Classes with dynamically allocated components

Implementation of a safe array type: dbl_vect (1)

////File: dbl_vect.h

```
class dbl_vect {
  public:
    explicit dbl_vect(int n = 10);
    ~dbl_vect() { delete []p; }
    double& element(int i);

    double dot_prod(const dbl_vect& v) const;
    int  ub() const { return (size - 1); }   //upper bound
    void print() const;

  private:
    double*  p;
    int      size;
};
```

notice return type: double&
i.e. the result is a
left-value/variable

/////// File dbl_vect.cpp

```
#include <iostream>
#include<cassert>
#include "dbl_vect.h"

dbl_vect::dbl_vect(int n) : size(n)
{ assert(n > 0);
  p = new double[size];
  assert(p != 0);
}

double& dbl_vect::element(int i)
{ assert (i >= 0 && i < size);
  return p[i];
}
```

returns the *variable* p[i]

Classes with dynamically allocated components

Implementation of a safe array type: dbl_vect (2)

/////// File dbl_vect.cpp (continued)

```
void dbl_vect::print()const
{ cout << " vector of size " << size << endl;
  for (int i = 0; i <size; ++i) cout << p[i] << "\t";
}

double dbl_vect::dot_prod(const dbl_vect& v) const
{ assert(size == v.size);
  double sum = 0.0;

  for (int i = 0; i < size; ++i) sum += p[i] * v.p[i];
  return sum;
}
```

///// File client.cpp:

```
#include <iostream>
//using namespace std;

#include "dbl_vect.h"

int main()
{ dbl_vect a(10), b(5);  dbl_vect c(6);

  for (int i = 0; i <= c.ub(); ++i)
      c.element(i) = i + 0.1;

  c.print();
  cout << " dot product = " << c.dot_prod(c) << endl;

  a.element(1) = 5;
  b.element(1) = a.element(1) + 7;
  a.print(); cout << endl; …
}
```

```
vector of size 6
0.1   1.1   2.1   3.1   4.1   5.1
dot product = 58.06
vector of size 10
0     5     0     0     0     0     0
      0     0     0
```

Classes with dynamically allocated components

Copying

```
///// File copying.cpp:
#include "dbl_vect.h"
#include <iostream>
```

dv default *copy constructor* called here

```
void P(dbl_vect dv)
  { for(int i=0; i<= dv.ub();++i){dv.element(i)= 1.0;}
    dv.print();
  }
```

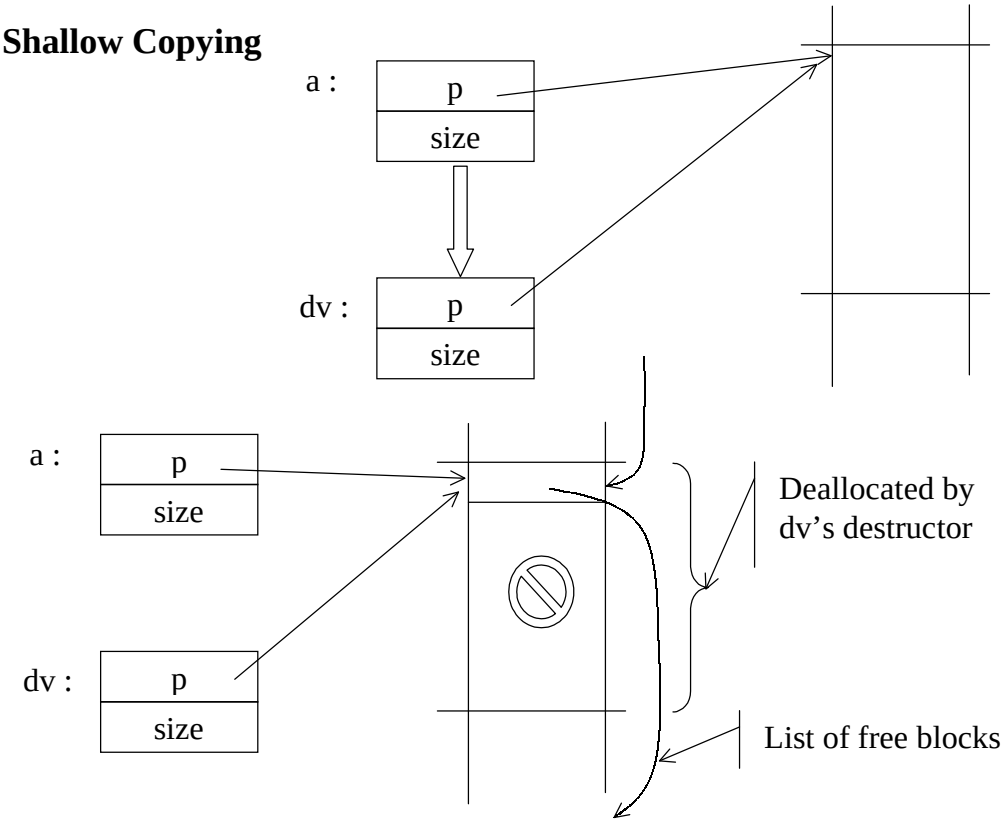
dv destructor called here

```
int main()
{ dbl_vect a(5);
  for (int i=0; i<=a.ub(); ++i){a.element(i)= i+ double(i)/10.0;}
  a.print();
  P(a);
  a.print();
}
```

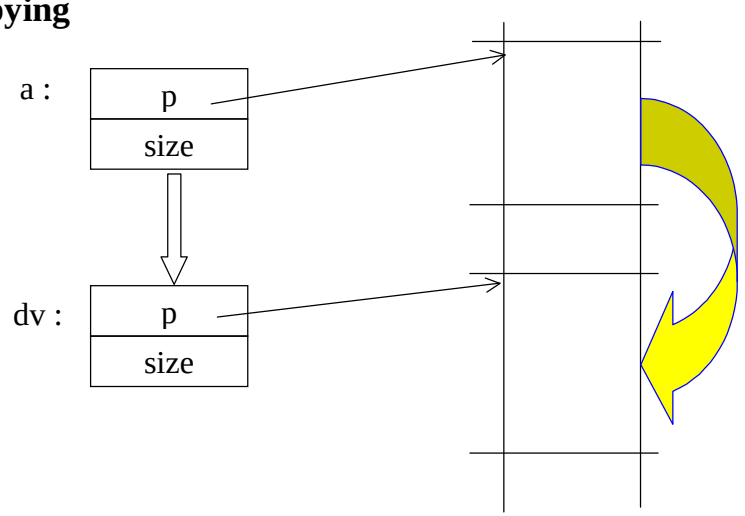
output:	
vector of size 5 0 1.1 2.2 3.3 4.4	a
vector of size 5 1 1 1 1 1	dv
vector of size 5 2.49092e-297 1 1 1 1	a ?

Classes with dynamically allocated components

Shallow Copying



Deep Copying



Copy Constructor

Every class has a *copy constructor*:

```
class A
{ public:
    A(const A & a)
    …
}
```

the copy constructor is used at function calls to
- initialise value parameters of type A
- to copy **return**-values of type A back to the caller

The copy constructor is supposed to initialise the new A-object so it is a copy of the argument object a.

The *default copy-constructor* makes memberwise/shallow initialisation, i.e. pointers are copied, but *not* the data pointed to by the pointers.

Redefine Copy Constructor to Use Deep Copying

```
class dbl_vect
{ public:
    explicit dbl_vect(int n = 10);     // constructor
    dbl_vect(const dbl_vect&);         // copy constructor
    ~dbl_vect() { delete []p; }        // destructor

    double& element(int i);
    double dot_prod(const dbl_vect& v) const;
    int  ub() const { return (size - 1); }   //upper bound
    void print() const;
  private:
    double*  p;
    int size;
};

dbl_vect::dbl_vect(const dbl_vect& dv):size(dv.size)
 {assert(size>0);
  p= new double[size];     // allocate new space for copy
  assert(p != 0);
  for(int i=0; i<size; ++i){p[i]= dv.p[i];}      //copy elements
 }
```


Copy Constructor

Redefine Copy Constructor to Use Deep Copying

Try Again:

```
int main()
{ dbl_vect a(5);
  for (int i=0; i<=a.ub(); ++i){a.element(i)= i + i / 10.0;}
  a.print();
  P(a);
  a.print();
}
```

output:	
vector of size 5 0 1.1 2.2 3.3 4.4	a
vector of size 5 1 1 1 1 1	dv
vector of size 5 0 1.1 2.2 3.3 4.4	a OK