A <Basic> C++ Course

8 – Object-oriented programming 2

Julien Deantoni

adapted from Jean-Paul Rigault courses

Outline

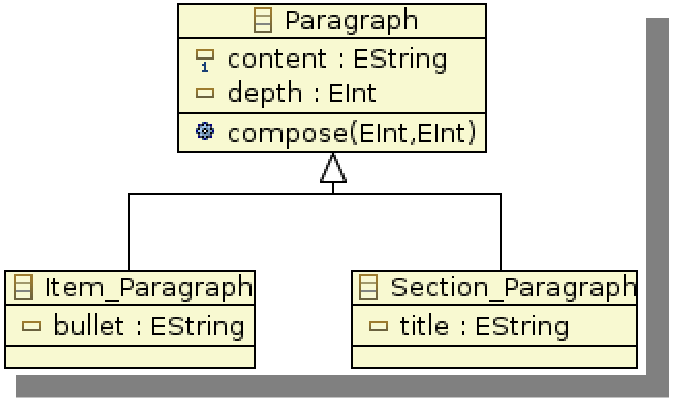
- Dynamic Typing
- Truncature
- Cast

Variants of class `Paragraph` Definition of derived classes (1)

- We wish to have several sorts of paragraphs
  - titles, sections, enumerations, items...

- We want to share as much as possible the common properties
  - contents as a string
  - possibility to compose (crude lay out)

- But specific properties should be possible
  - numbering, bullets...
  - page layout

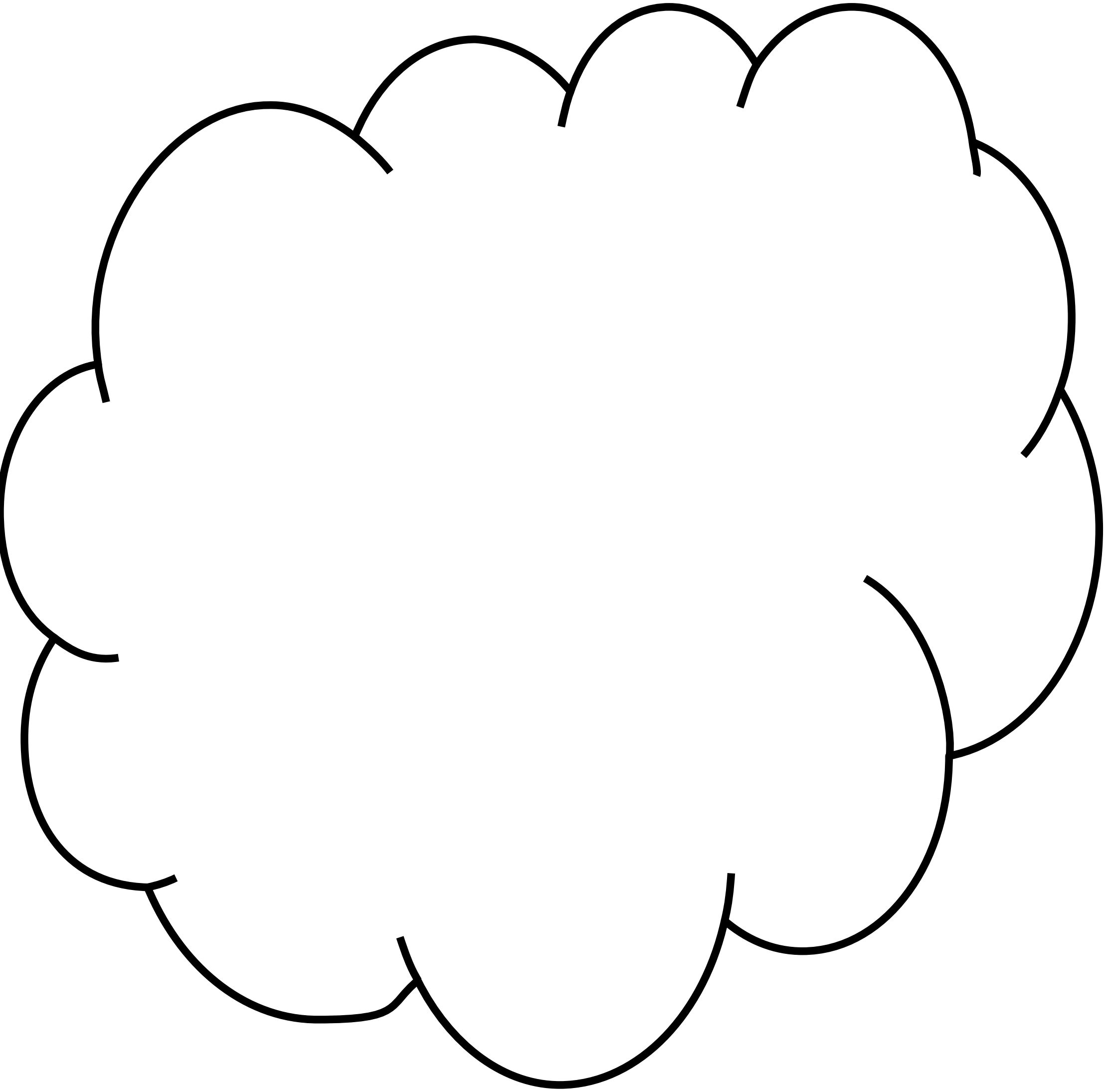


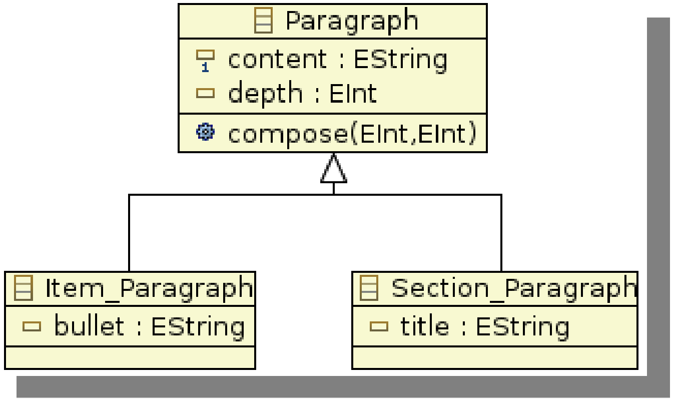
Variants of class `Paragraph` Definition of derived classes (5)

- A derived class may add new properties
  - data members
  - member-functions
  - friend functions
- A derived class may redefine (override) some inherited member-functions
- Derivation depth is unlimited
- Single and multiple inheritance
  - Single: only one base class
  - Multiple: several *distinct* base classes

What happens in memory
(at least conceptually)

If `compose(int, int)` is NON virtual




Paragraph
content : EString
depth : EInt
compose(EInt,EInt)

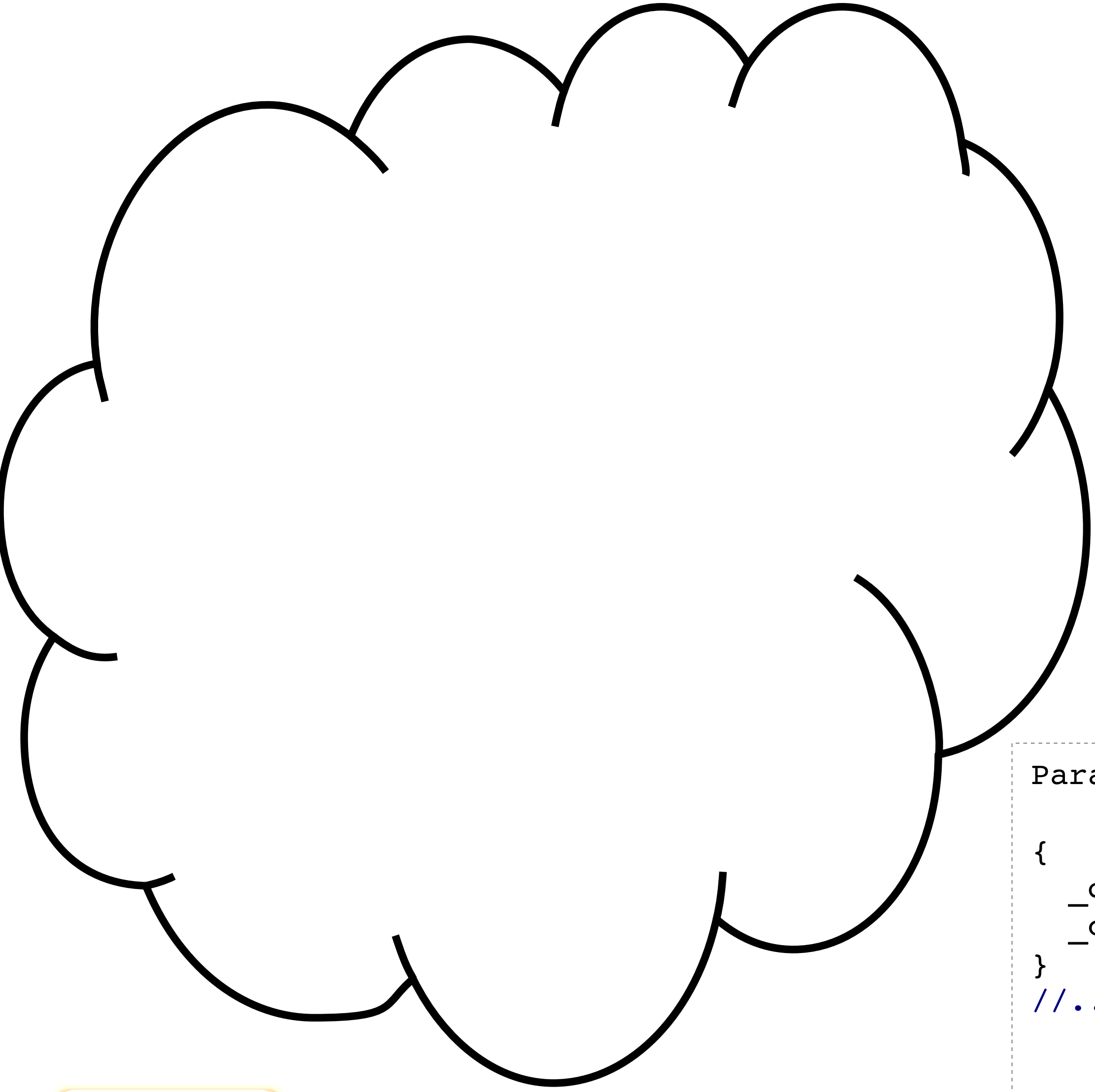
Item_Paragraph
bullet : EString

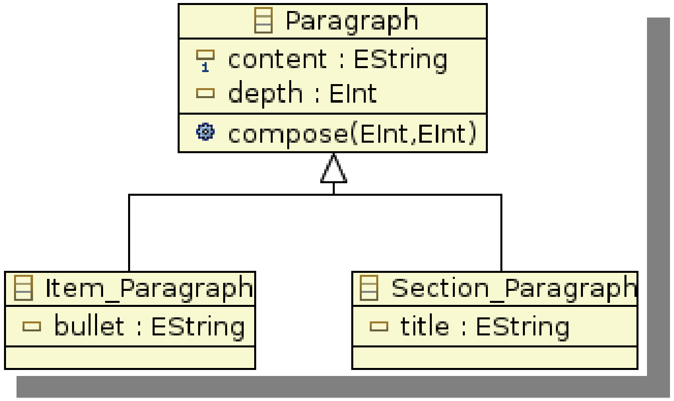
Section_Paragraph
title : EString

```
//...
Paragraph p(“test”,2);
```

What happens in memory (at least conceptually)

If `compose(int, int)` is NON virtual





Paragraph
content : EString
depth : EInt
compose(EInt,EInt)

Item_Paragraph
bullet : EString

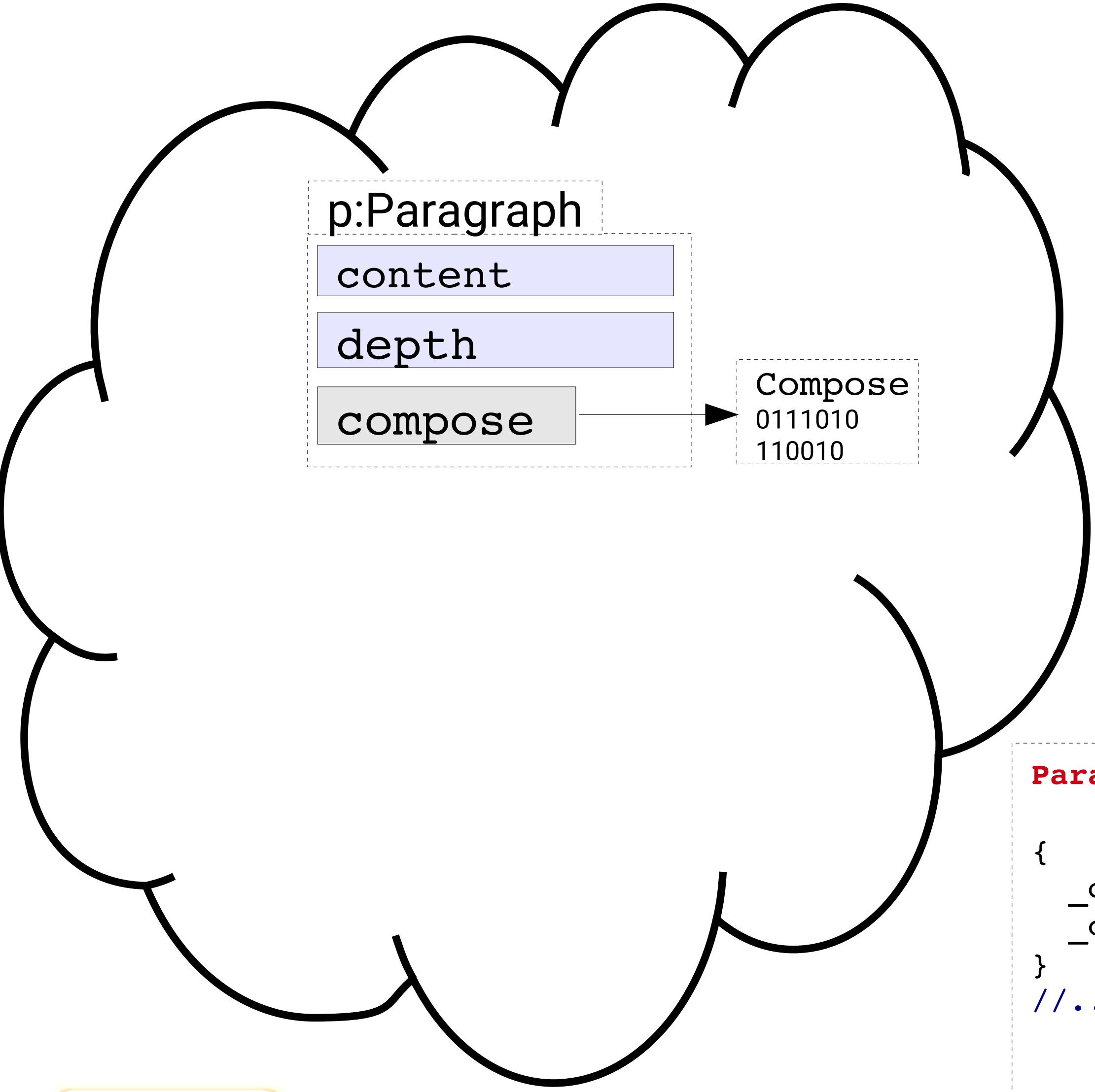
Section_Paragraph
title : EString

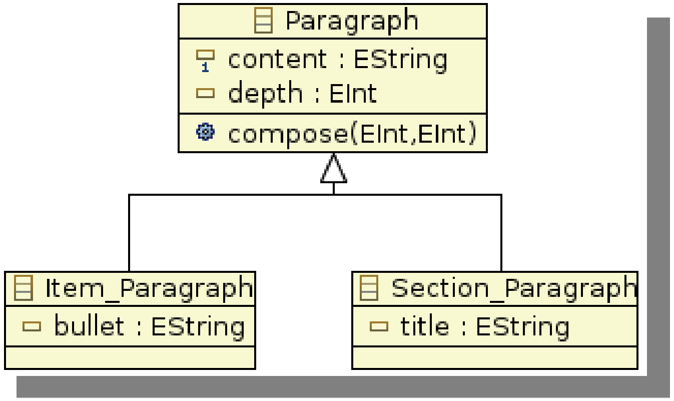
```
//...
Paragraph p("test",2);
```

```
Paragraph::Paragraph(std::string content
                        int depth)
{
  _depth = depth;
  _content = content;
}
//...
```

What happens in memory (at least conceptually)

If `compose(int, int)` is NON virtual



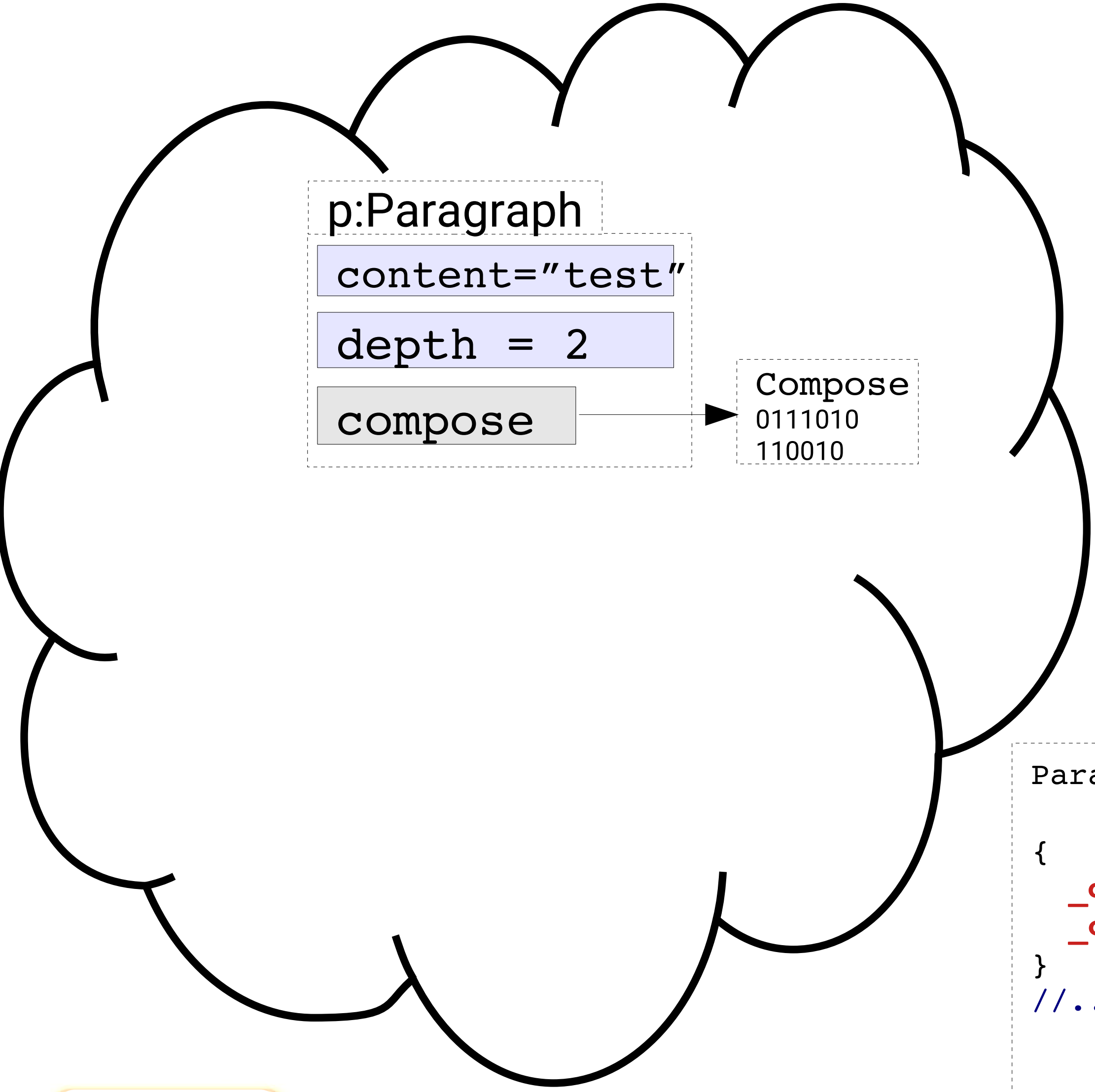


```
//...
Paragraph p("test",2);
```

```
Paragraph::Paragraph(std::string content
                     int depth)
{
  _depth = depth;
  _content = content;
}
//...
```

What happens in memory (at least conceptually)

If `compose(int, int)` is NON virtual



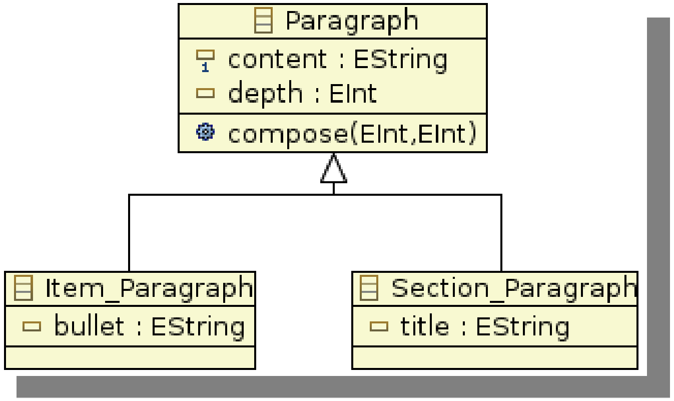
p:Paragraph

content=”test”

depth = 2

compose

Compose
0111010
110010



Paragraph
content : EString
depth : EInt
compose(EInt,EInt)

Item_Paragraph
bullet : EString

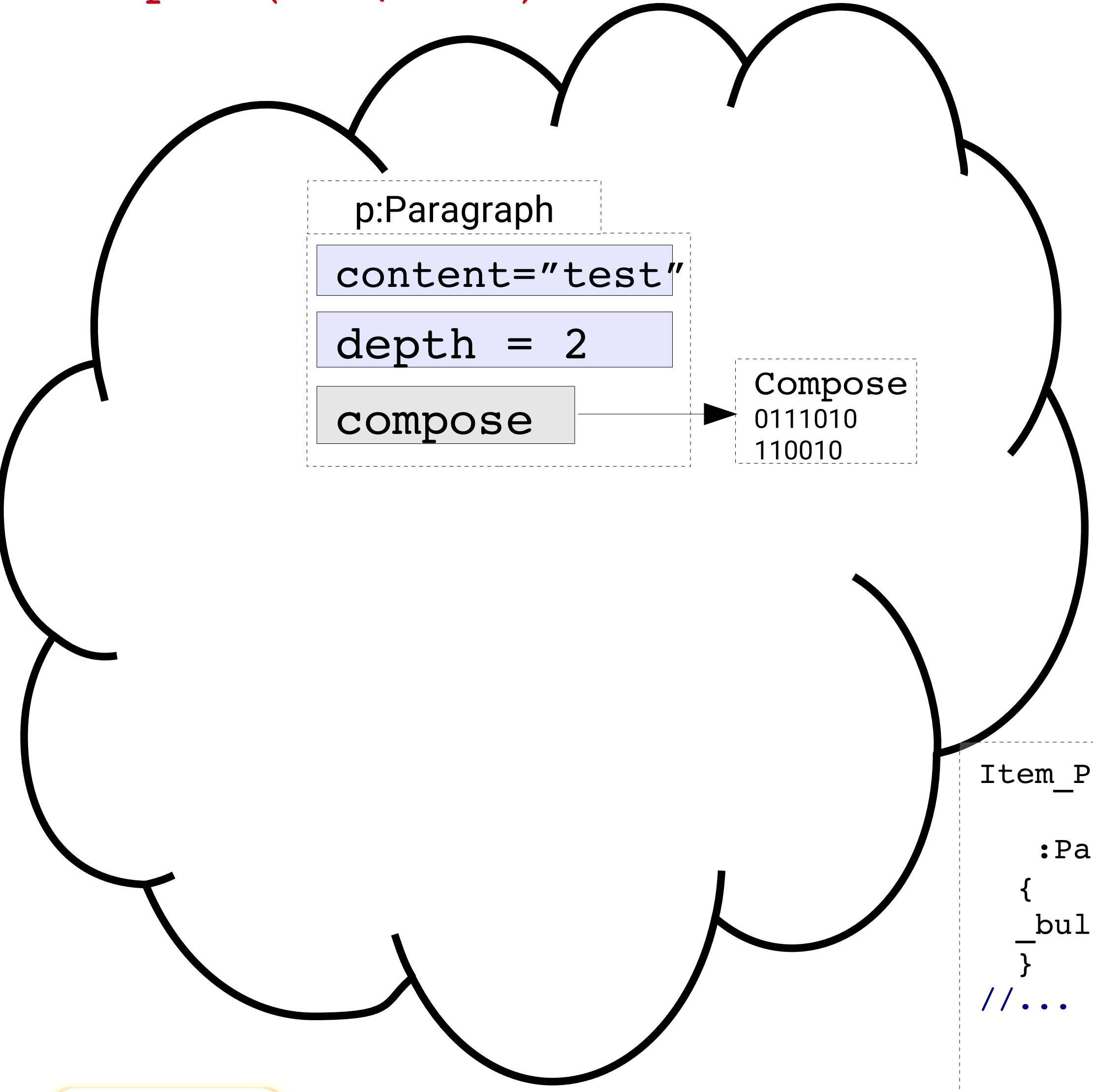
Section_Paragraph
title : EString

```
//...
Paragraph p(“test”,2);
```

```
Paragraph::Paragraph(std::string content
                     int depth)
{
  _depth = depth;
  _content = content;
}
//...
```

What happens in memory (at least conceptually)

If `compose(int, int)` is NON virtual and is not redefined in Item_Paragraph



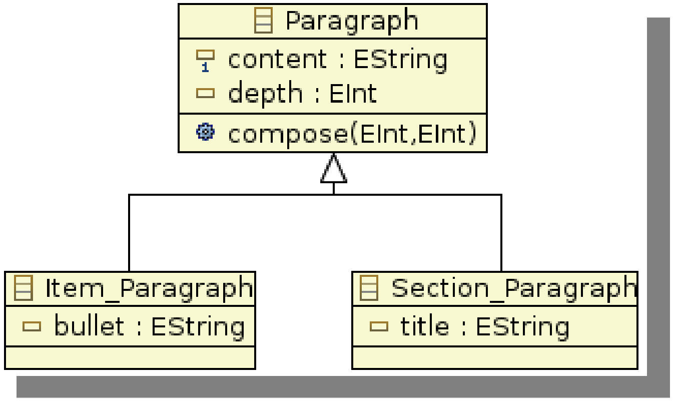
p:Paragraph

content="test"

depth = 2

compose

Compose
0111010
110010



Paragraph
content : EString
depth : EInt
compose(EInt,EInt)

Item_Paragraph
bullet : EString

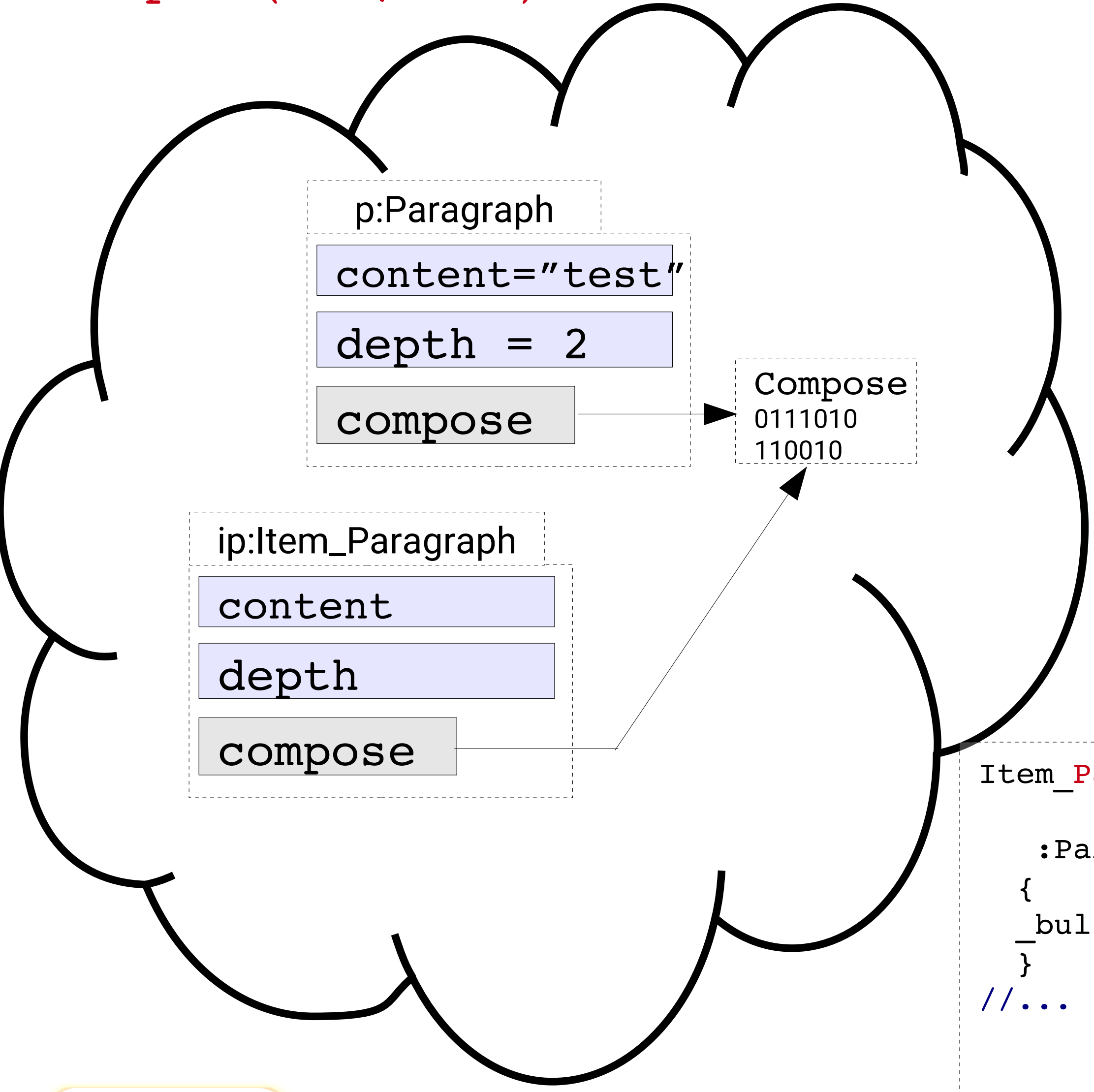
Section_Paragraph
title : EString

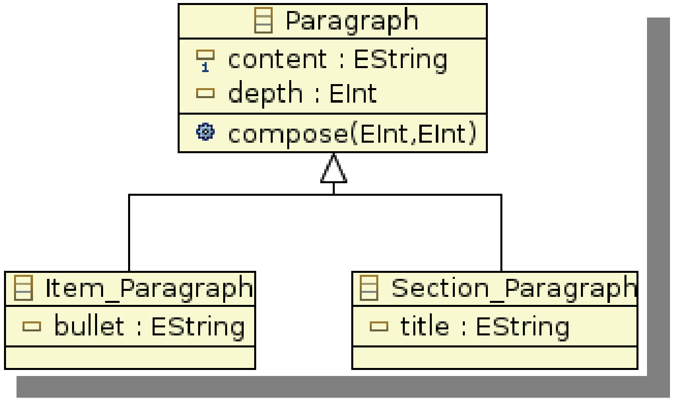
```
//...
Paragraph p("test",2);
Item_Paragraph ip("",1,'#');
```

```
Item_Paragraph::Item_Paragraph(string content,
                               int depth, char b)
    :Paragraph(content, depth)
   {
   _bullet = b;
   }
//...
```

What happens in memory (at least conceptually)

If `compose(int, int)` is NON virtual and is not redefined in Item_Paragraph



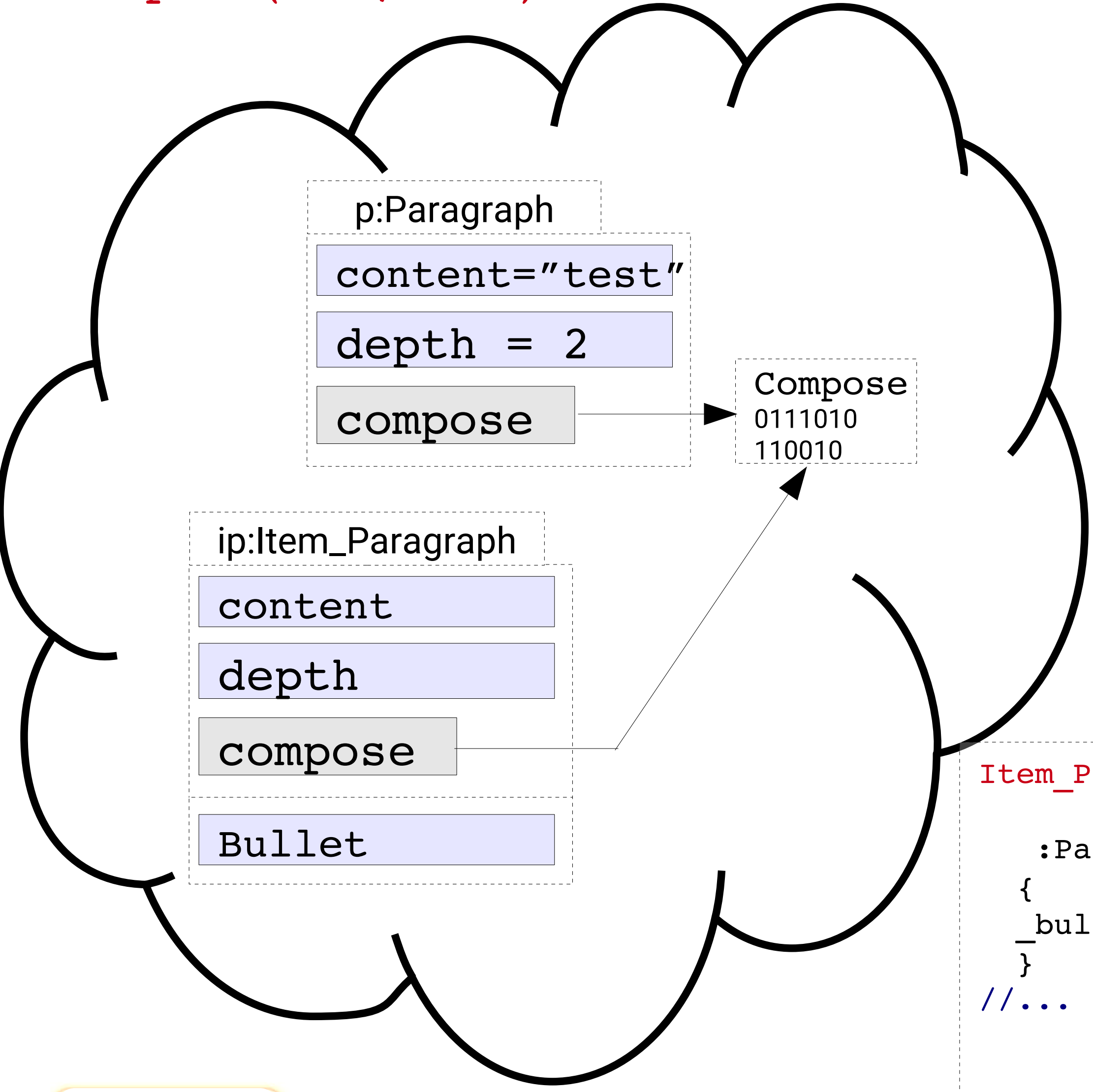


```
//...
Paragraph p("test",2);
Item_Paragraph ip("",1,'#');
```

```
Item_Paragraph::Item_Paragraph(string content,
                               int depth, char b)
   :Paragraph(content, depth)
  {
  _bullet = b;
  }
//...
```

What happens in memory (at least conceptually)

If `compose(int, int)` is NON virtual and is not redefined in Item_Paragraph



p:Paragraph

content="test"

depth = 2

compose

Compose
0111010
110010

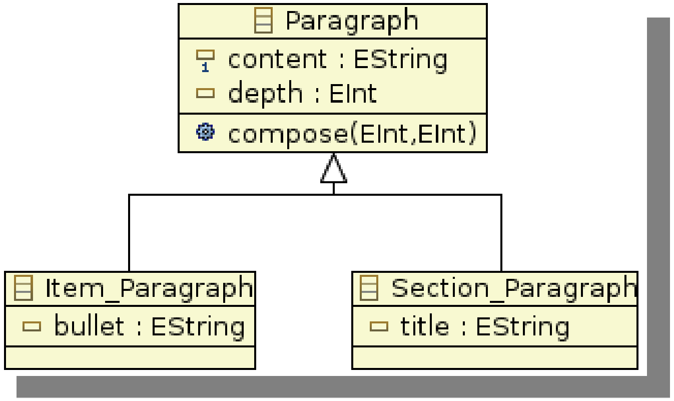
ip:Item_Paragraph

content

depth

compose

Bullet



Paragraph
content : EString
depth : EInt
compose(EInt,EInt)

Item_Paragraph
bullet : EString

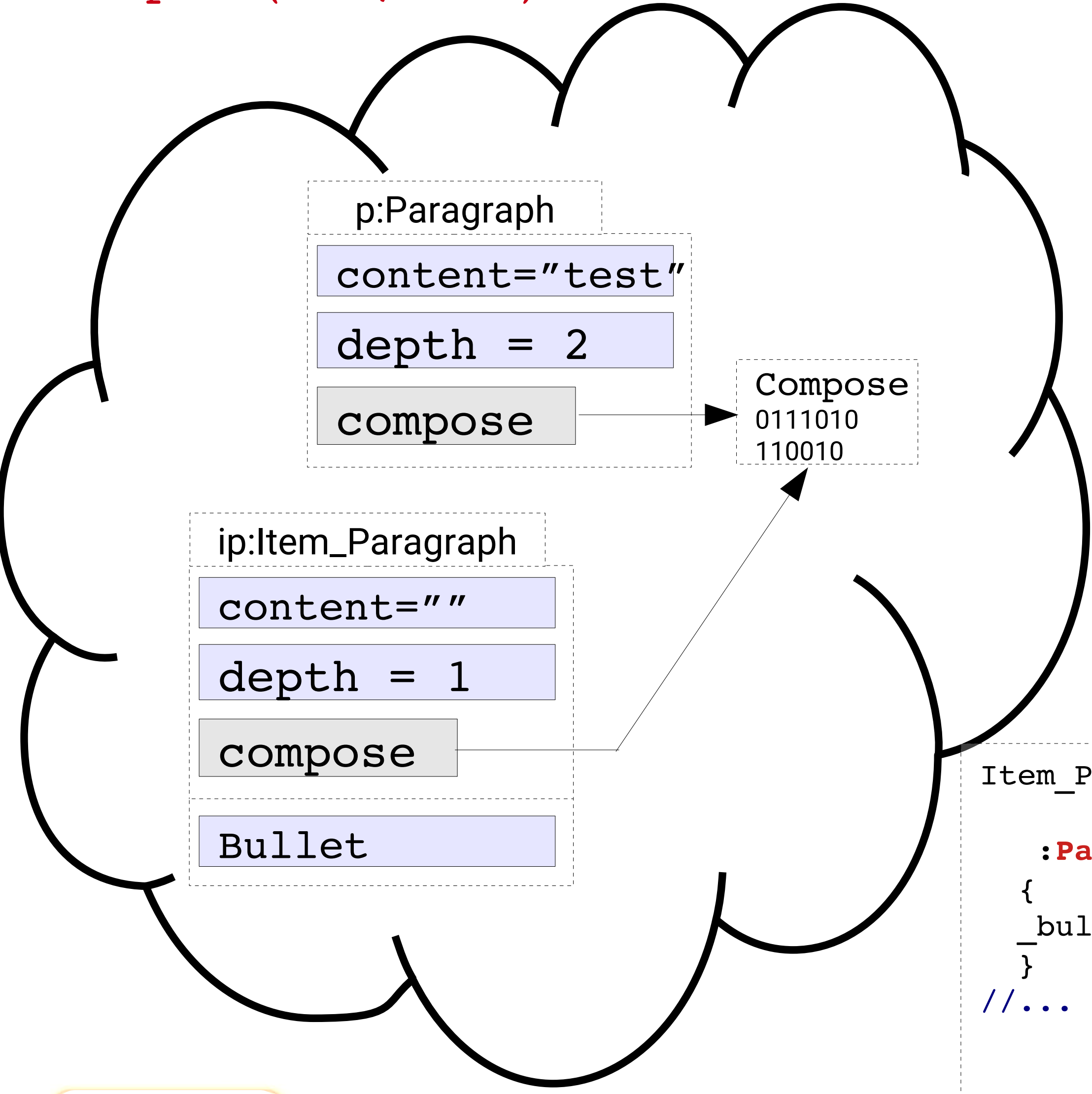
Section_Paragraph
title : EString

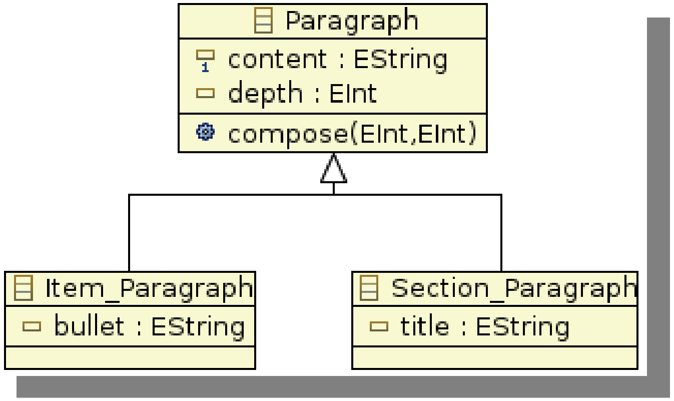
//...
Paragraph p("test",2);
Item_Paragraph ip("",1,'#');

```
Item_Paragraph::Item_Paragraph(string content,
                               int depth, char b)
   :Paragraph(content, depth)
  {
  _bullet = b;
  }
//...
```

What happens in memory (at least conceptually)

If `compose(int, int)` is NON virtual and is not redefined in Item_Paragraph



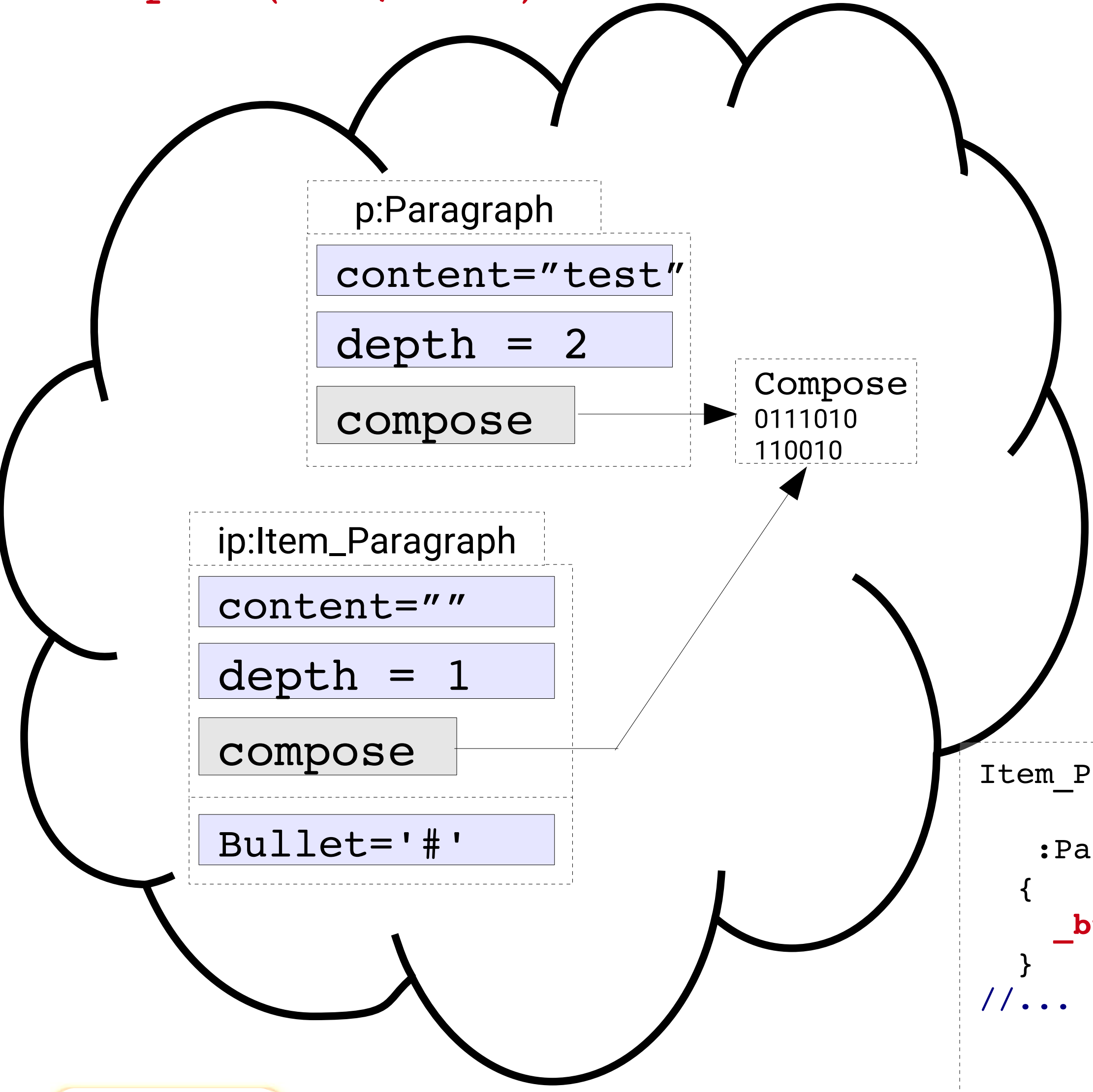


```
//...
Paragraph p("test",2);
Item_Paragraph ip("",1,'#');
```

```
Item_Paragraph::Item_Paragraph(string content,
                    int depth, char b)
   :Paragraph(content, depth)
  {
  _bullet = b;
  }
//...
```

What happens in memory (at least conceptually)

If `compose(int, int)` is NON virtual and is not redefined in Item_Paragraph



p:Paragraph

content="test"

depth = 2

compose

Compose
0111010
110010

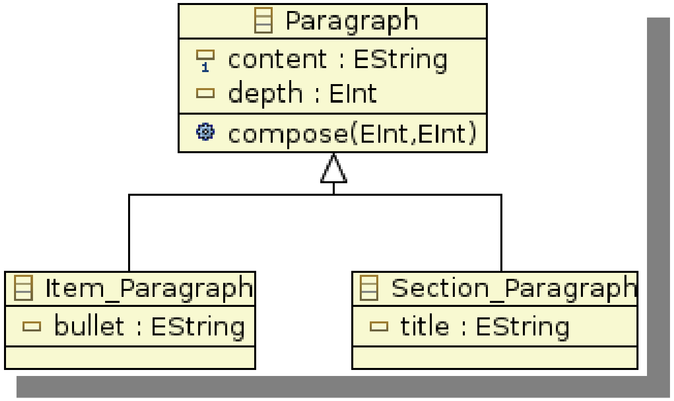
ip:Item_Paragraph

content=""

depth = 1

compose

Bullet='#'

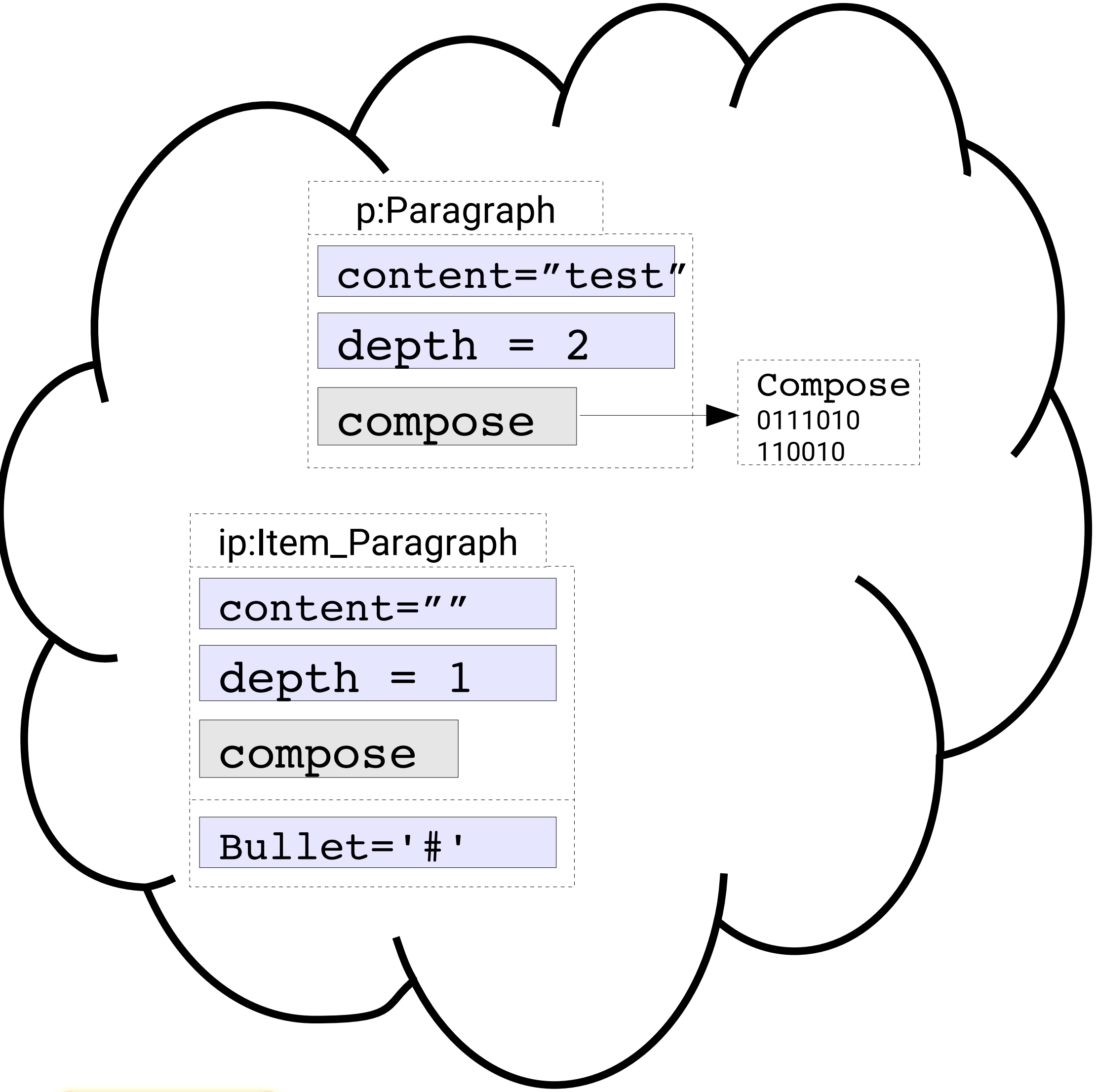


```
//...
Paragraph p("test",2);
Item_Paragraph ip("",1,'#');
```

```
Item_Paragraph::Item_Paragraph(string content,
                               int depth, char b)
   :Paragraph(content, depth)
  {
     _bullet = b;
  }
//...
```

What happens in memory (at least conceptually)

If `compose(int, int)` is NON virtual and is **REDEFINED** in Item_Paragraph



p:Paragraph

content="test"

depth = 2

compose

Compose
0111010
110010

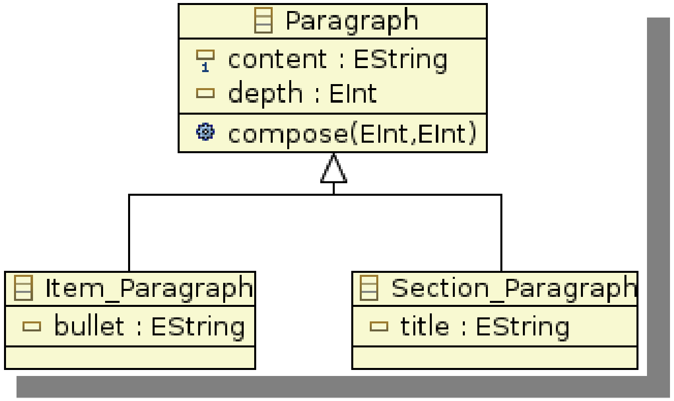
ip:Item_Paragraph

content=""

depth = 1

compose

Bullet='#'



Paragraph
content : EString
depth : EInt
compose(EInt,EInt)

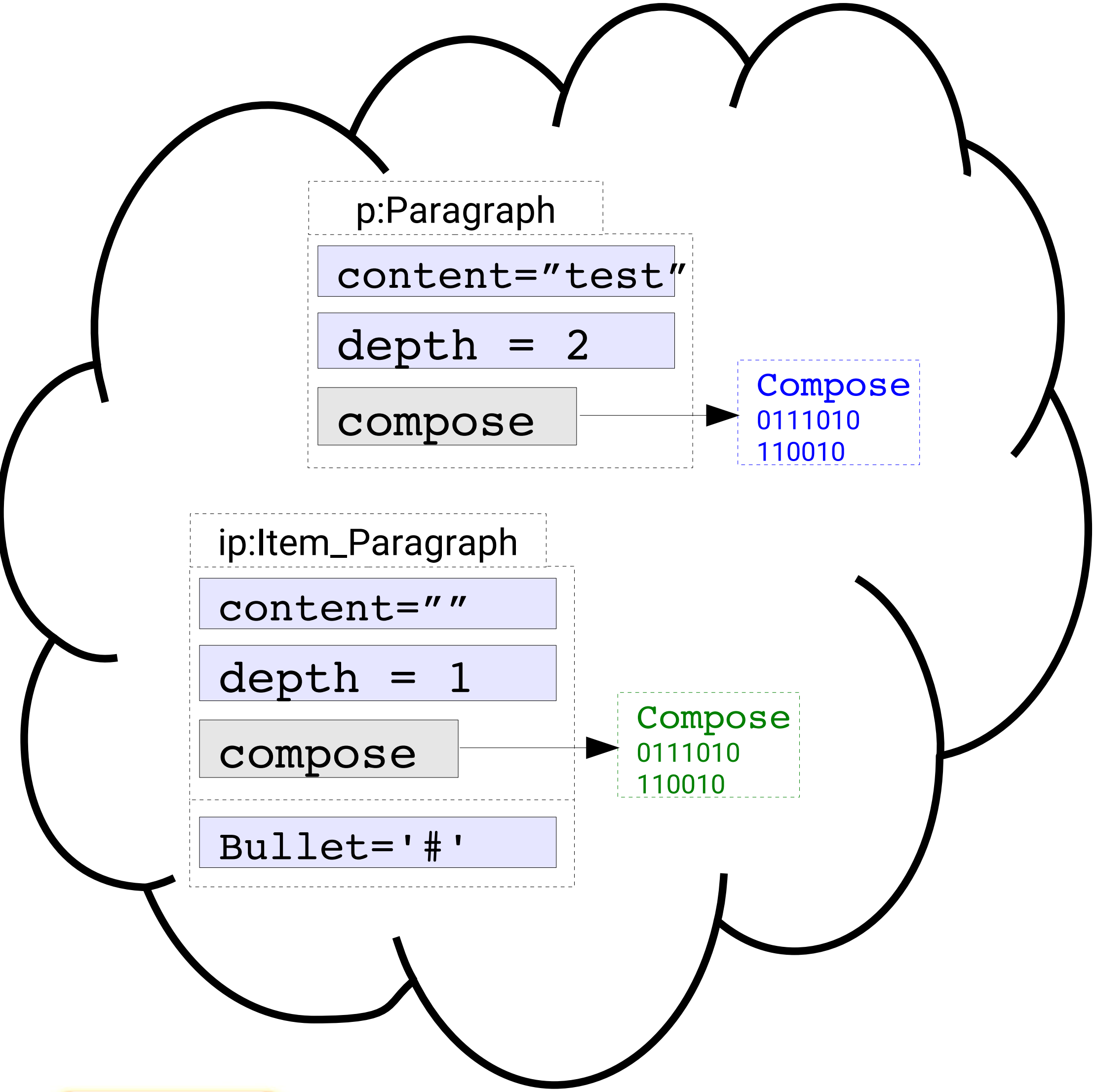
Item_Paragraph
bullet : EString

Section_Paragraph
title : EString

```
//...
Paragraph p("test",2);
Item_Paragraph ip("",1,'#');
```

What happens in memory (at least conceptually)

If `compose(int, int)` is NON virtual and is **REDEFINED** in Item_Paragraph



p:Paragraph

content="test"

depth = 2

compose

Compose
0111010
110010

ip:Item_Paragraph

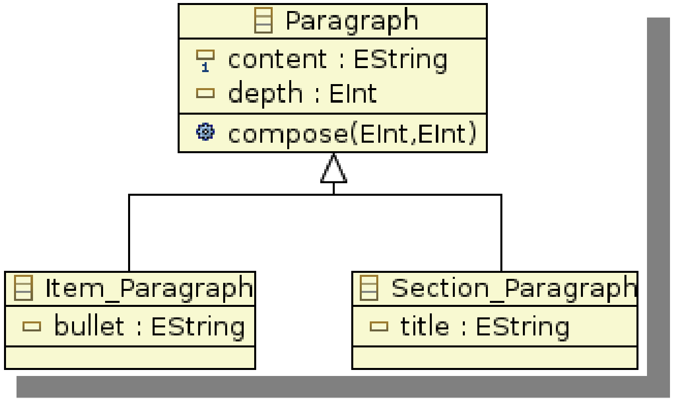
content=""

depth = 1

compose

Compose
0111010
110010

Bullet='#'



Paragraph
content : EString
depth : EInt
compose(EInt,EInt)

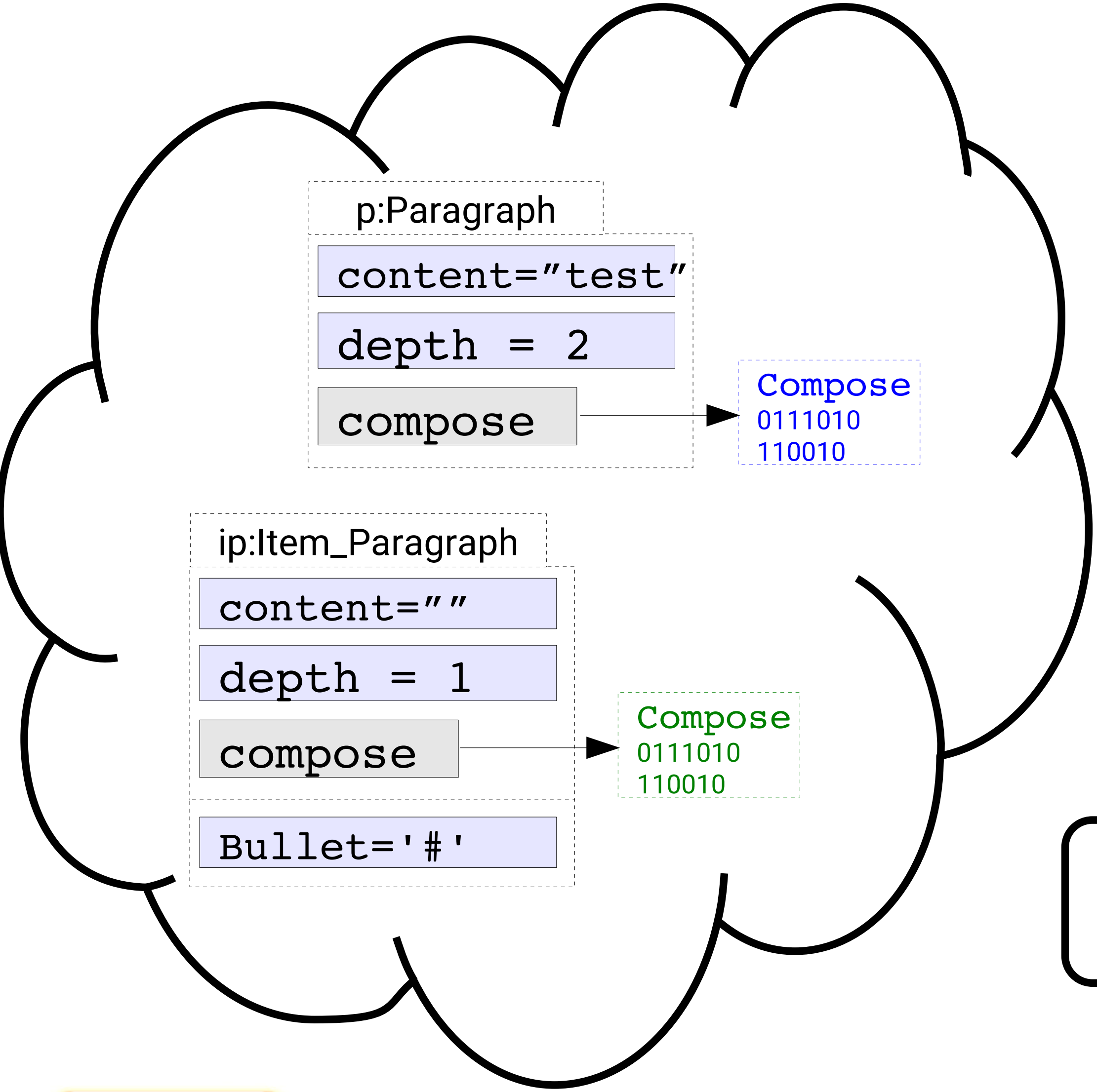
Item_Paragraph
bullet : EString

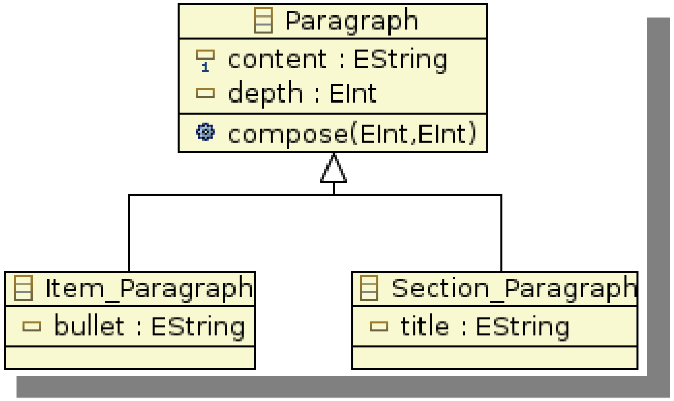
Section_Paragraph
title : EString

```
//...
Paragraph p("test",2);
Item_Paragraph ip("",1,'#');

p.compose();
ip.compose();
```

What happens in memory (at least conceptually)

If `compose(int, int)` is NON virtual and is **REDEFINED** in Item_Paragraph



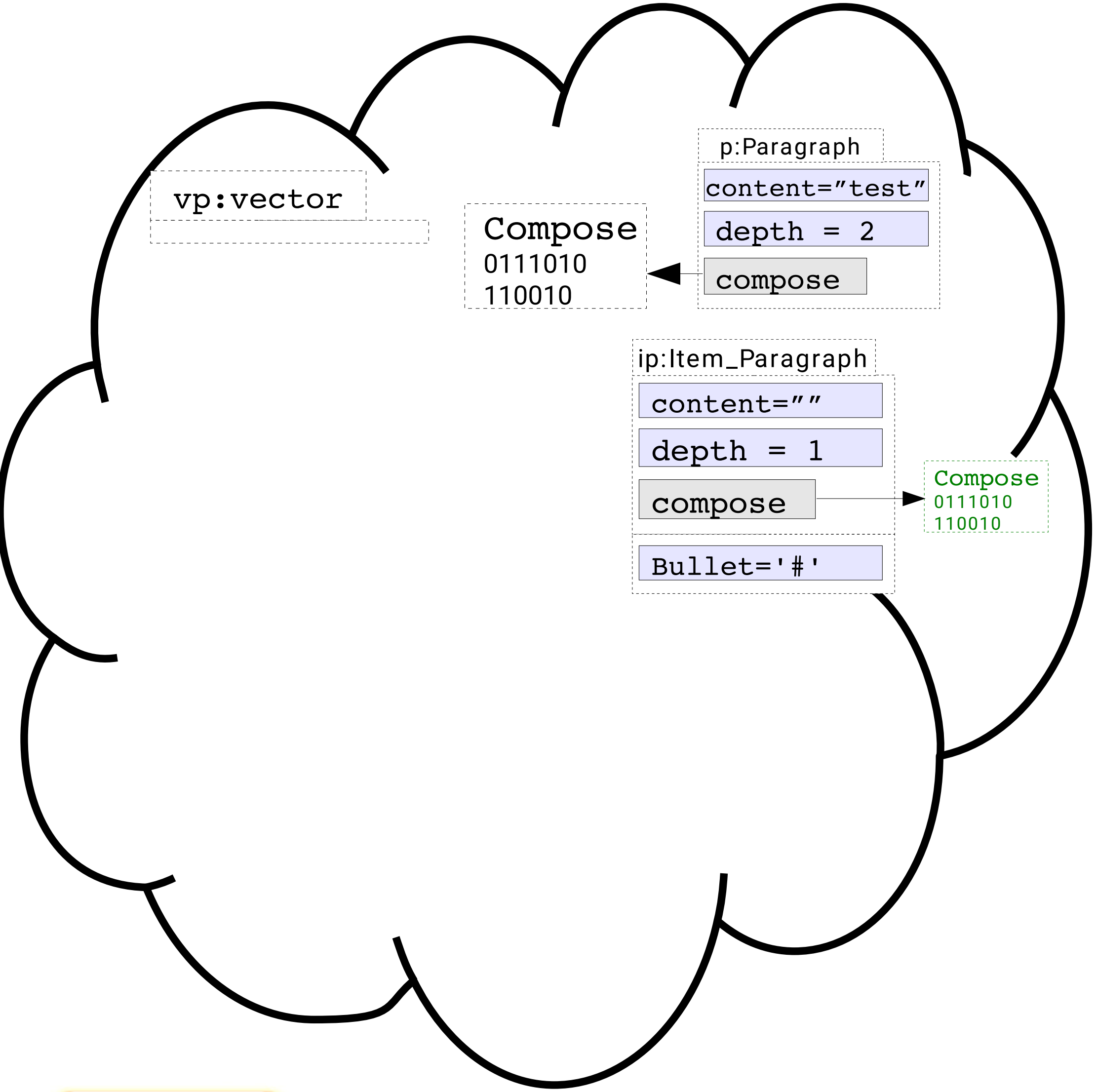


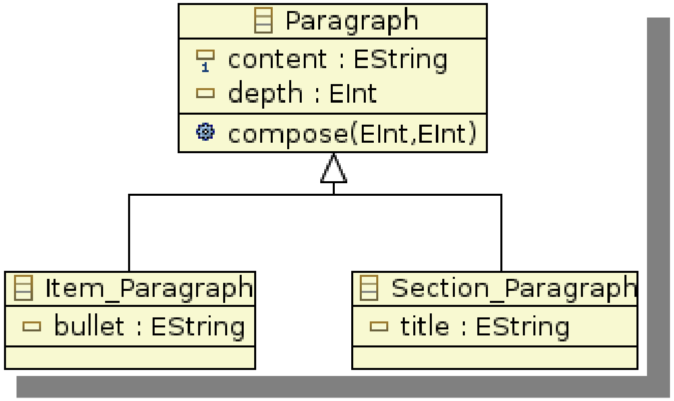
```
//...
Paragraph p("test",2);
Item_Paragraph ip("",1,'#');

p.compose();
ip.compose();
```

What if we create a vector of Paragraphs ??

What happens in memory (at least conceptually)

If `compose(int, int)` is NON virtual and is **REDEFINED** in Item_Paragraph

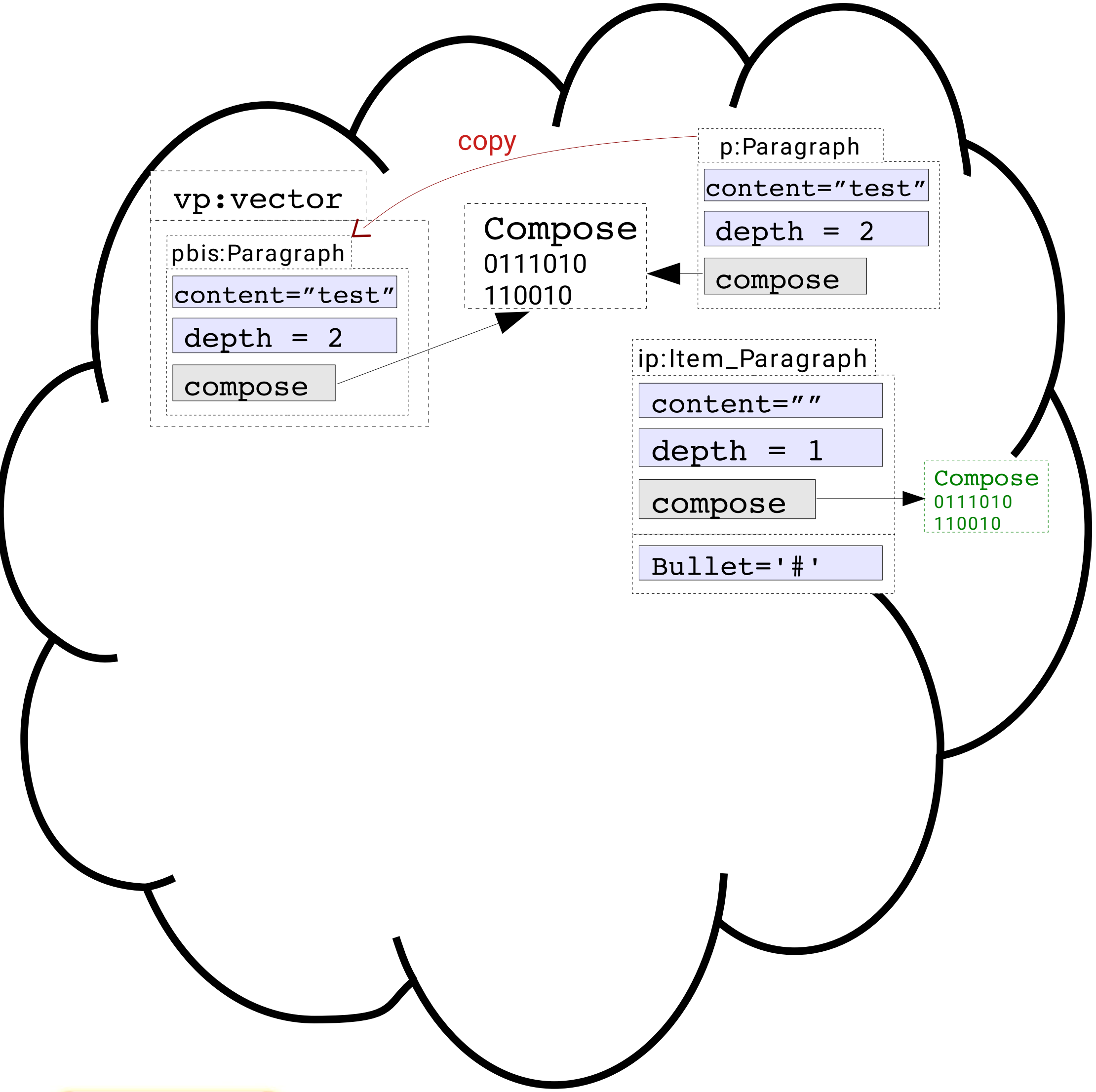


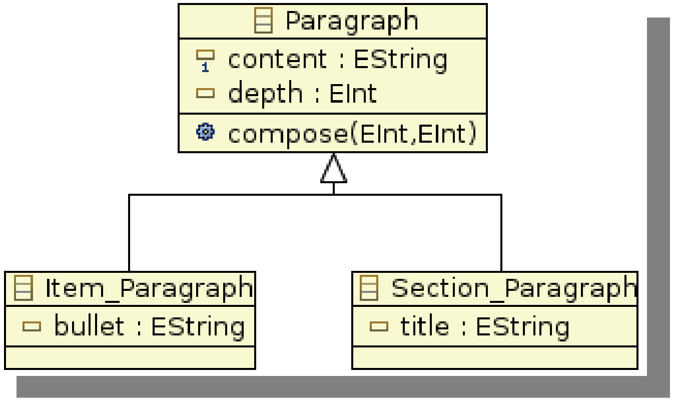


```
//...
Paragraph p("test",2);
Item_Paragraph ip("",1,'#');

vector<Paragraph> vp;
```

What happens in memory (at least conceptually)

If `compose(int, int)` is NON virtual and is **REDEFINED** in Item_Paragraph

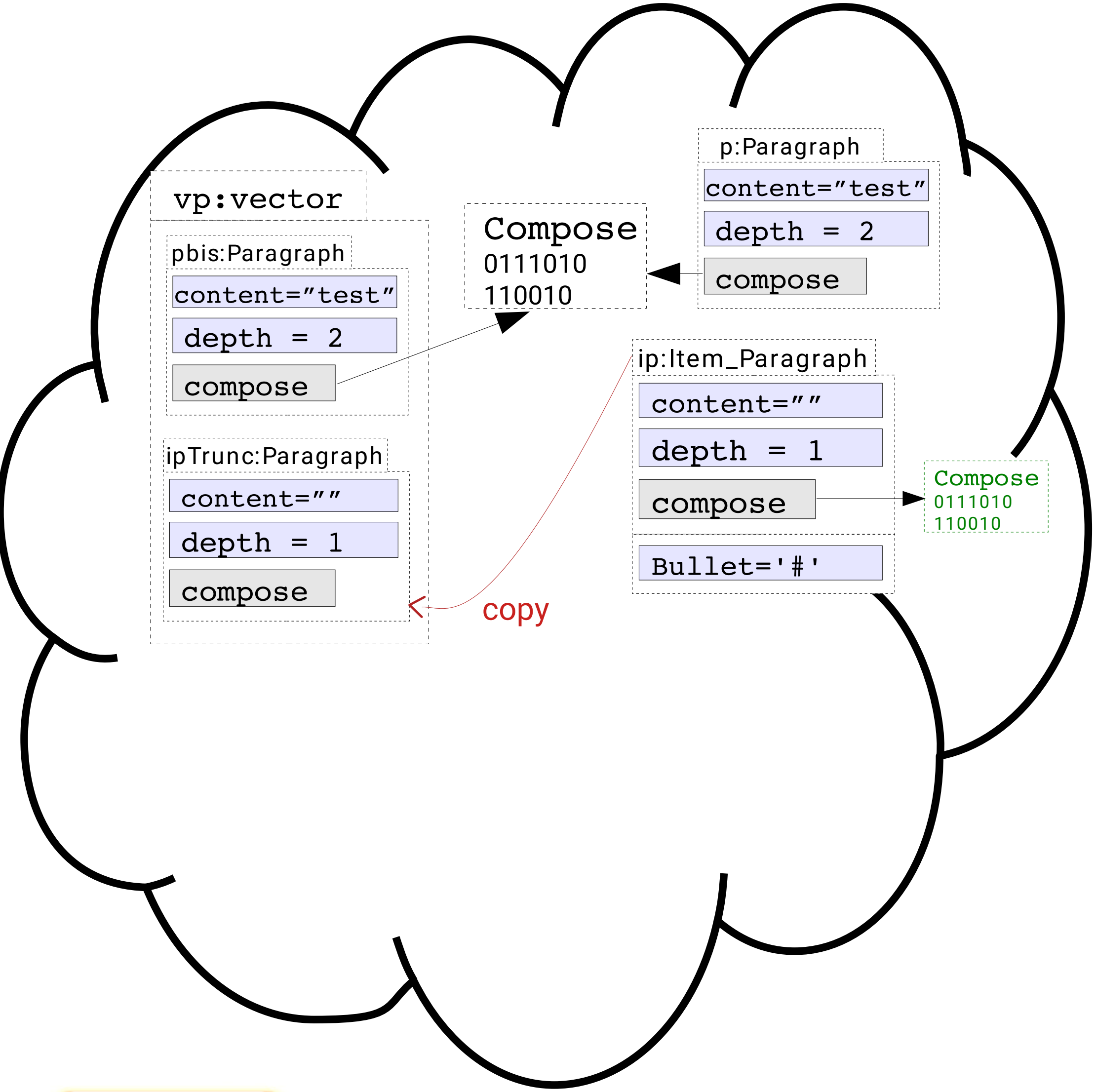


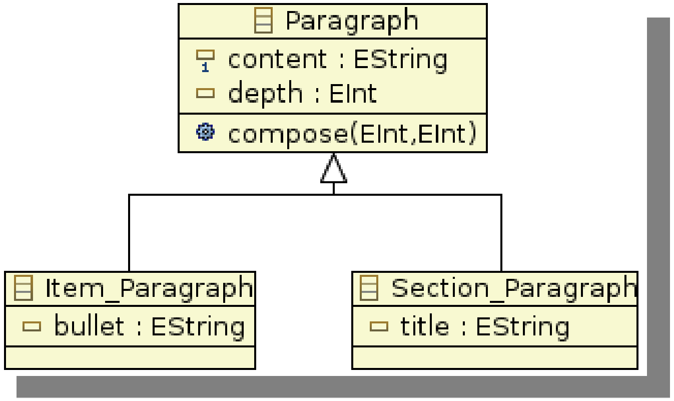


```
//...
Paragraph p("test",2);
Item_Paragraph ip("",1,'#');

vector<Paragraph> vp;
vp.push_back(p);
```

What happens in memory (at least conceptually)

If `compose(int, int)` is NON virtual and is **REDEFINED** in Item_Paragraph

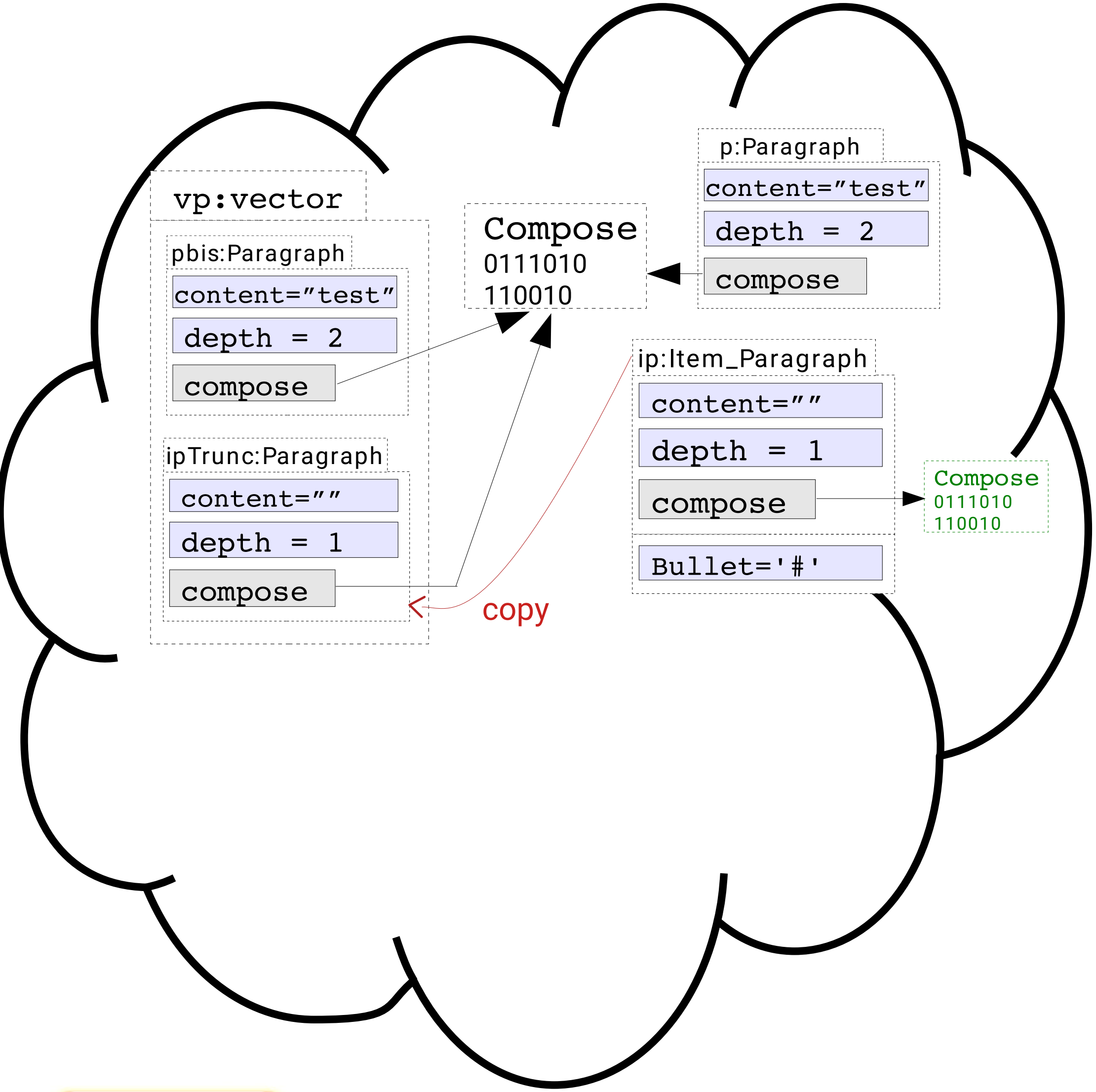


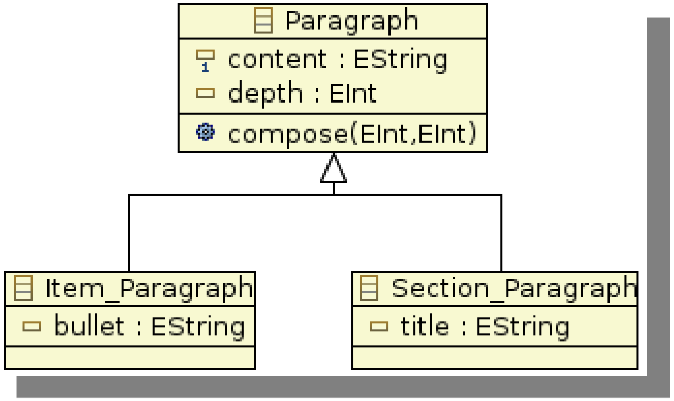


```
//...
Paragraph p("test",2);
Item_Paragraph ip("",1,'#');

vector<Paragraph> vp;
vp.push_back(p);
vp.push_back(ip);
```

What happens in memory (at least conceptually)

If `compose(int, int)` is NON virtual and is **REDEFINED** in Item_Paragraph

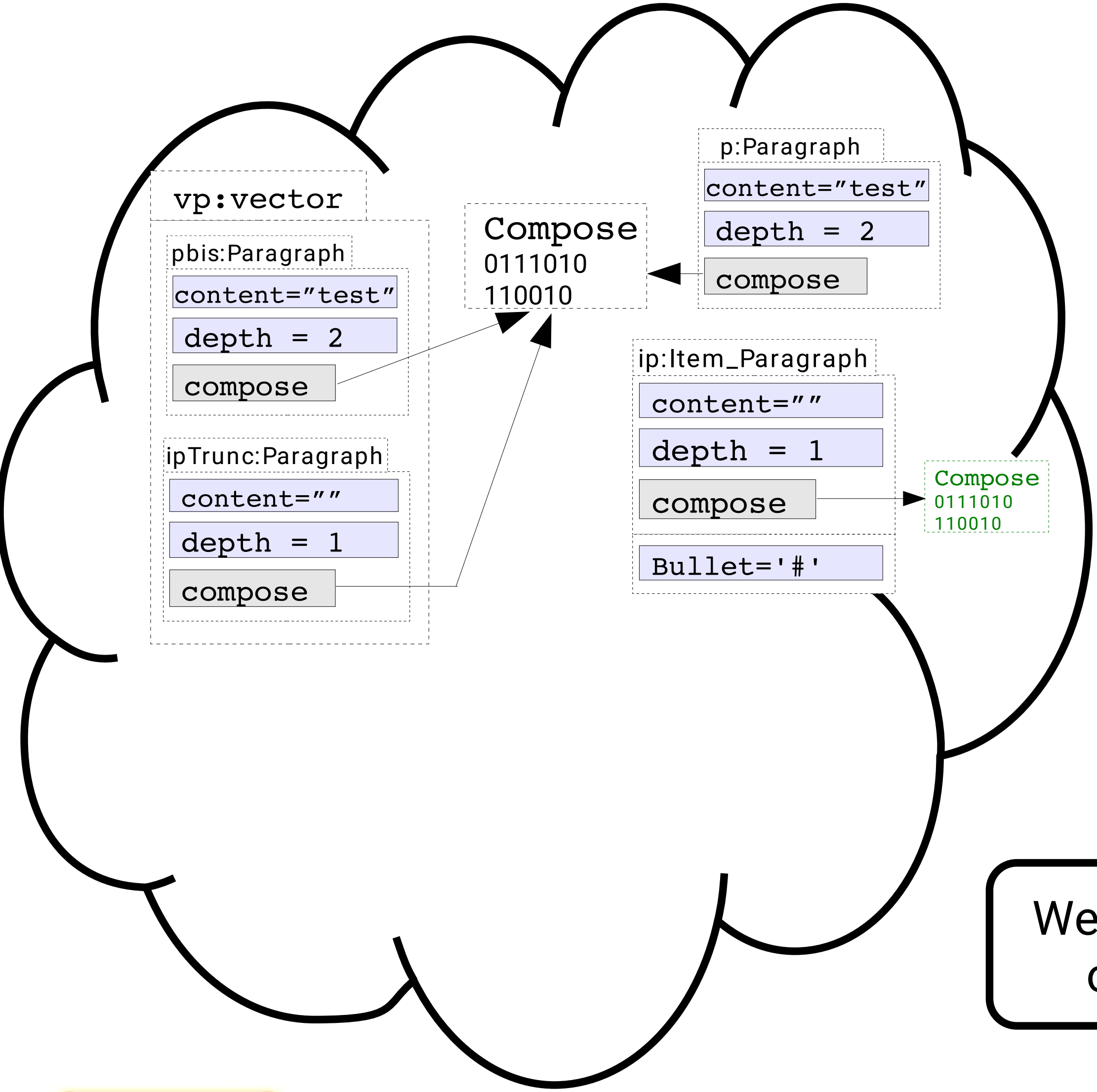


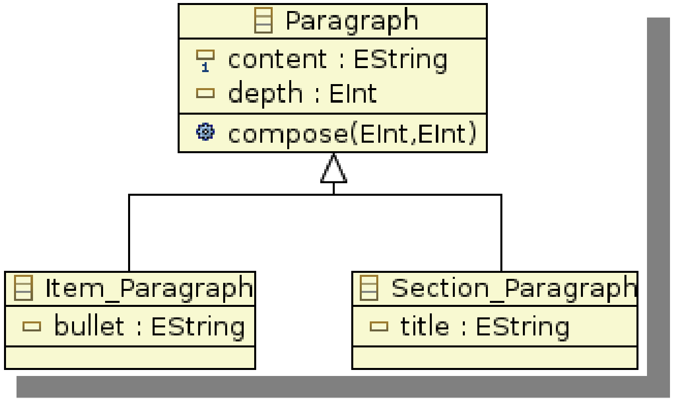


```
//...
Paragraph p("test",2);
Item_Paragraph ip("",1,'#');

vector<Paragraph> vp;
vp.push_back(p);
vp.push_back(ip);
```

What happens in memory (at least conceptually)

If `compose(int, int)` is NON virtual and is **REDEFINED** in Item_Paragraph



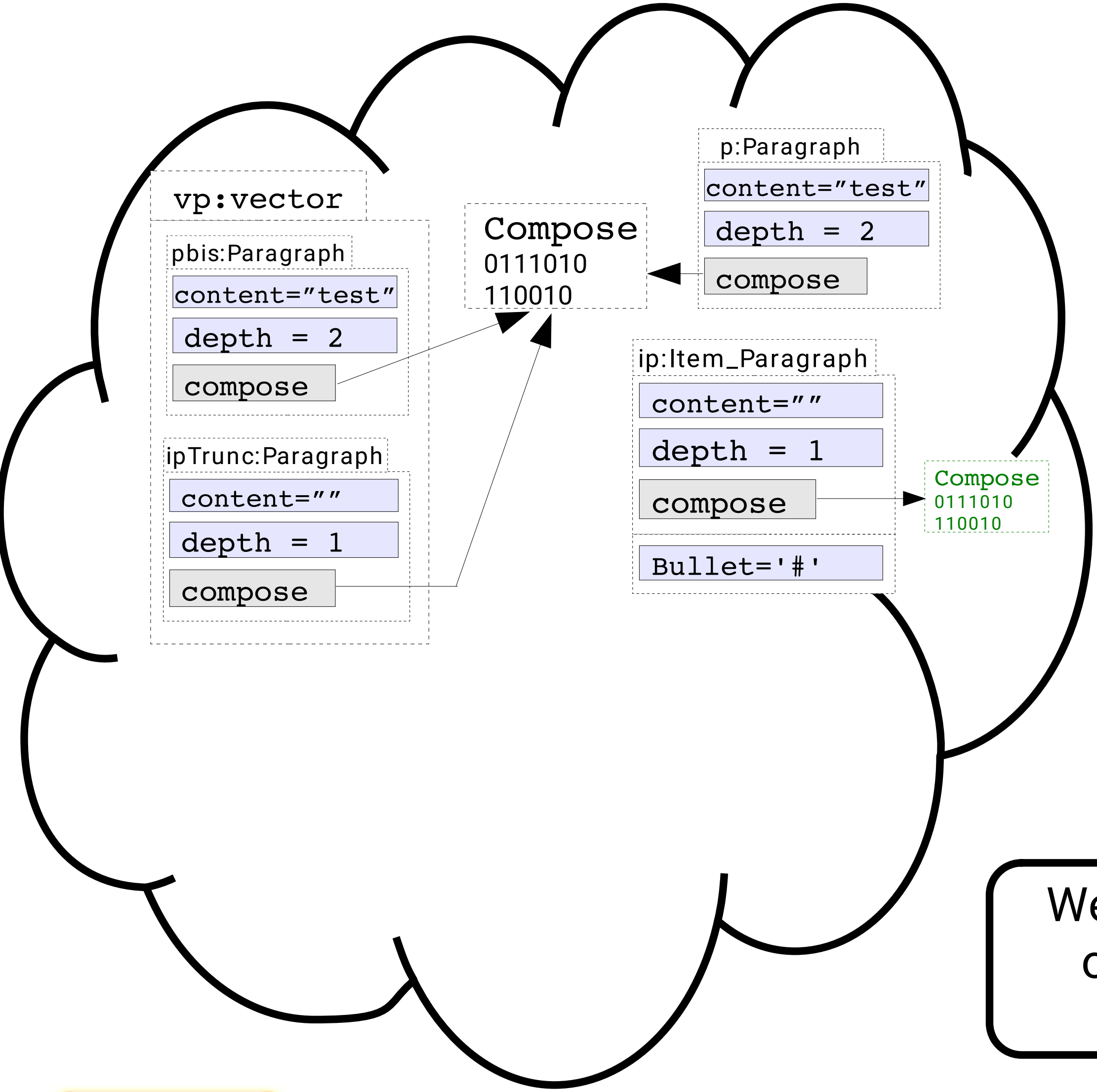


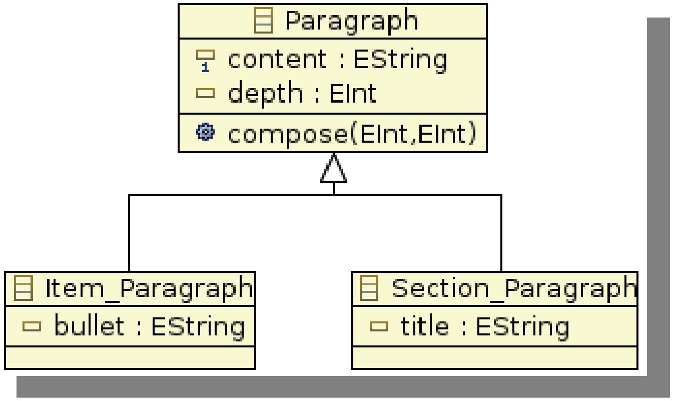
```
//...
Paragraph p("test",2);
Item_Paragraph ip("",1,'#');

vector<Paragraph> vp;
vp.push_back(p);
vp.push_back(ip);
```

We lost the specificities of Item_Paragraph

What happens in memory (at least conceptually)

If `compose(int, int)` is NON virtual and is **REDEFINED** in Item_Paragraph



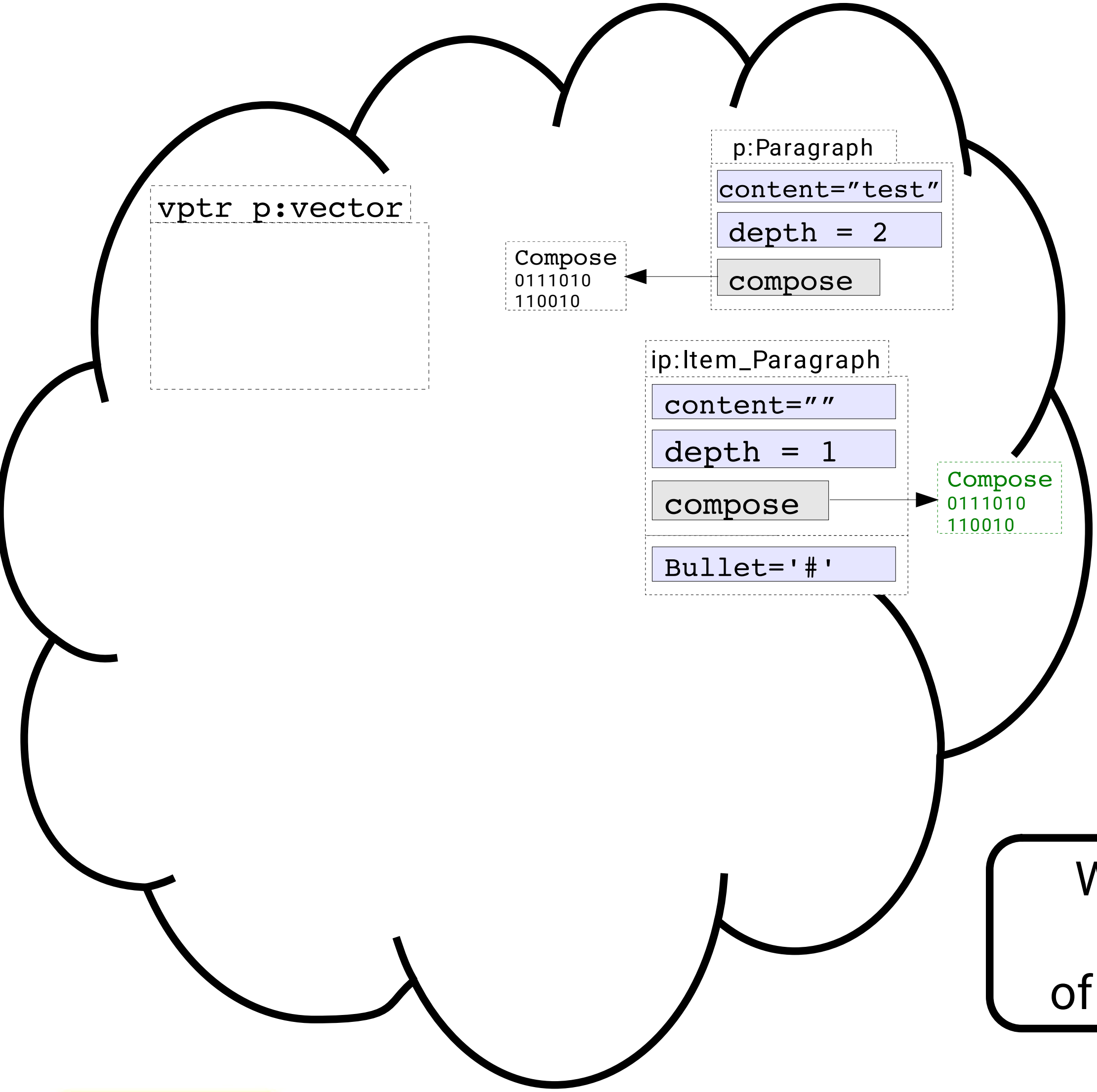


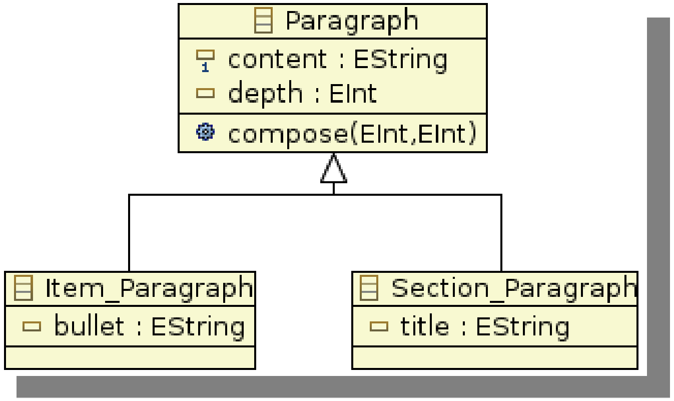
```
//...
Paragraph p("test",2);
Item_Paragraph ip("",1,'#');

vector<Paragraph> vp;
vp.push_back(p);
vp.push_back(ip);
```

We lost the specificities of Item_Paragraph...
Forever !

What happens in memory (at least conceptually)

If `compose(int, int)` is NON virtual and is **REDEFINED** in Item_Paragraph



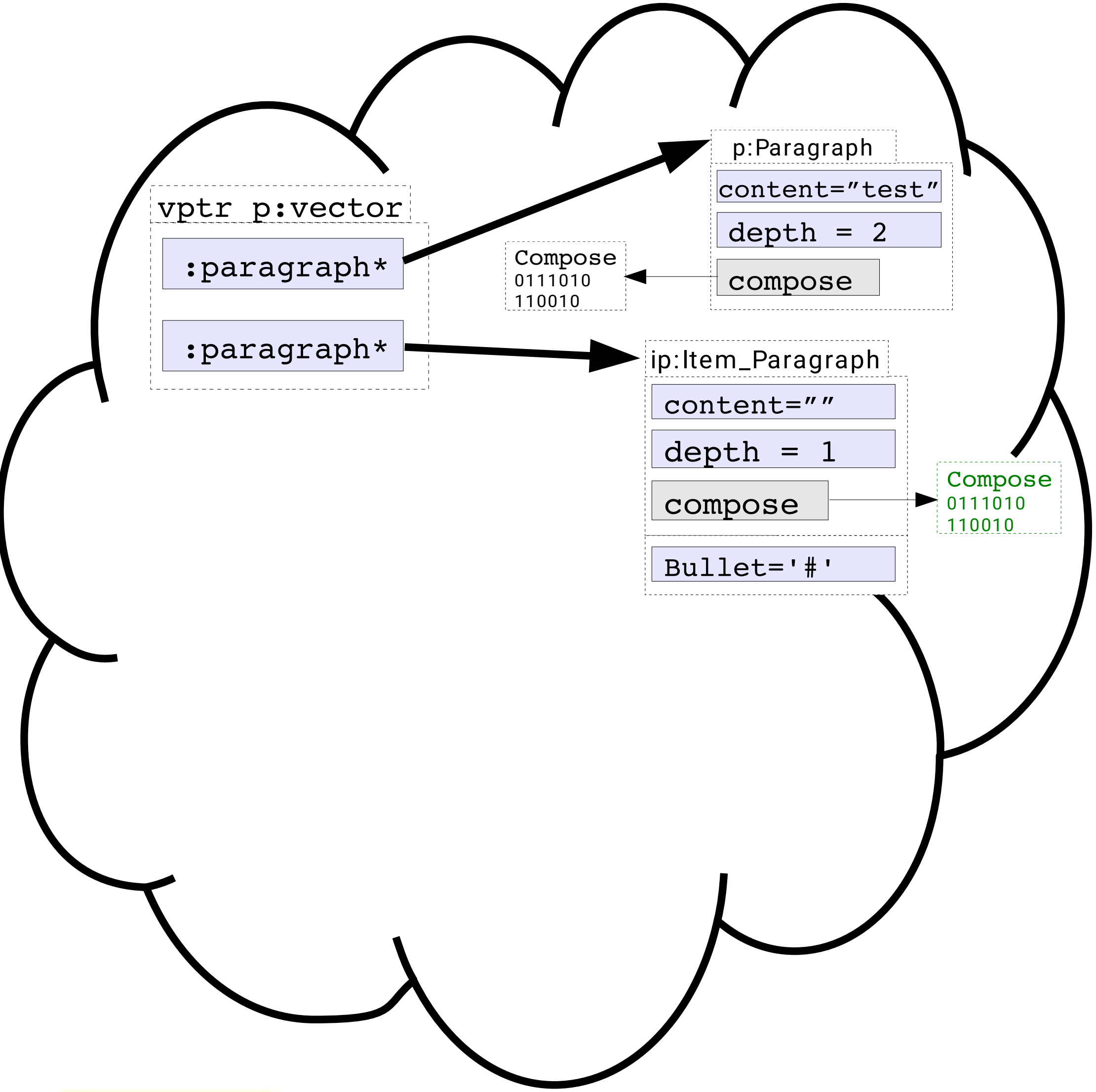


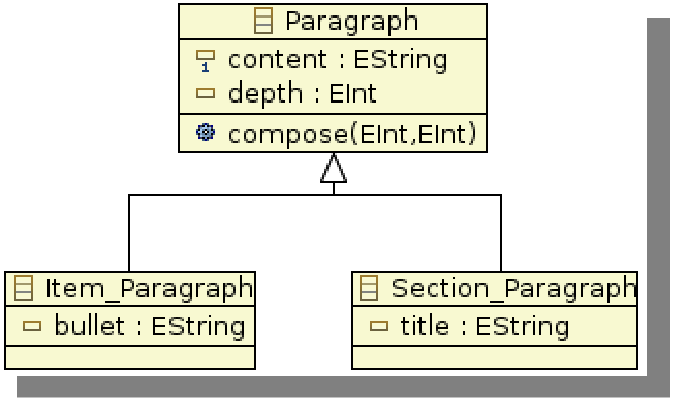
```
//...
Paragraph p("test",2);
Item_Paragraph ip("",1,'#');

vector<paragraph*> vptr_p
```

What happens if we create a vector of paragraph pointer ?

What happens in memory (at least conceptually)

If `compose(int, int)` is NON virtual and is **REDEFINED** in Item_Paragraph

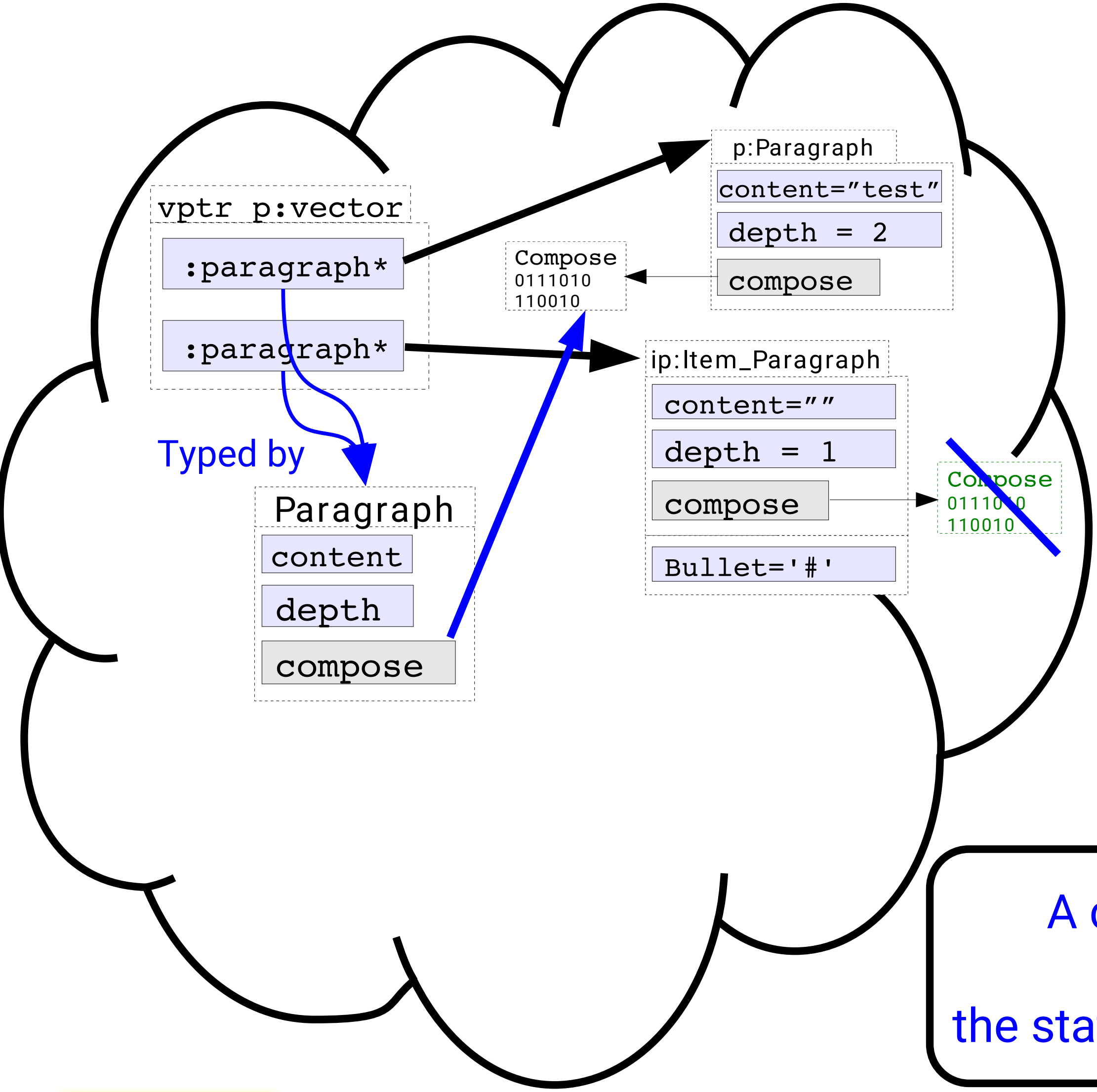


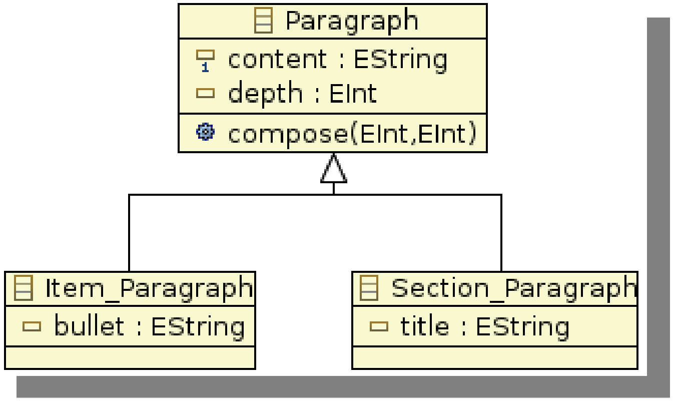


```
//...
Paragraph p("test",2);
Item_Paragraph ip("",1,'#');

vector<paragraph*> vptr_p
vptr_p.push_back(&p);
vptr_p.push_back(&ip);
```

What happens in memory (at least conceptually)

If `compose(int, int)` is NON virtual and is **REDEFINED** in Item_Paragraph



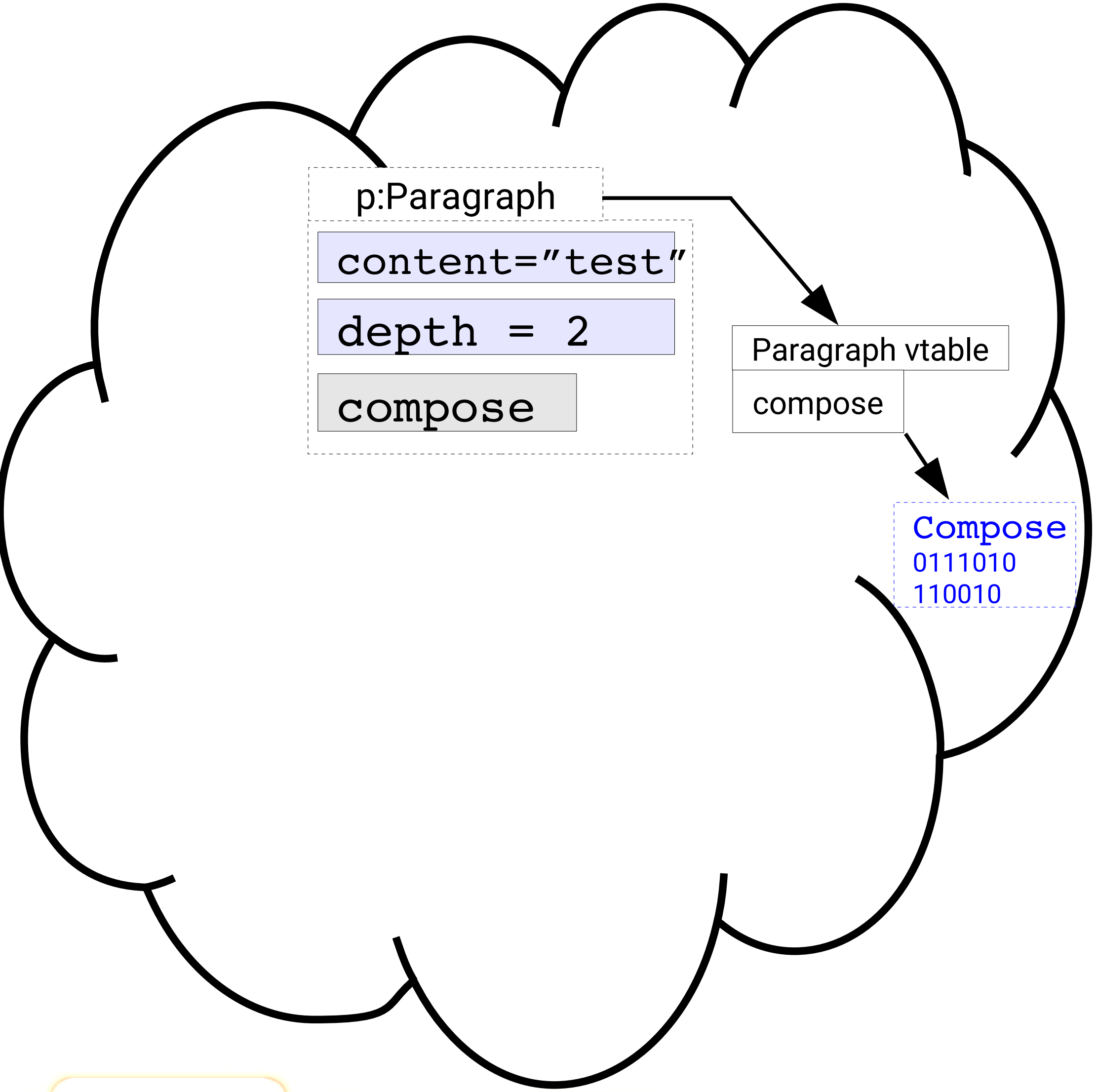


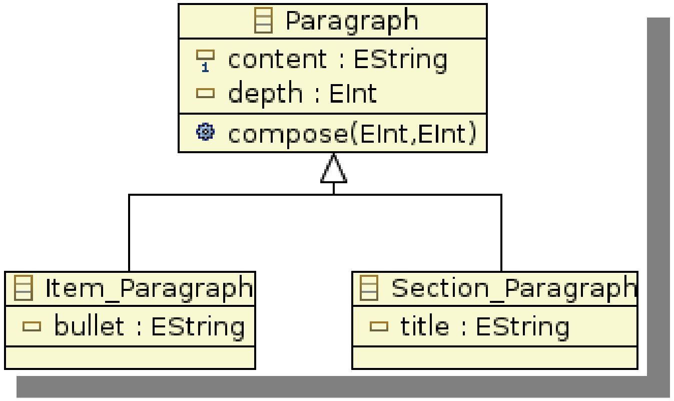
```
//...
Paragraph p("test",2);
Item_Paragraph ip("",1,'#');

vector<paragraph*> vptr_p
vptr_p.push_back(&p);
vptr_p.push_back(&ip);
```

A call to a non-virtual function
is resolved according to
the static type of the hidden parameter

What happens in memory (at least conceptually)

If `compose(int, int)` is **VIRTUAL** and is **REDEFINED** in Item_Paragraph

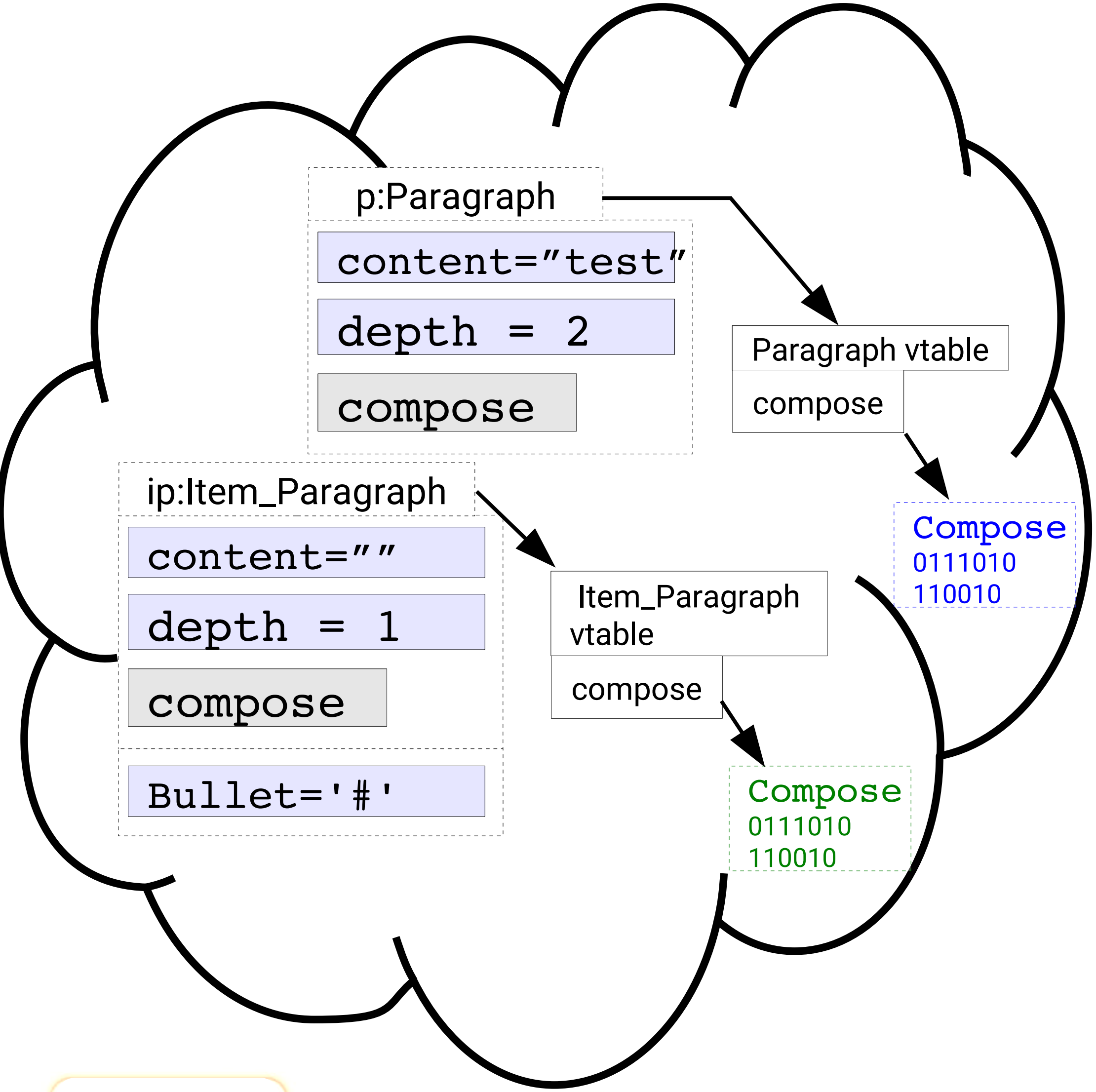


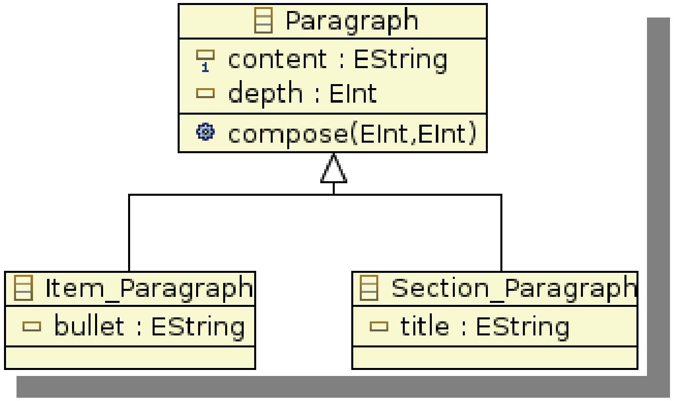


```
//...
Paragraph p("test",2);
p.compose();
```

What happens in memory (at least conceptually)

If `compose(int, int)` is **VIRTUAL** and is **REDEFINED** in Item_Paragraph

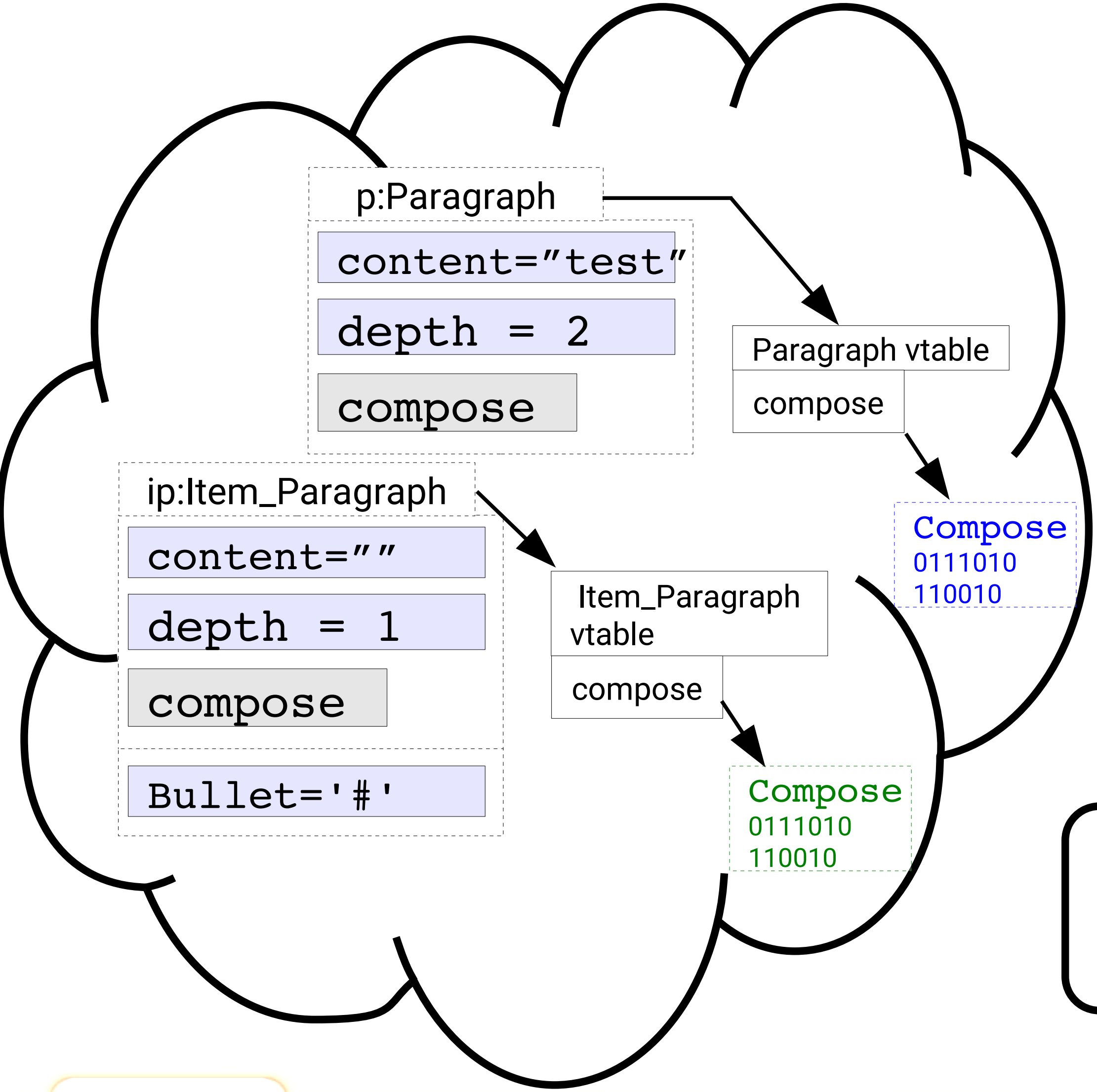


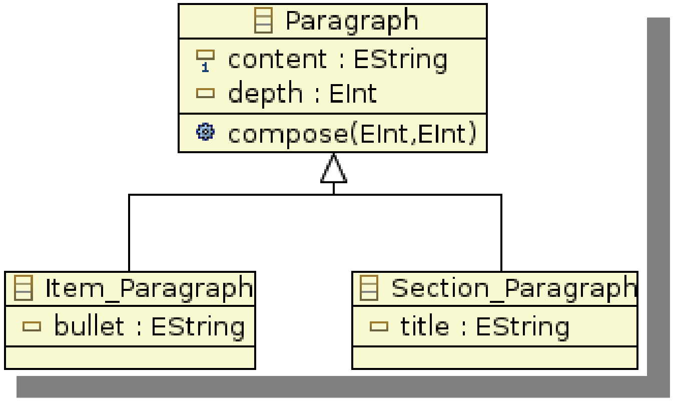


```
//...
Paragraph p("test",2);
Item_Paragraph ip("",1,'#');

p.compose();
ip.compose();
```

What happens in memory (at least conceptually)

If `compose(int, int)` is **VIRTUAL** and is **REDEFINED** in Item_Paragraph



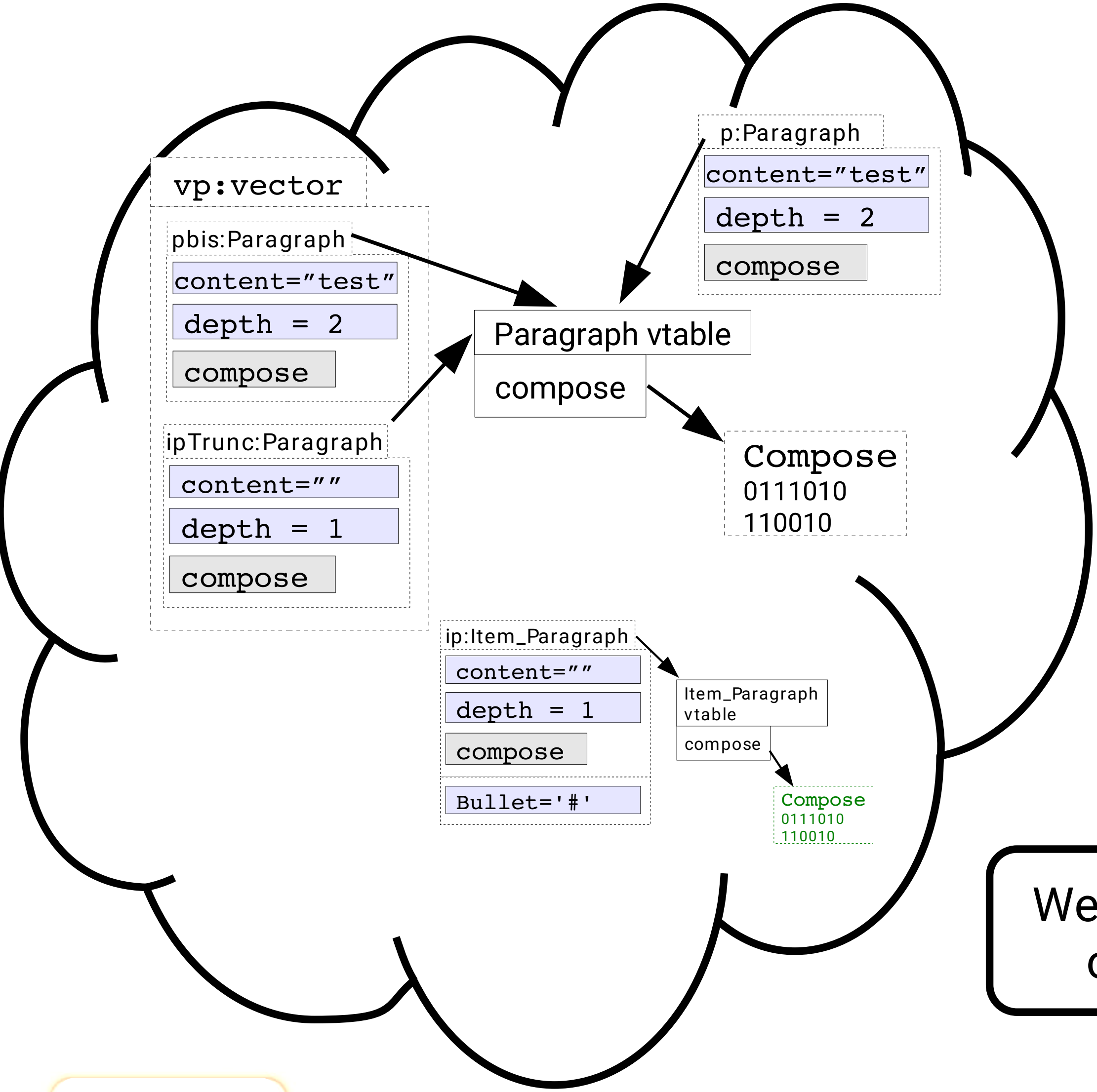


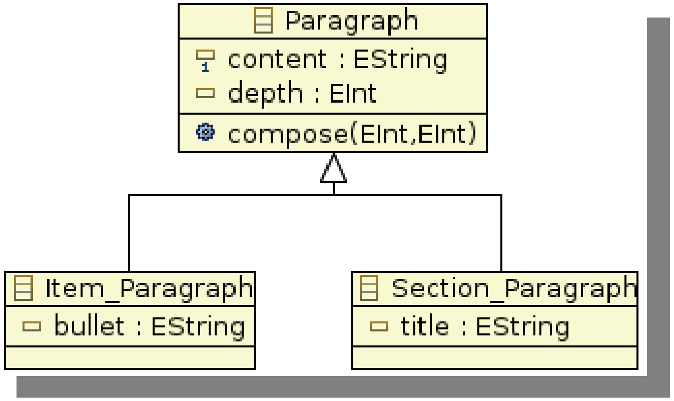
```
//...
Paragraph p("test",2);
Item_Paragraph ip("",1,'#');

p.compose();
ip.compose();
```

What happens if we create a vector of Paragraphs ??

What happens in memory (at least conceptually)

If `compose(int, int)` is **VIRTUAL** and is **REDEFINED** in Item_Paragraph



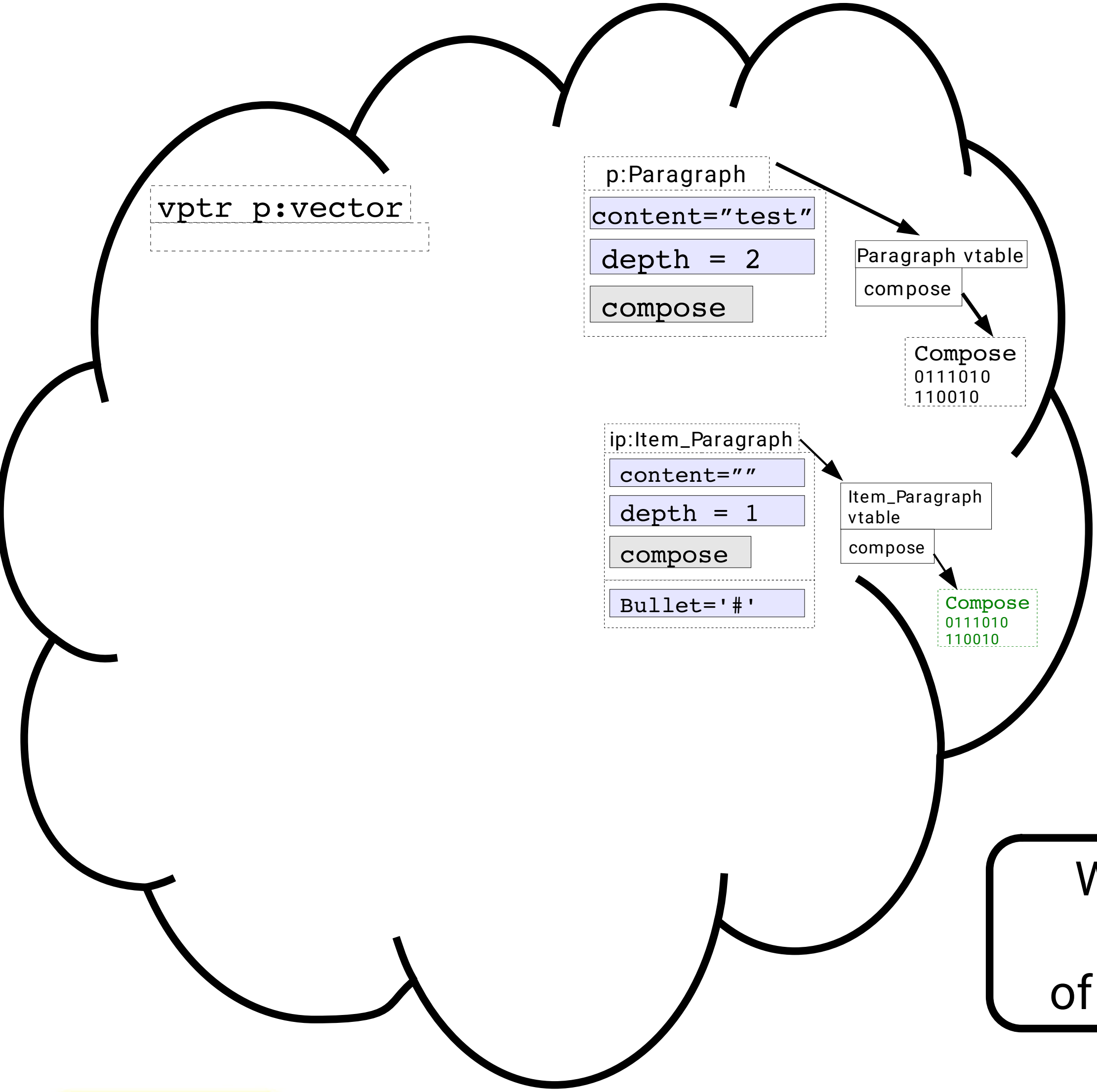


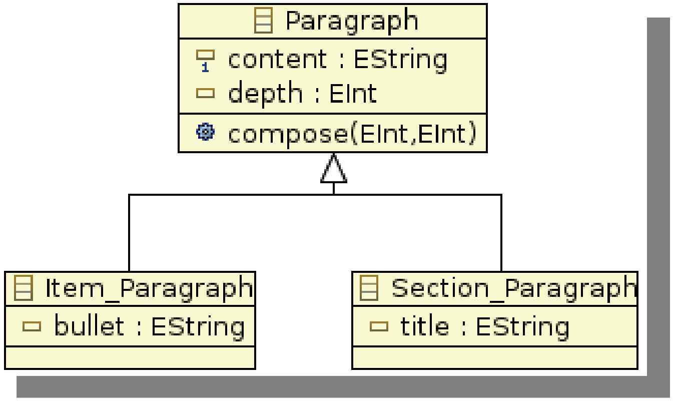
```
//...
Paragraph p("test",2);
Item_Paragraph ip("",1,'#');

vector<Paragraph> vp;
vp.push_back(p);
vp.push_back(ip);
```

We lost the specificities of Item_Paragraph

What happens in memory (at least conceptually)

If `compose(int, int)` is **VIRTUAL** and is **REDEFINED** in Item_Paragraph



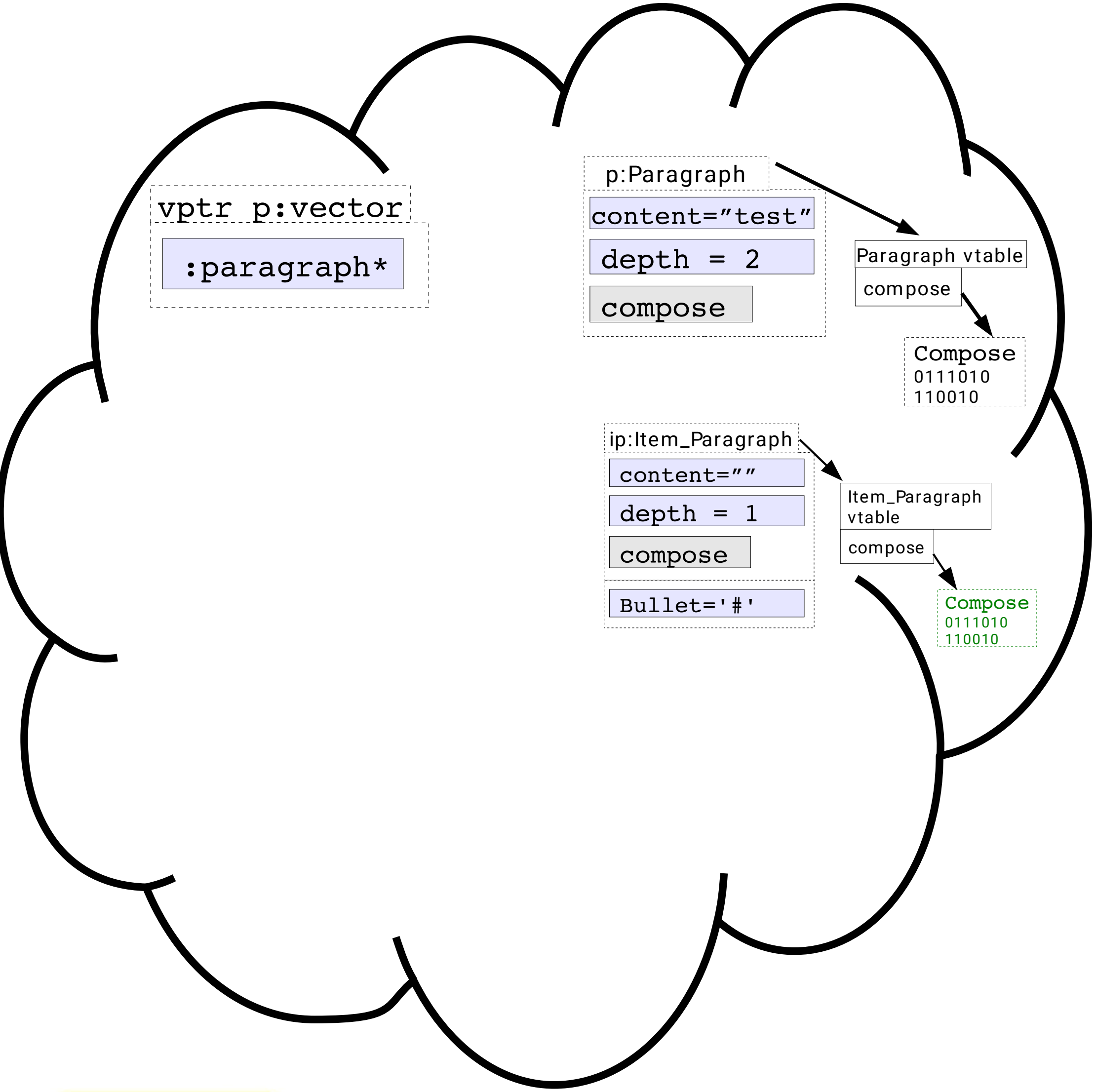


```
//...
Paragraph p("test",2);
Item_Paragraph ip("",1,'#');

vector<paragraph*> vptr_p
```

What happens if we create a vector of paragraph pointer ?

What happens in memory (at least conceptually)

If `compose(int, int)` is **VIRTUAL** and is **REDEFINED** in Item_Paragraph



vptr p:vector

:paragraph*

p:Paragraph

content="test"

depth = 2

compose

Paragraph vtable

compose

Compose
0111010
110010

ip:Item_Paragraph

content=""

depth = 1

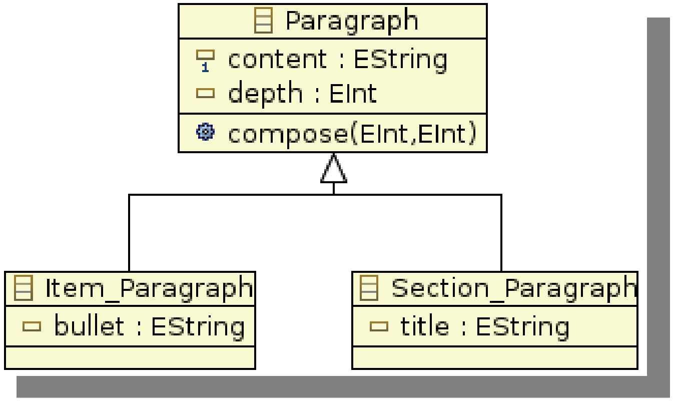
compose

Bullet='#'

Item_Paragraph vtable

compose

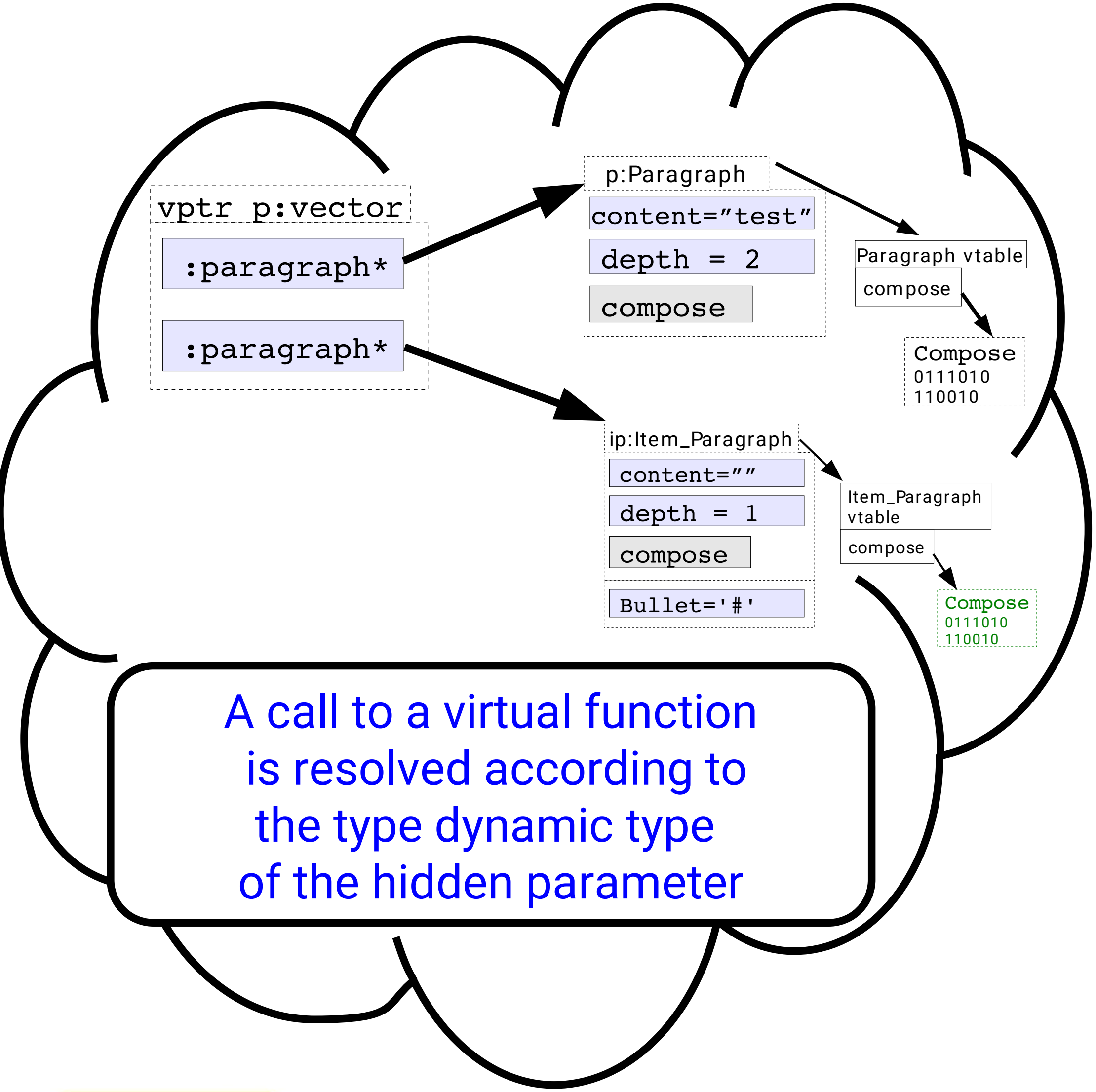
Compose
0111010
110010



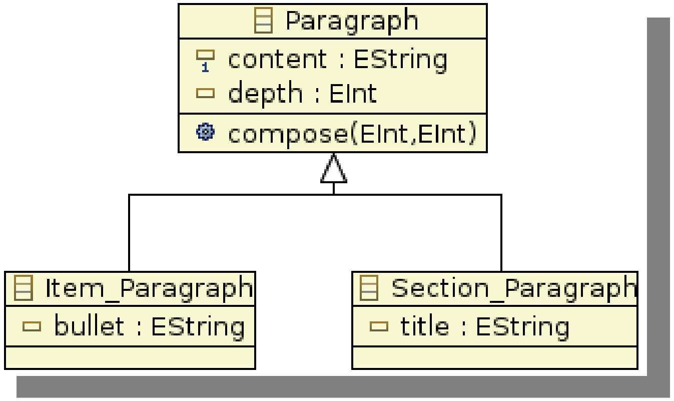
```
//...
Paragraph p("test",2);
Item_Paragraph ip("",1,'#');

vector<paragraph*> vptr_p
vptr_p.push_back(&p);
```

What happens in memory (at least conceptually)

If `compose(int, int)` is **VIRTUAL** and is **REDEFINED** in Item_Paragraph



A call to a virtual function is resolved according to the type dynamic type of the hidden parameter

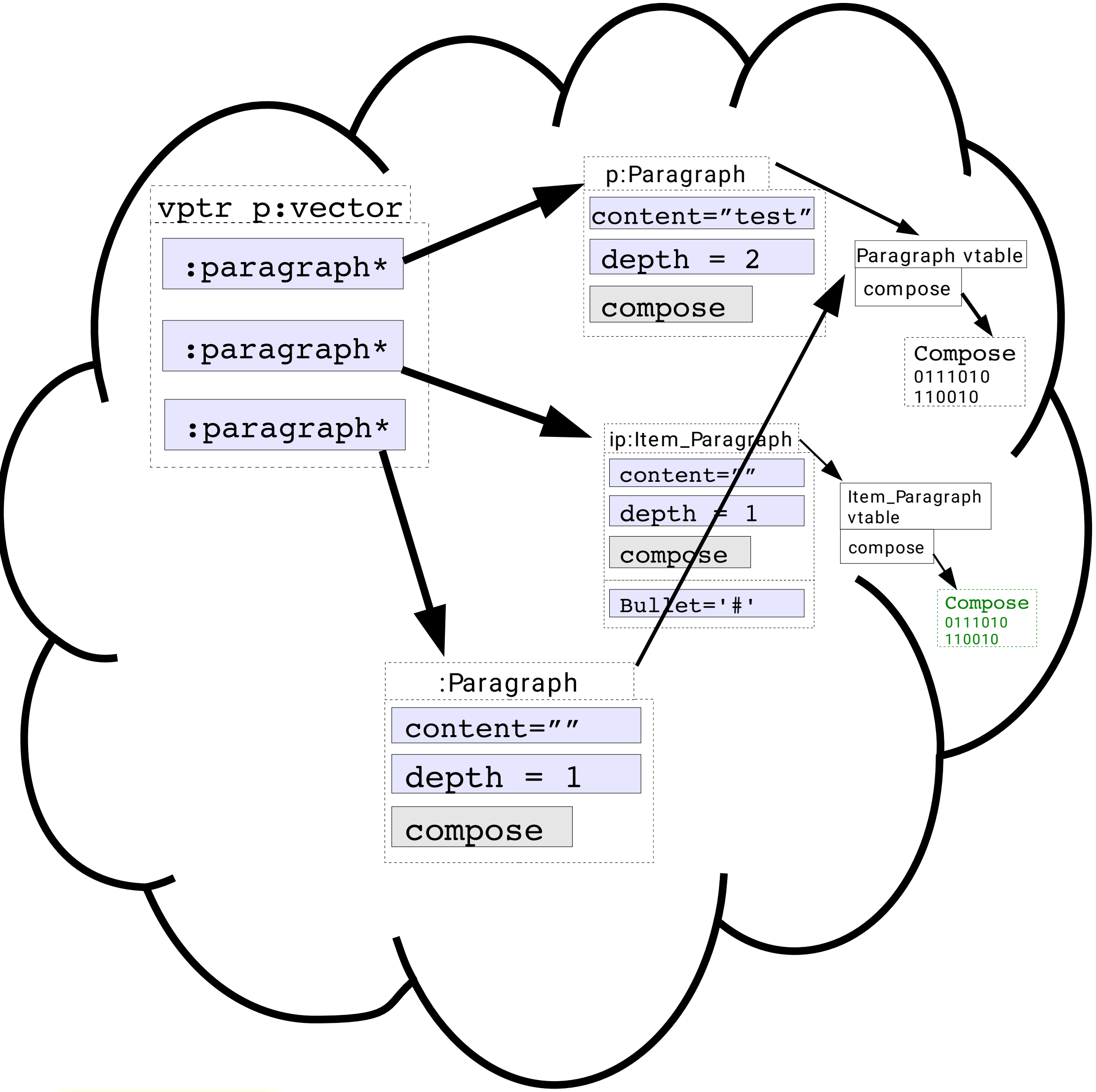


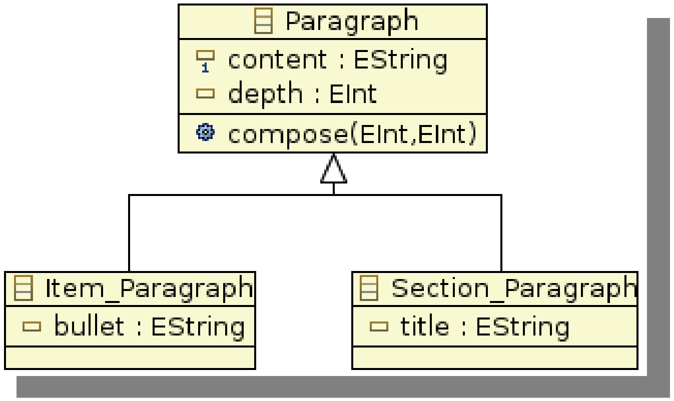
```
//...
Paragraph p("test",2);
Item_Paragraph ip("",1,'#');

vector<paragraph*> vptr_p
vptr_p.push_back(&p);
vptr_p.push_back(&ip);
```

✓

What happens in memory (at least conceptually)

If `compose(int, int)` is **VIRTUAL** and is **REDEFINED** in Item_Paragraph

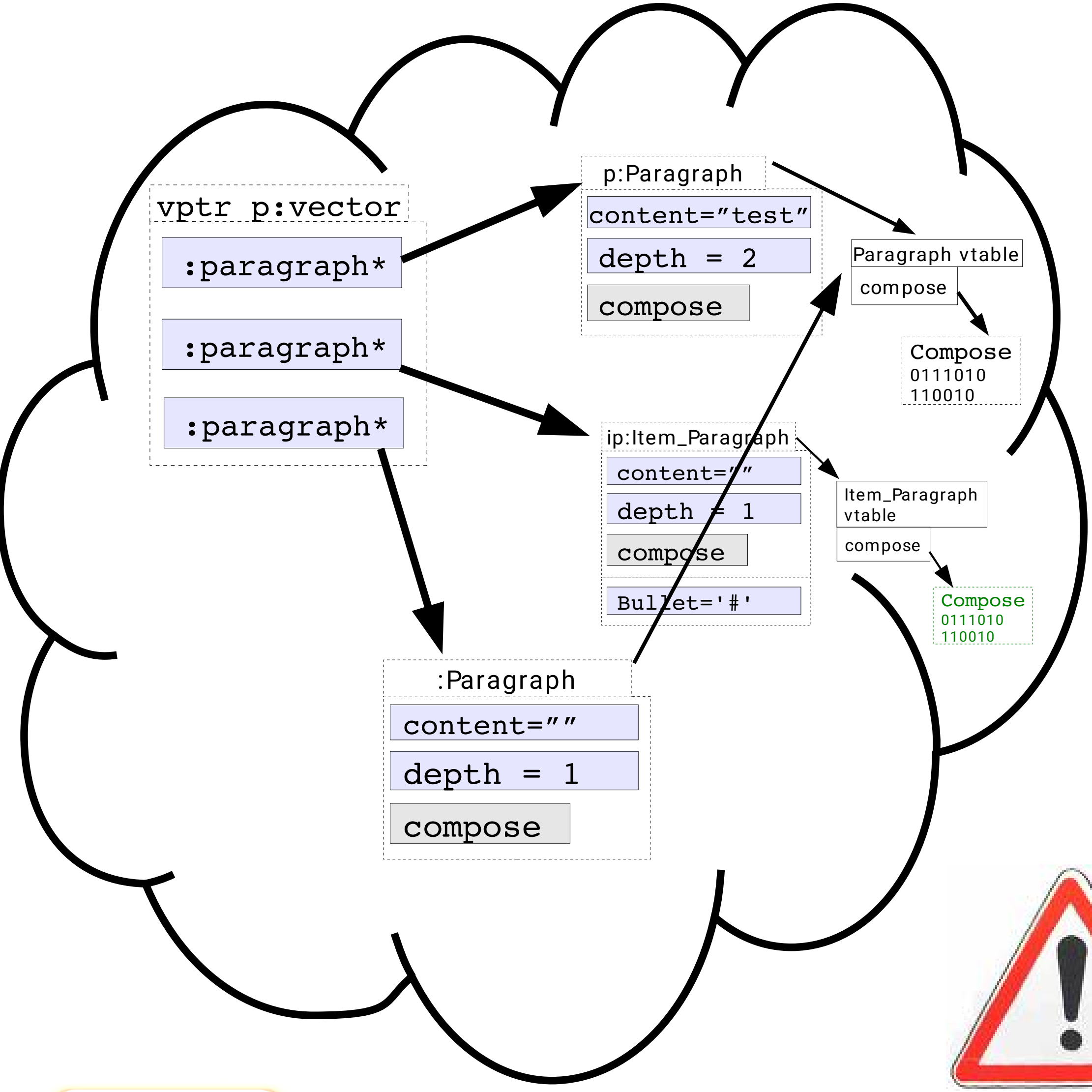


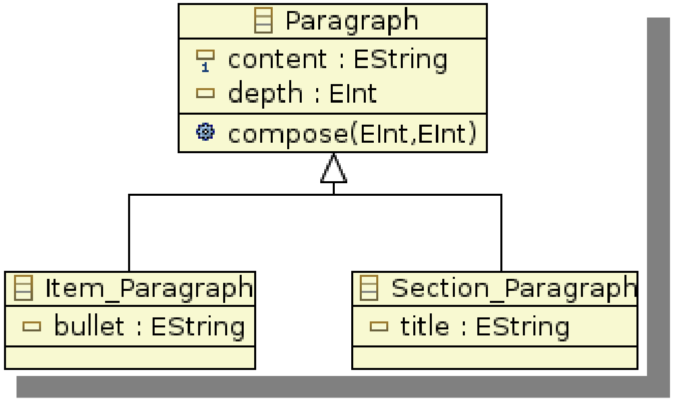


```
//...
Paragraph p(“test”,2);
Item_Paragraph ip(“”,1,'#');

vector<paragraph*> vptr_p
vptr_p.push_back(&p);
vptr_p.push_back(&ip);
vptr_p.push_back(new Paragraph(ip));
```

What happens in memory (at least conceptually)

If `compose(int, int)` is **VIRTUAL** and is **REDEFINED** in Item_Paragraph

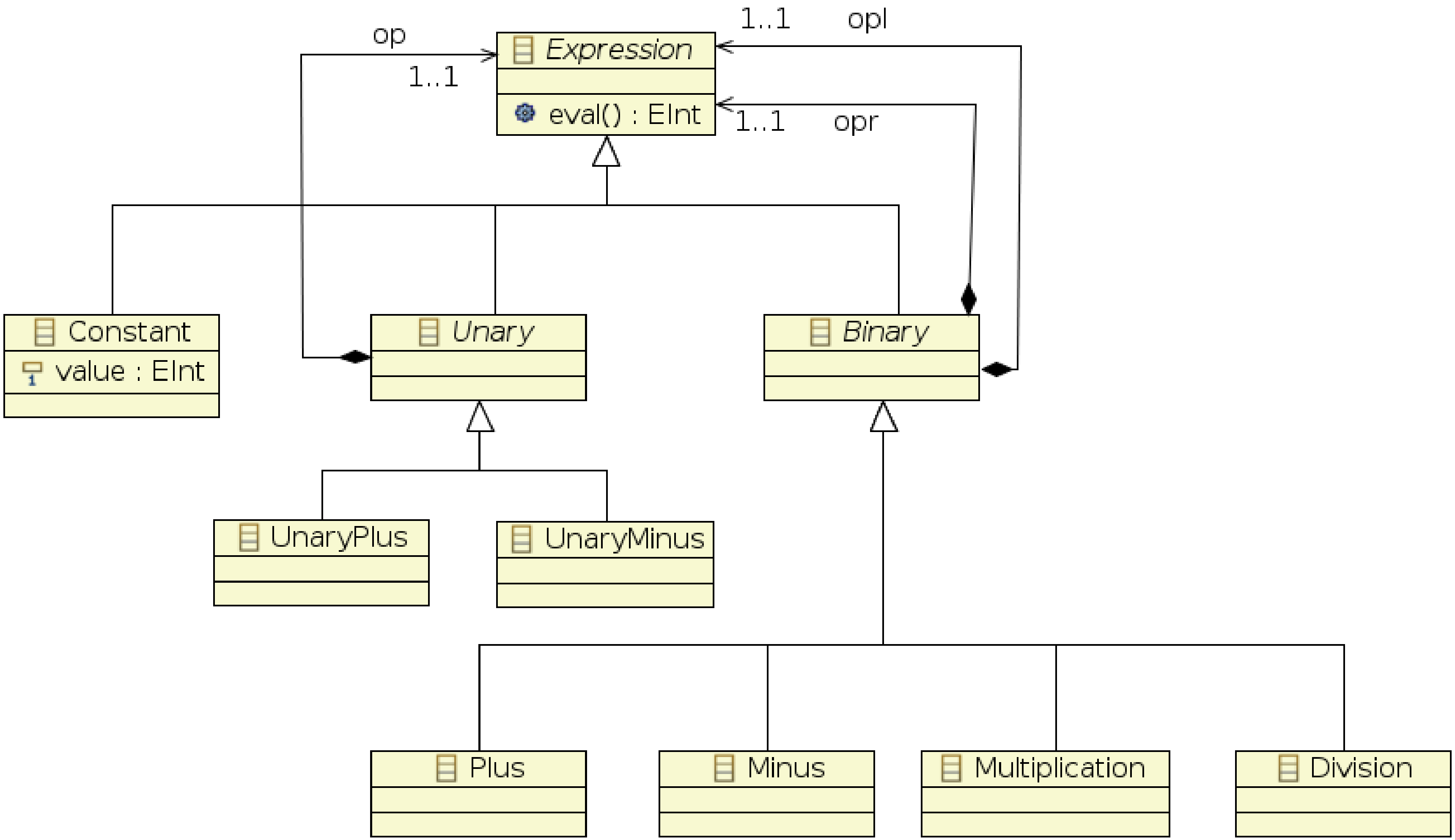




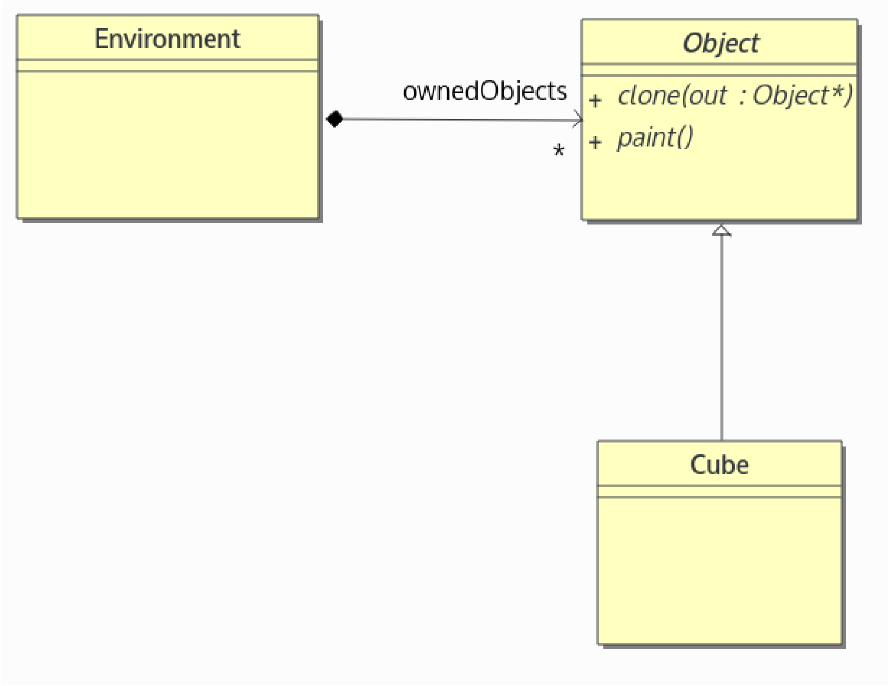
```
//...
Paragraph p("test",2);
Item_Paragraph ip("",1,'#');

vector<paragraph*> vptr_p
vptr_p.push_back(&p);
vptr_p.push_back(&ip);
vptr_p.push_back(new Paragraph(ip));
```

You have to use the `clone()` virtual function
(course 7-OO.pdf)

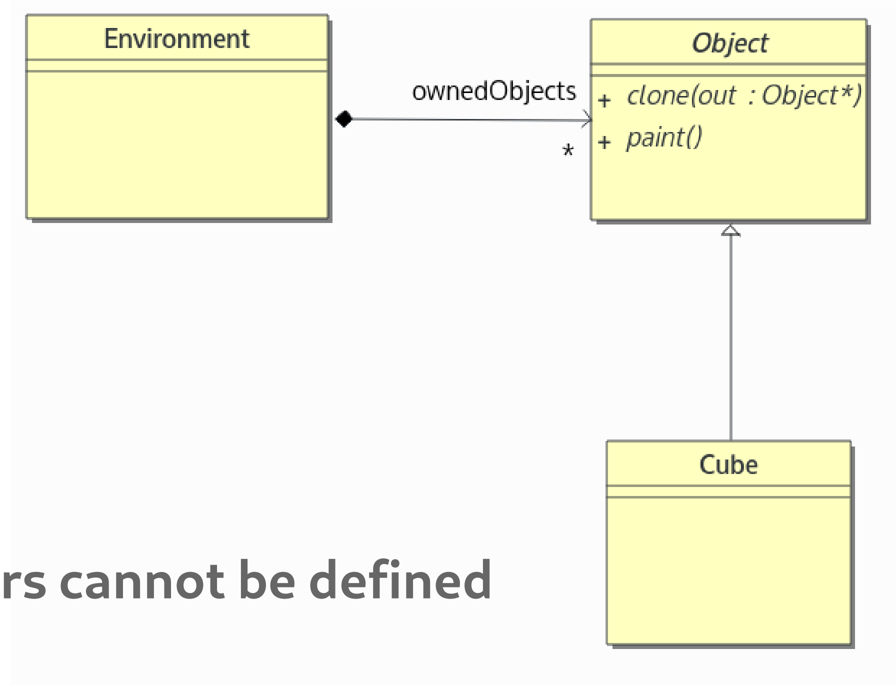
Abstract classes and pure virtual functions



Abstract classes and pure virtual functions



Abstract classes and pure virtual functions

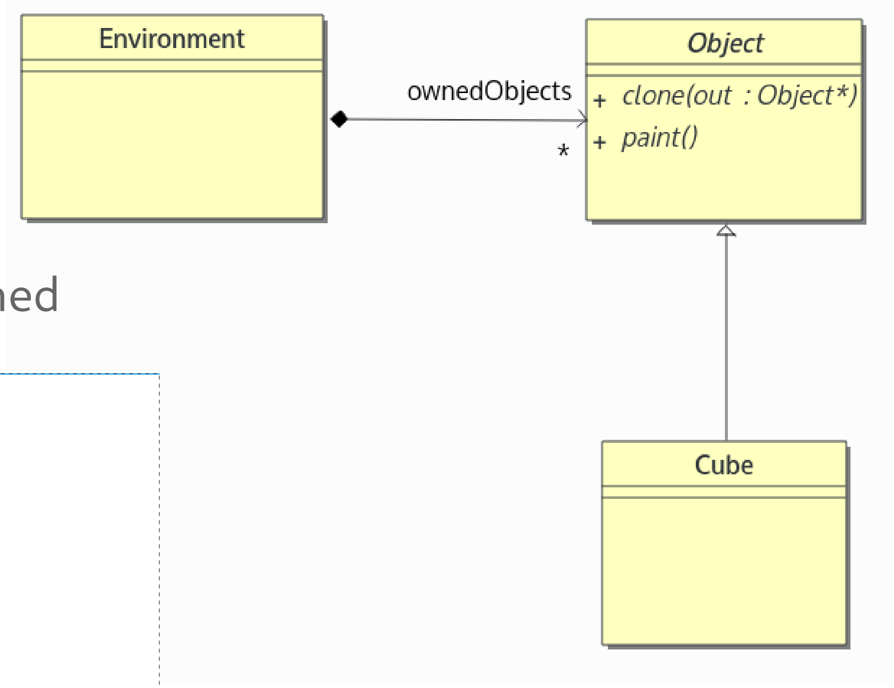


- Object is abstract
  - It cannot be instantiated
  - (because) **Some of its behaviors cannot be defined**

→ at least one of its member function is a pure virtual function

Abstract classes and pure virtual functions

- Object is abstract
  - It cannot be instantiated
  - Some of its behaviors cannot be defined

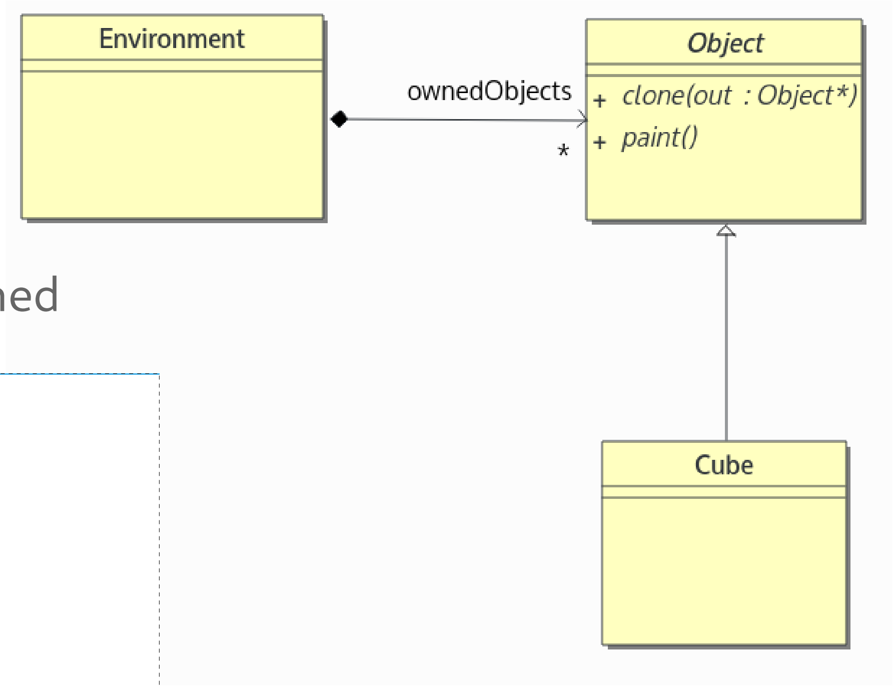


```
#ifndef _OBJECT_H
#define _OBJECT_H


class Object
{
    public:
//[...]
        virtual Object* clone() const=0;
        virtual void paint()=0;
        virtual ~Object();
};

#endif // _OBJECT_H
```

pure virtual functions

Abstract classes and pure virtual functions

- Object is abstract
  - It cannot be instantiated
  - Some of its behaviors cannot be defined



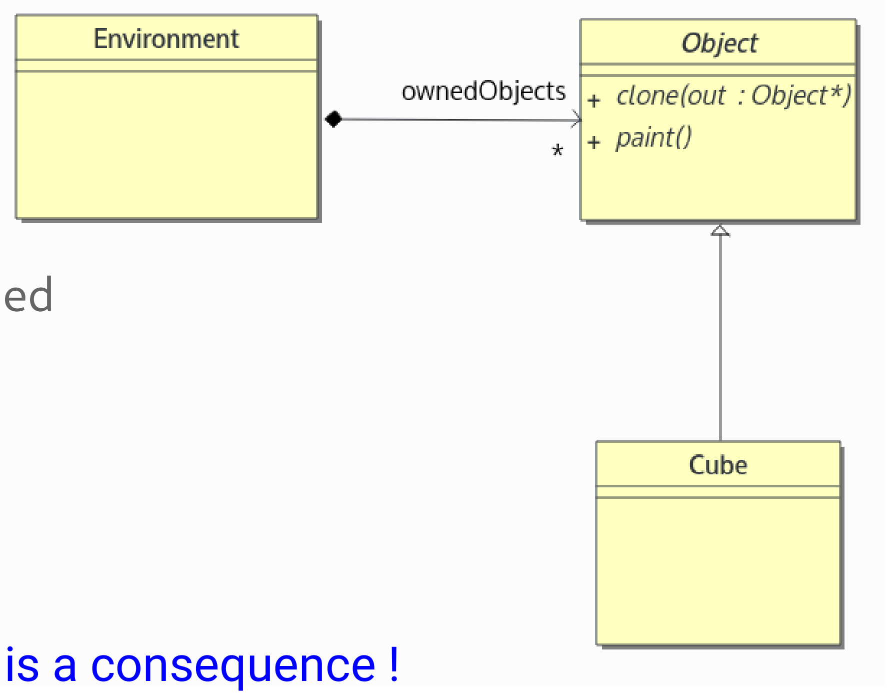
```
#ifndef _OBJECT_H
#define _OBJECT_H


class Object
{
    public:
//[...]
        virtual Object* clone() const=0;
        virtual void paint()=0;
        virtual ~Object();
};

#endif // _OBJECT_H
```

} pure virtual functions

It is not always the case that a Class has only pure virtual functions

Abstract classes and pure virtual functions

- Object is abstract
    - It cannot be instantiated
    - Some of its behaviors cannot be defined



In C++, a Class is not *a priori* Abstract. This is a consequence !
If some behavior cannot be defined, it means this class should not be instantiated and consequently it is an abstract class

Safe downward cast (1)

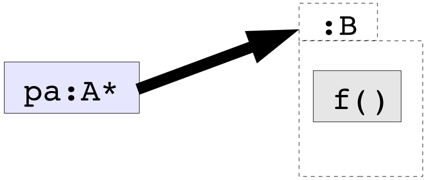
- **Downward cast may be dangerous**

```
class A { ... };
class B : public A {
   void f() { ... }      // f() not defined in A
};
```

Safe downward cast (1)

- Downward cast may be dangerous

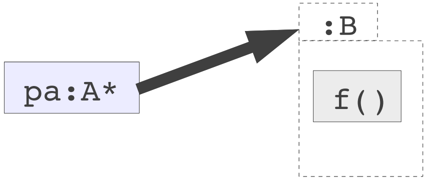
```
class A { ... };
class B : public A {
   void f() { ... }          // f() not defined in A
};

A* pa = new B();             // OK
```



Safe downward cast (1)

- Downward cast may be dangerous

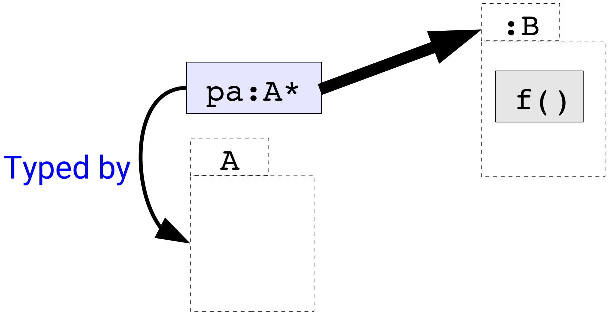
```
class A { ... };
class B : public A {
   void f() { ... }        // f() not defined in A
};

A* pa = new B();           // OK
pa->f();                   // KO
```



Safe downward cast (1)

- **Downward cast may be dangerous**

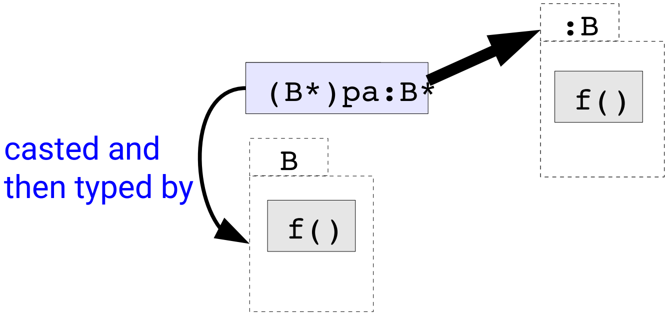
```
class A { ... };
class B : public A {
   void f() { ... }      // f() not defined in A
};

A* pa = new B();         // OK
pa->f();                 // KO
```



Safe downward cast (1)

- **Downward cast may be dangerous**

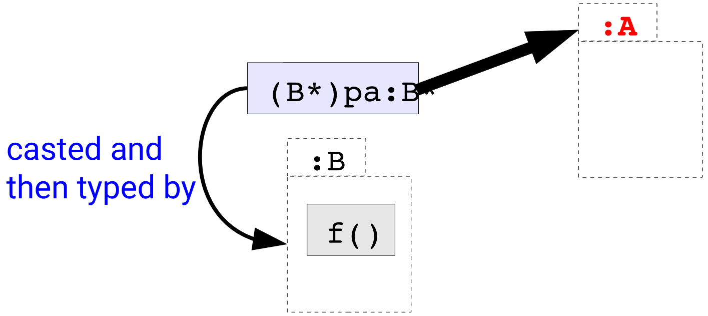
```
class A { ... };
class B : public A {
   void f() { ... }        // f() not defined in A
};

A* pa = new B();           // OK
pa->f();                   // KO
((B*)pa)->f();             // OK
```



Safe downward cast (1)

- **Downward cast may be dangerous**

```
class A { ... };
class B : public A {
   void f() { ... }          // f() not defined in A
};

A* pa = new A();             // OK
pa->f();                     // KO
((B*)pa)->f();               // OK
```

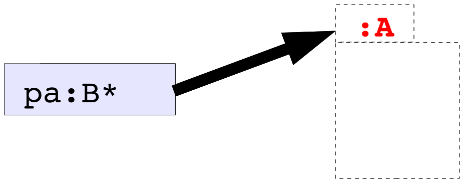


Unsafe !!

Safe downward cast (1)

- Downward cast may be dangerous

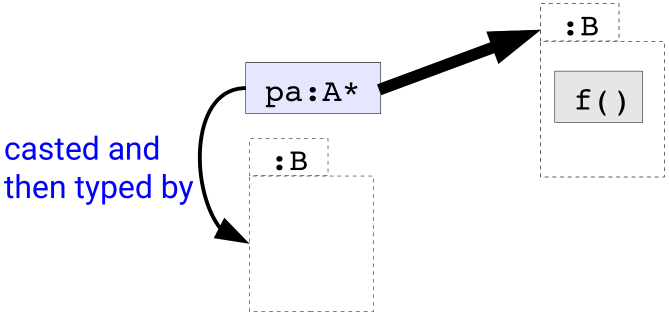
```
class A { ... };
class B : public A {
   void f() { ... }        // f() not defined in A
};

B* pa = new A();           // KO
```



Safe downward cast (1)

- Downward cast may be dangerous

```
class A { ... };
class B : public A {
   void f() { ... }       // f() not defined in A
};

A* pa = new B();          // OK
pa->f();                  // KO
((B*)pa)->f();            // OK
static_cast<B*>(pa)->f();// OK
```


casted and
then typed by

Unsafe !!

Safe downward cast (2)

- Operator `dynamic_cast`

```
B *pb = dynamic_cast<B*>(pa);
if (pb != nullptr)
{
  pb->f();        // OK and safe
}
```

Safe downward cast (2)

- Operator `dynamic_cast`

```
B *pb = dynamic_cast<B*>(pa);
if (pb != nullptr)
  pb->f(); // OK and safe

try {
  B& b = dynamic_cast<B&>(*pa);
  b.f();
} catch(bad_cast) {
  cerr << "bad conversion" << endl;
}
```

Safe downward cast (3)

- Limitation of `dynamic_cast`
  - Work only on classes with virtual functions (polymorphic types)
- Invoking `dynamic_cast` from a constructor or a destructor
  - **`dynamic_cast`** behaves like a virtual function
  - it is statically bound in a constructor or a destructor