Layering a Component A Detailed Example Using the Queue

Part 2 – Internal Representation

The Internal Representation

```
// Filename: Queue2.hpp
#pragma once
#include “List\List1.hpp”

template <class T>
class Queue2
{
public: // Standard Operations
   Queue2();
   ~Queue2();
   void clear (void);
   void transferFrom (Queue2& source);
   Queue2& operator = (Queue2& rhs);
// Queue2 Specific Operations
   void enqueue (T& x);
   void dequeue (T& x);
   void replacefront (T& x);
   T& front (void);
   Integer length (void);
private: // representation
   typedef List1<T> ListOfT;
   ListOfT s;
};
```

internal representation – dictates the format in which data is stored (or represented) internally inside the component

concrete representation – this term is interchangeable with *internal representation*

In the private part of the template class is the location where the declarations for layering a component are made

The Internal Representation

```
// Filename: Queue2.hpp
#pragma once
#include “List\List1.hpp”

template <class T>
class Queue2

{

public: // Standard Operations

   Queue2();

   ~Queue2();

   void clear (void);

   void transferFrom (Queue2& source);

   Queue2& operator = (Queue2& rhs);

// Queue2 Specific Operations

   void enqueue (T& x);

   void dequeue (T& x);

   void replacefront (T& x);

   T& front (void);

   Integer length (void);

private: // representation

   typedef List1<T> ListOfT;
   ListOfT s;

};
```

private:

- The private keyword tells the C++ compiler that the client program is *not* allowed any access to the data members or data types that are declared in this section
- Compiler enforced *encapsulation* – the compiler will raise a compiler error for those lines of code in the client program that attempt to access internal data members or declared types

The Internal Representation

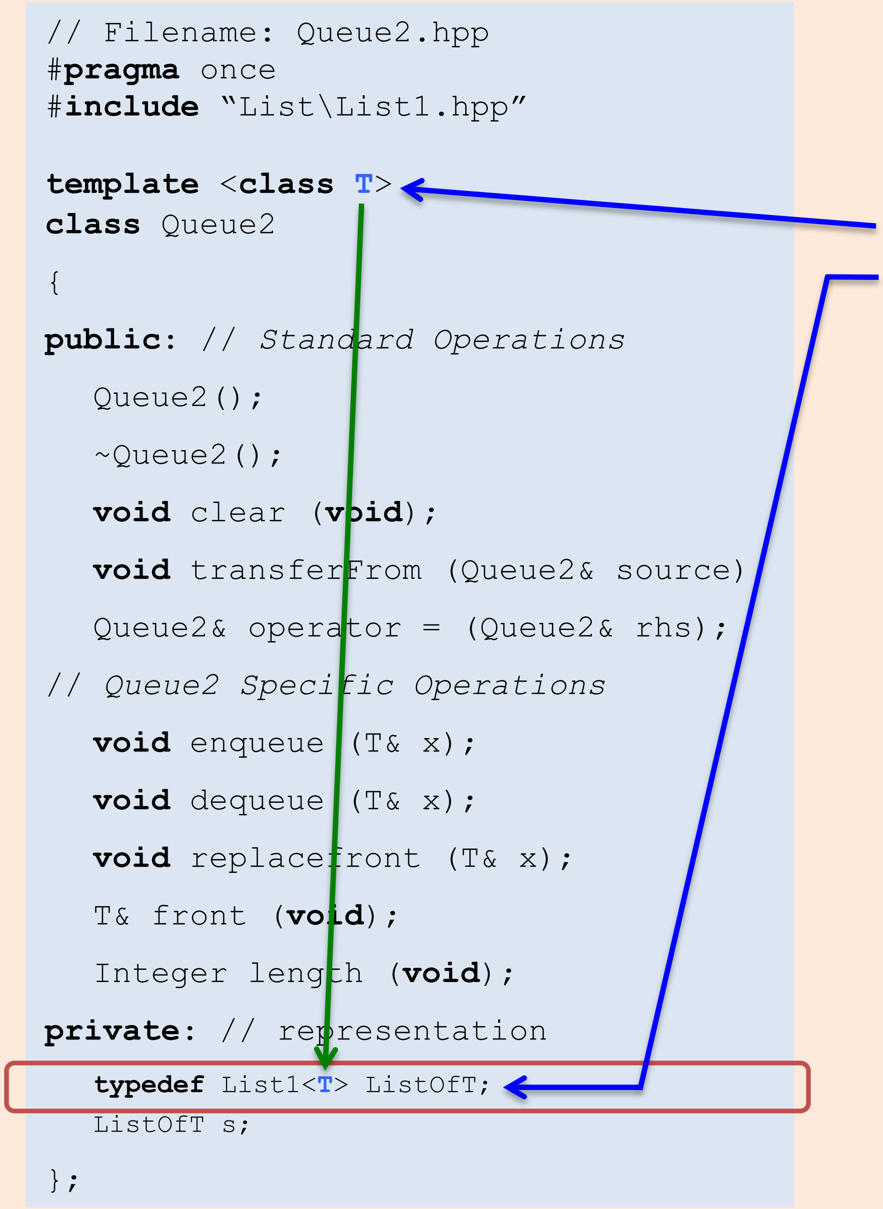
```
// Filename: Queue2.hpp
#pragma once
#include "List\List1.hpp"

template <class T>
class Queue2

{

public: // Standard Operations

   Queue2();

   ~Queue2();

   void clear (void);

   void transferFrom (Queue2& source);

   Queue2& operator = (Queue2& rhs);

// Queue2 Specific Operations

   void enqueue (T& x);

   void dequeue (T& x);

   void replacefront (T& x);

   T& front (void);

   Integer length (void);

private: // representation

   typedef List1<T> ListOfT;
   ListOfT s;

};
```

- When performing component layering, there are two parts that appear in the declarations of the internal representation:

  1. Instance creation – using the **typedef** statement to create an instance of the layered upon component

  2. Data member declaration – declaring internal data members from the instances created in Step 1 (above)

The Internal Representation



```
// Filename: Queue2.hpp
#pragma once
#include "List\List1.hpp"

template <class T>
class Queue2
{
public: // Standard Operations
   Queue2();
   ~Queue2();
   void clear (void);
   void transferFrom (Queue2& source)
   Queue2& operator = (Queue2& rhs);
// Queue2 Specific Operations
   void enqueue (T& x);
   void dequeue (T& x);
   void replaceFront (T& x);
   T& front (void);
   Integer length (void);
private: // representation
   typedef List1<T> ListOfT;
   ListOfT s;
};
```

Instance Creation:

- The **T** is the template parameter to Queue2
- ListOfT is an instance created in the private part of Queue2 using List1 and **T**

```
// Filename: Queue2.hpp
#pragma once
#include "List\List1.hpp"

template <class T>
class Queue2
{
public: // Standard Operations
   Queue2();
   ~Queue2();
   void clear (void);
   void transferFrom (Queue2& source);
   Queue2& operator = (Queue2& rhs);
// Queue2 Specific Operations
   void enqueue (T& x);
   void dequeue (T& x);
   void replaceFront (T& x);
   T& front (void);
   Integer length (void);
private: // representation
   typedef List1<T> ListOfT;
   ListOfT s;
};
```

The Internal Representation

Client of Queue2:

- The client below creates an instance of Queue2 and supplies **Integer** as the template parameter
- Queue2's **T** = **Integer**, and therefore the internal representation *ListOfT* is an instance of *List1<Integer>*

Example client:

```
{
   typedef Queue2<Integer> IntegerQueue;
   IntegerQueue q1;
   Integer y1;
   // ...
   // Suppose q1 = <3,88>
   y1 = 5;
   q1.enqueue(y1);
}
```

```
// Filename: Queue2.hpp
#pragma once
#include “List\List1.hpp”

template <class T>
class Queue2

{

public: // Standard Operations

   Queue2();

   ~Queue2();

   void clear (void);

   void transferFrom (Queue2& source);

   Queue2& operator = (Queue2& rhs);

// Queue2 Specific Operations

   void enqueue (T& x);

   void dequeue (T& x);

   void replacefront (T& x);

   T& front (void);

   Integer length (void);

private: // representation

   typedef List1<T> ListOfT;
   ListOfT s;

};
```

The Internal Representation

s is the data member declared from the instance of *ListOfT*

All data of type T inserted into a Queue variable (when a client program calls *enqueue*), will be stored internally in the data member *s*

Example:

- In the client below the value 5 is enqueued onto *q1*

- *enqueue*’s implementation will store the value 5 in Queue2’s internal data member, ListOfT *s*

Example client:

```
{
   typedef Queue2<Integer> IntegerQueue;
   IntegerQueue q1;
   Integer y1;
   // ...
   // Suppose q1 = <3,88>
   y1 = 5;
   q1.enqueue(y1);
}
```

The Internal Representation

```
// Filename: Queue2.hpp
#pragma once
#include "List\List1.hpp"

template <class T>
class Queue2
{
public: // Standard Operations
   Queue2();
   ~Queue2();
   void clear (void);
   void transferFrom (Queue2& source);
   Queue2& operator = (Queue2& rhs);
// Queue2 Specific Operations
   void enqueue (T& x);
   void dequeue (T& x);
   void replacefront (T& x);
   T& front (void);
   Integer length (void);
private: // representation
   typedef List1<T> ListOfT;
   ListOfT s;
};
```

Compiler enforced encapsulation – revisited

- The client program below will not compile correctly because the client is attempting to gain access to the privately declared data member *s*
- When the C++ compiler does not allow client access to private members, this is called *compiler enforced encapsulation*
- Data member *s* as well as the instance *ListOfT* are said to be *encapsulated*

Example client:

```
{
   typedef Queue2<Integer> IntegerQueue;
   IntegerQueue q1;
   Integer z;
   // ...
   z = q1.s.rightLength();
}
```

The Internal Representation

```
// Filename: Queue2.hpp
#pragma once
#include “List\List1.hpp”

template <class T>
class Queue2
{
public: // Standard Operations
   Queue2();
   ~Queue2();
   void clear (void);
   void transferFrom (Queue2& source);
   Queue2& operator = (Queue2& rhs);
// Queue2 Specific Operations
   void enqueue (T& x);
   void dequeue (T& x);
   void replacefront (T& x);
   T& front (void);
   Integer length (void);
private: // representation
   typedef List1<T> ListOfT;
   ListOfT s;
   //! correspondence:
   //!   self = s.left * s.right
};
```

The *correspondence*:

- Specifies how the abstract value (i.e., *self*) can be obtained from the value stored in the internal data member.
- In this example the list *s* is the internal data member
- The correspondence for Queue2 layered on List is:

```
self = s.left * s.right
```

The Internal Representation

```
// Filename: Queue2.hpp
#pragma once
#include "List\List1.hpp"

template <class T>
class Queue2
{
public: // Standard Operations
   Queue2();
   ~Queue2();
   void clear (void);
   void transferFrom (Queue2& source);
   Queue2& operator = (Queue2& rhs);
// Queue2 Specific Operations
   void enqueue (T& x);
   void dequeue (T& x);
   void replacefront (T& x);
   T& front (void);
   Integer length (void);
private: // representation
   typedef List1<T> ListOfT;
   ListOfT s;
   //! correspondence:
   //!   self = s.left * s.right
};
```

Example of *correspondence*: `self = s.left * s.right`

- In the client below, the outgoing q1 = <3,88,5>
- Internally to the client's variable *q1*, the list *s* could be configured in any one the following four ways:
  1. s = (<>,<3,88,5>)
  2. s = (<3>,<88,5>)
  3. s = (<3,88>,<5>)
  4. s = (<3,88,5>,<>)
- Why? Because the correspondence indicates that to obtain the abstract value for *self* (i.e., *q1* in this example), we just concatenate the concrete s.left with s.right

Example client:

```
{
   typedef Queue2<Integer> IntegerQueue;
   IntegerQueue q1;
   Integer y1 = 5;
   // ...
   // Suppose q1 = <3,88>
   q1.enqueue(y1);
   // Outgoing q1 = <3,88,5>
}
```

The Internal Representation

```
// Filename: Queue2.hpp
#pragma once
#include “List\List1.hpp”

template <class T>
class Queue2
{
public: // Standard Operations
   Queue2();
   ~Queue2();
   void clear (void);
   void transferFrom (Queue2& source);
   Queue2& operator = (Queue2& rhs);
// Queue2 Specific Operations
   void enqueue (T& x);
   void dequeue (T& x);
   void replacefront (T& x);
   T& front (void);
   Integer length (void);
private: // representation
   typedef List1<T> ListOfT;
   ListOfT s;
   //! correspondence:
   //!   self = s.left * s.right
};
```

internal contract:

Where does the correspondence have meaning?

- It is a contract that has meaning only within the component – therefore it is known as an internal contract

- It has no meaning to the client program of Queue because the client program is not permitted to access the internal data members, i.e., *s* in this example

The Internal Representation

```
// Filename: Queue2.hpp
#pragma once
#include “List\List1.hpp”

template <class T>
class Queue2
{
public: // Standard Operations
   Queue2();
   ~Queue2();
   void clear (void);
   void transferFrom (Queue2& source);
   Queue2& operator = (Queue2& rhs);
// Queue2 Specific Operations
   void enqueue (T& x);
   void dequeue (T& x);
   void replacefront (T& x);
   T& front (void);
   Integer length (void);
private: // representation
   typedef List1<T> ListOfT;
   ListOfT s;
   //! correspondence:
   //!   self = s.left * s.right
};
```

internal contract:

Who must follow this internal contract?

- The component implementer, i.e., the implementer of Queue2 in this example

When is this internal contract used by a component implementer?

1. When an exported operation (e.g., *enqueue*) is called:

   The implementer can *assume* the correspondence holds

2. When a called exported operation (e.g., *enqueue*) terminates:

   The implementer must *guarantee* that the correspondence holds

Member Function Implementations

```
// Filename: Queue2.hpp
#pragma once
#include “List\List1.hpp”

template <class T>
class Queue2
{
public: // Standard Operations
   Queue2();
   ~Queue2();
   void clear (void);
   void transferFrom (Queue2& source);
   Queue2& operator = (Queue2& rhs);
// Queue2 Specific Operations
   void enqueue (T& x);
   ...
   Integer length (void);
private: // representation
   typedef List1<T> ListOfT;
   ListOfT s;
};

// Member function implementations
//   not shown in this example
```

- The component implementer places all of the component’s member function implementations at the bottom of the component’s .hpp file

- There is one member function implementation for each header declared in the template class definition

- In this example there are 10 member function implementations:
  - One each for the 5 Standard Operations
  - One each for the 5 Queue Specific Operations