A <Basic> C++ Course

7 – Object-oriented programming

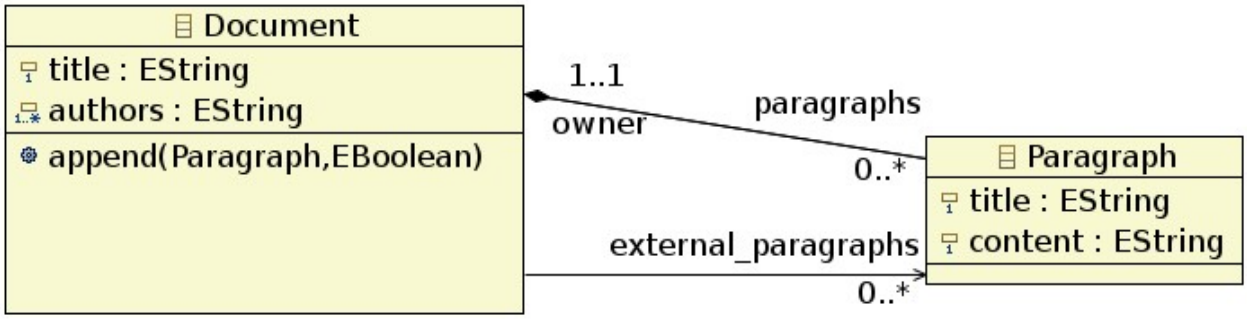
Julien Deantoni

Outline

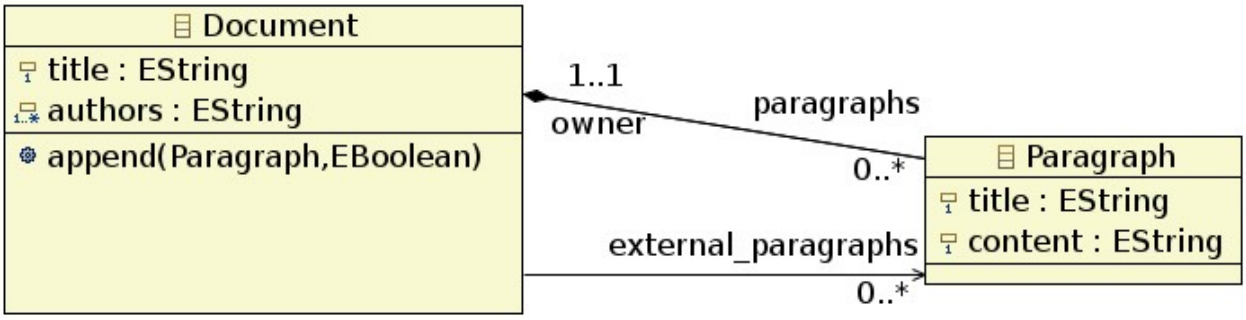
- Introduction to class derivation: variants of class Paragraph
- Dynamic typing and virtual functions:
  - Composing various sorts of paragraphs
  - Another example: the **Expr** class

Document example basic specifications

- Consider (unstructured) text documents. A **`Document`** is composed of:
  1. a title,
  2. A set of authors,
  3. an ordered collection of contained paragraphs,

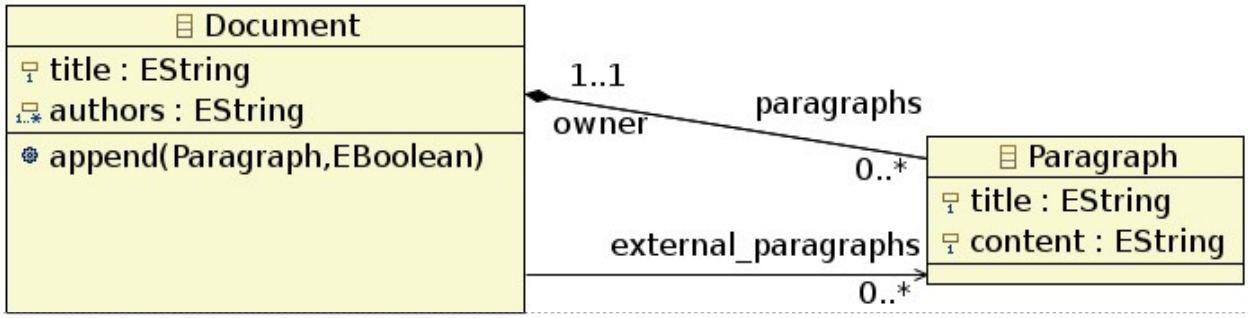


Document example basic specifications vs C++




```
class Document{
private:
  string _title;
  vector<string> _authors;
```

Document example basic specifications vs C++




```
class Document{
private:
  string _title;
  vector<string> _authors;
  vector<Paragraph> _paragraphs;
  vector<Paragraph*> _external_paragraphs;
public:
  /*! constructors */
  Document(string title="default_title",
         vector<string> new_authors= vector<string>(),
          vector<Paragraph> new_paragraphs=vector<Paragraph>(),
           vector<Paragraph*> new_external_paragraphs=vector<Paragraph*>());
  //! Copy constructor
  Document(const Document&);

  void append(Paragraph&, bool);
};
```

Document example
basic specifications vs C++

```
#include "document.h"
Document::Document(string title,
                   vector<string> new_authors,
                   vector<Paragraph> new_paragraphs,
                   vector<Paragraph*> new_external_paragraphs)
{
 _title = title;
 _authors = new_authors;
 _paragraphs = new_paragraphs;
 _external_paragraphs = new_external_paragraphs;
}


void Document::append(Paragraph& p, bool isOwned)
{
  if (isOwned == true)
  {
    _paragraphs.push_back(
  }
  else
  {
    _external_paragraphs.push_back(
  }
}
```

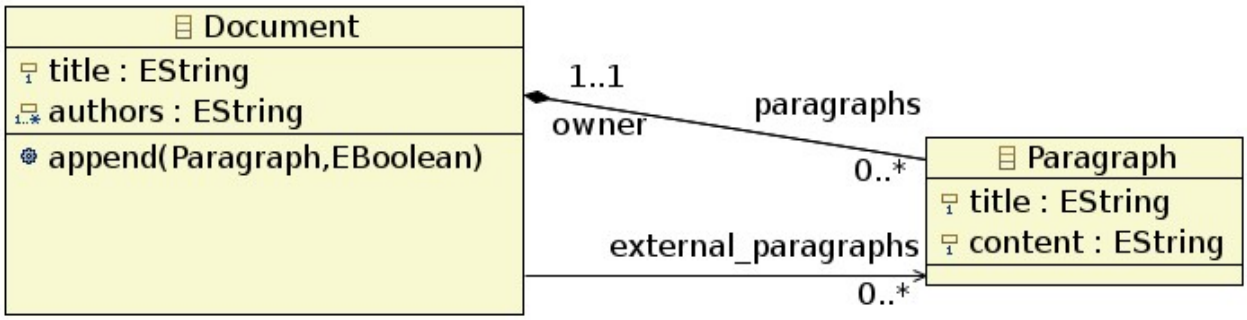
Document example
basic specifications vs C++

```
#include "document.h"
Document::Document(string title,
                   vector<string> new_authors,
                   vector<Paragraph> new_paragraphs,
                   vector<Paragraph*> new_external_paragraphs)
{
 _title = title;
 _authors = new_authors;
 _paragraphs = new_paragraphs;
 _external_paragraphs = new_external_paragraphs;
}


void Document::append(Paragraph& p, bool isOwned)
{
  if (isOwned == true)
  {
    _paragraphs.push_back(p);
  }
  else
  {
    _external_paragraphs.push_back(
  }
}
```

Document example
basic specifications vs C++

```
#include "document.h"
Document::Document(string title,
                   vector<string> new_authors,
                   vector<Paragraph> new_paragraphs,
                   vector<Paragraph*> new_external_paragraphs)
{
 _title = title;
 _authors = new_authors;
 _paragraphs = new_paragraphs;
 _external_paragraphs = new_external_paragraphs;
}


void Document::append(Paragraph& p, bool isOwned)
{
  if (isOwned == true)
  {
    _paragraphs.push_back(p);
  }
  else
  {
    _external_paragraphs.push_back(&p);
  }
}
```

Document example
another implementation




```
class Document{
private:
  string _title;
  vector<string> _authors;
  vector<Paragraph*> _paragraphs;
  vector<Paragraph*> _external_paragraphs;
public:
  ...
```

Document example
another implementation

A new place is reserved in the memory and the corresponding pointer is put in the container

```
#include "document.h"

void Document::append(Paragraph& p, bool owned)
{
  if (owned == true)
  {
     _paragraphs.push_back(new Paragraph(p));
  }
  else
  {
     _external_paragraphs.push_back(&p);
  }
}
```

Document example
another implementation

```
#include "document.h"

void Document::append(Paragra
{
  if (owned == true)
  {
     _paragraphs.push_back(new Paragraph(p));
  }
  else
  {
     _external_paragraphs.push_back(&p);
  }
}
```

A new place is reserved in the memory and the corresponding pointer is put in the container

The previous statement is always true:
For me, when there is a containment, the object life is under the responsibility of the container.
Otherwise it is not.

Document example another implementation

```
void Document::append(Paragr
{
  if (owned == true)
  {
     _paragraphs.push_b
  }
  else
  {
     _external_paragraphs.pus
  }
}

Document::~Document()
{
   for (Paragraph* ptr_p : paragraphs)
   {
      delete ptr_p;
   }
}
```

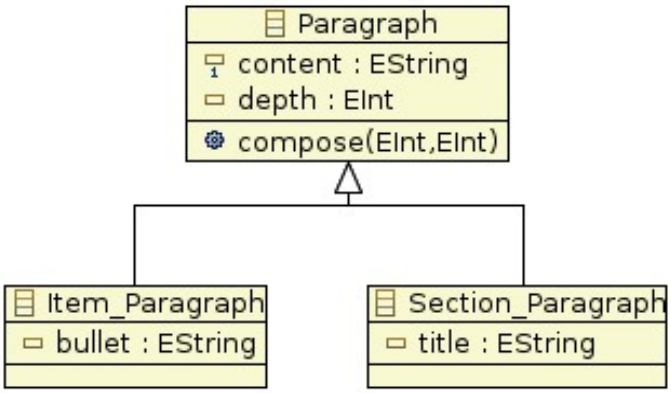
A new place is reserved in the memory and the corresponding pointer is putted in the container

The previous statement is always true:
For me, when there is a containment, the object life is under the responsibility of the container.
Otherwise it is not.

Consequently, the release of the memory is handled in destructor

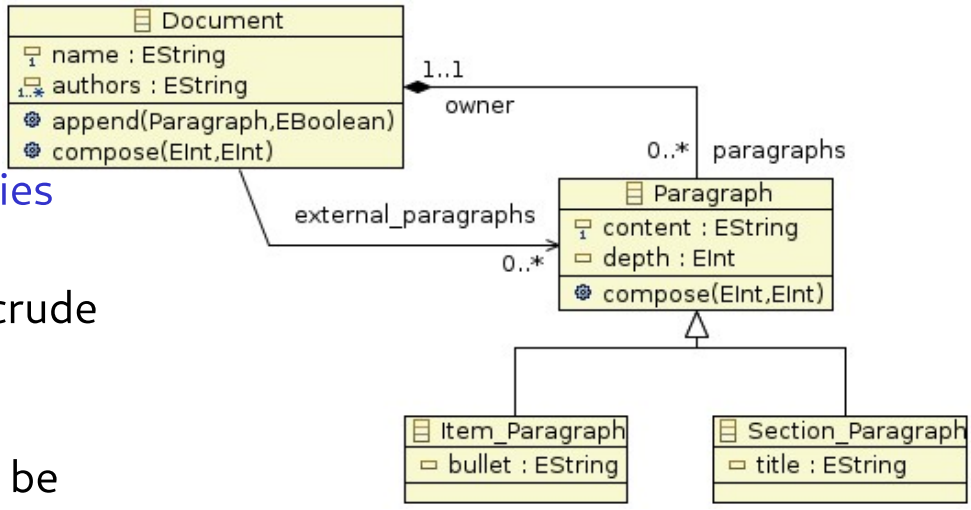
Variants of class Paragraph
Definition of derived classes

- 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



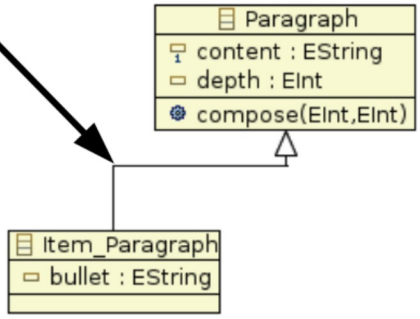
Variants of class Paragraph
Definition of derived classes

- 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



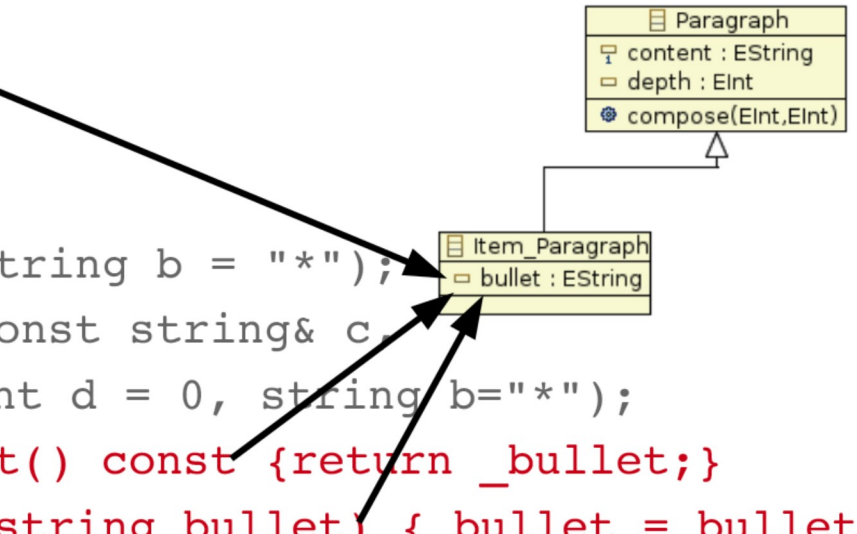
Variants of class Paragraph
Definition of derived classes

```
class Item_Paragraph : public Paragraph
{
private:
  string _bullet;

public:
   Item_Paragraph(string b = "*");
  Item_Paragraph(const string& c,
int d = 0, string b="*");
  string get_bullet() const {return _bullet;}
  void set_bullet(string bullet) {_bullet = bullet;}
   // ...
};
```



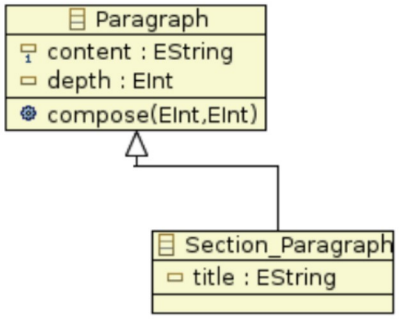
Variants of class Paragraph
Definition of derived classes

```
class Item_Paragraph : public Paragraph
{
private:
  string _bullet;

public:
  Item_Paragraph(string b = "*");
  Item_Paragraph(const string& c,
                 int d = 0, string b="*");
  string get_bullet() const {return _bullet;}
  void set_bullet(string bullet) {_bullet = bullet;}
  // ...
};
```

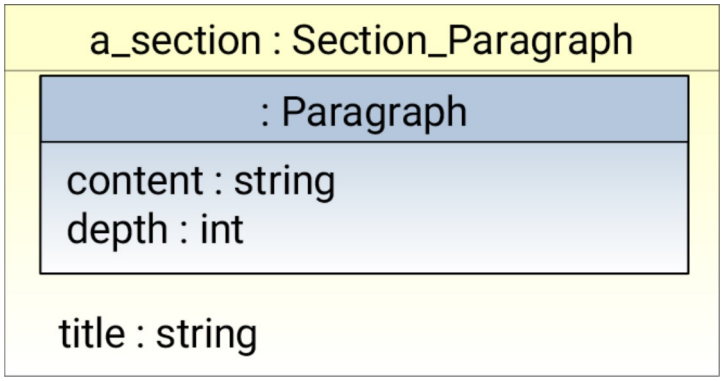


Variants of class Paragraph
Definition of derived classes

```
class Section_Paragraph : public Paragraph
{

private:
   string _title;

public:
   Item_Paragraph(string t = "default_title");
   Item_Paragraph(const string& c,
                  int d = 0, string t="default_title");
   string get_title() const {return _title;}
   void set_title(string title) {_title = title;}
// ...
};
```



- A `Section_Paragraph` **is a** `Paragraph`
- A `Section_Paragraph` **inherits** `Paragraph` properties
  - Its underlying C structure contains the `Paragraph` one plus all data members specific to `Section_Paragraph`



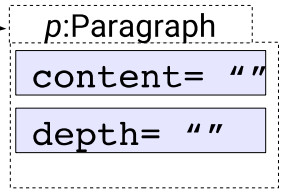
- A `Section_Paragraph` **is a** `Paragraph`
- A `Section_Paragraph` **inherits** `Paragraph` properties
  - Its underlying C structure contains the `Paragraph` one plus all data members specific to `Section_Paragraph`
  - One can apply to a `Section_Paragraph` object all public `Paragraph` member-functions
  - One may substitute to any instance of `Paragraph` an instance of `Section_Paragraph` (Substitutability principle) (*semantics known as sub-typing*)

Variants of class Paragraph
Definition of derived classes

- 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

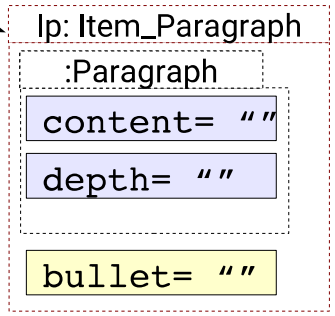
Definition of derived classes : What happens in the memory ?

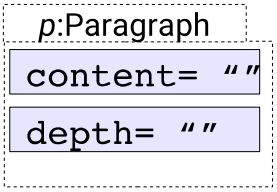
```
main(){
 Paragraph p;
 Item_Paragraph ip;
}
```



Definition of derived classes : What happens in the memory ?

```
main(){
 Paragraph p;
 Item_Paragraph ip;
}
```



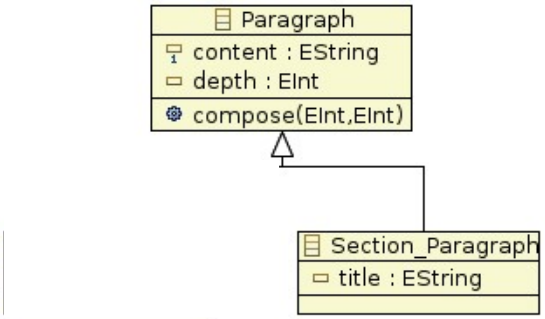


Variants of class Paragraph Protected members

- Accessing inherited member in a derived class:

```
Void Section_Paragraph::a_function(){
        _content = “blabla bla”;
}
```

  - ERROR: **`_content`** is private in **`Paragraph`**



Variants of class `Paragraph`
Protected members

- Accessing inherited member in a derived class:

```
Void Section_Paragraph::a_function(){
      _content = "blabla bla";
}
```

  - ERROR: **`_content`** is private in **`Paragraph`**

- Protected members:

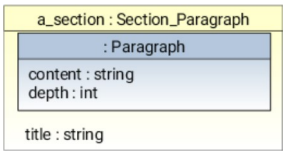
```
class Paragraph
{
protected:
  string _content;

public:
      // ...
};
```

  - A protected member is public to its class and its derivatives
  - Protected data members are as vulnerable as public ones if the class is not *final*

Variants of class Paragraph
Construction of derived classes

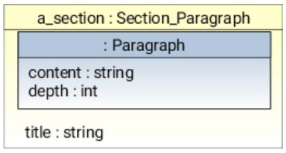
- Constructors of `Item_Paragraph`

In opposition with other members,
constructors are never inherited




```
Item_Paragraph::Item_Paragraph(const string& c="",
                               int d=0, string b = "*")
  : Paragraph(c,d), _bullet(b)
{}

Item_Paragraph::Item_Paragraph(string b = "*")
  : _bullet(b)
{}
```

Variants of class Paragraph
Construction of derived classes

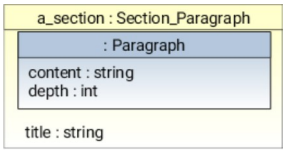


- Constructors of `Item_Paragraph`

Equivalent to `super()` in java (for single inheritance)

```
Item_Paragraph::Item_Paragraph(const string& c="",
                               int d=0, string b = "*")
  : Paragraph(c,d), _bullet(b)
{}

Item_Paragraph::Item_Paragraph(string b = "*")
  : _bullet(b)
{}
```

Variants of class Paragraph
Construction of derived classes

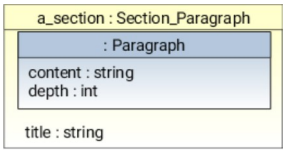


- Constructors of `Item_Paragraph`

```
Item_Paragraph::Item_Paragraph(const string& c="",
                               int d=0, string b = "*")
  : Paragraph(c,d), _bullet(b)
{}

Item_Paragraph::Item_Paragraph(string b = "*")
  : _bullet(b)
{}
```

Ok only if base class members and default constructors are public

Variants of class Paragraph
Construction of derived classes

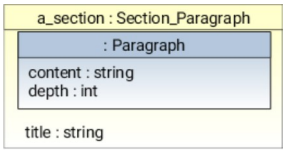


- Constructors of `Item_Paragraph`

```
Item_Paragraph::Item_Paragraph(const string& c="",
                               int d=0, string b = "*")
  : Paragraph(c,d), _bullet(b)
{}

Item_Paragraph::Item_Paragraph(string b = "*")
  : _bullet(b)
{}

Item_Paragraph::Item_Paragraph(string b = "*")
  : Paragraph(), _bullet(b)
{
}
```

equivalent

Variants of class Paragraph
Construction of derived classes



- Construction of Section_Paragraph

```
Section_Paragraph::Section_Paragraph( const string& c="",
                                      int d=0, string t="")
 : Paragraph(c,d), _title(t)
{}

Section_Paragraph::Section_Paragraph(string t="")
  : title(t)
{}
```

Variants of class Paragraph
Default construction of derived classes

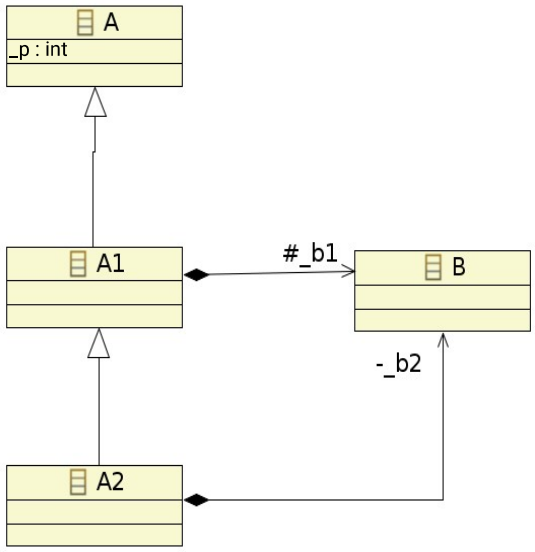
- If the derived class has no constructor, its members and base class are constructed by default construction
  - Everything is as if C++ creates a default default constructor
  - A class is constructible by default if
    - either it has a default constructor
    - or it has *no constructor at all*, and its members and immediate base classes are constructible by default
- If a derived class has no destructor, default destruction applies
  - Everything is as if C++ creates a default destructor

Variants of class Paragraph
Construction order of derived classes

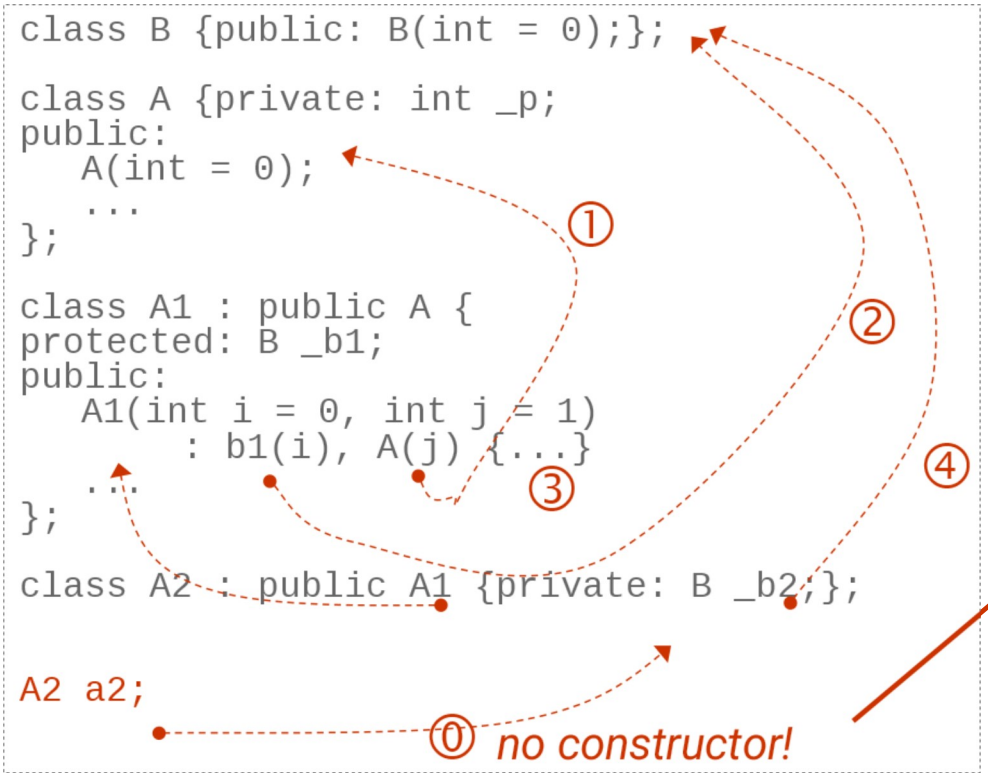
- Construction order
  1. the base class(es)
  2. the data members specific to derived class
  3. the body of the derived class constructor itself
- Destruction order: reverse of construction
- C++ applies these rules recursively
- A derived class constructor is entirely responsible for the construction of
  - its base class(es)
  - its specific members
  - the derived class itself (constructor body)

Variants of class Paragraph
Default construction of derived classes (2)

```
class B {public: B(int = 0);};

class A {private: int _p;
public:
   A(int = 0);
   // ...
};

class A1 : public A {
protected: B _b1;
public:
   A1(int i = 0, int j = 1)
        : _b1(i), A(j) {...}
   // ...
};

class A2 : public A1 {private: B _b2;};


A1 a1(2, 3);
```



Variants of class Paragraph
Default construction of derived classes (3)




```
A2:A2()
  : A1(),_b2()
{}
```

default default constructor

Variants of class Paragraph
Using publicly derived classes

- Standard conversions in case of public derivation
  - derived class instance → base class instance
  - pointer to derived class → pointer to base class
  - reference to derived class → reference to base class

```
//...
Item_Paragraph ip1;
Section_Paragraph sp2;
Paragraph p = ip1;       // initialization of Paragraph
// ...
cout << ip1 + sp2;       // + and << for Paragraph
```

Variants of class Paragraph
Using publicly derived classes

- Standard conversions in case of public derivation
  - derived class instance → base class instance
  - pointer to derived class → pointer to base class
  - reference to derived class → reference to base class

```
//...
Item_Paragraph ip1;
Section_Paragraph sp2;
Paragraph p = ip1;       // initialization of Paragraph
// ...
cout << ip1 + sp2;       // + and << for Paragraph
```

We do not want to *"print"* an `Item_Paragraph` and a `Section_Paragraph` in the same way

Variants of class Paragraph
Back to conversion of derived classes

- What if we redefine a print( ) member-function in variants of `Paragraph`?
  - Indeed each sort of paragraph has a different page layout
  - Note that the redefinition should have the same signature in the base and derived classes

```
Paragraph p;
Item_Paragraph ip;

ip.print();                 // Item_Paragraph::print()
p = ip;                     // Paragraph::operator=
p.print();                  // Paragraph::print()

Paragraph *ptr_p = &ip;     // standard conversion
ptr_p->print();

void f(Paragraph& p) {
   p.print();
}
```

Variants of class Paragraph
Back to conversion of derived classes

- What if we redefine a print( ) member-function in variants of `Paragraph`?
  - Indeed each sort of paragraph has a different page layout
  - Note that the redefinition should have the same signature in the base and derived classes

```
Paragraph p;
Item_Paragraph ip;

ip.print();                  // Item_Paragraph::print()
p = ip;                      // Paragraph::operator=
p.print();                   // Paragraph::print()

Paragraph *ptr_p = &ip;      // standard conversion
ptr_p->print();              // Paragraph::print()

void f(Paragraph& p) {
   p.print();
}
```

Variants of class Paragraph
Back to conversion of derived classes

- What if we redefine a print( ) member-function in variants of `Paragraph`?
  - Indeed each sort of paragraph has a different page layout
  - Note that the redefinition should have the same signature in the base and derived classes

```
Paragraph p;
Item_Paragraph ip;

ip.print();                  // Item_Paragraph::print()
p = ip;                      // Paragraph::operator=
p.print();                   // Paragraph::print()

Paragraph *ptr_p = &ip;      // standard conversion
ptr_p->print();              // Paragraph::print()

void f(Paragraph& p) {
   p.print();                // Paragraph::print()
}
```

Variants of class Paragraph
Back to conversion of derived classes

- What if we redefine a print( ) member-function in variants of `Paragraph`?
  - Indeed each sort of paragraph has a different page layout
  - Note that the redefinition should have the same signature in the base and derived classes

```
Paragraph p;
Item_Paragraph ip;

ip.print();                    // Item_Paragraph::print()
p = ip;                        // Paragraph::operator=
p.print();                     // Paragraph::print()

Paragraph *ptr_p = &ip;        // standard conversion
ptr_p->print();                // Paragraph::print()

void f(Paragraph& p) {
   p.print();                  // Paragraph::print()
}
```

static vs dynamic type !

Variants of class Paragraph
Virtual Functions

```
class Paragraph {
   // ...
   virtual print() const;
   ...
};

class Item_Paragraph : public Paragraph {
   // ...
  virtual print() const;
   ...
};
```

A virtual function is binded at run-time
(so called late-binding or dynamic typing)

→ each time we invoke a virtual member-function by accessing the object through a pointer or a reference, the dynamic type of the object determine (at run-time) which version of the member-function is to be used.

Variants of class Paragraph
Virtual functions

```
class Paragraph {
   // ...
   virtual print() const;
   ...
};


class Item_Paragraph : public Paragraph {
   // ...
  virtual print() const;
   ...
};
```

- One could says : every method must be virtual !
  - It could but it may not… (due to performance issue… on specific cases)
- Having virtual functions indicate that a class is meant to act as an interface to derived classes (Bjarne Stroustrup)

Variants of class Paragraph
Virtual functions

```
class Paragraph {
   // ...
   virtual print() const;
   ...
};

class Item_Paragraph : public Paragraph {
   // ...
  virtual print() const;
   ...
};
```

- One could says : every method must be virtual !
  - It could but it may not… (due to performance issue… on specific cases)
- Having virtual functions indicate that a class is meant to act as an interface to derived classes (Bjarne Stroustrup)

If the **destructor** is **not** declared **virtual** then **only the ~BaseClass() destructor may be called** leaving any allocated memory from the DerivedClass to persist and **leak**

Variants of class Paragraph
Virtual functions

```
class Paragraph {
   // ...
   virtual print() const;
   ...
};

class Item_Paragraph : public Paragraph {
   // ...
  virtual print() const;
   ...
};
```

- One could says : every method must be virtual !
  - It could but it may not… (due to performance issue… on specific cases)
- Having virtual functions indicate that a class is meant to act as an interface to derived classes (Bjarne Stroustrup)

virtual ~Paragraph(){};
virtual ~Item_Paragraph(){};
→ At least !!!

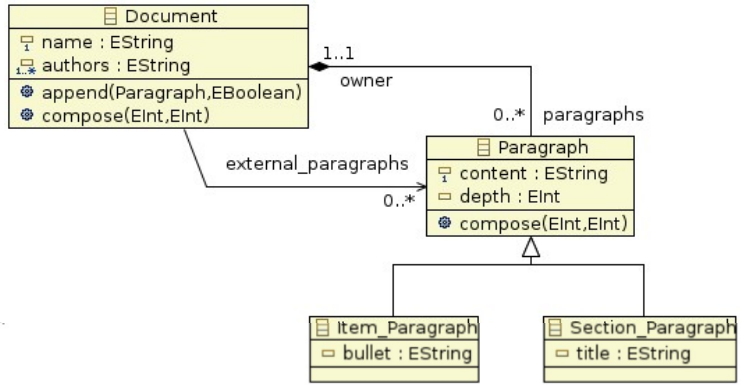
Variants of class Paragraph
Virtual functions

- What if we redefine a print( ) member-function in variants of `Paragraph`?
  - Indeed each sort of paragraph has a different page layout
  - Note that the redefinition should have the same signature in the base and derived classes

With *print* as virtual

```
Paragraph p;
Item_Paragraph ip;

ip.print();                    // Item_Paragraph::print()
p = ip;                        // Paragraph::operator=
p.print();                     // Paragraph::print()

Paragraph *ptr_p = &ip;        // standard conversion
ptr_p->print();                // Item_Paragraph::print()

void f(Paragraph& p) {
   p.print();                  // ??? the dynamic type of p
}
```

Document example
basic specifications vs C++



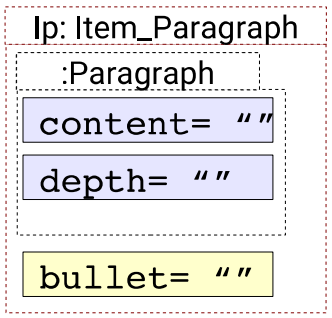
Classical problem

```
class Document{
private:
  string _title;
  vector<string> _authors;
  vector<Paragraph> _paragraphs;
  vector<Paragraph*> _external_paragraphs;
public:
  /*! constructors */
  Document(string title="default_title",
         vector<string> new_authors= vector<string>(),
          vector<Paragraph> new_paragraphs=vector<Paragraph>(),
           vector<Paragraph*> new_external_paragraphs=vector<Paragraph*>());

  void append(Paragraph&, bool);
};
```

Document example troncature problem

Dynamic type of p: `Item_Paragraph`

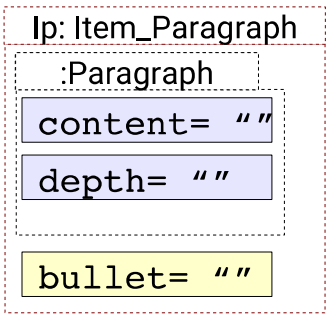
```
void Document::append(Paragraph& p, bool owned)
{
  if (owned == true)
  {
    _paragraphs.push_back(p);
  }
  else
  {
    _external_paragraphs.push_back(&p);
  }
}
```



Document example troncature problem

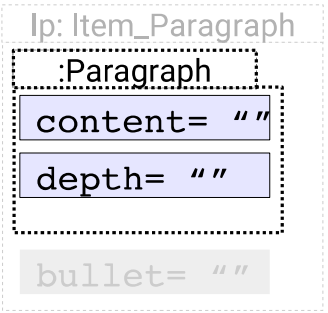
Dynamic type of p: `Item_Paragraph`

```
void Document::append(Paragraph& p, bool owned)
{
  if (owned == true)
  {
    _paragraphs.push_back((Paragraph)p);
  }
  else
  {
    _external_paragraphs.push_back(&p);
  }
}
```



Document example troncature problem

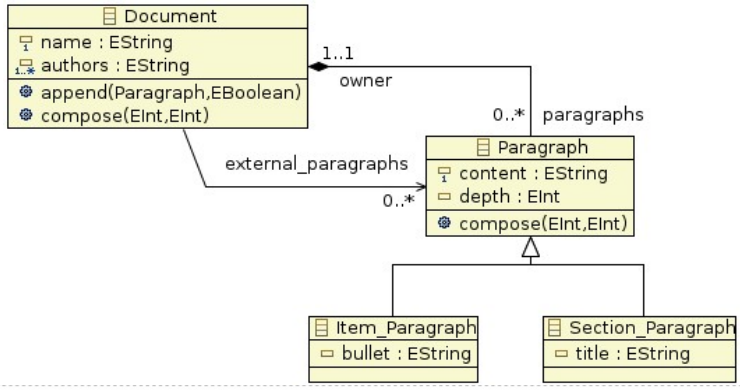
```
void Document::append(Paragraph& p, bool owned)
{
  if (owned == true)
  {
    _paragraphs.push_back(Paragraph(p));
  }
  else
  {
    _external_paragraphs.push_back(&p);
  }
}
```



The dynamic type of **p** is lost

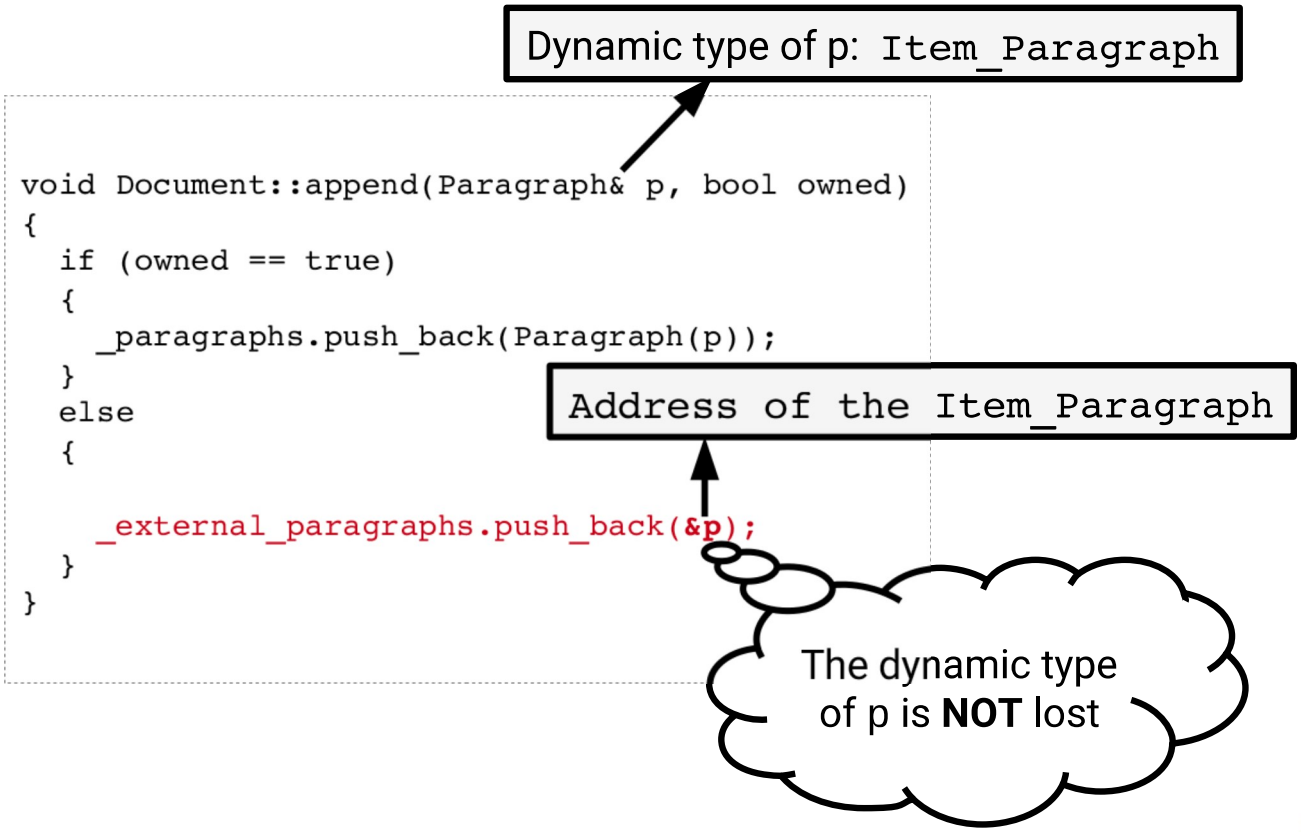
_bullet is truncated

Document example preserving dynamic type

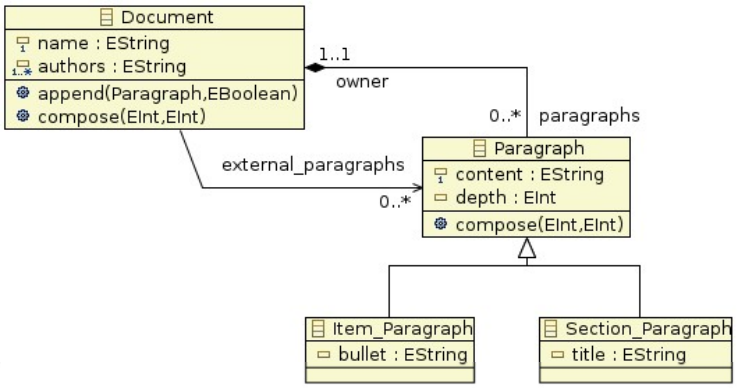



```
class Document{
private:
  string _title;
  vector<string> _authors;
  vector<Paragraph> _paragraphs;
  vector<Paragraph*> _external_paragraphs;
public:
  ...
```

Document example preserving dynamic type



Document example preserving dynamic type



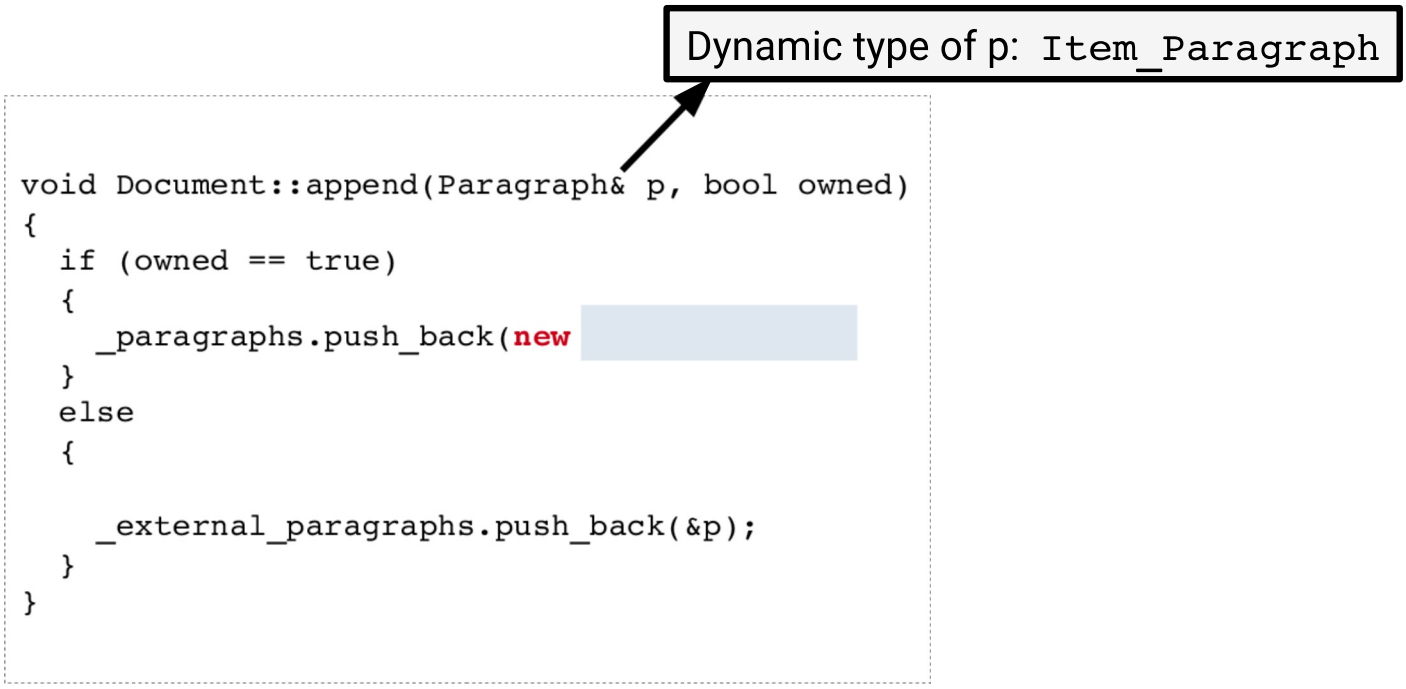

```
class Document{
private:
  string _title;
  vector<string> _authors;
  vector<Paragraph*> _paragraphs;
  vector<Paragraph*> _external_paragraphs;
public:
  /*! constructors */
  Document(string title="default_title",
        vector<string> new_authors= vector<string>(),
         vector<Paragraph*> new_paragraphs=vector<Paragraph*>(),
          vector<Paragraph*> new_external_paragraphs=vector<Paragraph*>());

  void append(Paragraph&, bool);
};
```

Document example preserving dynamic type

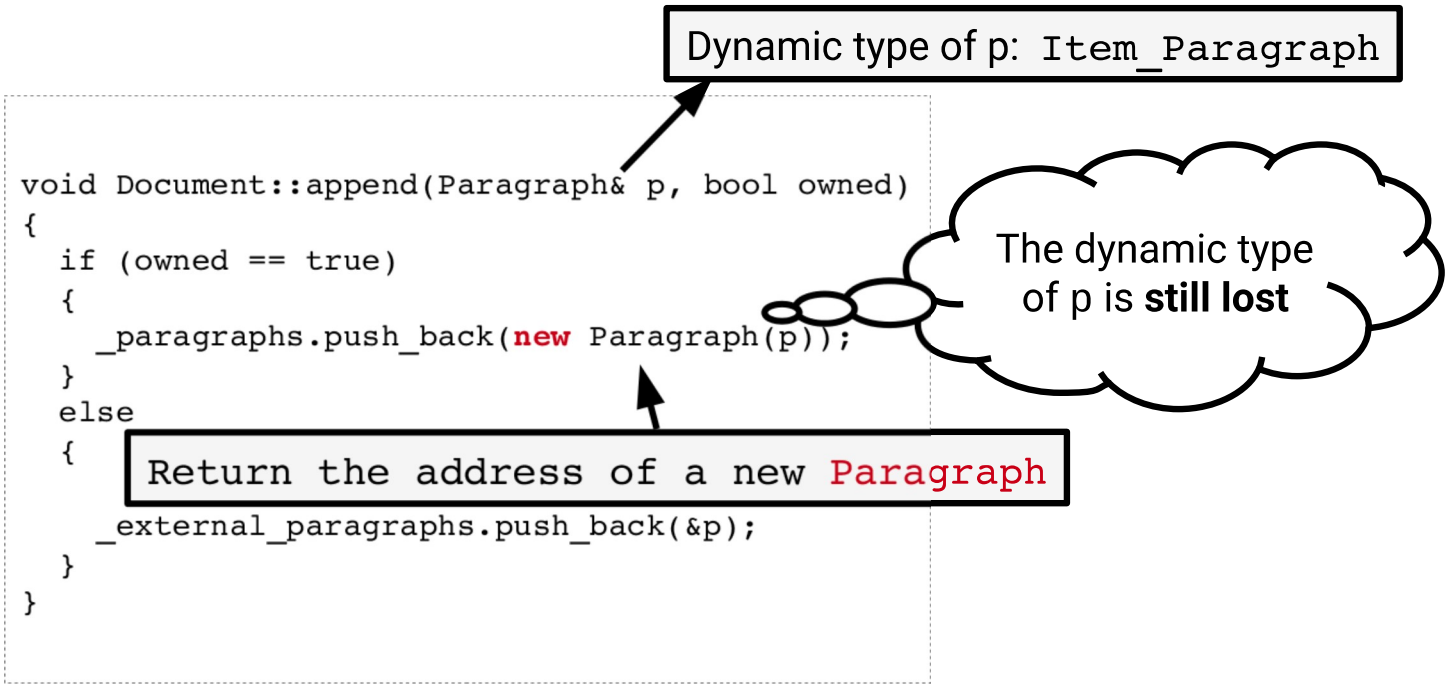
Dynamic type of p: `Item_Paragraph`

```
void Document::append(Paragraph& p, bool owned)
{
  if (owned == true)
  {
    _paragraphs.push_back(
  }
  else
  {

    _external_paragraphs.push_back(&p);
  }
}
```

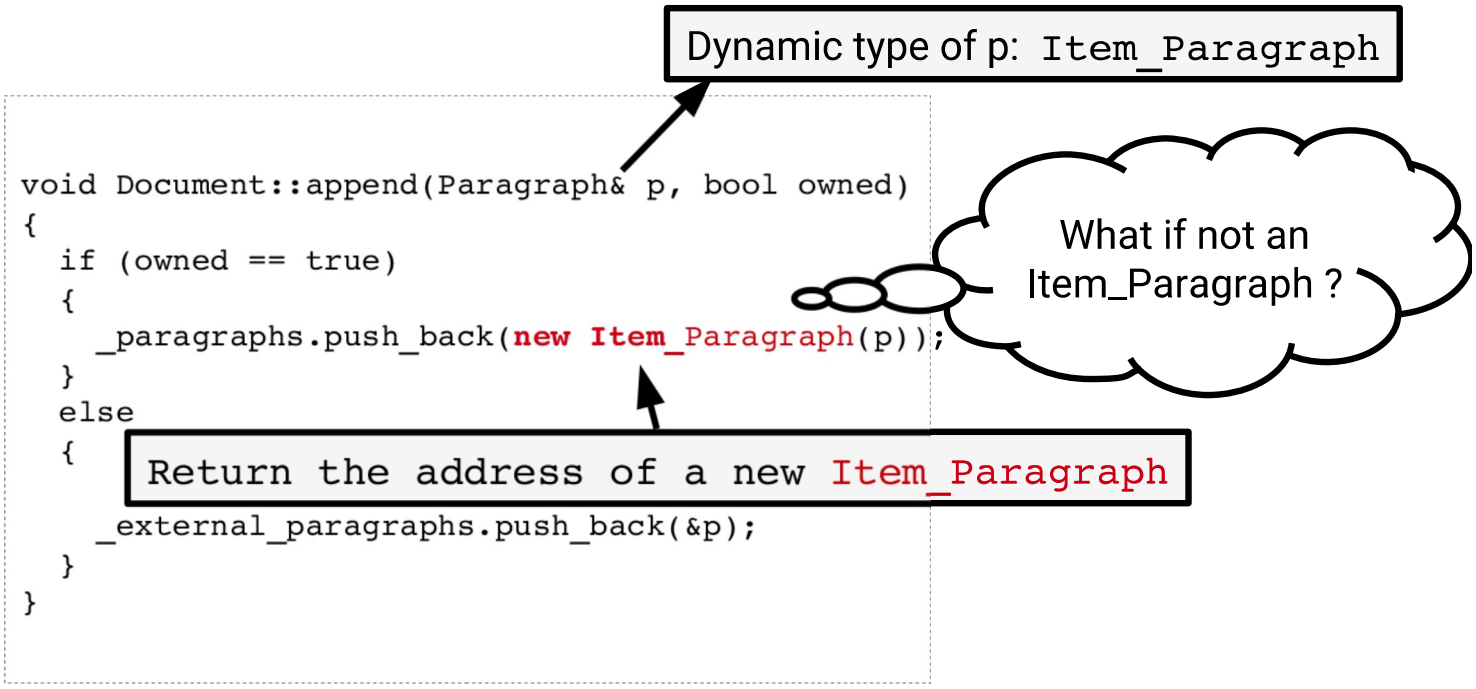
Document example preserving dynamic type



Document example preserving dynamic type



Document example preserving dynamic type




```
void Document::append(Paragraph& p, bool owned)
{
  if (owned == true)
  {
    _paragraphs.push_back(new Item_Paragraph(p));
  }
  else
  {
    _external_paragraphs.push_back(&p);
  }
}
```

Document example
preserving dynamic type : polymorphic copy (clone)

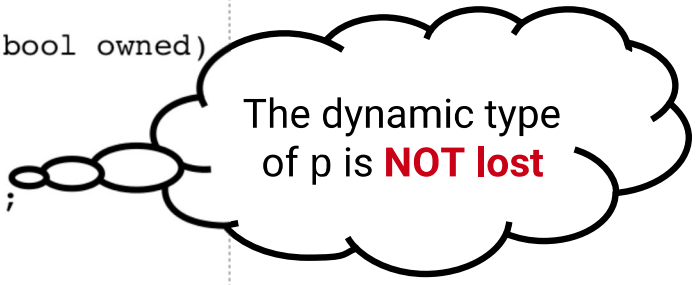
```
class Paragraph {
public:
    virtual Paragraph* clone() const {
        return new Paragraph(*this);
    }
};

class Item_Paragraph : public Paragraph {
public:
    virtual Paragraph* clone() const {
        return new Item_Paragraph(*this);
    }
};
```

A virtual function is binded at run-time
(so called late-binding or dynamic typing)

Document example
preserving dynamic type : polymorphic copy (clone)

```
class Paragraph {
public:
    virtual Paragraph *clone() const {
        return new Paragraph(*this);
    }
};

class Item_Paragraph : public Paragraph {
public:
    virtual Paragraph *clone() const {
        return new Item_Paragraph(*this);
    }
};
```

```
void Document::append(Paragraph& p, bool owned)
{
  if (owned == true)
  {
    _paragraphs.push_back(p.clone());
  }
  else
  {
    _external_paragraphs.push_back(&p);
  }
}
```



Return the address of a new Paragraph or a derived class

Variants of class Paragraph
Virtual functions and operator overload

- Only member-function can be virtual
  - How can we overload the “printing” function (operator<<)

Variants of class Paragraph
Friendship and derivation

- A derived class *does not* inherit its base class friends as friend
  - The friends of a derived class are not implicitly friends of its base class
  - Nevertheless, it is possible to use base class friends with an object of a publicly derived class as parameter

```
class A {
    friend void f(A);
    // ...
};
    class B : public A {...};
    B b;
    f(b);      // OK: equivalent to f((A)b)
```

Variants of class Paragraph
Virtual functions and operator overload

- Only member-function can be virtual
  - How can we overload the “printing” function (operator<<)

Variants of class Paragraph
Virtual functions and operator overload

- Only member-function can be virtual
  - How can we overload the “printing” function (operator<<)

```
class Paragraph {
   // ...
   virtual ostream& print(ostream&) const;

   std::ostream& operator<<(std::ostream& os, Paragraph p)
   {
      return p.print(os);
   }
};

class Item_Paragraph : public Paragraph {
   // ...
  virtual ostream& print(ostream&) const;
   ...
};
```

Est-ce correct ?

Variants of class Paragraph
Virtual functions and operator overload

- Only member-function can be virtual
  - How can we overload the “printing” function (operator<<)

```
class Paragraph {
   // ...
   virtual ostream& print(ostream&) const;

   std::ostream& operator<<(std::ostream& os, const Paragraph& p)
   {
      return p.print(os);
   }
};


class Item_Paragraph : public Paragraph {
   // ...
  virtual ostream& print(ostream&) const;
   ...
};
```

Variants of class Paragraph
Using publicly derived classes

- By using virtual functions

```
//...
Item_Paragraph ip1;
vector<Paragraph*> vp;
vp.push_back(ip1.clone());
// ...
cout << vp.at(0);          // operator<< from Paragraph
                           call print from Item_Paragraph
delete vp.at(0);
```

Variants of class Paragraph
Using publicly derived classes

- By using virtual functions

```
//...
Item_Paragraph ip1;
vector<Paragraph*> vp;
vp.push_back(ip1.clone());
// ...
cout << vp.at(0);          // operator<< from Paragraph
                           call print from Item_Paragraph
delete vp.at(0);
```

If the **destructor** is **not** declared **virtual** then **only the ~BaseClass() destructor is called** leaving any allocated memory from the DerivedClass to persist and **leak**