Modern C++ - POD and non-POD

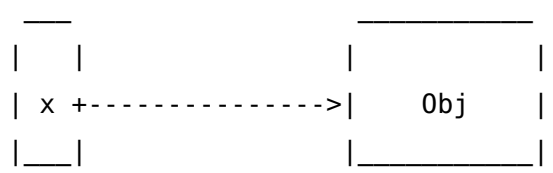
15. POD and non-POD classes

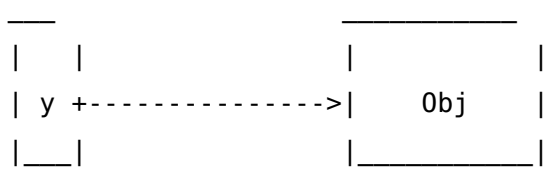
Value and reference semantics

Java and some other object-oriented languages use *reference* semantics. Reference semantic means that the declared variables represented by a single, fixed-sized *pointer* or *reference*, and the real object with all its fields are allocated in the *heap* by the **new** operation.

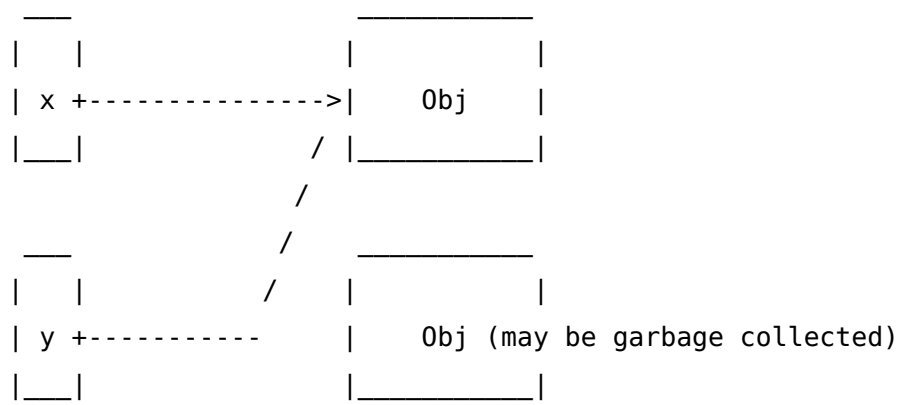
When we assign such a variable, we assign the references, therefore the assigned variable will refer to the new object.

```
Obj x = new Obj();
Obj y = new Obj();
```






```
y = x;
```



C++ however, uses *value-semantic*. Value semantic means, that the declared variable is stored directly in the computer's memory without any reference. The following object is stored in the memory as the fields appended after each other (with optionally padding between the fields).

```
1 struct MyStruct
2 {
3   int       i;
4   double    d;
5 };
```

When such object is being copied, then by default the memory area of the source is copied directly into the target. More precisely, the default assignment (and copy constructor) is implemented as a memberwise copy:

```
1 MyStruct x, y;
2
3 y = x;  // means: y.i = x.i; y.d = x.d;
```

POD and non-POD types

This default behavior works for many cases. However, there are situations when the default (memberwise) copy does not fit for the purposes.

Consider the following **DVector** class which stores up to **maxsize double** elements in a buffer dynamically allocated in the heap.

```
 1 // dvector.h
 2 #ifndef DVECTOR_H
 3 #define DVECTOR_H
 4
 5 // dynamically allocated vector storing double values up to 64 elements.
 6 class DVector
 7 {
 8 public:
 9   DVector();    // constructor
10
11   const int maxsize = 64;
12   int     size() const;    // actual size
13
14   double& operator[](int i);        // unchecked access
```

```
15   double  operator[](int i) const;  // unchecked access, const member
16
17   void    push_back(double d);  // append to end
18   void    pop_back();           // remove from end;
19
20 private:
21   int     _size;        // actual number of elements
22   int     _capacity;    // buffer size
23   int* _ptr;         // pointer to buffer
24 };
25 #endif /* DVECTOR_H */
```

We can insert new elements to the end of the buffer with the **push_back** function, while **pop_back** removes the last element. Inserted elements can be accessed with indexes between **0** and **size()-1** by the *index* operator) **operator[]**.

The constructor creates an empty DVector, allocating the buffer for **maxsize** elements. (Later we will create a version where the buffer automatically reallocated on push_back when full.)

The class has three fields: **capacity** to store the actual capacity (in this version it is always maxsize), **size** to store the elements actually inserted, and **ptr** pointing to the buffer allocated in heap.

The implementation in *dvector.cpp* is straightforward.

```
 1 // dvector.cpp
 2 #include <stdexcept>
 3 #include "dvector.h"
 4
 5 DVector::DVector()
 6 {
 7   _capacity = maxsize;
 8   _size = 0;
 9   _ptr = new double[_capacity];
10 }
11 int DVector::size() const
12 {
13   return _size;
14 }
15 double& DVector::operator[](int i)
16 {
```

```
17   return _ptr[i];
18 }
19 double DVector::operator[](int i) const
20 {
21   return _ptr[i];
22 }
23 void DVector::push_back(double d)
24 {
25   if ( _size == _capacity )
26     throw std::out_of_range("vector full");
27   _ptr[_size] = d;
28   ++_size;
29 }
30 void DVector::pop_back()
31 {
32   if ( 0 == _size )
33     throw std::out_of_range("vector empty");
34   --_size;
35 }
```

And here is the client code which tests the program:

```
 1 #include <iostream>
 2 #include "dvector.h"
 3
 4 void print(const DVec& dv, char *name)
 5 {
 6   std::cout << s << " = [ ";
 7   for (int i = 0; i < dv.size(); ++i )
 8     std::cout << dv[i] << " ";
 9   std::cout << "]" << std::endl;
10 }
11 int main()
12 {
13   DVector x;  // declare and fill x
14   for (int i = 0; i < 10; ++i )
15     x.push_back(i);
16
17   DVector y;  // declare and fill y
18   for (int i = 0; i < 15; ++i )
```

```
19     y.push_back(i+20);
20
21   print(x,"x");
22   print(y,"y");
23
24   std::cout << " x = y;" << std::endl;
25   x = y;
26
27   print(x,"x");
28   print(y,"y");
29
30   std::cout << "x[0] = 999;" << std::endl;
31   x[0] = 999;
32
33   print(x,"x");
34   print(y,"y");
35 }
```

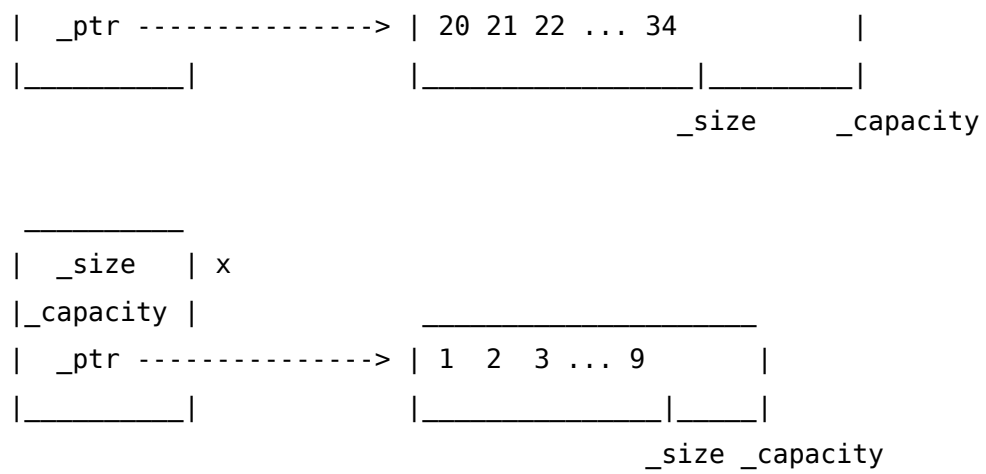
The result is not really what we expected:

```
1 $ ./a.out
2 x = [ 0 1 2 3 4 5 6 7 8 9 ]
3 y = [ 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 ]
4 x = y;
5 x = [ 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 ]
6 y = [ 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 ]
7 x[0] = 999;
8 x = [ 999 21 22 23 24 25 26 27 28 29 30 31 32 33 34 ]
9 y = [ 999 21 22 23 24 25 26 27 28 29 30 31 32 33 34 ]
```

Everything looks fine up to line 7. Both **x** and **y** objects have created, filled with integer literals automatically converted to doubles. When **y** is assigned to **x** in line 4 all elements of **y** seems to be copied to **x**. However, when **x[0]** is modified in line 7, surprisingly **y** is modified. This is definitely not what we expect.

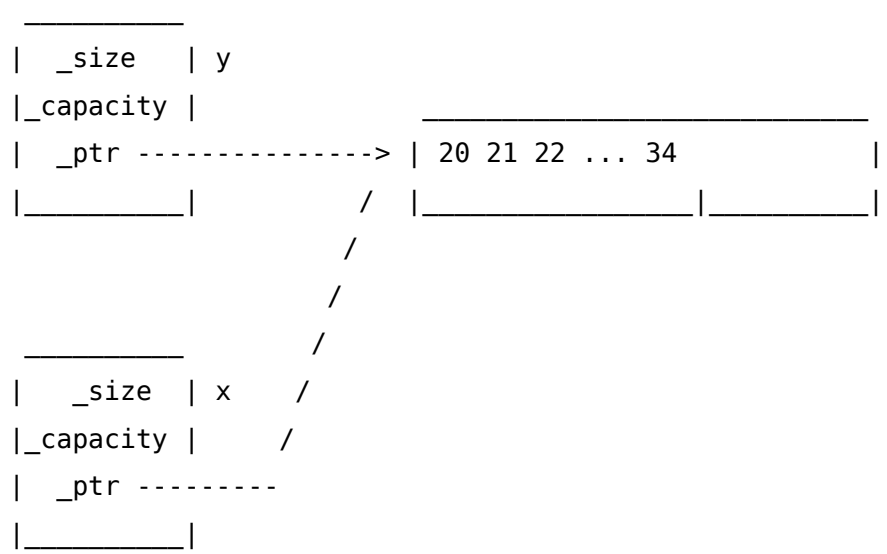
Where is the error?

Let consider the object layouts:



On the **x = y** assignment, we copy the values of all members of **y** to **x**, including the **ptr**, which means, that **x._ptr** will points to where **y._ptr** pointed to – the buffer of **y** object.

This happened on **x = y**:



As we see, the default memberwise copy is not always fits to the requirements. Those types where the memberwise copy is fine we call *POD types* (from hystorical *Plain Old Data* expression). Otherwise, we speak about a *non-POD* type.

Copy constructor and assignment operator

When the default memberwise copy operations are not sufficient, we have to provide our own copy operations: the *copy constructor* and the *assignment operator*. The role of these *special memberfunctions* is to implement the correct copy algorithms.

Copy constructor

The copy constructor has a strict signature: **MyType(const MyType &rhs)**. (In very special cases – like for some smart pointer – we can omit the *const* qualifier.) The copy constructor is called when a new object is created and initialized from the same type as argument.

```
MyType object1(object);    // copy constructor
MyType object2 = object1;  // copy constructor, non-explicit
```

Usually the semantics of the copy constructor is to allocate resources for the new object and initialize it copying from the argument object.

Assignment operator

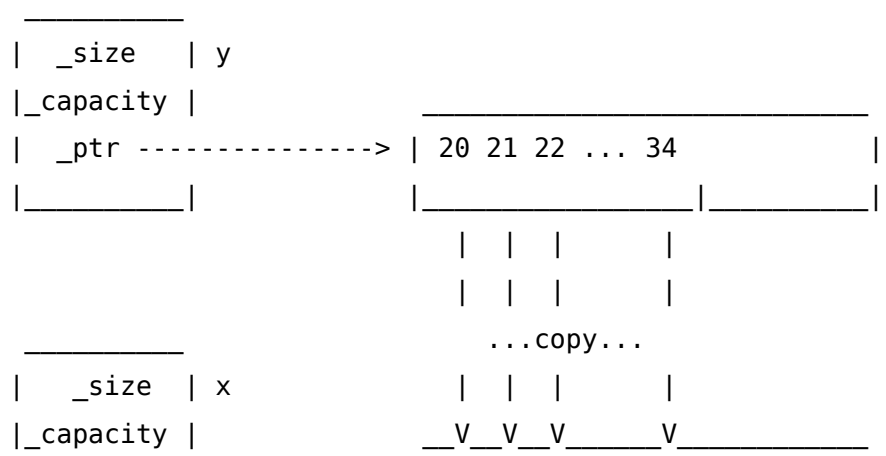
The assignment operator is a normal operator implementing the value semantics for assignment. The usual (but non-mandatory) syntax is **MyType& operator=(const MyType &rhs)**. The assignment operator is called when an existing object is assigned to an other existing object. It is possible, that the types of these objects are not the same, e.g. a *char** object is assignable to an *std::string*.

The parameter of the assignment operator is usually a refrence parameter, but contrary to the copy constructor this is not mandatory. The return type of the assignment is a refefence to the same type, and returns the freshly updated object – the one on the left hand side of the assignment.

```
MyType object1, object2, object3;   // constructor
object3 = object2 = object1;        // assignment
```

Usually the semantics of the assignment is to free the old resurces of the object, then allocate the new resources and copy the value from the argument object. The function returns with ***this**, returning the freshly updated object by reference.

With the correct operations defined the effect of the **x = y;** copy is the following:



```
|  _ptr ---------------> | 20 21 22 ... 34            |
|__________|             |____________________|__________|
```

Destructor

There is an other issue with our **DVector** class. When our object is going out of its lifetime the date members of the object: **capacity**, **size**, and **ptr** will be freed. However, the buffer in the heap, that is allocated in the constructor, remains allocated.

In C++ we have no *garbage collector* (contrary to Java and C#). There are no more reference to this memory anymore, therefore this memory is wasted, it is a *memory leak*. Memory leak is a critical issue in C++.

The constructor is the place when we initialize our objects. The special memberfunction to execute the reverse operations – deallocate resources, other clean-up activities, etc. – is the *destructor*. There might be only one destructor for a class, and its signature is **~MyType()**.

The destructor – if defined – is always executed when an object's lifetime ends.

```
{
  MyType object;   // constructor is called for object
  // ...
}                  // destructor is called for object
```

Implmenting the copy operations and the destructor

Here we can see the implementation ofthe DVector with proper copy operations and destructor.

```
 1 // dvector.h
 2 #ifndef DVECTOR_H
 3 #define DVECTOR_H
 4
 5 class DVector
 6 {
 7 public:
 8   DVector();                              // constructor
 9   DVector(const DVector& rhs);            // copy constructor
10   DVector& operator=(const DVector& rhs); // assignment operator
11
12   ~DVector();   // destructor
```

```
13
14   int     size() const;    // actual size
15
16   double& operator[](int i);        // unchecked access
17   double  operator[](int i) const;  // unchecked access, const member
18
19   void    push_back(double d);  // append to end
20   void    pop_back();           // remove from end;
21
22 private:
23   int     _size;        // actual number of elements
24   int     _capacity;    // buffer size
25   double* _ptr;         // pointer to buffer
26 };
27 #endif /* DVECTOR_H */
```

```
 1 // dvector.cpp
 2 #include <stdexcept>
 3 #include "dvector.h"
 4 DVector::DVector()
 5 {
 6   _capacity = 64;
 7   _size = 0;
 8   _ptr = new double[_capacity];
 9 }
10 DVector::DVector(const DVector& rhs)
11 {
12   _capacity = rhs._capacity;
13   _size = rhs._size;
14   _ptr = new double[_capacity];
15
16   for (int i = 0; i < _size; ++i)
17     _ptr[i] = rhs._ptr[i];
18 }
19 DVector& DVector::operator=(const DVector& rhs)
20 {
21   if ( this != &rhs )  // avoid x = x
22   {
23     delete [] _ptr;
24     _capacity = rhs._capacity;
```

```
25     _size = rhs._size;
26     _ptr = new double[_capacity];
27
28     for (int i = 0; i < _size; ++i)
29       _ptr[i] = rhs._ptr[i];
30   }
31   return *this;  // for x = y = z
32 }
33 DVector::~DVector()
34 {
35   delete [] _ptr;
36 }
37 //... rest of dvector.cpp is the same
```

Let recognize, the *self-assignment check* in line 21 of dvector.cpp. This is necessary to avoid the issues when someone accidentally try to execute **x = x**.

Make DVector growing dynamically

Our current implementation has a limited **capacity**. It is easy to remove this restriction making the capacity of the object dynamically growing on demand.

We check in **push_back** whether the logical *size* of the object reaches the physical *capacity*, and then let the buffer expanded by the newly defined **grow** method.

```
1 void DVector::push_back(double d)
2 {
3   if ( _size == _capacity )
4     grow();
5
6   _ptr[_size] = d;
7   ++_size;
8 }
```

The grow method doubles the capacity and allocate a new buffer, copies the elements from the old buffer, then deallocates the old buffer. The size does not change here.

```
1 void DVector::grow()
2 {
3   double *_oldptr = _ptr;
4   _capacity = 2 * _capacity;
```

```
 5   _ptr = new double[_capacity];
 6
 7   for ( int i = 0; i < _size; ++i)
 8     _ptr[i] = _oldptr[i];
 9
10   delete [] _oldptr;
11 }
```

This is very similar how the standard library **std::vector** is implemented.

The complete DVector class

To make our class complete, we do some refactoring work following the *DRY – Do not Repeat Yourself* – principle. We create the **copy** and **release** methods to implement common activies of the copy constructor, assignment operator and the destructor. Naturally, as neither of grow, copy, release are part of the intended interface of the class, we declare them as *private* methods.

We conclude with the following code:

```
 1 // dvector.h
 2 #ifndef DVECTOR_H
 3 #define DVECTOR_H
 4
 5 class DVector
 6 {
 7 public:
 8   DVector();    // constructor
 9   DVector(const DVector& rhs);            // copy constructor
10   DVector& operator=(const DVector& rhs); // assignment operator
11
12   ~DVector();   // destructor
13
14   int     size() const;    // actual size
15
16   double& operator[](int i);        // unchecked access
17   double  operator[](int i) const;  // unchecked access, const member
18
19   void    push_back(double d);  // append to end
20   void    pop_back();           // remove from end;
21
```

```
22 private:
23   int     _size;        // actual number of elements
24   int     _capacity;    // buffer size
25   double* _ptr;         // pointer to buffer
26
27   void copy(const DVector& rhs);   // private helper function
28   void release();                  // private helper function
29   void grow();                     // reallocate buffer
30 };
31 #endif /* DVECTOR_H */
```

```
 1 // dvector.cpp
 2 #include <stdexcept>
 3 #include "dvector.h"
 4
 5 DVector::DVector()
 6 {
 7   _capacity = 4;
 8   _size = 0;
 9   _ptr = new double[_capacity];
10 }
11 DVector::DVector(const DVector& rhs)
12 {
13   copy(rhs);
14 }
15 DVector& DVector::operator=(const DVector& rhs)
16 {
17   if ( this != &rhs )  // avoid x = x
18   {
19     release();
20     copy(rhs);
21   }
22   return *this;  // for x = y = z
23 }
24 DVector::~DVector()
25 {
26   release();
27 }
28 void DVector::copy(const DVector& rhs)
29 {
```

```
30   _capacity = rhs._capacity;
31   _size = rhs._size;
32   _ptr = new double[_capacity];
33
34   for (int i = 0; i < _size; ++i)
35     _ptr[i] = rhs._ptr[i];
36 }
37 void DVector::release()
38 {
39   delete [] _ptr;
40 }
41 void DVector::grow()
42 {
43   double *_oldptr = _ptr;
44   _capacity = 2 * _capacity;
45   _ptr = new double[_capacity];
46
47   for ( int i = 0; i < _size; ++i)
48     _ptr[i] = _oldptr[i];
49
50   delete [] _oldptr;
51 }
52 int DVector::size() const
53 {
54   return _size;
55 }
56 double& DVector::operator[](int i)
57 {
58   return _ptr[i];
59 }
60 double DVector::operator[](int i) const
61 {
62   return _ptr[i];
63 }
64 void DVector::push_back(double d)
65 {
66   if ( _size == _capacity )
67     grow();
68
```

```
69   _ptr[_size] = d;
70   ++_size;
71 }
72 void DVector::pop_back()
73 {
74   if ( 0 == _size )
75     throw std::out_of_range("vector empty");
76
77   --_size;
78 }
```