Double Linked Lists

The goal is to show how access control can be used for building reliable objects that hide their memory management.

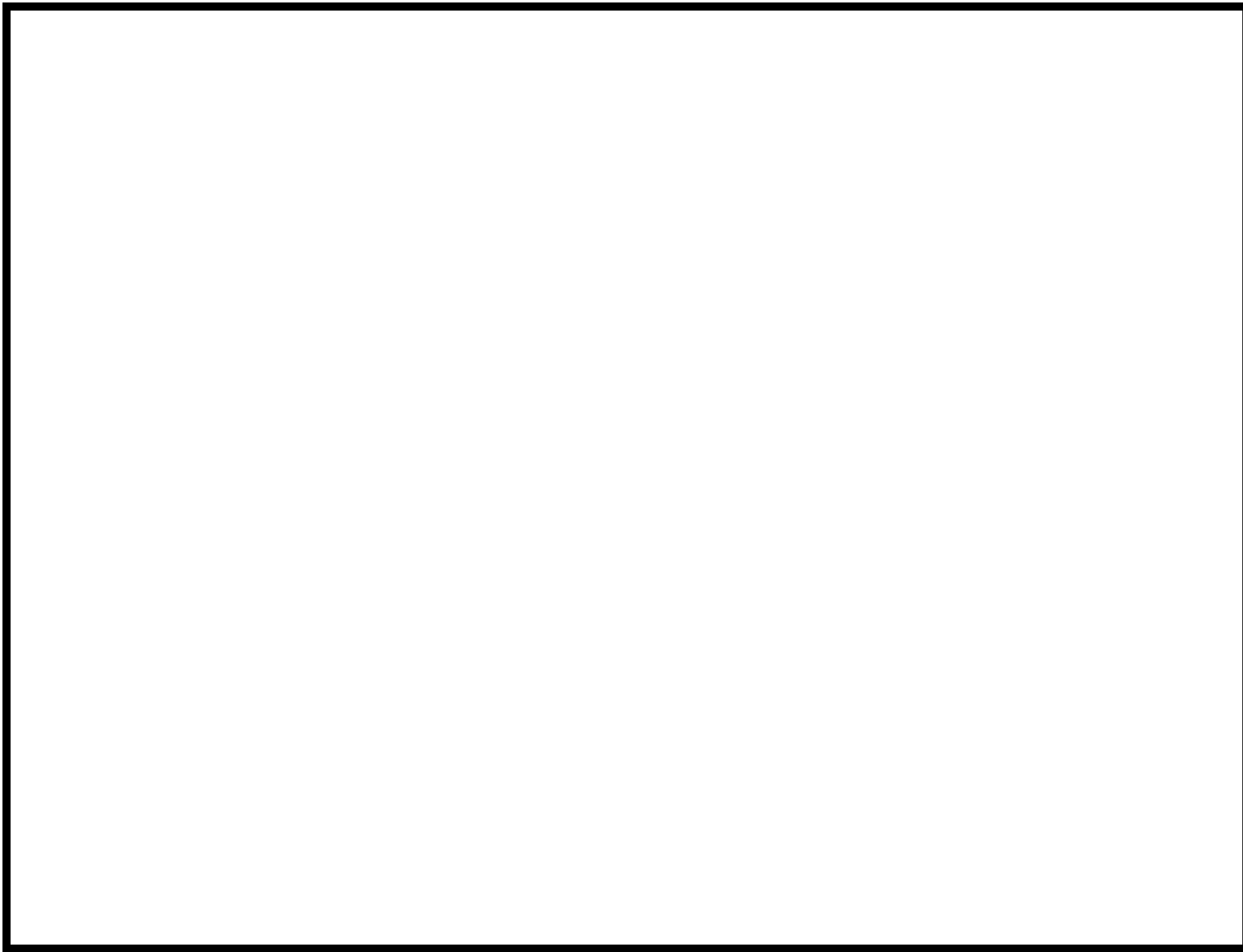
We will also use `operator [ ]`, and discuss some subtleties related to **const**ness.

Doubly Linked List

```
struct listnode
{
   std::string s;
   listnode* prev;
   listnode* next;
};

class dllist
{
   listnode* first;
   listnode* last;

   unsigned int len;
};
```



Doubly Linked Lists

We have seen (singly) linked lists before. In a doubly linked list, every node has two pointers. One points forward, and one points backward.

Doubly linked lists have the advantage that:

- Navigation (going backwards) is much easier.
- Insertion, deletion is easier.

The main disadvantage of dll's is that the extra pointers require a bit more space.

Visibility of listnode

In principle, the user should not use listnode. listnode can be hidden by declaring it inside dllist. This is not the same as in Java, because listnode will be not a subclass of dllist. It just replaces the name `listnode` by `dllist::listnode`, and one can make `dllist::listndoe` a private member.

We will not do this.

Constructors for listnode

It is useful to give constructors for listnode. This makes it harder to forget to initialize the pointers.

The constructors are simple, so they can be given at once in the **.h** file:

```
listnode( const std::string& s )
   : s(s), prev(0), next(0)
{ }

listnode( const std::string& s,
          listnode* p, listnode* n )
   : s(s), prev(p), next(n)
{ }
```

Name Lookup in Initializers

Normally, local parameters of a method override members. If you want the member, write `this -> mem`.

In an initializer of form `p(s)`, p has to be a member. Therefore, it is not necessary (and not allowed) to write `this -> p`.

Constructor for dllist

There is a reasonable default list, namely the empty list:

```
dllist( )
   : first(0), last(0), len(0)
{ }
```

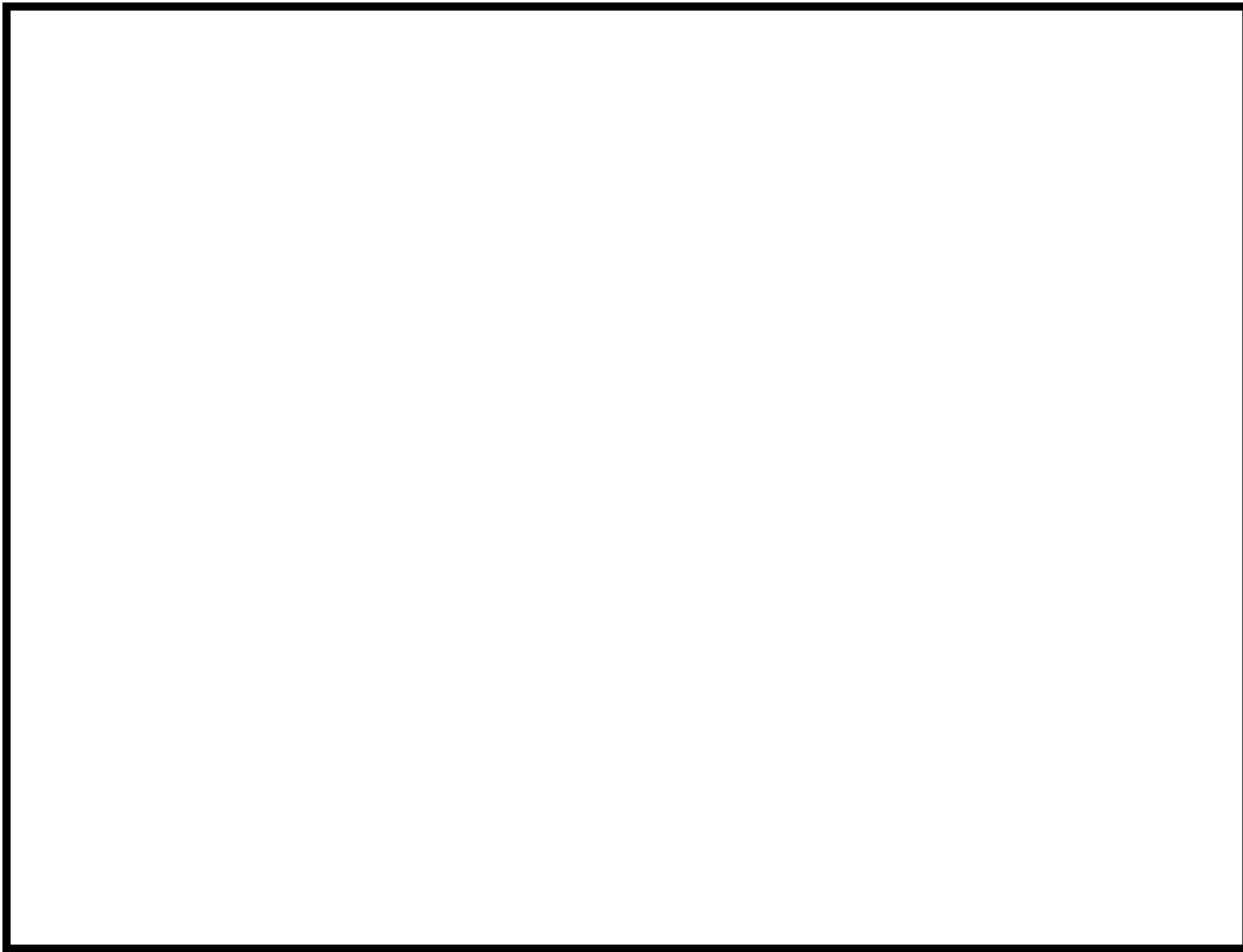
Copy Constructor of dllist

Copying a list is not so easy. The easiest way it to use the function `push_back( )`:

```
dllist( const dllist& dl )
   : first(0), last(0), len(0)
{
   for( listnode* p = dl. first; p; p = p -> next )
      push_back( p -> s );
}
```

push_back( )

```
void push_back( const std::string& s )
{
   if( first == 0 )
   {
      first = new listnode(s);
      last = first;
   }
   else
   {
      listnode* l = newlistnode( s, last, 0 );
      ( last -> next ) = l;
      last = l;
   };
   ++ len;
}
```

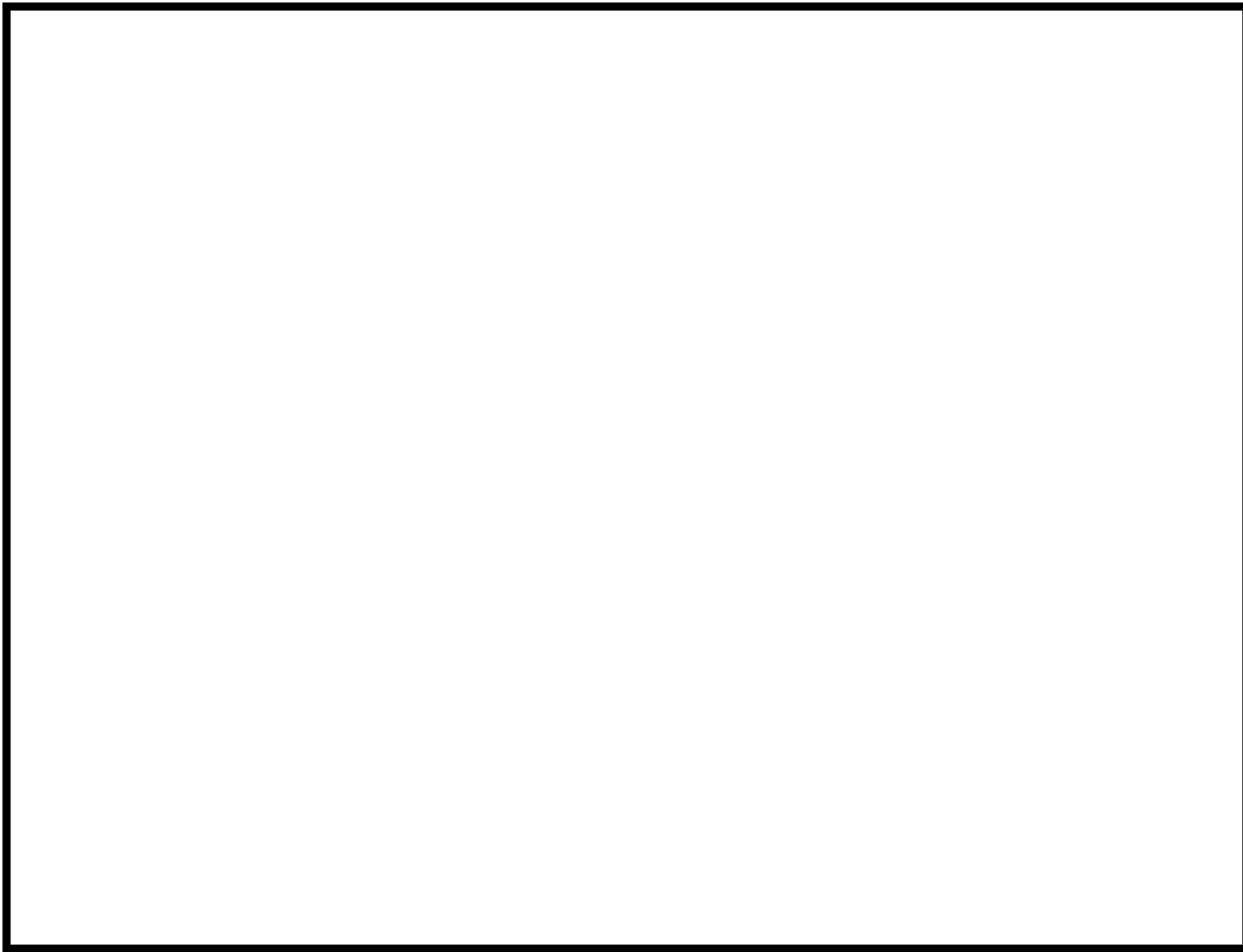


pop_back( )

```
void pop_back( )
{
   listnode *l = ( last -> prev );
      // This means that the program is not reliable
      // when the list is empty.

   delete last;
   last = l;
   if( last )
      last -> next = 0;
   else
      first = 0;

   -- len;
}
```



front( ) and back( )

If you retrieve an element from a container, it should be returned as reference. This avoids copying, and it allows assignment.

There is a subtlety with **const**ness. If you declare `const std::string& front( ) const,` then you cannot write

```
dll. front( ) = "somebody has to be first";
```

If you declare `std::string& front( ),` then you cannot call

```
dll. front( )
```

when dll is **const**.

Retrieving elements from containers

With the declaration `std::string front( ) const`, one can write

```
dll. front( ) = "assigning string to a useless copy";
```

but it is meaningless.

The solution is to define both the **const** and the non-**const** version. In case more than one definition exists for a method, the compiler uses the most specialized definition that fits.

const is more specialized than non-**const**.

front( )

```
// Returns the first element of the list, when the
// list is non-empty.

const std::string& front( ) const
{
   return first -> s;
}

std::string& front( ) const
{
   return first -> s;
}
```

What to do when preconditions are broken

Functions `pop_front( )`, `pop_back( )`, `front( )`, `back( )` crash when the list is non-empty.

What to do about this?

1. Invent some artificial behaviour. This solution is not acceptable, because every call that violates a precondition, is necessarily wrong.
2. Don't worry about it. Maybe the program crashes, maybe not.
3. Detect the problem and abort the program.

(3) is the best solution. We chose (2). Solution (1) is not acceptable. For (3), one can use the predicate `ASSERT(b)` . It checks the condition b, and if b is not true, it prints an error message and exits.

Assignment

`void operator = ( const dllist &dll )` does the following:

- Clean up `*this` list.
- Make copy of dll.

What if you assign a list to itself? $\Rightarrow$ list dll is first deleted, then copied.

Two solutions:

- Replace declaration by `void operator = ( dllist dll )`. This works, but it is inefficient.
- Check the situation where a list is assigned to itself. If this is the case, then do nothing.

The third solution, trusting that the user will not write $l = l$, is not acceptable, because of shared data structures.

void operator = ( )

```
void operator = ( const dllist& dll )
{
   if( *this != &dll )
   {
      while( size( ) > 0 )
         pop_front( );
   }
   for( listnode* p = dll. first;
                  p != 0;
                  p = p -> next )
      push_back( p -> s );
}
```

Destructor

The destructor is straightforward:

```
~dllist( )
{
   while( size( ))
      pop_front( );
}
```

Indexing

If one defines indexing through `operator [ ]`, the list can be indexed in the same way as an array.

As with **front** and **back**, one should define **const** and non-**const** versions.

Indexing

```
const std::string&
operator [ ] ( unsigned int k ) const
{
   ASSERT( k < size( ));
   listnode* p = dll. first;
   while(k)
   {
      p = p -> next;
      -- k;
   }
   return *p;
}
```

Indexing

```
std::string& operator [ ] ( unsigned int k )
{
   ASSERT( k < size( ));
   listnode* p = dll. first;
   while(k)
   {
      p = p -> next;
      -- k;
   }
   return *p;
}
```

Indexing

We can write

```
for( unsigned int i = 0; i < size( ); ++ i )
   std::cout << dll[i] << "\n";
```

and

```
for( unsigned int i = 0; i < size( ); ++ i )
{
   std::cout << "please type " << i << "th element: ";
   std::cin >> dll [i];
}
```

Efficiency

At present, the implementation is not efficient:

1. Deletion (through `pop_front( )` modifies pointers that are going to be deleted anyway.
2. When copying, one could first construct the `next` pointers, and only later the `prev` poiners.
3. Indexing takes linear time, while in an array it would be constant time.

Efficiency

Don't worry about small constants. Worry about orders.

But nothing can be done about (3). If indexing is important, one should use **vector**. For this reason, `std::list` in the standard library has no indexing operator.

size( )

It remains to define `size( )`:

```
unsigned int size( ) const
{
   return len;
}
```