

Queue

Constructor & Destructor
Two of the 5 Standard Operations


The Queue Component

Let's look at the 2 of the 5 Standard Operations

All C++ components will have a *constructor* and *destructor*

```
template <class T>
class Queue1
{
public: // Standard Operations
    Queue1();
    ~Queue1();

    void clear (void);
    void transferFrom (Queue1& source);
    Queue1& operator = (Queue1& rhs);
// Queue1 Specific Operations
    void enqueue (T& x);
    void dequeue (T& x);
    void replaceFront (T& x);
    T& front (void);
    Integer length (void);
private: // representation
    // ...
};
```



```

template <class T>
class Queue1
{
public: // Standard Operations
    Queue1();
        //! replaces self
        //! ensures: self = <>

    ~Queue1();

    void clear (void);

    void transferFrom (Queue1& source);
    Queue1& operator = (Queue1& rhs);
// Queue1 Specific Operations
    void enqueue (T& x);
    void dequeue (T& x);
    void replaceFront (T& x);
    T& front (void);
    Integer length (void);
private: // representation
    // ...
};

```

Queue's constructor



The *constructor* always has the same name as the class name, in this example, Queue1

Queue's constructor

```
template <class T>
class Queue1
{
public: // Standard Operations
    Queue1();
        //! replaces self
        //! ensures: self = <>
    ~Queue1();
    void clear (void);
    void transferFrom (Queue1& source);
    Queue1& operator = (Queue1& rhs);
// Queue1 Specific Operations
    void enqueue (T& x);
    void dequeue (T& x);
    void replaceFront (T& x);
    T& front (void);
    Integer length (void);
private: // representation
    // ...
};
```

← The spec's ensures clause indicates that the constructor for Queue initializes *self* to the empty string

replaces Parameter Mode

```
template <class T>
class Queue1
{
public: // Standard Operations
    Queue1();
        //! replaces self
        //! ensures: self = <>
    ~Queue1();
    void clear (void);
    void transferFrom (Queue1& source);
    Queue1& operator = (Queue1& rhs);
// Queue1 Specific Operations
    void enqueue (T& x);
    void dequeue (T& x);
    void replaceFront (T& x);
    T& front (void);
    Integer length (void);
private: // representation
    // ...
};
```

← The *replaces* parameter mode indicates that the empty queue initial value will take the place of whatever value was in the controlling object prior to the call

Queue's constructor

```
template <class T>
class Queue1
{
public: // Standard Operations
    Queue1();
    //! replaces self
    //! ensures: self = <>

    ~Queue1();
    void clear (void);
    void transferFrom (Queue1& source);
    Queue1& operator = (Queue1& rhs);
// Queue1 Specific Operations
    void enqueue (T& x);
    void dequeue (T& x);
    void replaceFront (T& x);
    T& front (void);
    Integer length (void);
private: // representation
    // ...
};
```

The compiler guarantees that the constructor will be called when a Queue variable is declared

Queue's constructor

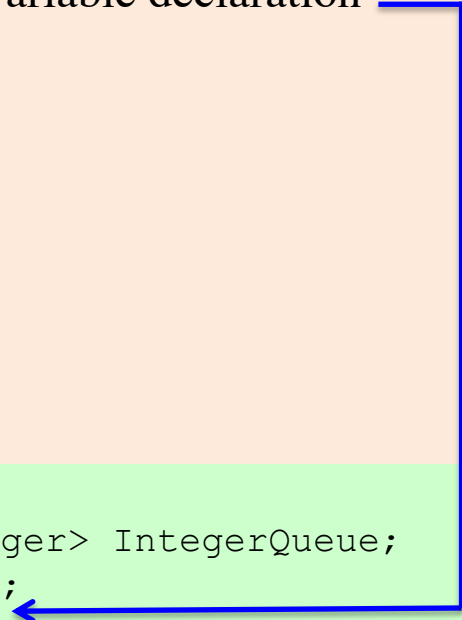
```
template <class T>
class Queue1
{
public: // Standard Operations
    Queue1();
        //! replaces self
        //! ensures: self = <>
    ~Queue1();
    void clear (void);
    void transferFrom (Queue1& source);
    Queue1& operator = (Queue1& rhs);
// Queue1 Specific Operations
    void enqueue (T& x);
    void dequeue (T& x);
    void replaceFront (T& x);
    T& front (void);
    Integer length (void);
private: // representation
    // ...
};
```

The C++ compiler automatically inserts a call to the constructor for each variable immediately after the variable declaration

Example client:

```
{
1  typedef Queue1<Integer> IntegerQueue;
2  IntegerQueue q1, q2;
}

```



Queue's constructor

```
template <class T>
class Queue1
{
public: // Standard Operations
    Queue1();
        //! replaces self
        //! ensures: self = <>
    ~Queue1();
    void clear (void);
    void transferFrom (Queue1& source);
    Queue1& operator = (Queue1& rhs);
// Queue1 Specific Operations
    void enqueue (T& x);
    void dequeue (T& x);
    void replaceFront (T& x);
    T& front (void);
    Integer length (void);
private: // representation
    // ...
};
```

- If we could see the *implicit* calls to the constructor (which we cannot), they would look like the following
- The programmer does not write this code, it is inserted automatically by the C++ compiler during compilation

Example client:

```
{
1  typedef Queue1<Integer> IntegerQueue;
2  IntegerQueue q1, q2;
3  q1.Queue1();
4  q2.Queue1();
}
```

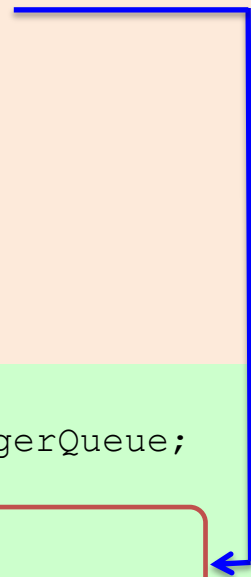

Queue's constructor

```
template <class T>
class Queue1
{
public: // Standard Operations
    Queue1();
        //! replaces self
        //! ensures: self = <>
    ~Queue1();
    void clear (void);
    void transferFrom (Queue1& source);
    Queue1& operator = (Queue1& rhs);
// Queue1 Specific Operations
    void enqueue (T& x);
    void dequeue (T& x);
    void replaceFront (T& x);
    T& front (void);
    Integer length (void);
private: // representation
    // ...
};
```

- By utilizing the constructor's spec, we can reason about the initial values of q1 and q2
- A comment containing the constructor's ensures clause has been added

Example client:

```
{
1  typedef Queue1<Integer> IntegerQueue;
2  IntegerQueue q1, q2;
3  q1.Queue1(); // self = <>
4  q2.Queue1(); // self = <>
}
```



Queue's constructor

```
template <class T>
class Queue1
{
public: // Standard Operations
    Queue1();
        //! replaces self
        //! ensures: self = <>
    ~Queue1();
    void clear (void);
    void transferFrom (Queue1& source);
    Queue1& operator = (Queue1& rhs);
// Queue1 Specific Operations
    void enqueue (T& x);
    void dequeue (T& x);
    void replaceFront (T& x);
    T& front (void);
    Integer length (void);
private: // representation
    // ...
};
```

- Substitute q1 and q2 for *self* in the first and second ensures clauses
- After the substitution, we see that the ensures clause guarantees that q1 and q2 are both initialized to the empty string

Example client:

```
{
    typedef Queue1<Integer> IntegerQueue;
    IntegerQueue q1, q2;
    q1.Queue1(); // q1 = <>
    q2.Queue1(); // q2 = <>
}
```

Queue's constructor

```
template <class T>
class Queue1
{
public: // Standard Operations
    Queue1();
        //! replaces self
        //! ensures: self = <>
    ~Queue1();
    void clear (void);
    void transferFrom (Queue1& source);
    Queue1& operator = (Queue1& rhs);
// Queue1 Specific Operations
    void enqueue (T& x);
    void dequeue (T& x);
    void replaceFront (T& x);
    T& front (void);
    Integer length (void);
private: // representation
    // ...
};
```

- As a reminder, the calls to the constructor are automatically inserted by the compiler and not explicitly written by the programmer
- Below a comment has been inserted indicating the initial values of q1 and q2

Example client:

```
{
1  typedef Queue1<Integer> IntegerQueue;
2  IntegerQueue q1, q2;
3  // q1 = <> and q2 = <>
}
```

```

template <class T>
class Queue1
{
public: // Standard Operations
    Queue1();
    ~Queue1();
    void clear (void);
    void transferFrom (Queue1& source);
    Queue1& operator = (Queue1& rhs);
    // Queue1 Specific Operations
    void enqueue (T& x);
    void dequeue (T& x);
    void replaceFront (T& x);
    T& front (void);
    Integer length (void);
private: // representation
    // ...
};

```

Queue's destructor



The job of the destructor is to return to the run-time system any resources that were allocated to the Queue variable during the variable's lifetime

Queue's destructor

```
template <class T>
class Queue1
{
public: // Standard Operations
    Queue1();
    ~Queue1();
    void clear (void);
    void transferFrom (Queue1& source);
    Queue1& operator = (Queue1& rhs);
// Queue1 Specific Operations
    void enqueue (T& x);
    void dequeue (T& x);
    void replaceFront (T& x);
    T& front (void);
    Integer length (void);
private: // representation
    // ...
};
```

The compiler guarantees that the destructor is called just prior to the variable going out of scope

Example client:

```
{
1  typedef Queue1<Integer> IntegerQueue;
2  IntegerQueue q1, q2;
3  // q1 = <> and q2 = <>
4
} ←
```

Queue's destructor

```
template <class T>
class Queue1
{
public: // Standard Operations
    Queue1();
    ~Queue1();
    void clear (void);
    void transferFrom (Queue1& source);
    Queue1& operator = (Queue1& rhs);
// Queue1 Specific Operations
    void enqueue (T& x);
    void dequeue (T& x);
    void replaceFront (T& x);
    T& front (void);
    Integer length (void);
private: // representation
    // ...
};
```

- If we could see the *implicit* calls to the destructor (which we cannot), they would look like the following
- The programmer does not write this code, it is inserted automatically by the C++ compiler during compilation

Example client:

```
{
1  typedef Queue1<Integer> IntegerQueue;
2  IntegerQueue q1, q2;
3
4  q1.~Queue1();
5  q2.~Queue1(); ←
}
```