ABSTRACT DATA TYPES

- **Overview**
  - Abstract Data Types
  - Stack, Queue, and Deque
- **References**
  - Richard F. Gilberg and Behrouz A. Forouzan: Data Structures - A Pseudocode Approach with C. 2nd Edition. Thomson (2005)
  - Russ Miller and Laurence Boxer: Algorithms Sequential & Parallel. 2nd Edition. Charles River Media Inc. (2005)
  - Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo: C++ Primer. 5th Edition. Addison-Wesley (2013)
  - Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, "Introduction to Algorithms", 4th Edition, The MIT Press (2022)
  - Scott Meyers: Effective Modern C++. O'Reilly (2014)
  - Anthony Williams: C++ Concurrency in Action - Practical Multithreading. Manning Publications Co. (2012)

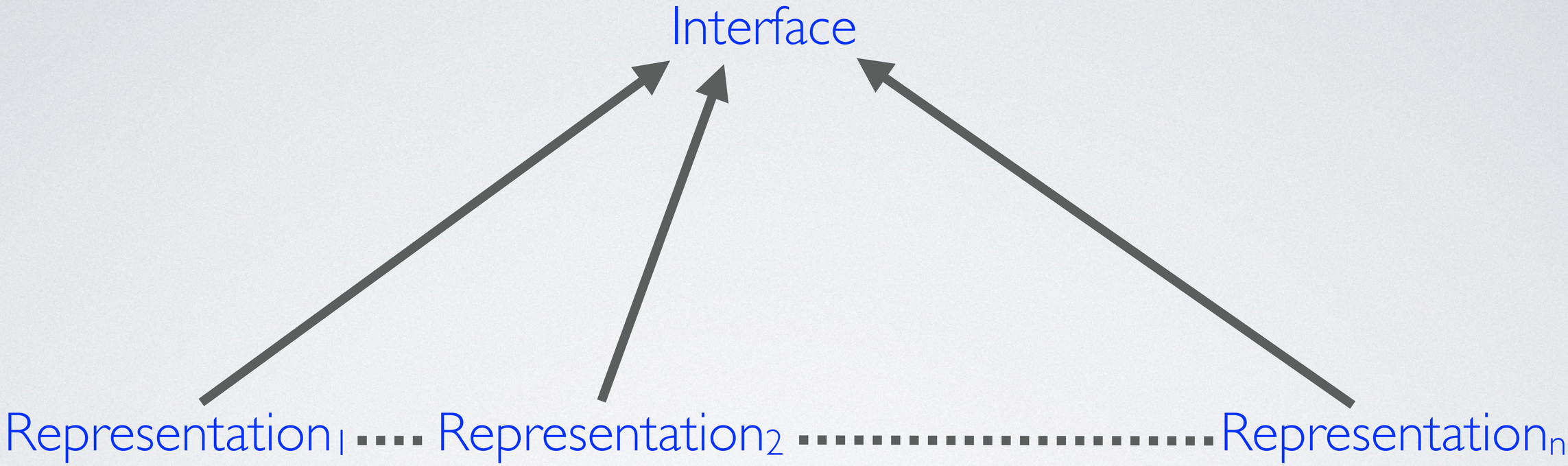
DATA ABSTRACTION

- Data abstraction divides a data type into two pieces:
  - An interface that tells us what values the data type represents, what the operations on the data are, and what properties these operations may be relied on to have.
  - The implementation provides a specific representation of the values, the operations supported, and the properties of the abstract data type.
- Entities that are abstract in this way are called abstract data types. A client of an abstract data type manipulates the new data only through the operations specified in the interface. If we wish to change the representation, we change the implementation of the operations in the interface.

OBJECT-ORIENTED ENCAPSULATION

- Object-oriented encapsulation is a principle that provides an effective means to construct abstract data types:
  - All member variables, the state of an object, have private visibility.
  - All member functions and the operations defined in the interface have public visibility.
- **Note**: The extent to which this scheme is used can differ.
- Object-oriented encapsulation does not necessarily mean that you cannot see the implementation. There are no pragmatic ways to use this knowledge.

REPRESENTATION STRATEGIES FOR ABSTRACT DATA TYPES



- Given an interface for a data type, we can change the underlying representation if needed using different strategies.

REQUIRED OPERATIONS

- To create values, verify that a given value is of the desired data type, and manipulate and access data, we need the following ingredients:
  - At least one constructor that allows us to create values of a given data type,
  - A type predicate that tests whether a given value is a representation of a particular data type, and
  - Some access procedures and manipulators allow us to extract particular information from a given representation and to update data of given value, respectively.

COPY AND MOVE SEMANTICS

- In addition to the required operations, we need to address copy semantics and move semantics of objects when using C++.
- Both relate to data transfer from one object to another, using shallow-copy or deep-copy, and whether the data is an lvalue or an rvalue reference.
- From a purely object-oriented viewpoint, copy semantics preserves control over object ownership. If the source object is an lvalue reference, copy semantics entails a copy constructor and a copy-assignment operator.
- Move semantics handles data transfer from source objects whose lifetime expires. If the source object is an rvalue reference, move semantics entails a move constructor and a move-assignment operator.

CANONICAL METHODS

- **Object creation and deletion**:
  - default constructor
  - destructor
- **Copy semantics**:
  - copy constructor
  - copy-assignment
- **Move semantics**:
  - move constructor
  - move-assignment

RVALUE REFERENCES

```
#define rvalue 42

int i = rvalue;              // use R-value to initialize i
int&& r1 = rvalue;           // R-value reference to a literal expression
int&& r2 = i * rvalue;       // R-value reference to result of multiplication
```

- To support move operations, rvalue references have been added to C++.
- An rvalue reference is a reference that must be bound to an rvalue (i.e, an expression that is not an lvalue like literals or results of function and operator call that return non-references).
- An rvalue reference is obtained by using && rather than &.
- Rvalue references may be bound only to an object about to be destroyed (i.e, whose lifetime expires).

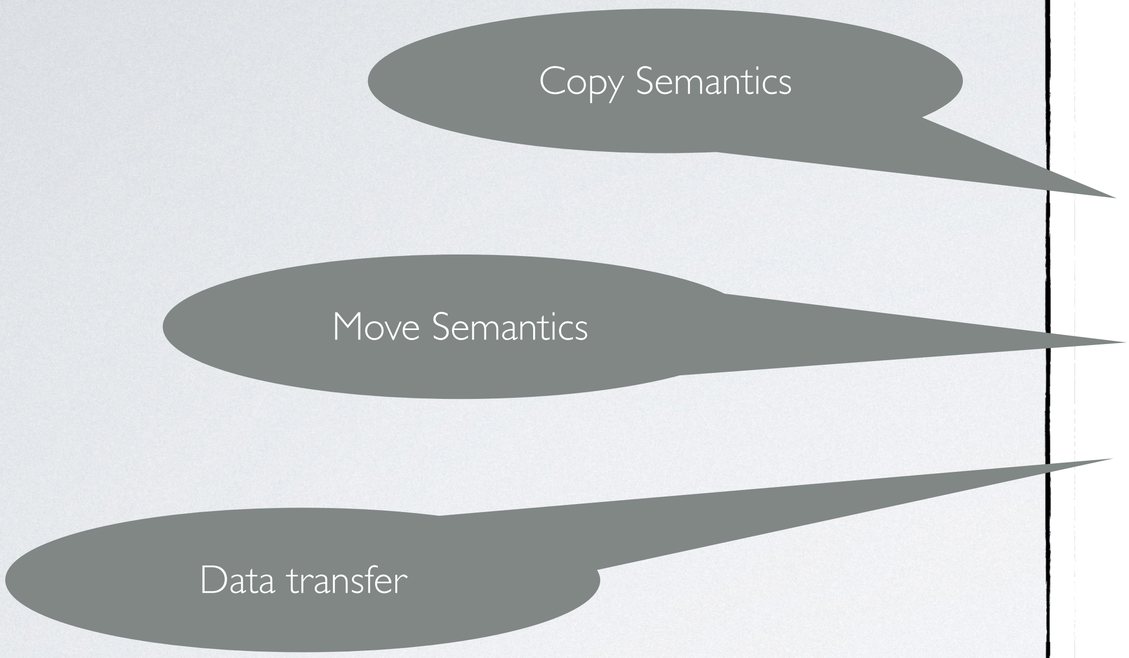
SPECIAL MEMBER FUNCTION RULES

- In C++, the compiler automatically generates the default constructor, copy constructor, copy-assignment operator, and destructor for a type if it does not declare its own. However,
  - If any constructor is explicitly declared, no default constructor is automatically generated.
  - If a virtual destructor is explicitly declared, no default destructor is automatically generated.
  - If a move constructor or move-assignment operator is explicitly declared, then:
    - No copy constructor is automatically generated.
    - No copy-assignment operator is automatically generated.
  - If a copy constructor, copy-assignment operator, move constructor, move-assignment operator, or destructor is explicitly declared, then:
    - No move constructor is automatically generated.
    - No move-assignment operator is automatically generated.

DYNAMIC STACK

- DynamicStack is an abstract data type.

DYNAMIC STACK




```
template<typename T>
class DynamicStack
{
public:
   // member type definitions relevant to DynamicStack

   DynamicStack();
   ~DynamicStack();

   DynamicStack( const DynamicStack& aOther );
   DynamicStack& operator=( const DynamicStack& aOther  );

   DynamicStack( DynamicStack&& aOther ) noexcept;
   DynamicStack& operator=( DynamicStack&& aOther );

   void swap( DynamicStack& aOther ) noexcept;

   std::optional<T> top() noexcept;
   void push( const T& aValue );
   void pop();

private:
   T* fElements;
   size_t fStackPointer;
   size_t fCapacity;

   void resize( size_t aNewCapacity );
   void ensure_capacity();
   void adjust_capacity();
};
```

- We extend the class DynamicStack, studied earlier, with copy and move semantics.

DYNAMICSTACK SWAP

```
void swap( DynamicStack& aOther ) noexcept
{
    std::swap( fElements, aOther.fElements );
    std::swap( fStackPointer, aOther.fStackPointer );
    std::swap( fCapacity, aOther.fCapacity );
}
```

- The swap function handles the data transfer between objects.
- The swap function never throws an exception. For this reason, it is marked **noexcept**.
- We need to swap the data element-wise to prevent an infinite recursion. The standard library defines std::swap using move semantics for the exchange of arguments.

DYNAMICSTACK COPY CONSTRUCTOR

```
DynamicStack( const DynamicStack& aOther ) :
    fElements(new T[aOther.fCapacity]),
    fStackPointer(aOther.fStackPointer),
    fCapacity(aOther.fCapacity)
{
    assert( fStackPointer <= fCapacity );

    for ( size_t i = 0; i < fStackPointer; i++ )
    {
        fElements[i] = aOther.fElements[i];
    }
}
```

- The copy constructor creates a fresh object and initializes all member variables by copying the data from the source object:
  - We allocate a new heap space to copy the data into. We use the values from the source object aOther.
  - The **for**-loop performs an element-wise deep-copy of the stack content. We only have to copy the elements up to the stack pointer.
  - The body of the copy constructor starts with an assert safety check verifying that the source stack is well-formed. The assert statement is a necessary input to the compiler's code analysis.

DYNAMICSTACK COPY-ASSIGNMENT

```
DynamicStack& operator=( const DynamicStack<T>& aOther )
{
    if ( this != &aOther )                          // self-assignment check
    {
        this->~DynamicStack();                      // free this stack

        new (this) DynamicStack( aOther );          // in-place new copy initialization
    }

    return *this;                                   // return updated object
}
```

- The copy-assignment operator always starts with a test for self-assignment. In this case, we return **this** object. If the test is missing, this might lead to a memory leak, especially if the object maintains heap memory. In this case, self-assignment would first free the memory and then attempt to assign the freed memory to **this** object.
- DynamicStack requires Deep-Copy data transfer. This can be achieved in two steps:
  - First, we free all resources acquired by **this** object by calling the class destructor on **this** object. It frees all resources but does not delete **this** object.
  - Second, we use in-place **new** copy initialization to perform a deep copy to transfer the data from aOther to **this** object. The in-place **new** operator applied to a pointer does not allocate new memory. Instead, the existing memory is reinitialized with the copy constructor.

DYNAMICSTACK MOVE CONSTRUCTOR

```
DynamicStack( DynamicStack<T>&& aOther ) noexcept :
    DynamicStack()
{
    swap( aOther );      // swap idiom
}
```

- The move constructor initializes all member variables by "stealing" the memory of the source object aOther.
- First, we initialize **this** object via the default constructor. This guarantees that all member variables are initialized with sensible values.
- Next, we swap the objects. This operation is guaranteed to succeed, hence the move constructor in marked **noexcept**. When the move constructor finishes, the lifetime of the source object aOther expires and it is destroyed.

DYNAMICSTACK MOVE-ASSIGNMENT

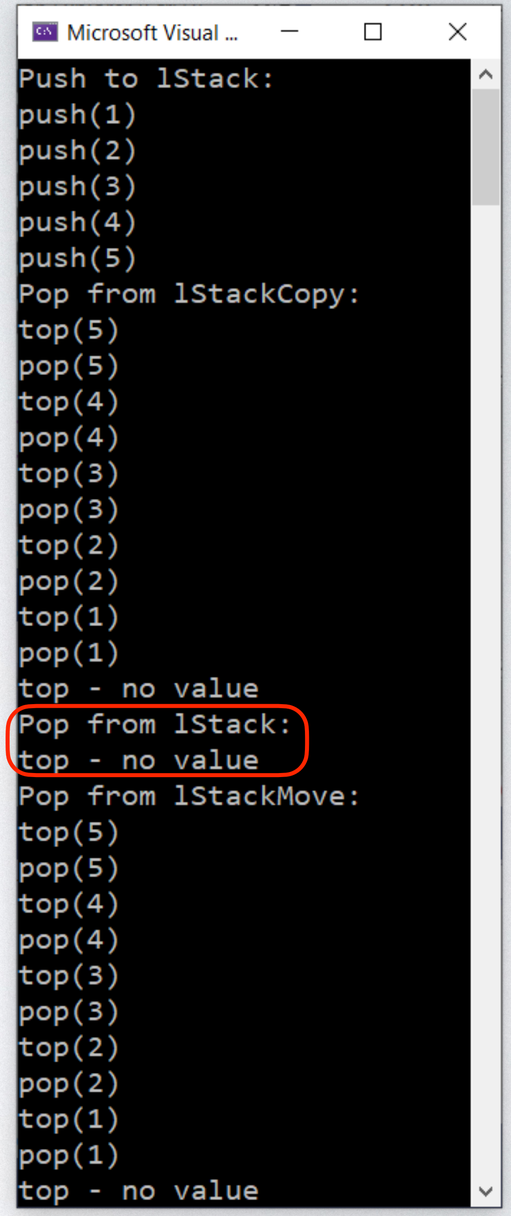
```
DynamicStack& operator=( DynamicStack<T>&& aOther ) noexcept
{
    if ( this != &aOther )   // self-assignment check
    {
        swap( aOther );      // swap idiom
    }

    return *this;
}
```

- The move-assignment operator, like the move constructor, "steals" the memory of the source object.
- First, we have to perform a self-assignment check. Next, we swap the objects, if necessary. This operation is guaranteed to succeed, hence the move-assignment operator in marked **noexcept**.
- **This** object is initialized. Its data is transferred to the source object, aOther, and vice versa. When the move-assignment operator finishes, the lifetime of the source object aOther expires and it is destroyed.

DYNAMICSTACK TEST

```
DynamicStack<int> lStack;

std::cout << "Push to lStack:" << std::endl;

for ( int i = 1; i < 6; i++ )
        lStack.push(i);

DynamicStack<int> lStackCopy = lStack;

std::cout << "Pop from lStackCopy:" << std::endl;

for ( auto lValue = lStackCopy.top(); lValue; )
{
    lStackCopy.pop();
    lValue = lStackCopy.top();
}

DynamicStack<int> lStackMove = std::move(lStack);

std::cout << "Pop from lStack:" << std::endl;

for ( auto lValue = lStack.top(); lValue; )
{
    lStack.pop();
    lValue = lStack.top();
}

std::cout << "Pop from lStackMove:" << std::endl;

for ( auto lValue = lStackMove.top(); lValue; )
{
    lStackMove.pop();
    lValue = lStackMove.top();
}
```

Warning: use of a moved from object.



STD::MOVE()

- The helper function std::move(arg) forces move semantics on arg, even if arg is an lvalue or lvalue reference.
- It is a compile-time function. No executable code is generated.
- The function std::move performs a type cast so that arg becomes an rvalue reference.

```
DynamicStack<int> IStackCopy = IStack;          // IStack is copied
DynamicStack<int> IStackMove = std::move(IStack);   // IStack is moved
```

TEMPLATE ARGUMENT DEDUCTION

- By default, the C++ compiler uses the arguments in a call to determine the template parameters for a function template.
- Determining the template arguments from the function arguments is template argument deduction.
- During template argument deduction, the compiler uses types of the arguments in the call to find the template arguments that generate a version of the function that best matches the given call.
- There are scenarios in which we must assist the compiler or explicitly specify template arguments. This includes also forwarding of arguments with their type unchanged.t

TYPE DEDUCTION EXAMPLE

```
template<typename T>
void g( T& v ) { std::cout << "g(T&)\n"; }

template<typename T>
void g( const T& v ) { std::cout << "g(const T&)\n"; }

template<typename T>
void g( T&& v ) { std::cout << "g(T&&)\n"; }

template<typename T>
class Foo
{
public:

    void f( T& v ) { std::cout << "f(T&)\n"; }
    void f( const T& v ) { std::cout << "f(const T&)\n"; }
    void f( T&& v ) { std::cout << "f(T&&)\n"; }

    template<typename U>
    void f( U&& v ) { std::cout << "ff(U&& " << v << ") -> "; g( v ); }

    template<typename U>
    void ff( U&& v ) { std::cout << "ff(U&& " << v << ") -> "; g( std::forward<U>(v) ); }
};
```

- Template function g is an overloaded function for lvalue references, constant lvalue references, and rvalue references.
- The template class Foo defines four overloaded member functions f for lvalue references, constant lvalue references, an rvalue reference, and a universal reference.
- The member function ff takes a universal reference and performs perfect forwarding to function g.

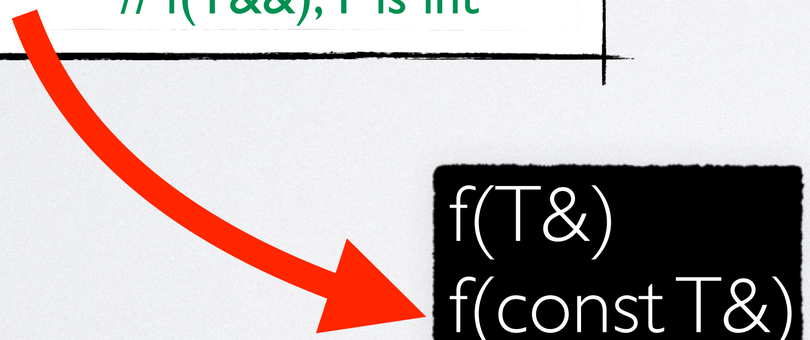
STD::FORWARD()

```
void f( const int& p ) { std:: cout << "[l-value]" }
void f ( int&& p) { std:: cout << "[r-value]" }

template<typename T> void f3(T&& p)
{
    f( p );
    f( std::forward<T>( p ) );
    std::cout << std::endl;
}

int a = 2;
f3( a ) ;          // [l-value][l-value]
f3( 4 );           // [l-value][r-value]
```

- The helper function std::forward(arg) allows perfect forwarding on arg.
- No executable code is generated. It is a compile-time function.
- It returns an rvalue reference to arg if arg is not an lvalue. If arg is an lvalue reference, it returns arg unchanged.

TYPE DEDUCTION: LVALUES AND RVALUES

```
Foo<int> obj;
int v = 20;
const int& cv = v;
double d = 2.2;
const double& cd = d;

obj.f( v );          // f(T&), T is int
obj.f( cv );         // f(const T&), T is int
obj.f( 20 );         // f(T&&), T is int
```

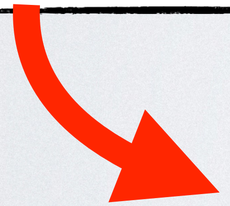


f(T&)
f(const T&)
f(T&&)

- In obj.f(v), v is an lvalue being passed as an lvalue reference.
- In obj.f(cv), cv is a constant lvalue reference.
- In obj.f(20), 20 is an rvalue and passed as an rvalue reference. This is the best match in the set of overloaded member functions.

TYPE DEDUCTION: UNIVERSAL REFERENCES

```
Foo<int> obj;
int v = 20;
const int& cv = v;
double d = 2.2;
const double& cd = d;

obj.f( 1.2 );                 // f(U&&), g(T&)
obj.f( d );                   // f(U&&), g(T&)
obj.f( cd );                  // f(U&&), g(const T&)
obj.f( "Hello, world!" );     // f(U&&), g(const T&)
```

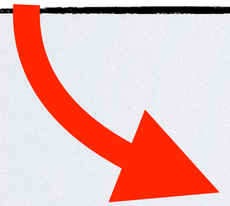


```
ff(U&& 1.2) -> g(T&)
ff(U&& 2.2) -> g(T&)
ff(U&& 2.2) -> g(const T&)
ff(U&& Hello, world!) -> g(const T&)
```

- In obj.f(1.2), 1.2 is an rvalue. It is passed as an rvalue reference to f. But it is forwarded to function g as an lvalue reference (i.e., reference to parameter).
- In obj.f(d), d is an lvalue pass as lvalue reference. It is forwarded to function g as an lvalue reference.
- In obj.f(cd), cd is a constant lvalue reference. It is forwarded to function g as a constant lvalue reference.
- In obj.f("Hello, world!"), "Hello, world!" is a constant lvalue reference (a constant reference to an array of 14 characters). It is passed to g as a constant lvalue reference.

TYPE DEDUCTION: PERFECT FORWARDING

```
Foo<int> obj;
int v = 20;
const int& cv = v;
double d = 2.2;
const double& cd = d;

obj.ff( 1.2 );                  // f(U&&), g(T&&)
obj.ff( d );                    // f(U&&), g(T&)
obj.ff( cd );                   // f(U&&), g(const T&)
obj.ff( "Hello, world!" );      // f(U&&), g(const T&)
```



```
ff(U&& 1.2) -> g(T&&)
ff(U&& 2.2) -> g(T&)
ff(U&& 2.2) -> g(const T&)
ff(U&& Hello, world!) -> g(const T&)
```

- In obj.ff(1.2), 1.2 is an rvalue. It is passed as an rvalue reference to f and forwarded to function g as an rvalue reference (perfect forwarding).
- In obj.ff(d), d is an lvalue pass as lvalue reference. It is forwarded to function g as an lvalue reference.
- In obj.ff(cd), cd is a constant lvalue reference. It is forwarded to function g as a constant lvalue reference.
- In obj.ff("Hello, world!"), "Hello, world!" is a constant lvalue reference (a constant reference to an array of 14 characters). It is passed to g as a constant lvalue reference.

REFERENCE COLLAPSING

- If we pass an lvalue to a function whose parameter is a universal reference, the compiler deduces the parameter type as the argument's lvalue reference type.

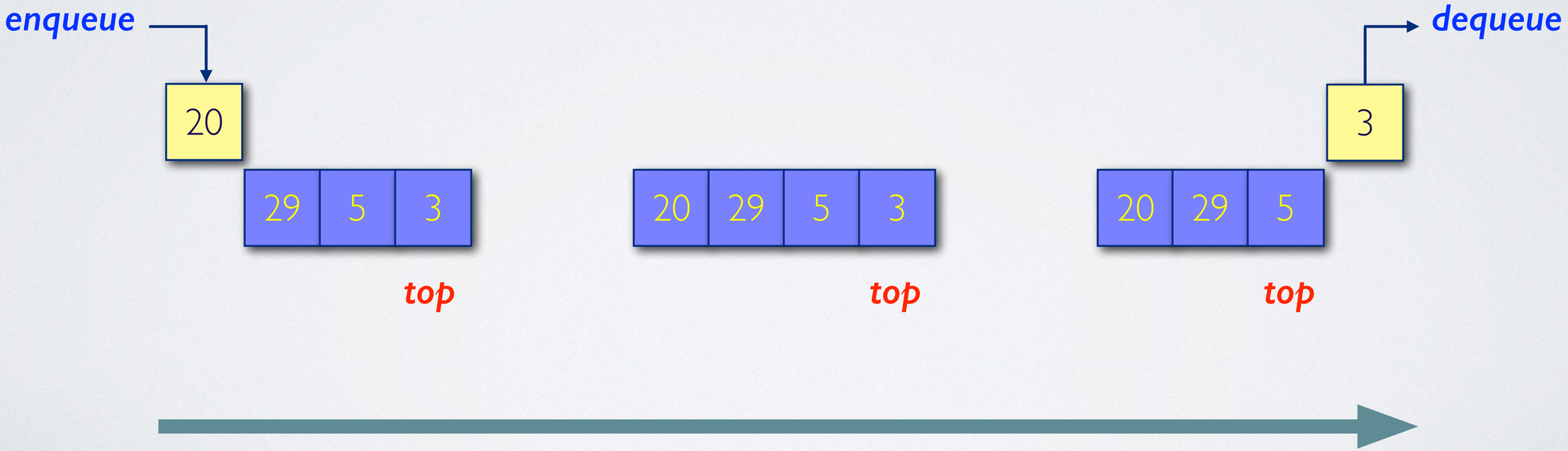
- If we indirectly create a reference to a reference, then those references "collapse." That is, for a given type X:

  - X& &, X& &&, and X&& & all collapse to type X&.

  - X&& && collapses to X&&.

ELEMENTARY DATA STRUCTURE QUEUE

- A queue is a dynamic set in which the element removed is always the one that has been in the set for the longest time: the queue implements a fist-in-first-out, or FIFO, policy.



IMPLEMENTATION OF QUEUE

- It is customary to define an interface for a queue similar to a stack. Hence, we need to provide:
  - A top() function that returns the first, or oldest, element in the queue, if it exists, without changing the queue.
  - An enqueue() method that inserts a new element into the queue. The newly added element is the last, or youngest, element in the queue.
  - A dequeue() method that removes the first, or oldest, element from the queue.
- Again, we use std::optional<T> as a result for top(), as top() may fail when the queue is empty. The vocabulary type std::optional allows us to implement top() as atomic actions rather than a two-step process that is not thread-safe.

QUEUE INTERFACE: FIRST VERSION

```
template<typename T, size_t N = 32>
class Queue
{
public:
    // member type definitions relevant to Queue

    Queue() noexcept;

    size_t size() const noexcept;
    bool empty() const noexcept;
    bool full() const noexcept;

    std::optional<T> top() noexcept;
    void enqueue( const T& aValue );
    void dequeue();

private:
    T fElements[N];
    size_t fHead;
    size_t fTail;
    size_t fSize;
};
```

Queue is trivially copyable

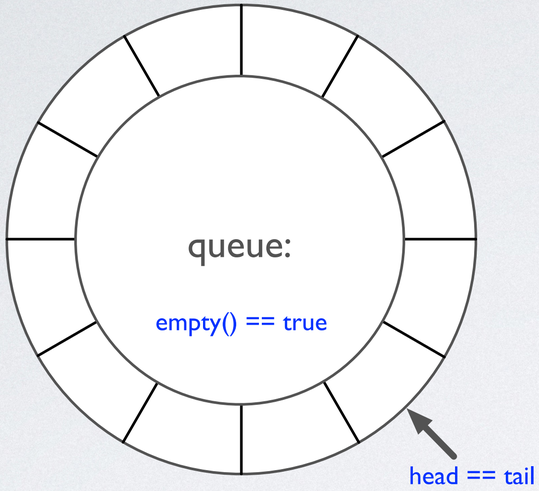
Service member functions

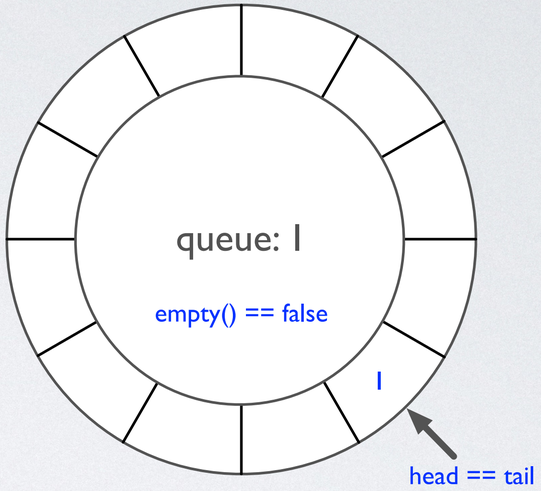
QUEUE CONSTRUCTOR

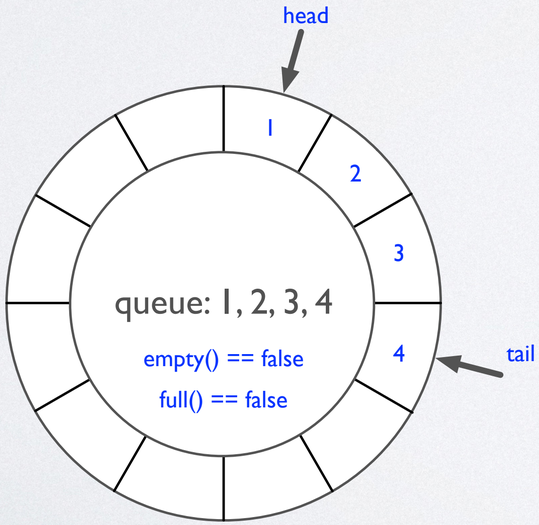
```
Queue() noexcept :
        fElements(),
        fHead(0),
        fTail(0),
        fSize(0)
{}
```

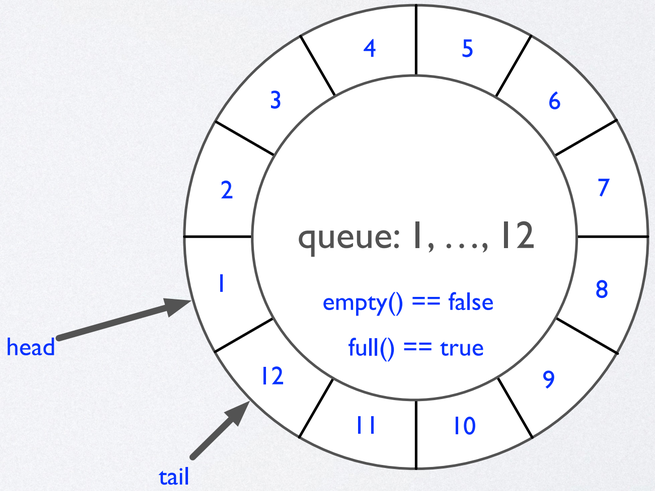
- The default constructor initializes all member variables with sensible values:
  - Each element in array fElements is set to the default value for type T. The expression fElements() is mapped by the C++ compiler to an initializer loop using T() for every element in the array.
  - The queue implements a circular buffer, in which fHead points to the first and oldest element, and fTail points to the last and newest element. Initially, the queue is empty, fHead == fTail == fSize.

QUEUE STATES









QUEUE SIZE

```
size_t size() const noexcept
{
    return fSize;
}
```

- The size getter returns the current number of elements in the queue.
- The running time of size() is O(1).

QUEUE EMPTY & FULL

```
bool empty() const noexcept
{
    return fSize == 0;
}

bool full() const noexcept
{
    return fSize == N;
}
```

- A queue is empty if it does not contain any elements.
- The running time of empty() is O(1).
- A queue is full if it contains N elements.
- The running time of full() is O(1).

QUEUE ENQUEUE

```
void enqueue( const T& aValue )
{
    assert( !full() );

    if ( !empty() )
    {
        if ( ++fTail == N )
        {
            fTail = 0;
        }
    }

    fElements[fTail] = aValue;
    fSize++;
}
```

- The enqueue operation takes a constant reference to a value of type T and copies the referenced value into the array fElements at the current tail position. Afterwards, the tail pointer points to the next free element. Advancing the tail pointer is in round-robin fashion.
- The assert statement checks for a queue overflow error. This test is run in Debug mode and suppressed in the Release version.
- The running time of push() is O(1).

QUEUE DEQUEUE

```
void dequeue()
{
    assert( !empty() );

    fSize—;

    if ( !empty() )
    {
        if ( ++fHead == N )
        {
            fHead = 0;
        }
    }
}
```

- The dequeue operation advances the head pointer in round-robin fashion. The first element will be freed later, on a later enqueue operation or when the queue object goes out of scope.
- The assert statement checks for a queue underflow error. This test is run in Debug mode and suppressed in the Release version.
- The running time of pop() is O(1).

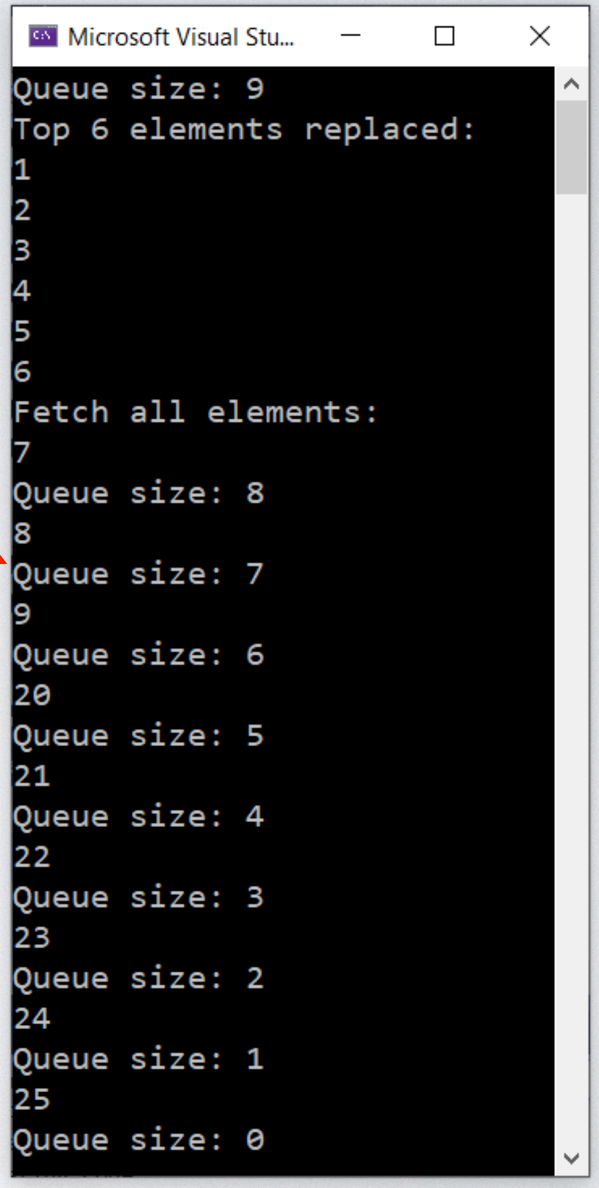
QUEUE TOP

```
std::optional<T> top()
{
    if ( !empty() )
    {
        return std::optional<T>( fElements[fHead] );
    }
    else
    {
        return std::optional<T>();
    }
}
```

- The top operation returns the first, oldest element if it exists.
- We use the vocabulary type std::optional<T> as the result type. Any instance of std::optional<T> at any given point in time either contains a value or does not contain a value. Through std::optional<T>, top becomes an atomic operation that tests for a value and returns a value, if it exists.
- The running time of top() is O(1).

QUEUE TEST

```
Queue<int, 10> IQueue;

for ( int i = 1; i < 10; i++ )
{
    IQueue.enqueue( i );
}

std::cout << "Queue size: " << IQueue.size() << std::endl;
std::cout << "Top 6 elements replaced:" << std::endl;

std::optional<int> IValue = IQueue.top();

for ( int i = 20; i < 26; i++ )
{
    std::cout << IValue.value() << std::endl;

    IQueue.dequeue();
    IQueue.enqueue( i );

    IValue = IQueue.top();
}

std::cout << "Fetch all elements:" << std::endl;

while ( IValue )
{
    std::cout << "Queue size: " << IQueue.size() << std::endl;
    std::cout << IValue.value() << std::endl;

    IQueue.dequeue();

    IValue = IQueue.top();
}

std::cout << "Queue size: " << IQueue.size() << std::endl;
```



INSERT ELEMENT: A POSSIBLE PERFORMANCE ISSUE

- If we have a container, like a stack or queue, it seems logical that when we add a new element via an insertion function, the element type remains the same in the container. However, this is not always true. Consider, for example, this code:

```
Queue<std::string> lQueue;

lQueue.enqueue( "Hello, world!" );
```

- Here, the container lQueue holds elements of type std::string, but we have used a c-string, "Hello, world!", which is of type **const char**[14].

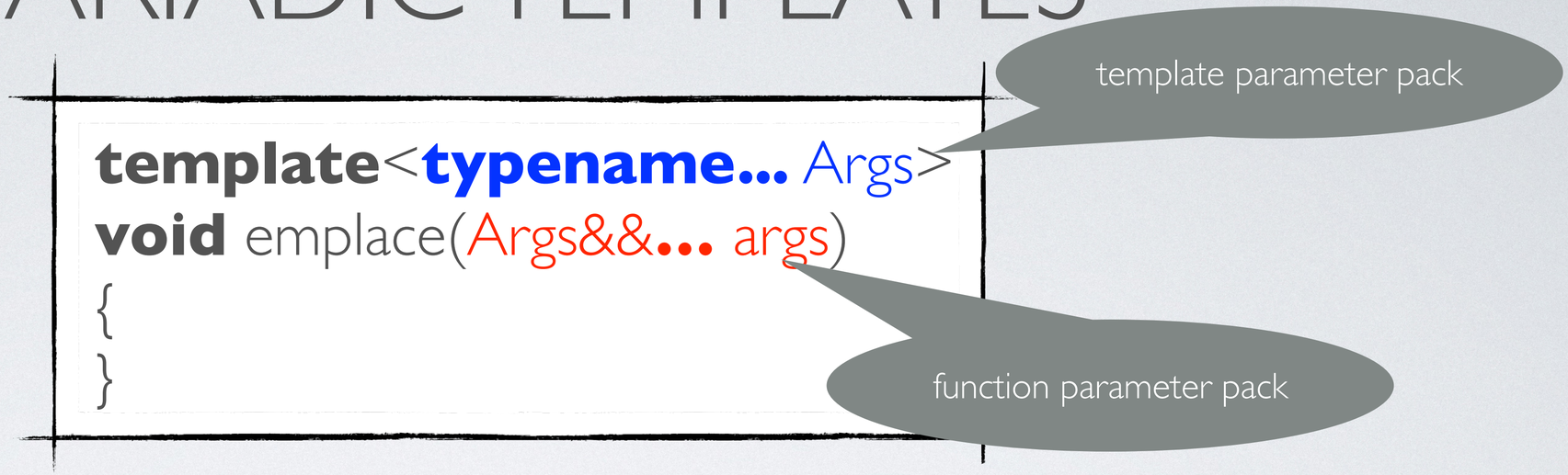
- The compiler sees a mismatch that needs to be resolved. The compiler creates a temporary std::string value, an rvalue which the function enqueue sees as an lvalue reference or a value to be copied into the queue. For performance, the c-string "Hello, world!" should be passed to as is, and a std::string object should be constructed 'in-place' inside the queue.

EMPLACE ELEMENTS

```
template<typename... Args>
void emplace(Args&&... args)                                    // allow for type deduction of arg types
{
    assert( !full() );

    if ( !empty() )
    {
        if ( ++fTail == N ) { fTail = 0; }
    }

    fElements[fTail].~T();                                      // free old entry using T's destructor

    new ( &fElements[fTail++]) T( std::forward<Args>(args)... );    // placement new (matching constructor)

    fSize++;
}
```

- The method emplace() works like enqueue, but the new element inserted is constructed, not copied.
- We define emplace() as a variadic template method and construct the element in-place via a matching element constructor via perfect forwarding of the arguments received by emplace().

VARIADIC TEMPLATES



- A variadic template is a template function or class that can take a parameter pack as the argument.
- In a template parameter list, **class...** or **typename...** indicates that the following parameter represents a list of zero or more types (e.g., Args is a list of types).
- The name of a type followed by an ellipsis represents a list of zero or more non-type parameters of the given type (e.g., args is a list of non-type parameters).

EMPLACE SEQUENCE

```
fElements[fTail].~T();                                          // free old entry using T's destructor

new ( &fElements[fTail]) T( std::forward<Args>(args)... );      // placement new (matching constructor)
```

- When we use emplace(), we first need to free the object currently stored in the target location. We do not free the object itself, just the resources it uses.

- Next, the in-place **new** operator constructs the new object. The operator does not acquire new memory. It reinitializes existing memory with updated data.

STACK & QUEUE REVISED

STACK REVISED

```
template<typename T>
class StackRevised
{
public:
    StackRevised();
    ~StackRevised();

    StackRevised( const StackRevised& aOther );
    StackRevised& operator=( const StackRevised<T>& aOther );

    StackRevised( StackRevised<T>&& aOther ) noexcept;
    StackRevised& operator=( StackRevised<T>&& aOther ) noexcept;

    void swap( StackRevised& aOther ) noexcept;

    size_t size() const noexcept;

    std::optional<T> top() noexcept;
    void push( const T& aValue );
    void pop();

    template<typename... Args>
    void emplace( Args&&... args );

private:
    T* fElements;
    size_t fStackPointer;
    size_t fCapacity;

    void resize( size_t aNewCapacity );
    void ensure_capacity();
    void adjust_capacity();
};
```

copy semantics

move semantics

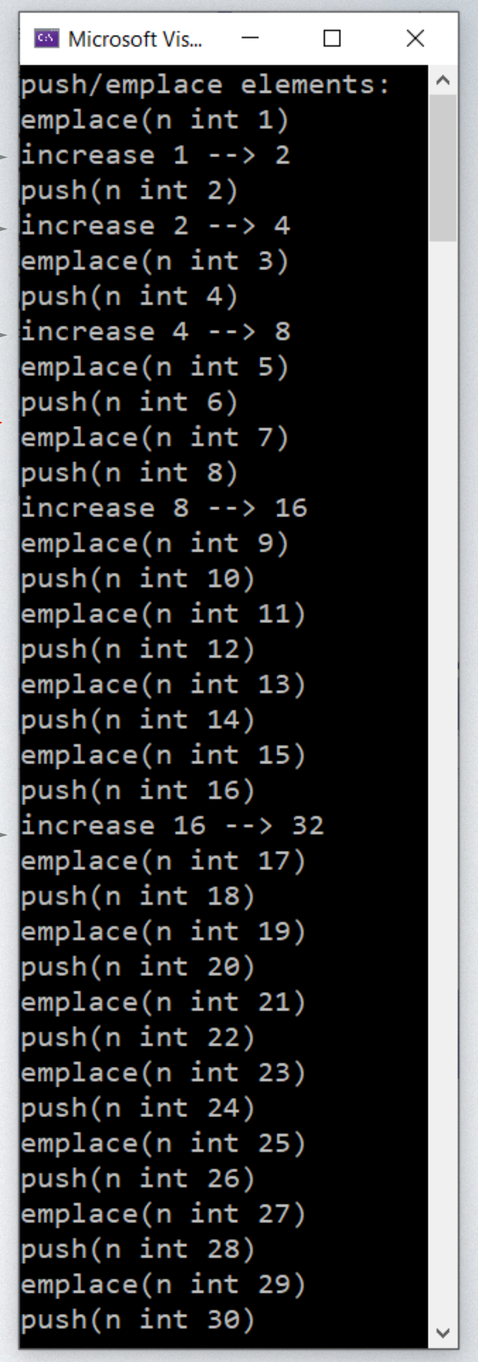
stack semantics

expansion/contraction

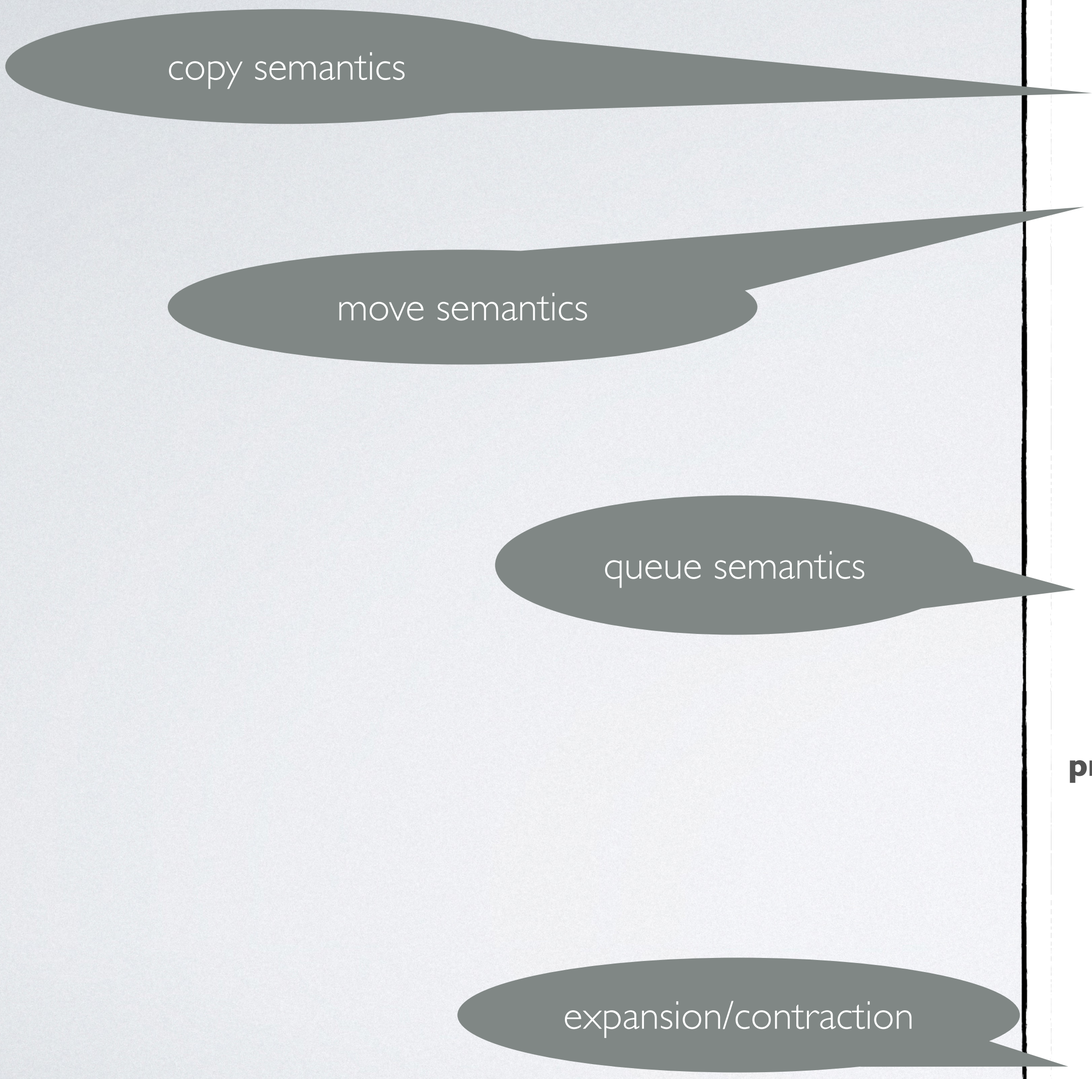
REVISED STACK TEST

```
struct Data
{
    std::string fName;
    std::string fType;
    int fValue;

    Data( const char* aName = "", const char* aType = "", int aValue = 0 ) :
        fName(aName),
        fType(aType),
        fValue(aValue)
    {}
};

std::ostream& operator<<( std::ostream& s, const Data& o )
{
    return s << o.fName << " " << o.fType << " " << o.fValue;
}
```

```
StackRevised<Data> IStack;

std::cout << "push/emplace elements:" << std::endl;

for ( int i = 1; i <= 30; i++ )
{
    if ( i % 2 == 0 )
    {
        IStack.push( Data( "n", "int", i ) );
    }
    else
    {
        IStack.emplace( "n", "int", i );
    }
}
```



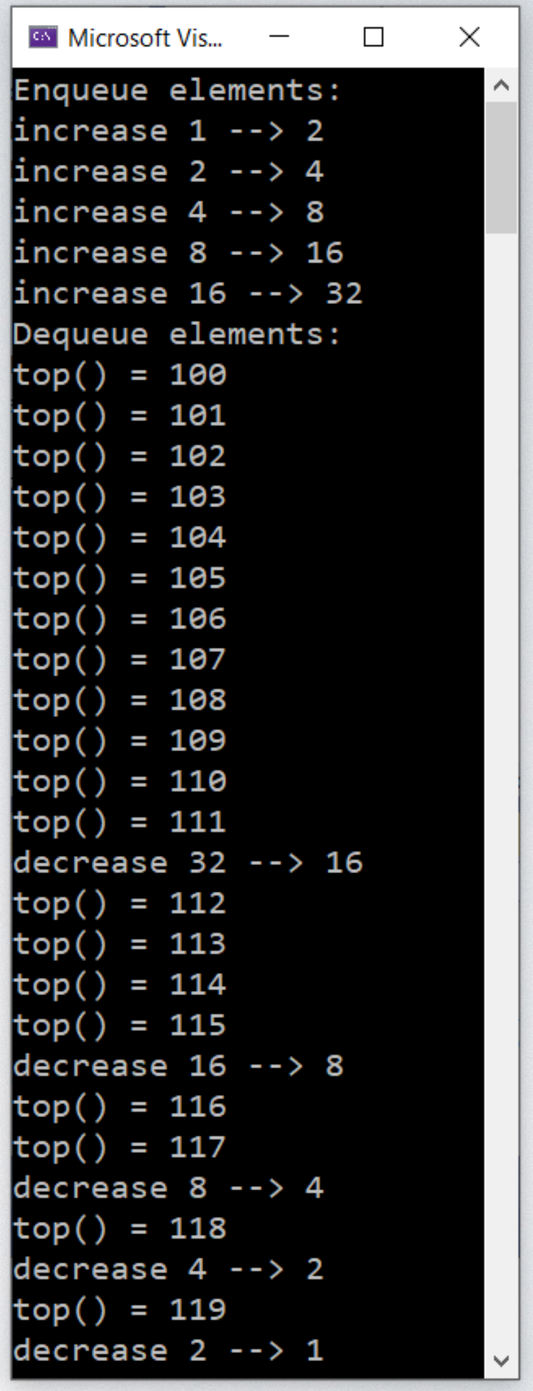
QUEUE REVISED




```
template<typename T>
class QueueRevised
{
public:
    QueueRevised();
    ~ QueueRevised();

    QueueRevised( const QueueRevised& aOther );
    QueueRevised& operator=( const QueueRevised <T>& aOther );

    QueueRevised( QueueRevised <T>&& aOther ) noexcept;
    QueueRevised& operator=( QueueRevised <T>&& aOther ) noexcept;

    void swap( QueueRevised& aOther ) noexcept;

    size_t size() const noexcept;
    bool empty() const noexcept;
    bool full() const noexcept;

    std::optional<T> top() noexcept;
    void enqueue( const T& aValue );
    void dequeue();

    template<typename... Args>
    void emplace( Args&&... args );


private:
    T* fElements;
    size_t fCapacity;
    size_t fHead;
    size_t fTail;
    size_t fSize;

    void resize( size_t aNewCapacity );
    void ensure_capacity();
    void adjust_capacity();
};
```

REVISED QUEUE TEST

```
QueueRevised<int> IQueue;

std::cout << "Enqueue elements:" << std::endl;

for ( int i = 100; i < 120; i++ )
{
    if ( i % 2 == 0 )
        IQueue.enqueue(i);
    else
        IQueue.emplace( i );
}

std::cout << "Dequeue elements:" << std::endl;

std::optional<int> IValue = IQueue.top();

while ( IValue )
{
    std::cout << "top() = " << IValue.value() << std::endl;

    IQueue.dequeue();

    IValue = IQueue.top();
}
```



NOTE ON STACK AND QUEUE

- Stacks and queues can have a shared representation. The implementations of the standard library classes std::stack and std::queue vividly demonstrate this. Both use std::deque, a double-ended queue, as an underlying container. Consequently, std::stack and std::queue are object adapters that use an object of type std::deque as delegate instance and forward calls to stack and queue to the suitable methods of std::deque.

- A deque can be implemented as a circular buffer. The standard library uses a different approach that avoids copying elements on expansion and contraction. Note, type std::vector employs an expansion/contraction technique similar to the one shown in COS30008.

- A naive implementation of a priority queue might use priority-value pairs for ordering. A better solution is to use a heap that can be viewed as a binary tree and an array.