

A Detailed Explanation Of the Sequence Component

Part 2

Sequence's Five Standard Operations

The Sequence Component

Sequence has 12 member functions:

- The 5 *Standard Operations*
- And 7 *Sequence Specific Operations*

```
template <class T>
class Sequence1
{
public: // Standard Operations
    Sequence1();
    ~Sequence1();

    void clear(void);
    void transferFrom(Sequence1& source);
    Sequence1& operator =(Sequence1& rhs);

// Sequence1 Specific Operations
    void add(Integer pos, T& x);
    void remove(Integer pos, T& x);
    void replaceEntry(Integer pos, T& x)
    T& entry(Integer pos);
    void append(Sequence1& sToApppend);
    void split(Integer pos,
               Sequence1& receivingS);

    Integer length(void);

private: // representation
    // ...
};
```

An Empty Sequence

The Sequence **constructor** initializes a sequence variable so that it is empty

Where:

$s1 = \langle \rangle$

Recall: The compiler guarantees that the constructor is automatically called when a Sequence variable is declared in a client program

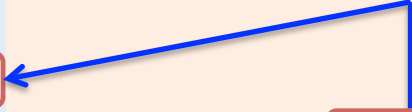
```
template <class T>
class Sequence1
{
public: // Standard Operations
    Sequence1();
        //! alters self
        //! ensures: self = < >
    ~Sequence1();
    void clear(void);
    void transferFrom(Sequence1& source);
    Sequence1& operator =(Sequence1& rhs);
// Sequence1 Specific Operations
    void add(Integer pos, T& x);
    void remove(Integer pos, T& x);
    void replaceEntry(Integer pos, T& x)
    T& entry(Integer pos);
    void append(Sequence1& sToAppend);
    void split(Integer pos,
                Sequence1& receivingS);
    Integer length(void);
private: // representation
    // ...
};
```

Reclaim Resources

```
template <class T>
class Sequence1
{
public: // Standard Operations
    Sequence1();
    ~Sequence1();

    void clear(void);
    void transferFrom(Sequence1& source);
    Sequence1& operator =(Sequence1& rhs);
// Sequence1 Specific Operations
    void add(Integer pos, T& x);
    void remove(Integer pos, T& x);
    void replaceEntry(Integer pos, T& x)
    T& entry(Integer pos);
    void append(Sequence1& sToAppend);
    void split(Integer pos,
               Sequence1& receivingS);

    Integer length(void);
private: // representation
    // ...
};
```



The job of the **destructor** is to reclaim any resources allocated to the Sequence variable during the variable's lifetime in the client program

Recall: The compiler guarantees that the destructor is automatically called just prior to the variable going out of scope in the client program

Resetting to Initial Value

```
template <class T>
class Sequence1
{
public: // Standard Operations
    Sequence1();
    //! alters self
    //! ensures: self = < >
    ~Sequence1();

    void clear(void);
    //! clears self

    void transferFrom(Sequence1& source);
    Sequence1& operator =(Sequence1& rhs);
// Sequence1 Specific Operations
    void add(Integer pos, T& x);
    void remove(Integer pos, T& x);
    void replaceEntry(Integer pos, T& x)
    T& entry(Integer pos);
    void append(Sequence1& sToAppend);
    void split(Integer pos,
                Sequence1& receivingS);

    Integer length(void);
private: // representation
    // ...
};
```

The job of the **clear** operation is to reset the Sequence variable back to its initial value

$s1 = \langle \rangle$

Recall: To determine the initial value of the variable, examine the constructor's ensures clause

Transferring a Value

```
template <class T>
class Sequence1
{
public: // Standard Operations
    Sequence1();
    ~Sequence1();

    void clear(void);

    void transferFrom(Sequence1& source);
        //! replaces self
        //! clears source
        //! ensures: self = #source

    Sequence1& operator =(Sequence1& rhs);
// Sequence1 Specific Operations
    void add(Integer pos, T& x);
    void remove(Integer pos, T& x);
    void replaceEntry(Integer pos, T& x)
    T& entry(Integer pos);
    void append(Sequence1& sToAppend);
    void split(Integer pos,
                Sequence1& receivingS);

    Integer length(void);
private: // representation
    // ...
};
```

The job of **transferFrom** is to move the value stored in parameter *source* to *self* and to clear *source*

Example:

```
typedef Sequence1<Integer> SequenceOfInteger;
SequenceOfInteger s1, s2;
...
// incoming s1 and s2
// s1 = <0,11,2>
// s2 = <73,105>
        s2.transferFrom(s1);
// outgoing s1 and s2
// s1 = <>
// s2 = <0,11,2>
```

Recall: transferFrom, moves the value, it does not copy it

Copying a Value

```
template <class T>
class Sequence1
{
public: // Standard Operations
    Sequence1();
    ~Sequence1();

    void clear(void);
    void transferFrom(Sequence1& source);

    Sequence1& operator =(Sequence1& rhs);
        //! replaces self
        //! restores rhs
        //! ensures: self = rhs

// Sequence1 Specific Operations
    void add(Integer pos, T& x);
    void remove(Integer pos, T& x);
    void replaceEntry(Integer pos, T& x)
    T& entry(Integer pos);
    void append(Sequence1& sToAppend);
    void split(Integer pos,
                Sequence1& receivingS);

    Integer length(void);
private: // representation
    // ...
};
```

The job of **operator =** is to copy the value stored in *rhs* to *self* and leave *rhs* unchanged

Example:

```
typedef Sequence1<Integer> SequenceOfInteger;
SequenceOfInteger s1, s2;
...
// incoming s1 and s2
// s1 = <0,11,2>
// s2 = <73,105>
    s2 = s1;
// outgoing s1 and s2
// s1 = <0,11,2>
// s2 = <0,11,2>
```

Recall: operator = uses infix syntax but the C++ compiler views the call using object oriented syntax:

```
s2 = s1; // infix
↓
s2.operator =(s1); // object oriented
```