

Because the constructor `Q` is not exported from `module ADT_Queue`, the implementation of `Queue` is inaccessible in any module that imports `ADT_Queue`.\*

The module interface thus forms a “logical firewall” that prevents changes within the ADT from spreading.

Note: The instance declaration

```
... deriving (Show)
```

is actually a breach of the abstraction (because it reveals the implementation); it's included for demonstration purposes.

Hence we can change `Queue`'s implementation without changing any software that uses `Queue`.

Why might we want to change `Queue`?

For efficiency reasons:

- Removing an item is *cheap*— requires access only to the list's *head*.
- Adding an item is *expensive*— requires access to the list's *last* element.

There's a faster implementation of `Queue`, such that

- adding an item is cheap— it requires access only to a list's *head*.
- removing an item is cheap *on average*— it (usually) requires access only to a list's *head*.

\*An ADT's implementation is visible on the Hugs command line.

```
> 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] [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 outs ins) = Q outs (x:ins)
> remQ :: Queue a -> (a, Queue a)
> remQ (Q (x:outs) ins) = (x, Q outs ins)
> remQ (Q [] _ ) = error "remQ: empty queue"
> remQ (Q [] _ ins) = remQ (Q (reverse ins) [])
```

Because the implementation is *hidden* behind the ADT's interface, changing it requires *no changes* in any module that uses it.

Example: ADT Parser

```
> module ParseLib
>   ( Parser, item, papply, (+++), sat, many, ... )
> where
>   ...
> newtype Parser a = P (String -> [(a,String)])
```

In Haskell, the **newtype** declaration creates a new type from an existing one. What distinguishes the new type from the existing one is its constructor.

Omitting `Parser`'s constructor, `P`, from `ParseLib`'s export list makes `Parser` an *abstract* type.

The **newtype** declaration of `Parser` is necessary if `Parser` is to be an instance of class `Monad`:

```
> instance Monad Parser where
>   ... signature functions ...
```

because an instance type must be a distinct type (not a mere synonym).

Among `ParseLib`'s interface functions is one we've used:

```
> papply :: Parser a -> String -> [(a,String)]
> papply (P parsef) string = parsef string
```

`papply` extracts the actual parsing function (here named `parsef`) from a `Parser` and applies it.

An example of a parser:

```
> item :: Parser Char
> item = P (\string -> case string of
>   [] -> []
>   (x:xs) -> [(x,xs)])
```

The Haskell `Queue` and `Parser` ADTs are polymorphic. In C++, polymorphic ADTs can be defined using...

### Class Templates

- A **class** is a pattern for *instances*.
- A **class template** is a pattern for *classes*.

*Instances* are built from *classes* during *execution*.

*Classes* are built from *templates* during *compilation*.

A class template looks like a class declaration with the key-word `template` and at least one **type-parameter**.

Example: a *polymorphic* array class.

The ideas behind `IntArray` would be equally useful for

- `ShortArray`
- `DoubleArray`
- etc.

Example:

```
template <typename T, int S>
class Array -- simplified IntArray
{
public:
    Array( int s = S )
    { size = s; data = new T[ size ]; }
    Array( const Array& );
    ~Array(){ delete [] data; }
    Array& operator=( const Array& );
    T& operator[]( int index );
    Array operator+( const Array& ) const;
    int getSize(){ return size; }
private:
    int size;
    T *data;
};
```

```
template<typename T, int S> Array<T,S>
Array<T,S>::operator+( const Array<T,S>& b ) const
{
    Array<T,S> r( size > b.size ? *this : b );
    if( size > b.size )
    { 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;
}
```

19 April 2005

How are class templates used? Examples:

```
Array<int,20> A0;
Array<double,10> A1(3*n+2);
Array<char,50> A2;
typedef Array<float,25> floatArray;
floatArray A3, A4, A5( 100 );
```

The type-*argument* (enclosed in < >) is substituted for the type-*parameter* during compilation to *instantiate* a class.

### Instantiation failure

Not all types can be used as template arguments.

Suppose we declare

```
Array<char*,100> A6;
```

In the definition of operator+ we find

```
r.data[i] += data[i];
```

in which (+) :: **char\*** -> **char\*** -> **char\***

But (+) isn't defined for pointers, so the instantiation may fail:

- If the client program actually calls `Array<char*>::operator+`, a compilation failure is inevitable.
- otherwise, it depends on the compiler...

Hence what C++ templates provide is not always parametric polymorphism. In cases where a template uses an overloaded function (or operator), it provides a form of overloading (i.e., ad-hoc polymorphism).

19 April 2005

A proposal for expressing instance-type restrictions in C++:

```
template<typename T>
constraints {
    T* tp; // T must have...
    B* bp = tp; // an accessible base B
    tp->f(); // a member function f
    T a(1); // a constructor from int
    a = *tp; // assignment
    // ...
}
class X {
    // ...
}; // --Stroustrup, 1994
```

In Haskell, such polymorphism restrictions are expressed through Type Classes.

A sketch (**purely imaginary**):

```
> class Incrementable t where
>   (+) :: t -> t -> t

> newtype Incrementable a => Array a = ...
> Array :: Int -> Array a
> Array :: Array a -> Array a
> ([]) :: Array a -> Int -> a&
> (+) :: Incrementable a =>
    Array a -> Array a -> Array a
> getSize :: Array a -> Int
```

19 April 2005

**Class derivation:** a new kind of sharing.

Old sharing ("contains"/"is a part of"): A class's definition is *used* in its clients.

Example: Class cell is used in List, linklist,...

New sharing ("is a kind of"): A class's definition is *extended* by new classes **derived** from it. The new classes **inherit** members from their base class.

Derivation and inheritance in zoology/taxonomy:

```
class mammal
{ warm blooded, vertebrate, breathes
  air, suckles young, has hair
};
```

```
class elephant : mammal
{ large, thick skin,
  lives in herds,
  has a long trunk
};
```

```
class mouse : mammal
{ small, furry,
  solitary, rodent,
  has a long tail
};
```

Derivation and Inheritance in C++

```
class B
{ public:
    B(){ x = 1; }
    int x;
    char f();
};
```

```
class D : public B
// D inherits B::x, B::f
{ public: int g();
};
// ...and has its own new
// member g
```

D also inherits B::B() (implicitly, not by name).

19 April 2005