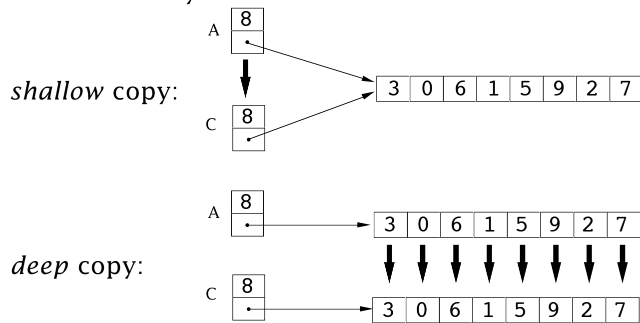
The assignment `C := A` could be implemented in either of two ways:



Shallow copying yields *reference* semantics; value semantics requires deep copying. (Sethi's `List` class is vulnerable to shallow copying.)

Shallow copying leads to *aliasing*: two (variable) names for the same storage. In Java, Modula–2, and Oberon, aliasing is a way of life.

Storage-management implications: If storage is shared between variables which have different lifetimes, when should it be recycled?

- In Pascal, Modula–2, and C, it's the client programmer's problem.
- Java, Oberon, Eiffel, etc. eliminate the problem by providing garbage collection: Heap storage is reclaimed as automatically as stack storage.
- C++ enables an ADT designer to prevent the problem by using deep copying to avoid sharing storage.

Declarations with `IntArray` arguments—

```
IntArray C = A;
IntArray C( A );
```

invoke the *copy* constructor:

```
IntArray::IntArray( const IntArray &b )
{
    size = b.size;
    data = new int[ size ];
    for ( int i = 0; i < size; ++i )
        data[ i ] = b.data[ i ];
}
```

Other occasions at which the copy constructor is invoked:

- an `IntArray` argument is passed to a function *by value*
- a function returns an `IntArray` value

The copy-constructor code is

```
data = new int[ size ];
for( int i = 0; i < size; ++i )   // expensive
    data[ i ] = b.data[ i ];
```

instead of simply

```
data = b.data;                    // cheap
```

in order to provide value semantics, ensuring that two `IntArray` variables share no storage.

The `IntArray`-assignment function also copies deeply:

```
IntArray&  IntArray::operator=( const IntArray &b )
{
   if( data != b.data )                // note 1
   {
      delete data;                     // note 2
      size = b.size;
      data = new int[ size ];
      for( int i = 0; i < size; ++i )  // note 3
            data[ i ] = b.data[ i ];
   }
   return *this;                       // note 4
}
```

Notes:

1. detects `x = x;` (why?)
2. recycles target's heap storage
3. copies source data to target data (*deep* copy)
4. permits `x = y = z;`
   i.e., `x.operator=( y.operator=( z ) )`

The *default* copy constructor and assignment operator simply copy all member variables, including pointers. Result: reference semantics.

How could an ADT designer avoid deep copying's cost without introducing reference semantics?

The Law of The Big Three

> If a class needs any of
> a **destructor**,
> a **copy constructor**,
> an **assignment operator**,
> ***it needs them all***.

Cline & Lomow, *C++ FAQs* (A-W, 1995)

The `IntArray`-indexing function:

```
int& IntArray::operator[]( int index )
{
  return data[index];
}
```

Note that the function returns an ***l***-value.

Example:

```
IntArray M( 20 );
// ...
M[12] = M[8] + M[18];
```

That's "syntactic sugar" for:

```
int* IntArray::operator[]( int index )
{
  return &(data[ index ]);
}
// ...
*(M[12]) = *(M[8]) + *(M[18]);
```

The IntArray-addition function

```
IntArray IntArray::operator+( const IntArray& b )
   const                                          // 1
{
  IntArray r( size > b.size ? *this : b );        // 2

  if( size > b.size )                             // 3
  {
     for( int i = 0; i < b.size; i++ )
        r.data[i] += b.data[i];
  }
  else
  {
     for( int i = 0; i < size; i++ )
        r.data[i] += data[i];
  }

  return r;                                       // 4
}
```

Notes

1. declares that function does not alter its implicit argument (which is passed by reference)
2. the result is as long as the longer argument
3. adds elements of shorter argument to r
4. returns r (*by value* ⇒ automatic call of copy constructor)

The examples again:

```
int main()
{
   IntArray A( 30 ), B( A.getSize() + 7 );
   IntArray *p = new IntArray( A );
   IntArray C = A;
              // array initialization by
              //  C.IntArray( A )
   C = B;     // array assignment, including
              // deletion of C's previous value,
              // by C.operator=( B ).
   B = C + A + C;
   // B.operator=(C.operator+(A).operator+(C))

        // temp.IntArray(C.operator+(A));
        // B.operator=(temp.operator+(C));
        // temp.~IntArray();

   delete p;  // p->~IntArray()

     // At exit from block, compiler inserts
     //   C.~IntArray();
     //   B.~IntArray();
     //   A.~IntArray();
}
```

In C++, internalization of ADTs' heap-storage management allows ADT objects to be used like instances of basic types— their *heap storage* is reclaimed as automatically as their *activation-record storage*.

An important aspect of ADTs is that their implementations are *hidden*. This hiding is accomplished by restricting the scope of names.
In a C++ `class` or `struct` definition,

- `public` names have the same scope as the name of the class
- `private` names are visible only in the definitions of the class's own member functions
- `protected` names are visible in the definitions of the class's own member functions *and* of member functions of its immediate subclasses

Names that are not declared `public`, `protected`, or `private`:

In a `class`, they are `private`;
in a `struct`, they are `public`.

ADTs in Haskell

In Haskell, the mechanism for hiding is the **module**. A module is a script (i.e., a text file) that begins

```
> module XYZ ...
```

A module that begins

```
> module XYZ where ...
```

exports *all* of its top-level names, i.e., makes those names available for importation into other modules.

A module that begins

```
> module XYZ (name1, name2, ...) where ...
```

exports *only* the names appearing in the export list (before the **where**). All other names defined in the module are *private*.

Example: ADT **Queue**

```
> module ADT_Queue
>   ( Queue,
>     emptyQ,   -- Queue a
>     isEmptyQ, -- Queue a -> Bool
>     addQ,     -- a -> Queue a -> Queue a
>     remQ      -- Queue a -> (a, Queue a)
>   ) where

> data Queue a = Q [a] deriving (Show)

> emptyQ :: Queue a
> emptyQ = Q []

> isEmptyQ :: Queue a -> Bool
> isEmptyQ (Q []) = True
> isEmptyQ _      = False

> addQ :: a -> Queue a -> Queue a
> addQ x (Q xs) = Q (xs ++ [x])

> remQ :: Queue a -> (a, Queue a)
> remQ (Q (x:xs)) = (x, Q xs)
> remQ _          = error "remQ: empty queue"
```

An example:

```
> q1 = addQ 1 (addQ 2 (addQ 3 emptyQ))

Main> q1
Q [3,2,1]
```