UEE1303
Objective-Oriented Programming

C++_Lecture 11:

Inheritance (II) –

Multiple Inheritance

Object-Oriented Programming
Using C++ for Engineering and Technology

Agenda

- Concepts of Multiple Inheritance
  - disadvantages of multiple inheritance
  - order of constructors and destructors
- Ambiguity in Multiple Inheritance
  - call same-name members of base classes
  - members from common base class
- Overload Same-Name Members
  - two side effects
- Virtual Base Class
  - initialization and calling orders

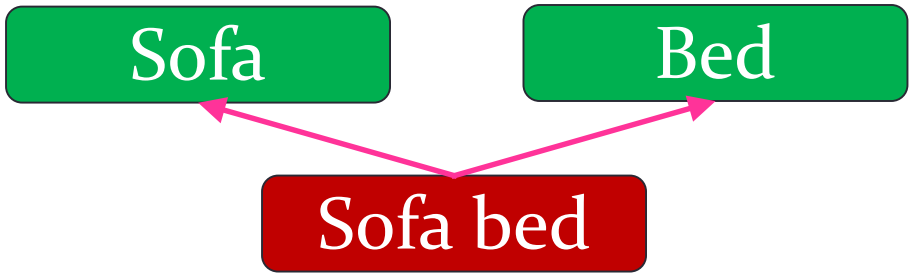
Multiple Inheritance

- Defining a class to have multiple parent classes is very simple
  - list parent classes one by one
- Example

```
class Sofabed: public Sofa, public Bed
{
    //specify properties of its own
};
```

  - support public/protected members of `Sofa` and `Bed`
  - a `Sofabed` object can be upcast to `Sofa` or `Bed`
  - creating a new `Sofabed` object calls the `Sofa`'s and `Bed`'s default constructors
  - call destructors in reverse order

Example of Sofa bed




```
class Sofa {
public: void sit() {
                cout << "sit!"<< endl;    }
};
class Bed {
public: void lie() {
                cout << "lie!" << endl;    }
};
class Sofabed : public Sofa, public Bed
{
    //specify properties of its own
};
```

Example of Sofa bed (cont.)

```
int main()
{
    Sofabed myfur;

    myfur.sit();
    myfur.lie();

    return 0;
}
```

- Using objects of classes with multiple parents is no different from using those without multiple parents
- All that really matters are the properties and behaviors supported by the class

Disadvantages of Multiple Inheritance

- Multiple inheritance is never required to solve a programming problem
  - The `Sofabed` class could be written to inherit from `Sofa` but could contain a `Bed`
- If two parent classes contain same-name members ⇨ must use the resolution operator (`::`) when working with those members
- The definition of a class that inherits from a single parent is almost easier to understand and less prone to error

Constructor of Multiple Inheritance

- Derived class must provide initial arguments for the constructors of each base class
  - same as single inheritance
- Format:

```
CDerived::CDerived(arg_B1, arg_B2, ...,
          arg_Bn, arg_Derived) :
    B1(arg_B1),B2(arg_B2),...,Bn(arg_Bn)
{
    //initilize CDerived data members
}
```

  - call constructors of base classes in its declaration order (left → right)
  - define initialization of new data members

Constructor of Multiple Inheritance (cont.)

- Copy constructors
  - If a derived-class object calls default copy constructor, compiler calls the default copy constructors of base classes automatically
- Format:

```
CDerived::CDerived(Derived & c1) :
    B1(c1),B2(c1),...,Bn(c1)
{
    //copy the rest data members
}
```

  - pass correspondent arguments to the copy constructor of base classes

Example of Multiple Inheritance

- Base class B1, B2 and B3 and their definitions

```
class B1 {
    int x;
protected: int GetX() { return x; }
public:    void SetX(int a=1) { x=a; }
};
class B2 {
    int y;
public: int GetY() { return y; }
        void SetY(int a=1) { y=a; }
};
class B3 {
    int z;
public: int GetZ() { return z; }
        void SetZ(int a=1) { z=a; }
};
```

Example of Multiple Inheritance (cont.)

- Derived class `D4` and the `main` function

```
class D4:public B1,public B2,public B3 {
    int w;
public:
    void SetW(int a) { w=a; }
    void ShowVal() {
        cout << GetX() << " " << GetY() << " "
             << GetZ() << " " << w << endl;
    }
};
```

```
void main() {
    D4 obj;
    obj.SetX(1); obj.SetY(2);
    obj.SetZ(3); obj.SetW(4);
    obj.ShowVal();
}
```

Example of Sofa Bed

```
class Sofa {
protected: int weight;
public:
    void sit() { cout << "sit!" << endl; }
    void SetWeight(int a=0) { weight=a; }
    int GetSofaWeight() { return weight; }
    void ShowWeight() {
        cout << "Sofa weight=" << weight; }
};
class Bed {
protected: int weight;
public:
    void lie() { cout << "lie!" << endl; }
    void SetWeight(int a=0) { weight=a; }
    int GetBedWeight() { return weight; }
    void ShowWeight() {
        cout << "Bed weight=" << weight; }
};
```

Example of Sofa Bed (cont.)

```
class Sofabed : public Sofa, public Bed
{
public:
    void fold() { cout << "fold!" << endl; }
};
int main()
{
    Sofabed myfur;

    myfur.sit();
    myfur.lie();
    myfur.fold();

    myfur.SetWeight(100); //call which one?

    return 0;
}
```

Ambiguity in Multiple Inheritance

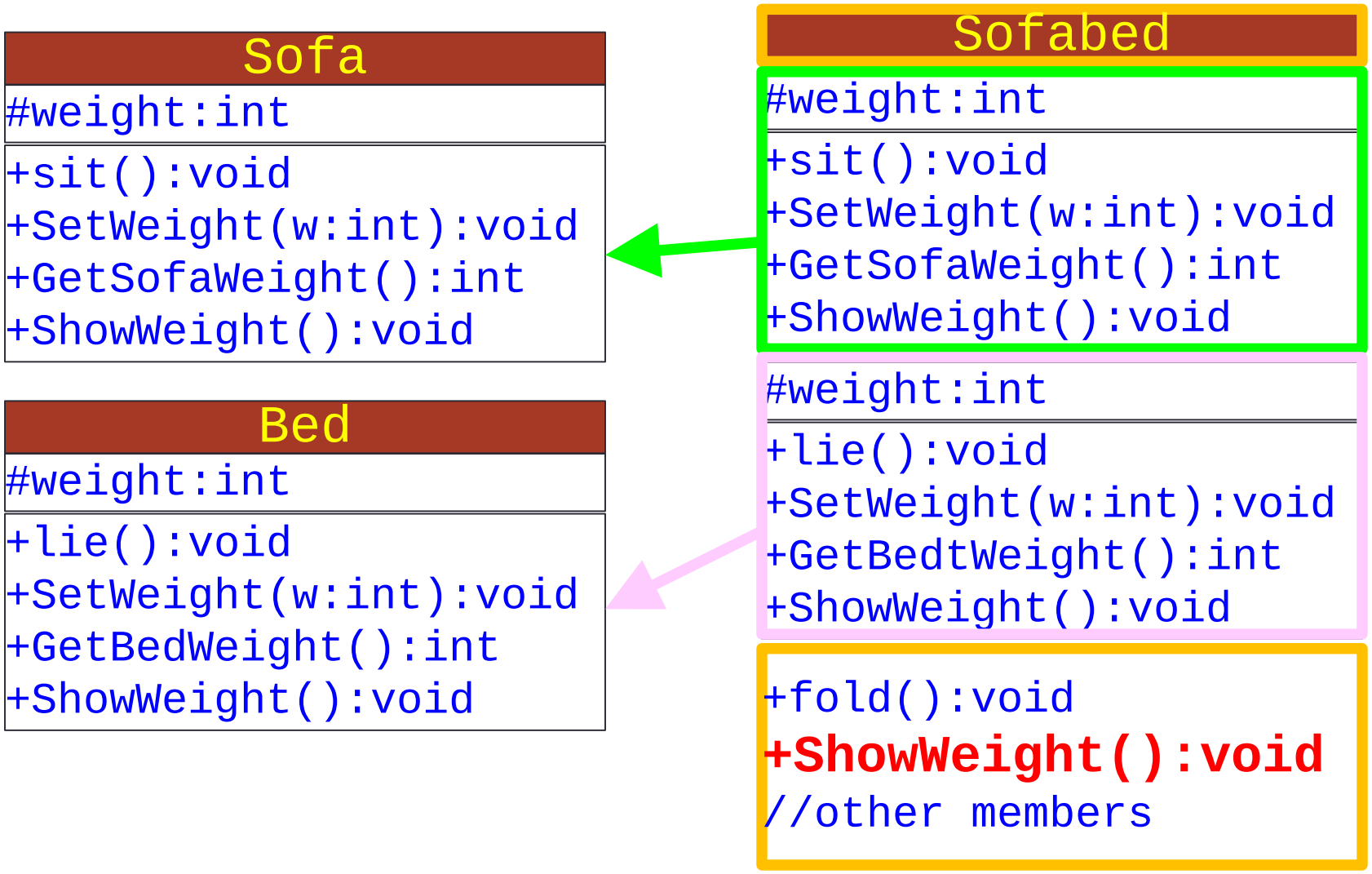
- Multiple inheritance resolves many complex scenarios but becomes ambiguous when
  - (case 1) calling same-name members from different parent classes
  - (case 2) calling the member of the common base class inherited by parent classes of the current class

Solutions for Case 1

- (case 1) calling same-name members from different parent classes
  - Use `Sofabed` as an example
- (Solution 1-1) use scope resolution (`::`)

```
myfur.Sofa::SetWeight(200);
myfur.Bed::SetWeight(300);
```

UML Class Diagram of Sofabed



Solutions for Case 1 (cont.)

- (Solution 1-2) overload the same-name member

```
class Sofabed : public Sofa, public Bed
{
public:
    void ShowWeight() {
        Sofa::ShowWeight();
        cout << "&";
        Bed::ShowWeight();
        cout << endl;
    }
};
```

```
weight = 200 & weight = 300
weight = 200
weight = 300
```

```
myfur.Sofa::SetWeight(200);
myfur.Bed::SetWeight(300);
myfur.ShowWeight();
myfur.Sofa::ShowWeight(); cout << endl;
myfur.Bed::ShowWeight(); cout << endl;
```

Overload Same-Name Members

- Side effect 1: overloaded members in base classes cannot be accessed directly

```
class CB {
public:
    void f() { cout<< "CB's f()" <<endl; }
    void f(int x) {
        cout<< "CB's f(x)" <<endl; }      };
class CD : public CB {
public:
    void f() { cout<< "CD's f()" <<endl;} };
```

```
// in main()
CD obj;
obj.f();
obj.f(5); //what happens?
//what if call 'obj.CB::f(5);'
```

```
candidate expects 0 arguments,
1 provided
```

Overload Same-Name Members (cont.)

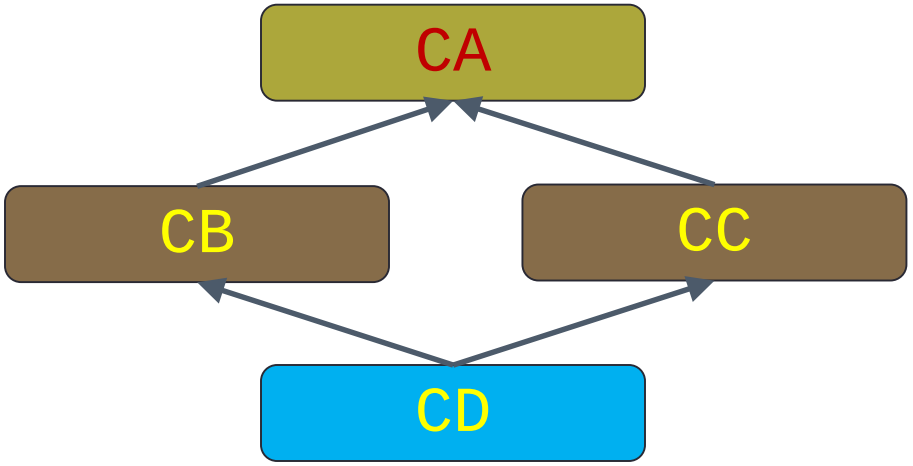
- Side effect 2: the pointer points to base-class members, not the one in the derived class

```
SofaBed obj;
obj.Sofa::SetWeight(25);
Bed *ptr;
ptr = new Bed;

ptr->SetWeight(70);
ptr->ShowWeight();

ptr = &obj;
ptr->ShowWeight(); //call which??
```

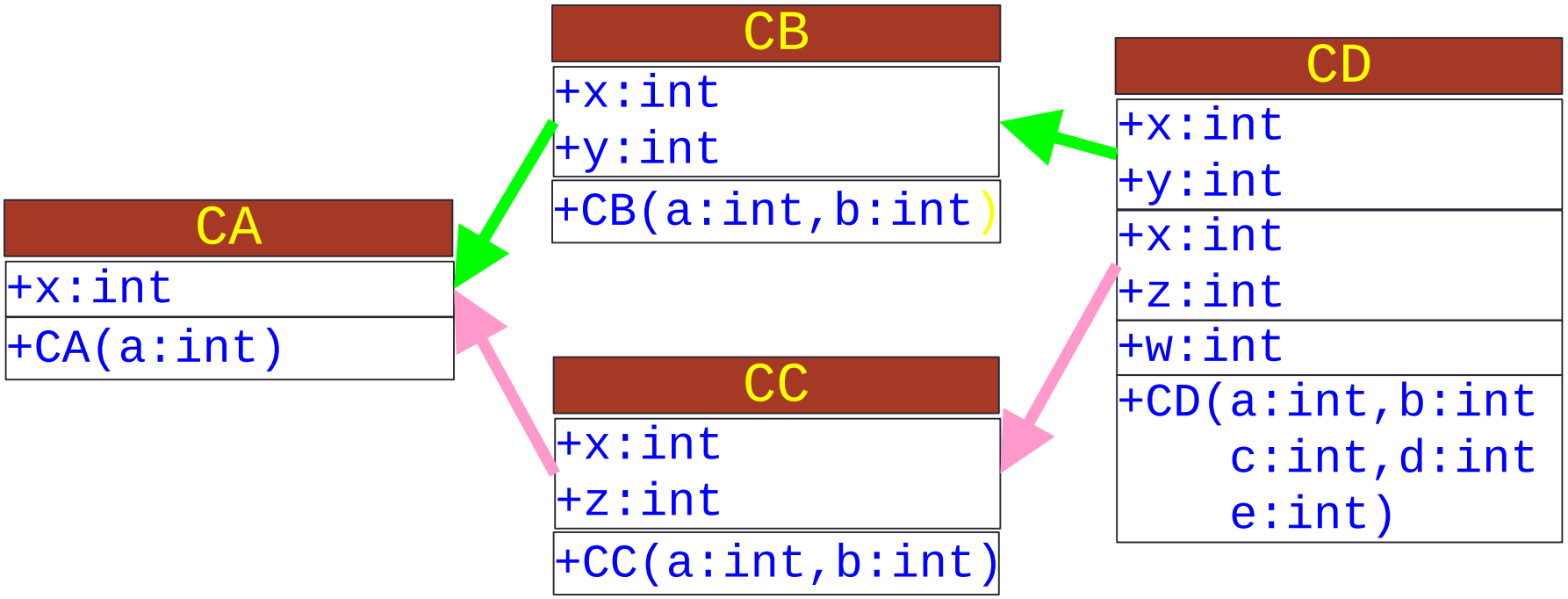
  - `ptr` points to `obj` but call `ShowWeight()` in class `Bed`
  - so not recommended ⇨ *virtual functions*

Problem from Common Base Class

- A derived class inherits multiple base classes and all derived from one common base class
  - ambiguity from calling members from the common base class
  - Introduce virtual base class



Common-Base-Class Problem



- Class CD contains two copies of class CA
  - redundant memory space
  - cause ambiguity

Example of Common-Base-Class Problem

```
class CA  //common base class of CB and CC
{public:
    int x;
    CA(int a=0) { x=a; }
};
class CB : public CA //one base class of CD
{public:
    int y;
    CB(int a=0, int b=0):CA(a) { y=b; }
};
class CC : public CA //one base class of CD
{public:
    int z;
    CC(int a=0, int b=0):CA(a) { z=b; }
};
```

Example of Common-Base-Class Problem (cont.)

```
class CD: public CB, public CC
{public:
    int w;
    CD(int a=0, int b=0, int c=0, int d=0,
       int e=0): CB(a,b), CC(c,d) { w=e; }
    void ShowVal() {
        cout << "x=" << CB::x << " y=" << y;
        cout << "x=" << CC::x << " z=" << z;
        cout << "w=" << w << endl;
    }
};
```

```
// in main()
CD obj(5,4,3,2,1);
obj.ShowVal(); //what happens?
```

```
x=5 y=4 x=3 z=2 w=1
```

Virtual Base Class

- Virtual base class keeps only one copies of members when the derived class inherited one common base class
  - used when defining a derived class
- Format:

```
class ⟨CDerived⟩:
      virtual ⟨Acc Spe. 1⟩⟨CBase 1⟩,
      ...
{
    //specify properties of its own
};
```

  - members in virtual base class maintain only one copy in the derived class

Example of Virtual Base Class

```
class CA  //common base class of CB and CC
{public:
    int x;
    CA(int a=0) { x=a; }
};
class CB : virtual public CA
{public:
    int y;
    CB(int a=0, int b=0):CA(a) { y=b; }
};
class CC : virtual public CA
{public:
    int z;
    CC(int a=0, int b=0):CA(a) { z=b; }
};
```

Example of Virtual Base Class (cont.)

```
class CD: public CB, public CC
{public:
    int w;
    CD(int a=0, int b=0, int c=0, int d=0,
       int e=0): CA (a), CB(a,b), CC(c,d)
    { w=e; }
    void ShowVal() {
        cout << "x=" << CB::x << " y=" << y;
        cout << "x=" << CC::x << " z=" << z;
        cout << "w=" << w << " x=" << x;
    }
};
```

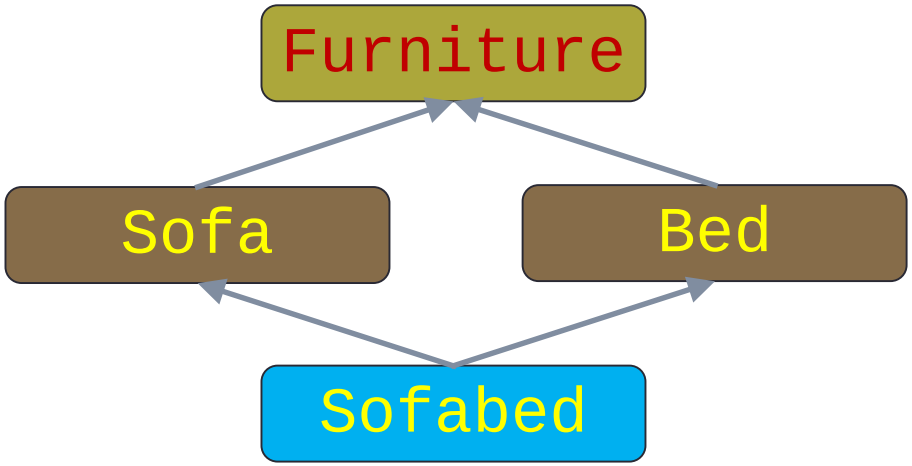
```
// in main()
CD obj(5,4,3,2,1);
obj.ShowVal(); //what happens?
```

```
x=5 y=4 x=5 z=2 w=1 x=5
```

Example of Virtual Base Class (cont.)

- After declaring `CA` as a virtual base class
  - only one copy of `x` in `CD` ⇨ no ambiguity when print the value of `x`
  - *constructor for `CA` is executed only once*
- To guarantee the correctness of virtual base class, your C++ program should
  - call constructor for virtual base classes first
  - omit the initialization from constructors for regular base classes on members of virtual base class ⇨ guarantee to execute the constructor of virtual base class once

Redesign Sofa Bed




```
class Furniture { //common base class
protected: int weight;
public:
    void SetWeight(int a=0) { weight=a; }
    int GetWeight() { return weight; }
    void ShowWeight() {
        cout << "weight=" << weight; }
};
```

Redesign Sofa Bed (cont.)

```
class Sofa : virtual public Furniture {
public:
    void sit() { cout << "sit!" << endl; }
//     void ShowWeight() {
//         cout << "Sofa weight=" << weight; }
};

class Bed : virtual public Furniture {
public:
    void lie() { cout << "lie!" << endl; }
//     void ShowWeight() {
//         cout << "Bed weight=" << weight; }
};
```

Redesign Sofa Bed (cont.)

```
class Sofabed : public Sofa, public Bed {
public:
    void fold() { cout << "fold!" << endl; }
};
```

```
int main()
{
    Sofabed obj;
    obj.sit();
    obj.lie();
    obj.fold();

    obj.SetWeight(100); //call which one?
    obj.ShowWeight(); //what happens?

    return 0;
}
```

Initialize Virtual Base Class

- If a constructor with arguments is defined in the virtual base class, need to use such constructor to initialize all derived classes
- Example:

```
class CA {
    public: CA(int a=0) { ... }                    };
class CB : virtual public CA {
    public: CB(int a=0):CA(a) { ... }              };
class CC : virtual public CA {
    public: CC(int a=0):CA(a) { ... }              };
class CD : public CB, public CC {
    public: CD(int a=0):CA(a), CB(a), CC(a)
                  {... }                           };
```

Initialize Virtual Base Class (cont.)

```
class CD : public CB, public CC {
    public: CD(int a=0):CA(a), CB(a), CC(a)
                    {... }                    };
```

- Before, derived-class constructors only initialize members in direct base classes, i.e. `CB(a), CC(a)` ⇨ those direct base classes initialize the indirect base classes (`CA(a)`)
- Now, derived-class constructors need to call `CA(a)` due to only one copy of data member ⇨ a strict rule enforced by C++
- Question: Is `CA(a)` called three times?
  No! C++ ignores `CA(a)` in `CB` and `CC` automatically

Order of Constructors/Destructors

- Similar to single inheritance, the constructors to be called starts from
  - first, virtual base classes in declaration order

    ⇨ not *initialization* order
  - then, other base classes in declaration order

- Destructors are called in the reverse order of the constructors

Example of Calling Order

```
class C1 {
public:
    C1() { cout << "construct C1\n"; }
    ~C1() { cout << "destruct C1\n"; }
};

class C2 {
public:
    C2() { cout << "construct C2\n"; }
    ~C2() { cout << "destruct C2\n"; }
};

class C3 {
public:
    C3() { cout << "construct C3\n"; }
    ~C3() { cout << "destruct C3\n"; }
};
```

Example of Calling Order (cont.)

```
class C4 {
public:
    C4() { cout << "construct C4\n"; }
    ~C4() { cout << "destruct C4\n"; }
};
class CD: public C3, virtual public C4,
          virtual public C2 { //decl. order
    C1 obj; // use a private C1 object
public:
    CD():obj(),C2(),C3(),C4() //init. order
        { cout << "construct CD\n";    }
    ~CD() { cout << "destruct CD\n"; }
};
```

```
int main() { //what happens?
    CD dd; cout << "here!\n"; return 0;
}
```

Example of Calling Order (cont.)

- screen output

```
construct C4  //1st virtual base class
construct C2  //2nd virtual base class
construct C3  //1st other base class
construct C1  //private member of CD
construct CD
here
destruct CD
destruct C1   //in reverse order
destruct C3
destruct C2
destruct C4
```

(Practice) Example of Copier

```
class PoweredDevice {
public:
    PoweredDevice(int nPower)   {
        cout << "PoweredDevice: "
             << nPower << endl;    }
};
class Scanner: public PoweredDevice {
public:
    Scanner(int nScanner, int nPower)
        : PoweredDevice(nPower)  {
        cout << "Scanner: "
             << nScanner << endl; }
};
```

Example of Copier (cont.)

```
class Printer: public PoweredDevice {
public:
    Printer(int nPrinter, int nPower)
        : PoweredDevice(nPower)   {
        cout << "Printer: "
             << nPrinter << endl;  }
};
class Copier: public Scanner, public Printer{
public:
    Copier(int nScanner, int nPrinter,
           int nPower)
        : Scanner(nScanner, nPower),
          Printer(nPrinter, nPower) {}
};
```

Example of Copier (cont.)

```
int main()
{
    Copier cCopier(1, 2, 3);
    return 0;
}
```

```
PoweredDevice: 3
Scanner: 1
PoweredDevice: 3
Printer: 2
```

Summary

- Concepts of multiple inheritance
  - example of sofabed
- Disadvantages of multiple inheritance
  - never required for programming
  - resolution operator (`::`) to resolve same-name members
- Calling order of constructors and destructors
  - call constructors from left to right
  - call destructors in the reverse order of constructors

Summary (cont.)

- Ambiguity in multiple inheritance
  - call same-name member of base classes

  ⇨ scope resolution and overloading same-name members
  - members from common base class
- Overload same-name members
  - two side effects
- Virtual base class
  - initialize member once for virtual base class
  - calling orders: first virtual then regular

References

- Paul Deitel and Harvey Deitel, “C++ How to Program (late objects version)” Seventh Edition
  - Chapter 24.7: Multiple Inheritance
  - Chapter 24.8: Multiple Inheritance and virtual Base Classes

- W. Savitch, “Absolute C++,” Fourth Edition
  - Chapter 14