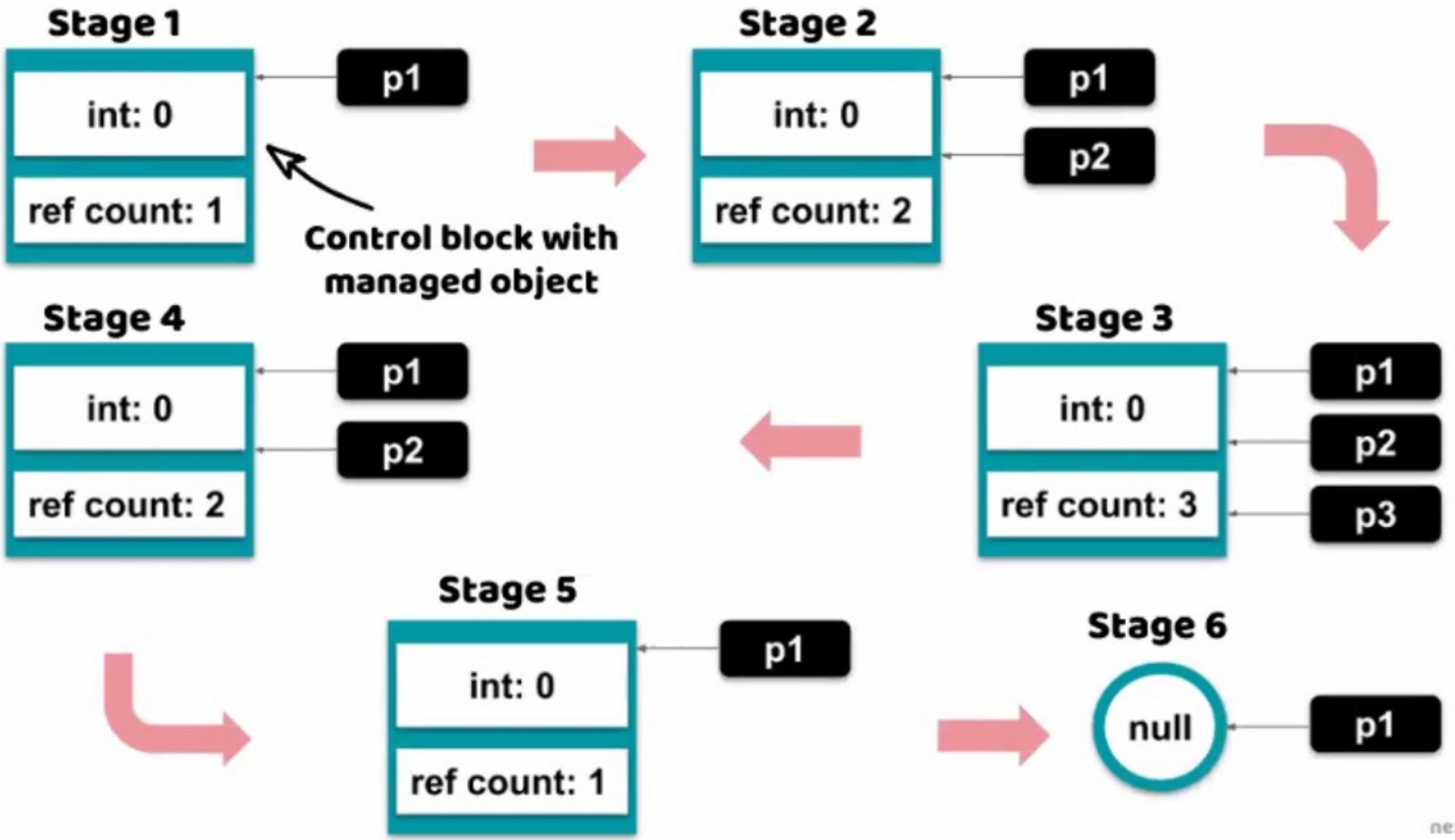
std::shared_ptr



```
class C { // tracciamento costruttori, distruttore, assegnazione
public:
  int x;
  C(int y=0): x(y) {std::cout << "C(" << x <<") \n";}
  ~C() {std::cout << "~C() \n";}
  C(const C& x) {std::cout << "Cc \n";}
  C& operator=(const C& x) {std::cout << "C= \n"; return *this;}
};
```

```
#include  <memory> // da includere per usare std::shared_ptr
int main() {
  std::shared_ptr<C> p1 = std::make_shared<C>(5); // C(5)
  std::cout << "p1->x = " << p1->x << std::endl;  // 5
  std::cout << "p1 Reference count = " << p1.use_count() << std::endl; // 1
```

```cpp
#include  <memory> // da includere per usare std::shared_ptr
int main() {
  std::shared_ptr<C> p1 = std::make_shared<C>(5); // C(5)
  std::cout << "p1->x = " << p1->x << std::endl;  // 5
  std::cout << "p1 Reference count = " << p1.use_count() << std::endl; // 1

  std::shared_ptr<C> p2(p1); std::shared_ptr<C> p3(p2);
  std::cout << "p1 Reference count = " << p1.use_count() << std::endl; // 3
  std::cout << "p2 Reference count = " << p2.use_count() << std::endl; // 3
  std::cout << "p3 Reference count = " << p3.use_count() << std::endl; // 3
  if (p1 == p3) std::cout << "p1 == p3\n"; // p1 == p2

  std::cout<<"Reset p1 "<<std::endl; p1.reset();
  std::cout << "p1 Reference Count = " << p1.use_count() << std::endl; // 0
  std::cout << "p2 Reference Count = " << p2.use_count() << std::endl; // 2
  std::cout << "p3 Reference Count = " << p2.use_count() << std::endl; // 2

  p1.reset(new C(7)); // C(7)
  std::cout << "p1 Reference Count = " << p1.use_count() << std::endl; // 1

  p1 = nullptr; // ~C()
  std::cout << "p1 Reference Count = " << p1.use_count() << std::endl; // 0
  if (!p1) std::cout << "p1 is NULL" << std::endl; // p1 is NULL

  p2 = std::make_shared<C>(2); // C(2)
  std::cout << "p1 Reference Count = " << p1.use_count() << std::endl; // 0
  std::cout << "p2 Reference Count = " << p2.use_count() << std::endl; // 1
  std::cout << "p3 Reference Count = " << p3.use_count() << std::endl; // 1

  std::shared_ptr<C> p4; p4=p3;
  std::cout << "p3 Reference Count = " << p3.use_count() << std::endl; // 2
  std::cout << "p4 Reference Count = " << p4.use_count() << std::endl; // 2
  // ~C() ~C()
}
```

shared_ptr in bolletta

```
class bolletta {
public:
  // parte pubblica non cambia
private:
  class nodo { // definizione di nodo
    public:
    nodo();
    nodo(const telefonata&,const std::shared_ptr<nodo>&);
    telefonata info;
    std::shared_ptr<nodo> next; // smart ptr next
  }; // fine classe nodo

  std::shared_ptr<nodo> first; // smart ptr first
};
```


```
// Metodi di bolletta
// Le definizioni sono semplici

void bolletta::Aggiungi_Telefonata(const telefonata& t){
  first = make_shared<nodo>(t,first);
}

bool bolletta::Vuota() const {
  return first == nullptr;
}
```

```
telefonata bolletta::Estrai_Una() {
  if(first==nullptr) throw EmptyList();
  telefonata aux = first->info;
  first = first->next; // assegnazione di shared_ptr
  return aux;
  // ATTENZIONE: nessuna distruzione
  // esplicita del primo nodo
}
```

```
ostream& operator<<(ostream& os, const bolletta& b) {
  if(b.Vuota()) os << "BOLLETTA VUOTA" << endl;
  else {
    os << "TELEFONATE IN BOLLETTA" << endl;
    shared_ptr<nodo> p = b.first; // per amicizia
    int i = 1;
    while (p!=0) {
      os << i++ << ") " << p->info << endl;
      p = p->next;
    }
  }
  return os;
}
```

std::unique_ptr

Defined in header <memory>

```
template<
    class T,
    class Deleter = std::default_delete<T>
> class unique_ptr;
```

(1) (since C++11)

`std::unique_ptr` is a smart pointer that owns and manages another object through a pointer and disposes of that object when the `unique_ptr` goes out of scope.

The object is disposed of, using the associated deleter when either of the following happens:

- the managing `unique_ptr` object is destroyed
- the managing `unique_ptr` object is assigned another pointer via `operator=` or `reset()`.

std::weak_ptr

Defined in header <memory>

```
template< class T > class weak_ptr;
```

(since C++11)

Scrivere un programma consistente di esattamente tre classi `A`, `B` e `C` e della sola funzione `main()` che soddisfi le seguenti condizioni:

1. la classe `A` è definita come:

   ```
   class A { public: virtual ~A(){} };
   ```

2. le classi `B` e `C` devono essere definite per ereditarietà e non contengono alcun membro

3. la funzione `main()` definisce le tre variabili:

   ```
   A* pa = new A; B* pb = new B; C* pc = new C;
   ```

   e nessuna altra variabile (di alcun tipo)

4. la funzione `main()` può **utilizzare solamente** espressioni di tipo `A*`, `B*` e `C*`, **non può** sollevare eccezioni mediante una `throw` e **non può** invocare l'operatore `new`

5. il programma deve compilare correttamente

6. l'esecuzione di `main()` **deve provocare un errore run-time**.

Scrivere un programma consistente di esattamente tre classi `A`, `B` e `C` e della sola funzione `main()` che soddisfi le seguenti condizioni:

1. la classe `A` è definita come:

   ```
   class A { public: virtual ~A(){} };
   ```

2. le classi `B` e `C` devono essere definite per ereditarietà e non contengono alcun membro
3. la funzione `main()` definisce le tre variabili:

   `A* pa = new A; B* pb = new B; C* pc = new C;`

   e nessuna altra variabile (di alcun tipo)
4. la funzione `main()` può **utilizzare solamente** espressioni di tipo `A*`, `B*` e `C*`, **non può** sollevare eccezioni mediante una `throw` e **non può** invocare l'operatore `new`
5. il programma deve compilare correttamente
6. l'esecuzione di `main()` **deve provocare un errore run-time**.

```
class B: public A {};
class C: public A {};
int main() {/* ...*/ dynamic_cast<C&>(*pb);}
```

Soluzione

Dereferencing a NULL pointer is undefined behavior.

In fact the standard calls this exact situation out in a note (8.3.2/4 "References"):

> Note: in particular, a null reference cannot exist in a well-defined program, because the only way to create such a reference would be to bind it to the "object" obtained by dereferencing a null pointer, which causes undefined behavior.

Ognuno dei seguenti frammenti è il codice di uno o piú metodi pubblici di una qualche classe C. La loro compilazione provoca errori?

`C f(C& x) {return x;}`	OK/NC?
`C& g() const {return *this;}`	OK/NC?
`C h() const {return *this;}`	OK/NC?
`C* m() {return this;}`	OK/NC?
`C* n() const {return this;}`	OK/NC?
`void p() {}` `void q() const {p();}`	OK/NC?
`void p() {}` `static void r(C *const x) {x->p();}`	OK/NC?
`void s(C *const x) const {*this = *x;}`	OK/NC?
`static C& t() {return C();}`	OK/NC?
`static C *const u(C& x) {return &x;}`	OK/NC?

```
class Z {
 public: Z(int x) {}
};
```

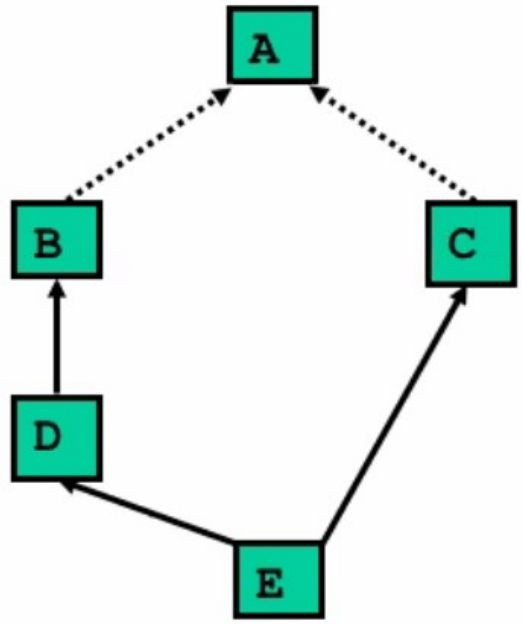
```
class A {
 public:
  void f(int) {cout << "A::f(int) "; f(true);}
  virtual void f(bool) {cout <<"A::f(bool) ";}
  virtual A* f(Z) {cout <<"A::f(Z) "; f(2); return this;
  A() {cout <<"A() "; }
};
```

```
class B: virtual public A {
 public:
  void f(const bool&){cout<< "B::f(const bool&) ";}
  void f(const int&){cout<< "B::f(const int&) ";}
  virtual B* f(Z) {cout <<"B::f(Z) "; return this;}
  virtual ~B() {cout << "~B ";}
  B() {cout <<"B() "; }
};
```

```
class C: virtual public A {
 public:
  C* f(Z){cout <<"C::f(Z) "; return this;}
  C() {cout <<"C() "; }
};
```

```
class D: public B {
public:
  virtual void f(bool) const {cout <<"D::f(bool) ";}
  B* f(Z) {cout << "D::f(Z) "; return this;}
  ~D() {cout <<"~D ";}
  D() {cout <<"D() "; }
};
```

```
class E: public D, public C {
 public:
  void f(bool){cout<< "E::f(bool) ";}
  E* f(Z){cout <<"E::f(Z) "; return this;}
  E() {cout <<"E() "; }
  ~E() {cout <<"~E ";}
};
```




```
B* pb=new B; C* pc = new C; D* pd = new D; E* pe = new E; A *pa1=pb, *pa2=pc, *pa3=pd, *pa4=pe; B *pb1=pe;
```

```
/*
class Z {
 public: Z(int x) {}
};
```

```
class A {
 public:
  void f(int) {cout << "A::f(int) "; f(true);}
  virtual void f(bool) {cout <<"A::f(bool) ";}
  virtual A* f(Z) {cout <<"A::f(Z) "; f(2); return this;}
  A() {cout <<"A() "; }
};
```

```
class B: virtual public A {
 public:
  void f(const bool&){cout<< "B::f(const bool&) ";}
  void f(const int&){cout<< "B::f(const int&) ";}
  virtual B* f(Z) {cout <<"B::f(Z) "; return this;}
  virtual ~B() {cout << "~B ";}
  B() {cout <<"B() "; }
};
```

```
class C: virtual public A {
 public:
  C* f(Z){cout <<"C::f(Z) "; return this;}
  C() {cout <<"C() "; }
};
```

```
class D: virtual public A {
public:
  virtual void f(bool) const {cout <<"D::f(bool) ";}
  A* f(Z) {cout << "D::f(Z) "; return this;}
  ~D() {cout <<"~D ";}
  D() {cout <<"D() ";}
};
```

```
class E: public C {
 public:
  C* f(Z){cout <<"E::f(Z) "; return this;}
  ~E() {cout <<"~E ";}
  E() {cout <<"E() ";}
};
```

```
class F: public B, public E, public D  {
 public:
  void f(bool){cout<< "F::f(bool) ";}
  F* f(Z){cout <<"F::f(Z) "; return this;}
  F() {cout <<"F() "; }
  ~F() {cout <<"~F ";}
};

 */
```

```
B* pb=new B; C* pc = new C; D* pd = new D; E* pe = new E;
F* pf = new F; B *pb1= new F;
A *pa1=pb, *pa2=pc, *pa3=pd, *pa4=pe, *pa5=pf;
```

```
E* puntE = new F;
```

```
A* puntA = new F;

pa3->f(3);

pa5->f(3);

pb1->f(true);

pa4->f(true);

pa2->f(Z(2));

pa5->f(Z(2));

(dynamic_cast<E*>(pa4))->f(Z(2));

(dynamic_cast<C*>(pa5))->f(Z(2));

pb->f(3);

pc->f(3);

(pa4->f(Z(3)))->f(4);

(pc->f(Z(3)))->f(4);

delete pa5;

delete pb1;
```

```
A* pa5 = new F;
pa5->f(3); // A::f(int) F::f(bool)

B* pb1 = new F;
pb1->f(true); // B::f(const bool&)

A* pa4 = new E;
pa4->f(true); // "A::f(bool)

A* pa2 = new C;
pa2->f(Z(2)); // C::f(Z)

A* pa5 = new F;
pa5->f(Z(2)); // F::f(Z)

(dynamic_cast<E*>(pa4))->f(Z(2));

(dynamic_cast<C*>(pa5))->f(Z(2));

pb->f(3);

pc->f(3);

(pa4->f(Z(3)))->f(4);

(pc->f(Z(3)))->f(4);

delete pa5;

delete pb1;
```