ ERROR: 'y' is not in A
}
```

- Even though B1 has both x and y, `APtr` can **only access x**, because `APtr` is of type A*.
- It knows only what's defined in A. So, accessing y (from B) is not allowed.
- **Memory safety**: `APtr` can only access the memory of A part inside B1, which is safe.

📌 **Important:** This is common in C++ — for example, in polymorphism.

❌ Why derived class pointer → base object is not allowed

```
// DerivedPtr01.cpp
#include "B.h"

void main() {
    A A1;
    B* BPtr;
    BPtr = &A1;     // ❌ ERROR: Can't convert A* to B*
    BPtr->x = 10;   // ❌ Dangerous
    BPtr->y = 20;   // ❌ Very dangerous
}
```

- `BPtr` is expecting a B-type object, which has both x and y.
- But A1 is only 4 bytes (for x). `BPtr->y = 20;` will try to write to memory that doesn't belong to A1 — this is **undefined behavior**.

💥 That's why C++ **doesn't allow** this.

⚠ You can force it using typecasting, but it’s dangerous

```
// DerivedPtrTypeCast.cpp
#include "B.h"

void main() {
    A A1;
    B* BPtr;
    BPtr = (B*)&A1; // ⚠️ Forced typecast — unsafe!
}
```

This compiles, but you're risking a crash or corrupt data.

📌 With Private Data and Functions (Safer Example)

```
// A.h
class A {
    int x;
public:
    void setx(const int = 0);
};

// B.h
#include "A.h"
class B : public A {
    int y;
public:
    void sety(const int = 0);
};
```

Now in main:

```
// BasePtr02.cpp
#include "B.h"

void main() {
    A* APtr;
```

```
    B B1;
    APtr = &B1;        // OK
    APtr->setx(10);    // OK
    APtr->sety(20);    // ❌ ERROR: sety() is not in A
}
```

APtr only knows about functions in A.

⚠ Derived class pointer pointing to base class object

```
// DerivedPtr02.cpp
#include "B.h"

void main() {
    A A1;
    B* BPtr;
    BPtr = &A1;          // ❌ ERROR: Can't convert A* to B*
    BPtr->setx(10);      // DANGEROUS
    BPtr->sety(20);      // DANGEROUS
}
```

Even if you force this, the memory layout is wrong, and you risk writing to an invalid memory location.

🔖 What about the this pointer?

When you call:

```
B1.setx(10);
```

It internally becomes:

```
setx(&B1, 10);
```

The `this` pointer in `A::setx()` is of type `A* const`. So it works even if `B1` is a derived object.

```
void setx(A* const this, const int p) {
    this->x = p;
}
```

That's why base class functions can safely work on derived class objects — they only use the base part.

🔁 What is Function Overriding?

Function overriding happens when:

1. A **base class** has a member function.
2. The **derived class** defines a function with **the same name and signature**.

Example:

```
class A {
public:
    void show() {
        cout << "show() function of class A called\n";
    }
};

class B : public A {
public:
    void show() { // overrides A::show()
        cout << "show() function of class B called\n";
    }
};
```

Now, if you do:

```
B b1;
b1.show();  // Calls B::show()
```

📍 How the Compiler Works Behind the Scenes

When you write:

```
b1.show();
```

The compiler first looks in class B. If B has a function called `show()`, it calls that.

If it didn't exist in B, the compiler would then check class A.

So, in your example:

```
B1.A::show(); // Forcefully calling the base version
```

This works because even though B *overrides* `A::show()`, the base version still exists and can be accessed using the **scope resolution operator**.

🧠 Hidden Insight: Different Signatures Internally

Even though the signatures look the same:

```
void show()
```

Internally, due to the `this` pointer:

- `A::show()` is really: `void show(A * const this)`
- `B::show()` is really: `void show(B * const this)`

So they are technically **different functions**, bound to different object types. That's why **you need `virtual`** if you want **polymorphism** — where the function call is resolved at runtime.

🤔 Why Function Overriding Without `virtual`?

You're right to wonder — *"what's the point if this doesn't do runtime polymorphism?"*

✅ It's still useful when:

- You're working with **objects** directly (not pointers or references).
- You want derived classes to provide different behavior.

But if you want:

```
A* ptr = new B;
ptr->show(); // Calls A::show() — NOT what we want!
```

You need this:

```
class A {
public:
    virtual void show() { ... }
};
```

Then it works like magic:

```
A* ptr = new B;
ptr->show();  // Now calls B::show() — thanks to 'virtual'
```

☑️ Summary

Concept	What Happens
Function overriding	Derived class replaces base class method
No `virtual`	Function resolved at compile time
With `virtual`	Function resolved at **runtime**
`Scope Resolution (::)`	Call base class version manually

5.4 Base Class Initialization

When you create an object of a derived class, it contains data members from both the base class and the derived class. We often need to initialize both sets of data members. Here's how this works:

Constructors and Object Creation

- When you create an object of a derived class, the compiler automatically calls the constructor of the base class first, followed by the constructor of the derived class.
- For example, if A is the base class and B is the derived class, when you write:

```
B B1;
```

This is internally converted into:

```
B B1;  // Memory allocated for the object
B1.A(); // Base class constructor called (implicitly)
B1.B(); // Derived class constructor called (implicitly)
```

Destructors

- Destructors are called in the reverse order.
- The base class destructor is called first when the object is destroyed, followed by the derived class destructor.

Issue in Base Class Initialization (Listing 5.15)

In this example, we are going to initialize base class data members, but we make a mistake by not passing the correct parameters to the base class constructor.

Listing 5.15: Unsuccessful Initialization of Base Class Members

A.h (Base Class)

```
class A
{
    int x;  // Data member x
public:
```

```
    A(const int = 0);  // Constructor with a default parameter for x
    void setx(const int = 0);  // Method to set x
    int getx() const;  // Method to get the value of x
};
```

- A is the base class with a private integer x.
- The constructor `A(int p)` takes an optional parameter to initialize x.
- The method `setx()` sets the value of x.
- The method `getx()` returns the value of x.

A.cpp (Base Class Implementation)

```
#include "A.h"

A::A(const int p)
{
    x = p;  // Initialize x with the value of p (defaults to 0 if no
value is passed)
}

void A::setx(const int p)
{
    x = p;
}

int A::getx() const
{
    return x;
}
```

- The constructor `A::A(const int p)` initializes x with the given value (default is 0).
- `setx()` and `getx()` are simple getter and setter methods for x.

B.h (Derived Class)

```
#include "A.h"

class B : public A  // Class B inherits from A
{
```

```cpp
    int y;  // Data member y
public:
    B(const int = 0);  // Constructor with a default parameter for y
    void sety(const int = 0);  // Method to set y
    int gety() const;  // Method to get the value of y
};
```

- B is the derived class from A with an additional integer y.
- `B::B(int q)` is the constructor of B that initializes y.

B.cpp (Derived Class Implementation)

```cpp
#include "B.h"

B::B(const int q)
{
    y = q;  // Initialize y with the value of q
}

void B::sety(const int q)
{
    y = q;
}

int B::gety() const
{
    return y;
}
```

- `B::B(const int q)` initializes y with the value of q.

baseinit01.cpp (Main Program)

```cpp
#include "B.h"
#include <iostream>
using namespace std;

int main()
{
    B B1(20);  // Create an object of class B, passing 20 to the
```

```
constructor
    cout << B1.getx() << endl;  // Print the value of x (from class
A)
    cout << B1.gety() << endl;  // Print the value of y (from class
B)
}
```

Output

```
0
20
```

Explanation of the Output

1. When you create the object B1 using the constructor `B1(20)`, the object B1 is created as:

```
B B1;       // Memory allocated for the object
B1.A();     // Base class constructor called, initializing x to
default 0
B1.B(20);   // Derived class constructor called, initializing y to 20
```

2. **Base Class Constructor (`A::A(int p)`)**: The base class constructor is called with the default value (`0`), so x is initialized to 0.
3. **Derived Class Constructor (`B::B(int q)`)**: The derived class constructor is called with the value `20`, so y is initialized to 20.

Thus, when `B1.getx()` is called, it returns `0` (default value for x), and when `B1.gety()` is called, it returns `20` (value passed to y).

Fixing the Initialization

In the previous example, the base class x was initialized with the default value of `0`. If we want to explicitly pass a value to the base class constructor from the derived class, we need to modify the constructor of the derived class.

Correcting the Derived Class Constructor (Listing 5.16)

To fix the issue and properly initialize the base class data member x, we modify the derived class constructor to pass a value to the base class constructor.

B.h (Corrected Derived Class)

```
#include "A.h"

class B : public A
{
public:
    B(const int p, const int q);  // Constructor with two parameters
for both x and y
};
```

B.cpp (Corrected Derived Class Implementation)

```
#include "B.h"

B::B(const int p, const int q) : A(p)  // Pass p to the base class
constructor
{
    y = q;  // Initialize y with the value of q
}
```

Now, the derived class constructor takes two parameters:

- p is passed to the base class constructor `A(p)`, which initializes x.
- q is used to initialize y in the derived class.

baseinit02.cpp (Corrected Main Program)

```
#include "B.h"
#include <iostream>
using namespace std;

int main()
{
    B B1(10, 20);  // Create an object of class B, passing 10 to the
base class and 20 to the derived class
    cout << B1.getx() << endl;  // Prints 10 (value passed to base
class constructor)
    cout << B1.gety() << endl;  // Prints 20 (value passed to
derived class constructor)
```

```
}
```

Output

```
10
20
```

Final Explanation

When the object `B1` is created with `B B1(10, 20);`, the constructor calls happen in this order:

1. `B1.A(10)` — The base class constructor initializes x with `10`.
2. `B1.B(20)` — The derived class constructor initializes y with `20`.

Thus, the output correctly prints:

- `10` for x (base class data member)
- `20` for y (derived class data member)

Protected Access Specifier

In C++, the `protected` access specifier is the third access modifier, in addition to `public` and `private`. Here's an overview of how it works:

- **Private members**: Can only be accessed by the member functions of the class itself.
- **Protected members**: Can be accessed by the member functions of the class and its derived classes. However, they are not accessible by non-member functions.
- **Public members**: Can be accessed by any function.

Accessing Protected Members (Listing 5.18)

In this example, we demonstrate the use of protected members in a class:

A.h (Base Class)

```
class A
{
private:
```

```
    int x;       // Private member, inaccessible outside class A
protected:
    int y;       // Protected member, accessible in class A and
derived classes
public:
    int z;       // Public member, accessible everywhere
};
```

- **Private x**: Only accessible within class A.
- **Protected y**: Accessible in class A and any derived classes.
- **Public z**: Accessible from anywhere.

B.h (Derived Class)

```
#include "A.h"

class B : public A   // Derived class B from class A
{
public:
    void xyz();      // Method to access members of A
};
```

B.cpp (Derived Class Implementation)

```
#include "B.h"

void B::xyz() // Member function of derived class B
{
    x = 1;  // ERROR: private member 'x' of base class A
    y = 2;  // OK: protected member 'y' of base class A
    z = 3;  // OK: public member 'z' of base class A
}
```

- **`x = 1;`**: This line gives an error because x is private in A, so it is inaccessible in B.
- **`y = 2;`**: This works because y is protected in A, so it is accessible in B.
- **`z = 3;`**: This also works because z is public in A, so it is accessible in B.

protected.cpp (Non-member Function)

```
#include "A.h"

void main()
{
    A *Aptr;
    Aptr->x = 10; // ERROR: private member 'x' of A
    Aptr->y = 20; // ERROR: protected member 'y' of A
    Aptr->z = 30; // OK: public member 'z' of A
}
```

- **`Aptr->x = 10;`**: Error because x is private in A.
- **`Aptr->y = 20;`**: Error because y is protected in A.
- **`Aptr->z = 30;`**: This works because z is public in A.

Deriving by Different Access Specifiers

5.6.1 Deriving by the Public Access Specifier

When you derive a class using the `public` access specifier, the access level of the base class members is retained. Here's what happens:

- **Private members**: Not accessible in the derived class or non-member functions.
- **Protected members**: Accessible in the derived class and its derived classes.
- **Public members**: Accessible in the derived class and its derived classes, and even from non-member functions.

publicInheritance.cpp (Public Inheritance)

```
class A
{
private:
    int x;
protected:
    int y;
public:
    int z;
};
```

```
class B : public A
{
public:
    void f1()
    {
        x = 1;  // ERROR: private member remains private
        y = 2;  // OK: protected member remains protected
        z = 2;  // OK: public member remains public
    }
};

class C : public B
{
public:
    void f2()
    {
        x = 1;  // ERROR: private member remains private
        y = 2;  // OK: protected member remains protected
        z = 2;  // OK: public member remains public
    }
};

void xyz() // Non-member function
{
    B B1;
    B1.z = 100; // OK: Can access public member of A
    A *APtr;
    APtr = &B1;
    APtr->z = 100; // OK: Can access public member of A through base
pointer
}
```

- The public members from class A remain accessible even through a base class pointer.
- Protected members can be accessed within the derived classes.
- Private members remain inaccessible.

5.6.2 Deriving by the Protected Access Specifier

When a class is derived using the `protected` access specifier, the base class members' access levels change:

- **Private members**: Still inaccessible.
- **Protected members**: Become protected in the derived class and accessible by its member functions.
- **Public members**: Become protected in the derived class, making them inaccessible by non-member functions.

```
class A
{
private:
    int x;
protected:
    int y;
public:
    int z;
};

class B : protected A  // Derived with protected access
{
public:

void f1()
    {
        y = 2;  // OK: protected member 'y' remains protected
        z = 2;  // OK: public member 'z' becomes protected
    }
};

void xyz() // Non-member function
{
    B B1;
    B1.z = 100; // ERROR: Cannot access public member 'z' through
protected derived class
    A *APtr;
    APtr = &B1; // ERROR: Cannot make a base class pointer point at
a protected derived class
    APtr->z = 100; // OK: Access public member of A through a base
pointer
}
```

- A base class pointer cannot point to an object of a class derived with protected access.

- Public members become protected in derived classes, making them inaccessible by non-member functions.

5.6.3 Deriving by the Private Access Specifier

When a class is derived using the `private` access specifier, the base class's public and protected members become private in the derived class. Here's what happens:

- **Private members**: Not accessible.
- **Protected members**: Become private in the derived class.
- **Public members**: Also become private in the derived class.

```
class A
{
private:
    int x;
protected:
    int y;
public:
    int z;
};

class B : private A  // Derived with private access
{
public:
    void f1()
    {
        y = 2;  // OK: protected member 'y' becomes private
        z = 2;  // OK: public member 'z' becomes private
    }
};

void xyz() // Non-member function
{
    B B1;
    B1.z = 100; // ERROR: Cannot access public member 'z'
    A *APtr;
    APtr = &B1; // ERROR: Cannot make a base class pointer point at
a private derived class
    APtr->z = 100; // OK: Access public member of A through a base
pointer
```

```
}
```

- Public and protected members from the base class become private in the derived class.
- A base class pointer cannot point to an object of a class derived with private access.

5.7 Different Kinds of Inheritance

5.7.1 Multiple Inheritance

In **multiple inheritance**, a class is derived from more than one base class.

Syntax:

```
class Derived : access Base1, access Base2, ...
{
    // class body
};
```

Diagram

```
A      B
 \    /
   C
```

Example:

```
class A {
private:
    int x;
protected:
    int y;
public:
```

```
    int z;
};

class B : private A {
public:
    void f1() {
        // x = 1; // Not allowed
        y = 2;    // OK
        z = 2;    // OK
    }
};

class C : public B {
public:
    void f2() {
        // y = 2; // Not allowed
        // z = 2; // Not allowed
    }
};
```

Notes:

- Private inheritance: public and protected members of A become private in B.
- In `main()`, object of class B cannot access z directly.
- A base class pointer can point to base class, not derived class privately inherited.

5.7.2 Ambiguities in Multiple Inheritance

1. Identical Members in Base Classes

When two base classes have same function name, it causes ambiguity.

```
class A {
public:

void show() {
        cout << "A\n";
    }
};
```

```
class B {
public:
    void show() {
        cout << "B\n";
    }
};

class C : public A, public B {};

int main() {
    C obj;
    // obj.show(); // ERROR: ambiguous
    obj.A::show(); // Calls A's show
    obj.B::show(); // Calls B's show
}
```

Resolved by Overriding:

```
class C : public A, public B {
public:
    void show() {
        cout << "C\n";
    }
};
```

Now, `obj.show()` calls C's `show()`.

You can still access A and B's show()

```
obj.A::show();
obj.B::show();
```

2. Diamond-Shaped Inheritance

When two base classes inherit from same class, and another class inherits from them.

Diagram

```
    A
   / \
  B   C
   \ /
    D
```

Example Without Virtual

```
class A {
public:
    void show();
};

class B : public A {};
class C : public A {};
class D : public B, public C {};

int main() {
    D d;
    d.show(); // ERROR: ambiguous
}
```

Solution: Virtual Inheritance

```
class B : virtual public A {};
class C : virtual public A {};
```

Now D has only one copy of A, and `d.show()` works fine.

5.7.3 Multi-Level Inheritance

One class is derived from another derived class.

Diagram

```
A
|
B
|
C
```

Example:

```
class A {
public:
    void fA() { cout << "fA()\n"; }
};

class B : public A {
public:
    void fB() { cout << "fB()\n"; }
};

class C : public B {
public:
    void fC() { cout << "fC()\n"; }
};

int main() {
    C obj;
    obj.fA(); // from A
    obj.fB(); // from B
    obj.fC(); // from C
}
```

5.7.4 Hierarchical Inheritance

One base class has multiple derived classes.

Diagram

```
   A
  / \
 B   C
```

Example:

```
class A {
public:
    void fA() { cout << "fA()\n"; }
};

class B : public A {
public:
    void fB() { cout << "fB()\n"; }
};

class C : public A {
public:
    void fC() { cout << "fC()\n"; }
};

int main() {
    B b;
    C c;
    b.fA(); b.fB();
    c.fA(); c.fC();
}
```

5.7.5 Hybrid Inheritance

A **combination** of multiple types of inheritance (e.g., multiple, hierarchical, multilevel).

Diagram

```
    A
   / \
  B   C
```

```
  \ /
   D
```

This can involve diamond-shaped problems and requires careful design, often solved with **virtual inheritance**.

Order of Invocation of Constructors and Destructors

Constructor invocation order:

1. **Virtual base class constructors** – In the order they are inherited.
2. **Non-virtual base class constructors** – In the order they are inherited.
3. **Member object constructors** – In the order they are declared in the class.
4. **Derived class constructor**

Destructor invocation order:

- Exactly the reverse of constructor invocation.

Example: Listing 5.33 – cd_order.cpp

```
#include<iostream.h>

class A {
public:A() {
cout << "Constructor of class A called\n";
 }
 ~A() {
   cout << "Destructor of class A called\n";
 }
};

class B {
public:
 B() {
   cout << "Constructor of class B called\n";
 }
```

```
 ~B() {
   cout << "Destructor of class B called\n";
 }
};

class C : virtual public A {
public:
 C() {
   cout << "Constructor of class C called\n";
 }
 ~C() {
   cout << "Destructor of class C called\n";
 }
};

class D : virtual public A {
public:
 D() {
   cout << "Constructor of class D called\n";
 }
 ~D() {
   cout << "Destructor of class D called\n";
 }
};

class E {
public:
 E() {
   cout << "Constructor of class E called\n";
 }
 ~E() {
   cout << "Destructor of class E called\n";
 }
};

class F : public B, public C, public D {
private:
 E Eobj; // member object
public:
 F() {
   cout << "Constructor of class F called\n";
 }
```

```
 ~F() {
   cout << "Destructor of class F called\n";
 }
};

void main() {
 F Fobj;
}
```

Explanation of the Output:

When object `Fobj` of class F is created, the constructors run in this order:

1. Constructor of **class A** – (Virtual base class)
2. Constructor of **class B** – (Non-virtual base class)
3. Constructor of **class C**
4. Constructor of **class D**
5. Constructor of **class E** – (Member object)
6. Constructor of **class F**

When `Fobj` is destroyed, destructors run in **reverse** order:

1. Destructor of **class F**
2. Destructor of **class E**
3. Destructor of **class D**
4. Destructor of **class C**
5. Destructor of **class B**
6. Destructor of **class A**

Output:

```
Constructor of class A called
Constructor of class B called
Constructor of class C called
Constructor of class D called
Constructor of class E called
Constructor of class F called
Destructor of class F called
Destructor of class E called
```

```
Destructor of class D called
Destructor of class C called
Destructor of class B called
Destructor of class A called
```