A Detailed Explanation Of the Sequence Component

Part 4

Inspecting the State of the Sequence

The Sequence Component

```
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
   // ...
};
```

Two of the 7 *Sequence Specific Operations* have to do with inspecting the state of the Sequence

entry

```
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);
      //! restores self, pos
      //! requires: 0 ≤ pos < |self|
      //! ensures: <entry> =
      //!  self[pos, pos+1)

   void append(Sequence1& sToApppend);

   void split(Integer pos,
              Sequence1& receivingS);

   Integer length(void);

private: // representation

   // ...

};
```

The job of *entry* is to return to the client program a reference to the item stored in *self* at location *pos*

Example:

```
typedef Sequence1<Text> TextSeq;
TextSeq s1;
Integer k;
...
// incoming s1 and k
// s1 = <"C343","C251","C455"> and k = 0
cout << s1.entry(k);
// outgoing s1
// s1 = <"C343","C251","C455"> and k = 0
// "C343" is displayed by the cout statement
```

entry's return type

```
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);
      //! restores self, pos
      //! requires: 0 ≤ pos < |self|
      //! ensures: <entry> =
      //!  self[pos, pos+1)

   void append(Sequence1& sToApppend);

   void split(Integer pos,
              Sequence1& receivingS);

   Integer length(void);

private: // representation

   // ...

};
```

T& is *entry*'s return type

T, the template parameter, is the type of the item stored in *self*

Recall that *T* is specified by the client program by using C++'s *typedef* construct – in the example below *T* has been set by the client to be type *Text*

Example:

```
typedef Sequence1<Text> TextSeq;
TextSeq s1;
Integer k;
...
// incoming s1 and k
// s1 = <"C343","C251","C455"> and k = 0
cout << s1.entry(k);
// outgoing s1
// s1 = <"C343","C251","C455"> and k = 0
// "C343" is displayed by the cout statement
```

The & in entry's return type

```
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);
      //! restores self, pos
      //! requires: 0 ≤ pos < |self|
      //! ensures: <entry> =
      //!  self[pos, pos+1)

   void append(Sequence1& sToApppend);

   void split(Integer pos,
              Sequence1& receivingS);

   Integer length(void);

private: // representation

   // ...

};
```

The & in the return type indicates that *entry* returns a reference to (address of) item stored in *self* at location *pos*

In the example below, type *T* has been set by the client program to be *Text*, so *entry* returns to the *cout* statement a reference to a *Text* object – which is the address to the Text object "C343" in this example

Example:

```
typedef Sequence1<Text> TextSeq;
TextSeq s1;
Integer k;
...
// incoming s1 and k
// s1 = <"C343","C251","C455"> and k = 0
cout << s1.entry(k);
// outgoing s1
// s1 = <"C343","C251","C455"> and k = 0
// "C343" is displayed by the cout statement
```

entry's requires clause

```
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);
      //! restores self, pos
      //! requires: 0 ≤ pos < |self|
      //! ensures: <entry> =
      //!  self[pos, pos+1)

   void append(Sequence1& sToApppend);

   void split(Integer pos,
              Sequence1& receivingS);

   Integer length(void);

private: // representation

   // ...

};
```

entry requires that the location to be accessed in *self* by parameter *pos* be within the bounds of *self*

The client below is defective because the call to *entry* violates the requires clause

Example:

```
typedef Sequence1<Text> TextSeq;
TextSeq s1;
Integer k;
...
// incoming s1 and k
// s1 = <"C343","C251","C455"> and k = 10
cout << s1.entry(k);
// outgoing s1 and k
// s1 = ???
// k = ???
```

length

```
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);
      //! restores self
      //! ensures: length = |self|

private: // representation

   // ...

};
```

The job of *length* is to return to the client an Integer representing the number of items in *self*, and leave *self* unchanged

Example:

```
typedef Sequence1<Text> TextSeq;
TextSeq s1;
Integer z;
...
// incoming s1 and z
// s1 = <"C343","C251","C455","B438">
// z = 0
z = s1.length();
// outgoing s1 and z
// s1 = <"C343","C251","C455","B438">
// z = 4
```

length's return type

```
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);
      //! restores self
      //! ensures: length = |self|
private: // representation
   // ...
};
```

Integer is the return type of *length*

The client program must be written so that it *catches* the returned Integer

Below, the client catches the Integer with an assignment statement and stores it in *z*, the locally declared Integer variable

Example:

```
typedef Sequence1<Text> TextSeq;
TextSeq s1;
Integer z;
...
// incoming s1 and z
// s1 = <"C343","C251","C455","B438">
// z = 0
z = s1.length();
// outgoing s1 and z
// s1 = <"C343","C251","C455","B438">
// z = 4
```