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", 3rd Edition, The MIT Press (2009)
  - Scott Meyers: Effective Modern C++. O'Reilly (2014)
  - Anthony Williams: C++ Concurrency in Action - Practical Multithreading. Manning Publications Co. (2012)

© Dr Markus Lumpe, 2024

DATA ABSTRACTION

- Data abstraction divides a data type into two pieces:
  - An interface that tells us what the data of the 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 data and code for the operations that make use of the specific data representation.
- A data type that is abstract in this way is said to be an abstract data type. A client of the data type manipulates the new data only through the operations specified in the interface. If we wish to change the representation, all we must do is change the implementation of the operations in the interface.

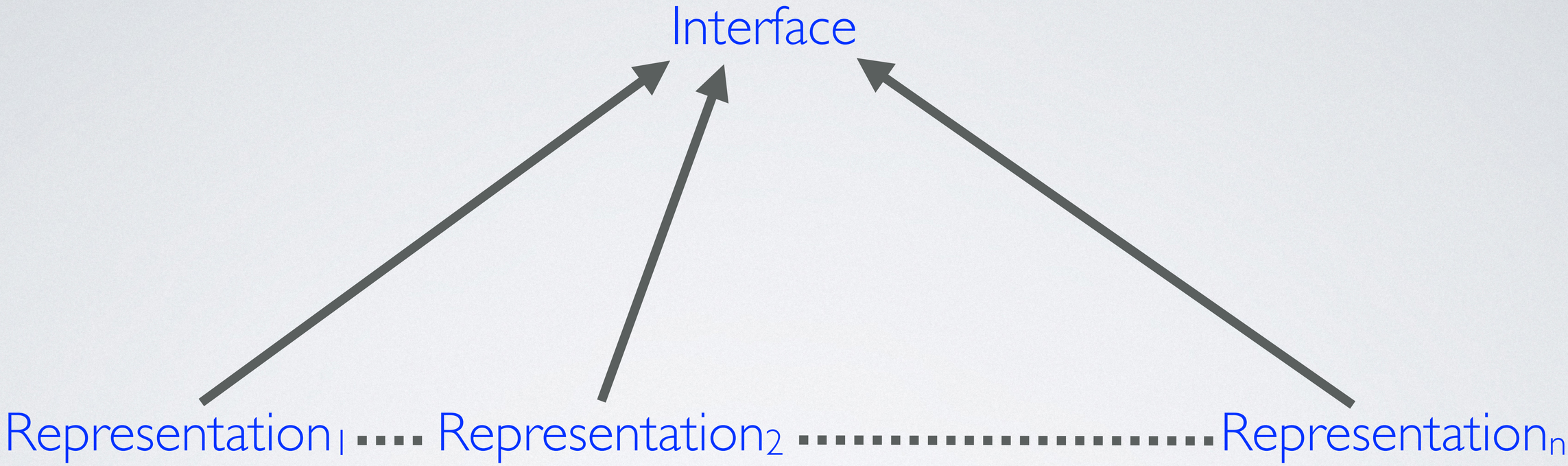
© Dr Markus Lumpe, 2024

OBJECT-ORIENTED ENCAPSULATION

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

© Dr Markus Lumpe, 2024

REPRESENTATION STRATEGIES FOR ABSTRACT DATA TYPES



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

© Dr Markus Lumpe, 2024

REQUIRED OPERATIONS

- In order to create values, verify that a given value is of the desired data type, 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 that allow us to extract a particular information from a given representation and to update data of given value, respectively.

© Dr Markus Lumpe, 2024

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 the transfer of data from one object to another using Shallow-Copy or Deep-Copy and whether the source of the data transfer is an L-value reference or an R-value reference.
- From a purely object-oriented viewpoint, copy semantics preserves control over object ownership. The source object is an L-value reference and copy semantics entails a copy constructor and a copy-assignment operator.
- Move semantics handles data transfer from source objects whose lifetime expires. The source object is an R-value reference and move semantics entails a move constructor and a move-assignment operator.

© Dr Markus Lumpe, 2024

CANONICAL METHODS

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

© Dr Markus Lumpe, 2024

R-VALUE REFERENCES

```
#define rvalue 42

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

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

© Dr Markus Lumpe, 2024

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, then no default constructor is automatically generated.
  - If a virtual destructor is explicitly declared, then 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.

© Dr Markus Lumpe, 2024

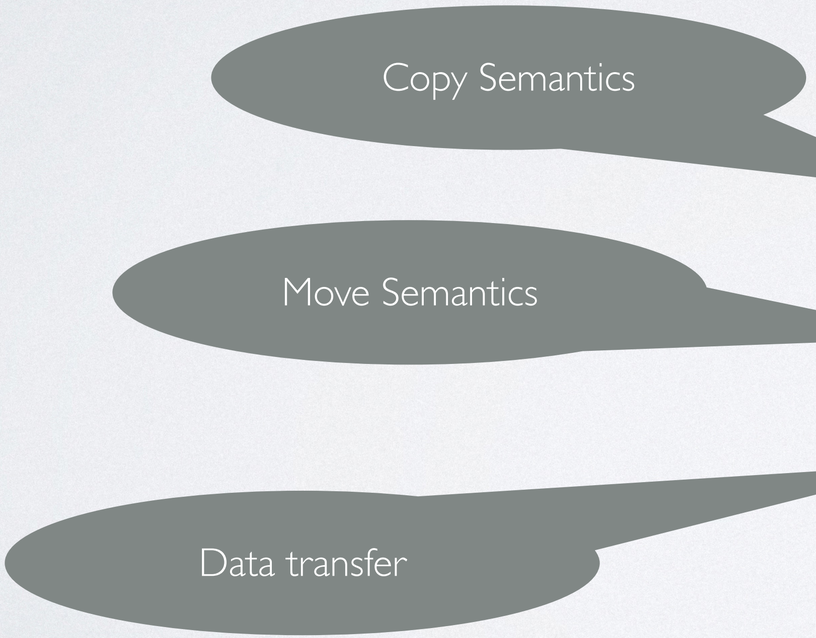
DYNAMIC STACK

- The type DynamicStack is an abstract data type.

© Dr Markus Lumpe, 2024

DYNAMIC STACK

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




```
template<typename T>
class DynamicStack
{
private:
    T* fElements;
    size_t fStackPointer;
    size_t fCurrentSize;

    void resize( size_t aNewSize );
    void ensure_capacity();
    void adjust_capacity();

public:
    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();
};
```

DYNAMICSTACK SWAP

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

- The swap function handles the data transfer between objects.
- The swap function no never throw an exception. For this reason, it is marked **noexcept**.
- We need to swap the data element-wise in order the prevent an infinite recursion. The standard library defines std::swap using move semantics for the exchange of arguments.

© Dr Markus Lumpe, 2024

DYNAMICSTACK COPY CONSTRUCTOR

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

    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 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. This check guarantees, for the newly created object, that the stack pointer does not exceed the size of the stack. We may known this for the source object, but the compiler does not know this for the newly created object. We need to explicitly state this assumption.

© Dr Markus Lumpe, 2024

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 just return **this**-object as result. If the test is missing, then 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 the **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. When new is applied to a pointer, then no new memory is allocated. Instead the exiting memory is reinitialized with the copy constructor.

© Dr Markus Lumpe, 2024

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 simply swap the objects. This operation is guaranteed to succeed, hence the move constructor in marked **noexcept**. Once the move constructor finishes, the source object aOther is destroyed. The lifetime of aOther expires.

© Dr Markus Lumpe, 2024

DYNAMICSTACK MOVE-ASSIGNMENT

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

    return *this;
}
```

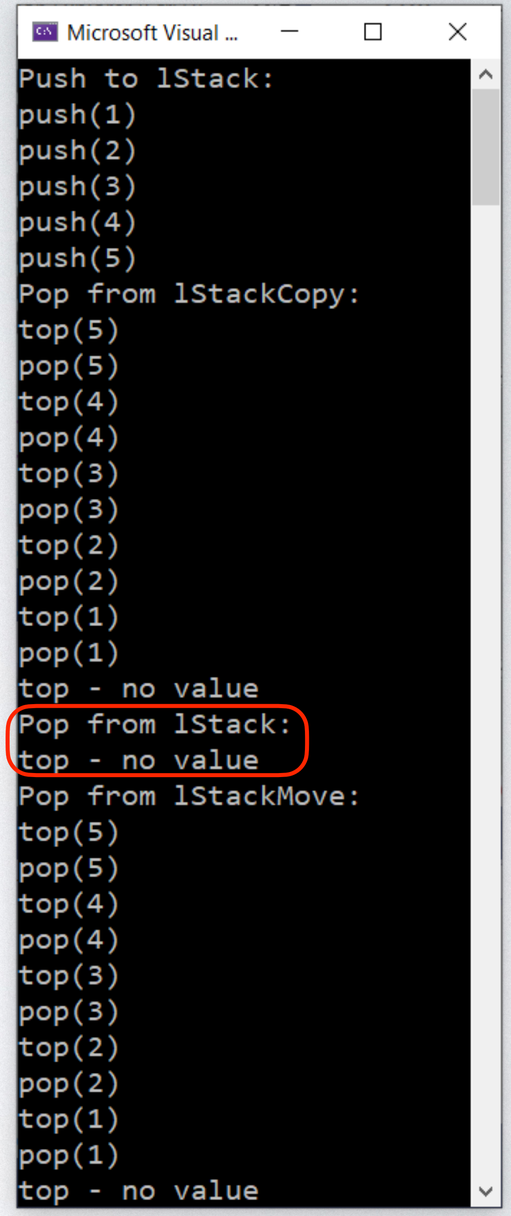
- The move-assignment operator, like the move constructor, "steals" the memory of the source object.
- Naturally, we have to perform a self-assignment check first. Next, we simply swap the objects, if necessary. This operation is guaranteed to succeed, hence the move-assignment operator in marked **noexcept**.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

© Dr Markus Lumpe, 2024

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; )
{
    Stack.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.



© Dr Markus Lumpe, 2024

STD::MOVE()

- The helper function std::move(arg) forces move semantics on arg, even if arg is an L-value or L-value 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 R-value reference.

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


© Dr Markus Lumpe, 2024

TEMPLATE ARGUMENT DEDUCTION

- The C++-compiler, by default, uses the arguments in a call to determine the template parameters for a function template.
- The process of determining the template arguments from the function arguments is known as 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, however, scenarios in which we have to assist the compiler or have to explicitly specify template arguments. This includes also forwarding of arguments with their type unchanged.

© Dr Markus Lumpe, 2024

TEMPLATE ARGUMENT DEDUCTION L-VALUE REFERENCES

```
template<typename T> void f1 (T& p) { ... }
```

- What is the type of parameter p in a call to function f1 after template argument deduction?
  - When a function parameter is an ordinary L-value reference to a template type parameter (i.e., it has the form T&), the binding rules state that we pass only an L-value. If the argument type is **const**, then T will be deduced as a **const** type.

```
int i = 2;
const int ci =3;

f1 (i);          // i is an int, T is int
f1 (ci);         // ci is a const int, T is const int
f1 (5);          // error: 5 is not an L-value.
```


© Dr Markus Lumpe, 2024

TEMPLATE ARGUMENT DEDUCTION CONST L-VALUE REFERENCES

```
template<typename T> void f2(const T& p) { … }
```

- What is the type of parameter p in a call to function f2 after template argument deduction?
  - When a function parameter has type **const** T&, the binding rules state that we pass any type of argument, and object const or otherwise, a temporary, or a literal value. When the function parameter is **const**, then T will be deduced as a non-const type.

```
int i = 2;
const int ci =3;

f2(i);          // i is an int, T is int
f2(ci);         // ci is a const int, T is int
f2(5);          // a const & parameter can be bound to an R-value, T is int
```


© Dr Markus Lumpe, 2024

TEMPLATE ARGUMENT DEDUCTION R-VALUE REFERENCES

```
template<typename T> void f3(T&& p) { ... }
```

- What is the type of parameter p in a call to function f3 after template argument deduction?
  - When a function parameter is an R-value reference to a template type parameter (i.e., it has the form T&&), the binding rules state that we pass an R-value.

```
f3(5);          // argument is an R-value of type int, T is int.
```


© Dr Markus Lumpe, 2024

REFERENCE COLLAPSING

- Passing an L-value would seem to be impossible to pass to function f3. It's parameter is an R-value reference. However, there are two binding rule exceptions:
  - If we pass an L-value to a function parameter that is an R-value reference to a template type parameter, then the compiler deduces the template type parameter as the argument's L-value 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&&.

```
int i = 2;
const int ci =3;

f3(i);          // i is an L-value of type int, T is int&
f3(ci);         // ci is an L-value of type const int, T is const int &
```


© Dr Markus Lumpe, 2024

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.

© Dr Markus Lumpe, 2024

IMPLEMENTATION OF QUEUE

- It is customary to define an interface for a queue similar to 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.
  - A enqueue() method that inserts a new element into the queue. The newly added elements is the last 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 as result for top(), as top() may fail. This happens 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.

© Dr Markus Lumpe, 2024

QUEUE INTERFACE: FIRST VERSION

```
template<typename T, size_t N = 32>
class Queue
{
private:
    T fElements[N];
    size_t fHead;
    size_t fTail;

public:
    Queue() noexcept;

    size_t size() const noexcept;

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

Queue is trivially copyable

We add a size() getter.

© Dr Markus Lumpe, 2024

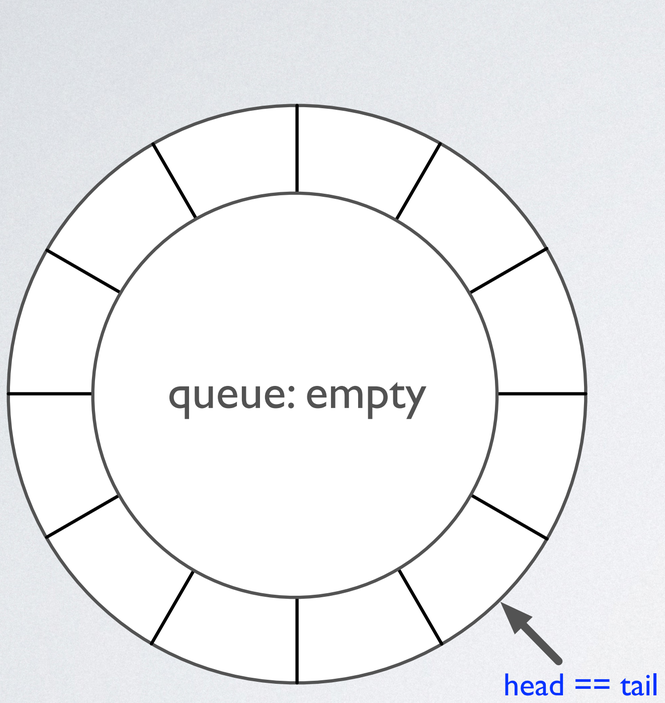
QUEUE CONSTRUCTOR

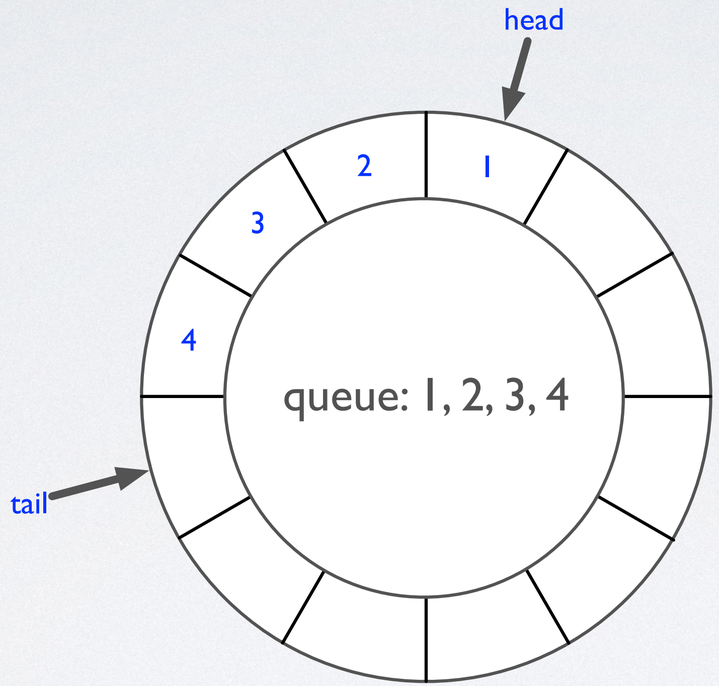
```
Queue() noexcept :
        fElements(),
        fHead(0),
        fTail(0)
{}
```

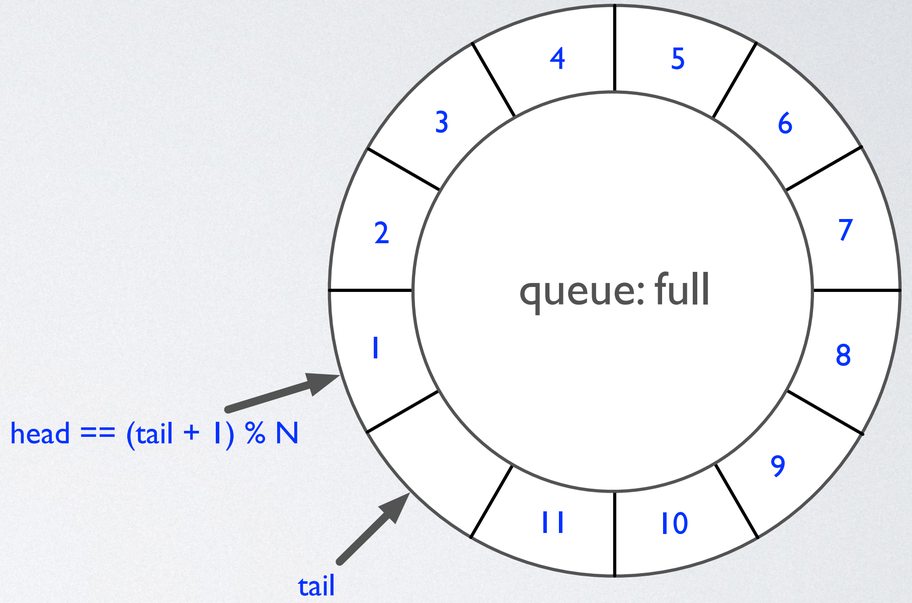
- The default constructor initializes all member variables with sensible values:
  - Each element in array fElements is set to the default of type T. The expression fElements() is mapped by the C++-compiler to an initializer loop using T() for every element in the array.
  - The queue implements a circular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

© Dr Markus Lumpe, 2024

QUEUE STATES







© Dr Markus Lumpe, 2024

QUEUE SIZE

```
size_t size() const noexcept
{
    return (N - fHead + fTail) % N;
}
```

- The size getter returns the current number of elements in the queue.
- We insert elements in clockwise order. The expression (N - fHead + fTail) % N yields the number of elements in the queue, when the elements are stored in clockwise order.
- The running time of size is O(1).

© Dr Markus Lumpe, 2024

QUEUE ENQUEUE

```
void enqueue( const T& aValue )
{
    assert( fHead != (fTail + 1) % N );

    fElements[fTail++] = aValue;

    if ( fTail == N )
    {
        fTail = 0;
    }
}
```

- 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 queue overflow. This test is run in debug mode. In the Release version, this test is suppressed.
- The running time of push is O(1).

© Dr Markus Lumpe, 2024

QUEUE DEQUEUE

```
void dequeue()
{
    assert( fHead != fTail );

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

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

© Dr Markus Lumpe, 2024

QUEUE TOP

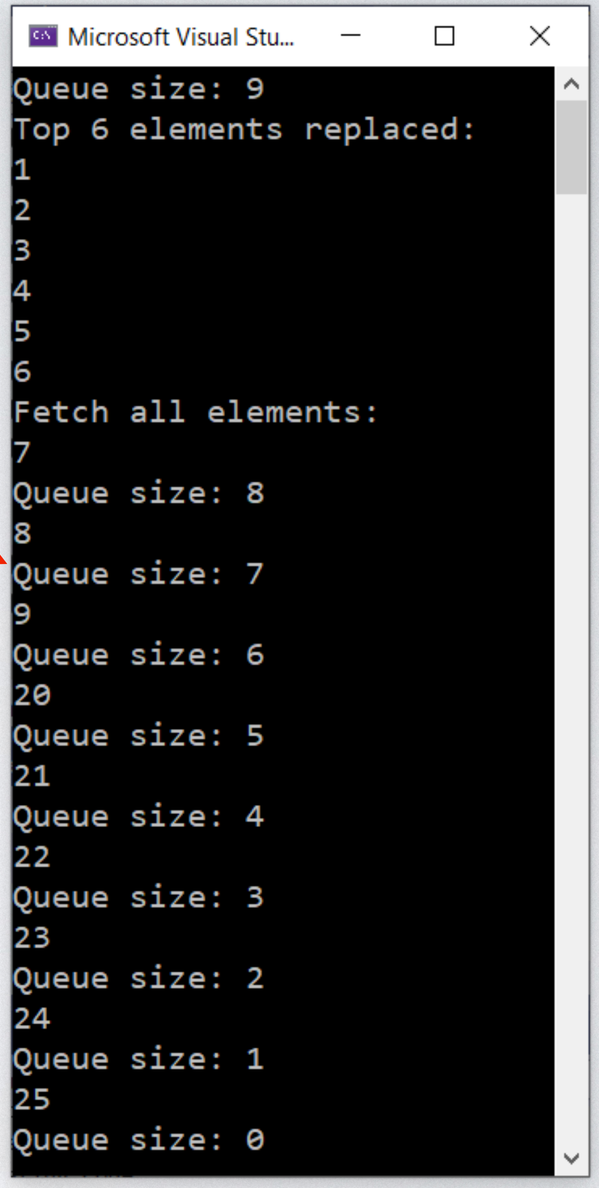
```
std::optional<T> top()
{
    if ( fHead != fTail )
    {
        return std::optional<T>( fElements[fHead] );
    }
    else
    {
        return std::optional<T>();
    }
}
```

- The top operation returns the first, oldest, element, if it exits.
- We use the vocabulary type std::optional<T> as 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 both tests for a value and returns a value, if it exists.
- The running time of top is O(1).

© Dr Markus Lumpe, 2024

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 << IValue.value() << std::endl;

    IQueue.dequeue();

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

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



© Dr Markus Lumpe, 2024

INSERT ELEMENT: A POSSIBLE PERFORMANCE ISSUE

- If we have a container, like stack of queue, it seems logical that when we add a new element via an insertion function, the type of the element remains the same. 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 string literal, "Hello World!", which is of type **const char**[13].

- The compiler sees a mismatch that needs to be resolved. In this case, an implicit type conversion occurs via std::string("Hello World!"). In the context of the enqueue call, the compiler creates a temporary of type std::string and passes it as constant reference to enqueue to be copied into the queue. With respect to performance, it would be better to pass the string literal directly to the queue and construct a std::string object in-place inside the queue.

© Dr Markus Lumpe, 2024

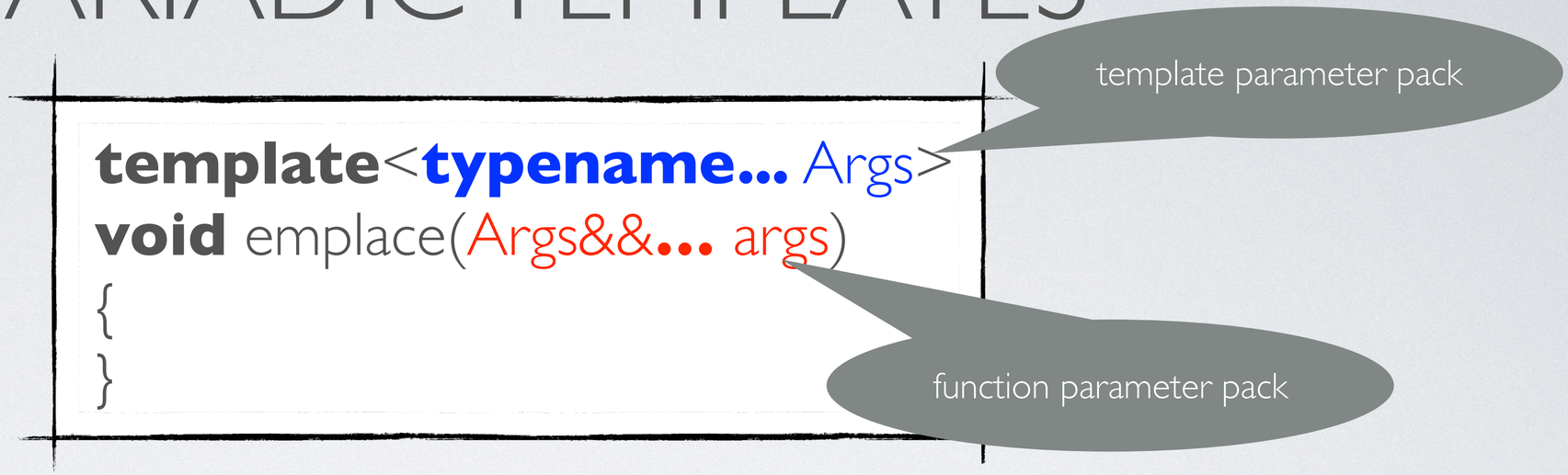
EMPLACE ELEMENTS

```
template<typename... Args>
void emplace(Args&&... args)                                    // allow for type deduction of arg types
{
    assert( fHead != (fTail + 1) % N );

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

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

    if ( fTail == N )
    {
        fTail = 0;
    }
}
```

- The method emplace() works like enqueue, but the new element inserted is construct into the queue, 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().

© Dr Markus Lumpe, 2024

VARIADIC TEMPLATES



- A variadic template is a template function or class that can take a varying number of parameters. The varying parameters are known as a parameter pack.

- 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).

© Dr Markus Lumpe, 2024

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 R-value reference to arg, if arg in not an L-value. If arg is an L-value reference, it returns arg unchanged.

© Dr Markus Lumpe, 2024

EMPLACE SEQUENCE

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

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

- We when use emplace, we first need to free the object that we are overriding. We do not free the object itself, just the resources it uses.

- Next, we use the new in-place operator to construct the new object in-place. The new in-place operator does not acquire new memory. It simply reinitializes existing memory with new data.

© Dr Markus Lumpe, 2024

STACK & QUEUE

© Dr Markus Lumpe, 2024

STACK REVISED

```
template<typename T>
class StackRevised
{
private:
   T* fElements;
   size_t fStackPointer;
   size_t fCurrentSize;

   void resize( size_t aNewSize );
   void ensure_capacity();
   void adjust_capacity();

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 );

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

   void pop();
};
```

expansion/contraction

copy semantics

move semantics

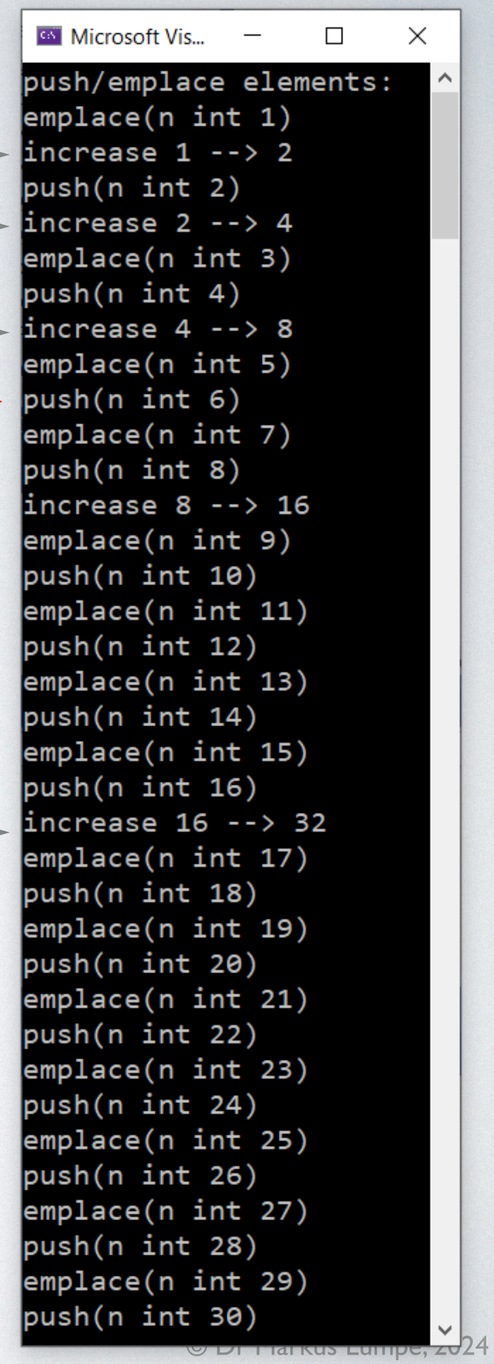
stack semantics

© Dr Markus Lumpe, 2024

REVISED STACK TEST

```
class Data
{
private:
    std::string fName;
    std::string fType;
    int fValue;

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

```
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 );
    }
}
```



© Dr. Markus Lumpe, 2024

QUEUE REVISED

```
template<typename T>
class QueueRevised
{
private:
    T* fElements;
    size_t fHead;
    size_t fTail;
    size_t fCurrentSize;

    void resize( size_t aNewSize );
    void ensure_capacity();
    void adjust_capacity();

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;

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

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

    void dequeue();
};
```

expansion/contraction

copy semantics

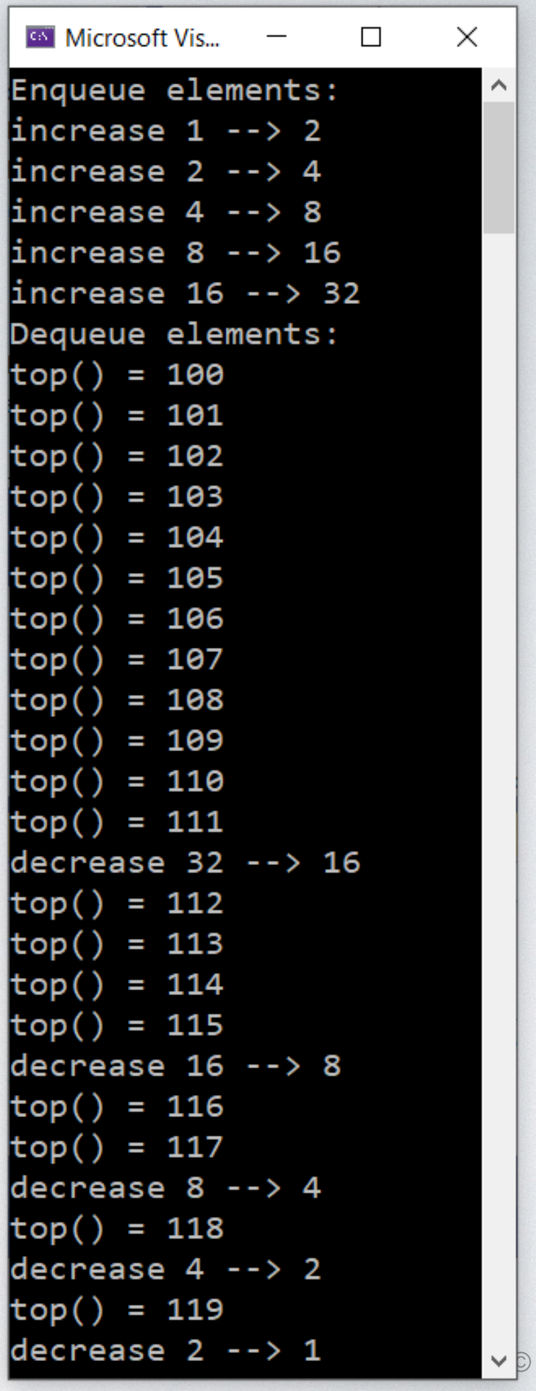
move semantics

queue semantics

© Dr Markus Lumpe, 2024

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();
}
```



© Dr Markus Lumpe, 2024

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 demonstrate this vividly. Both use std::deque, a double ended queue, as 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 circular buffer. The standard library uses different approach that avoids copying elements on expansion and contraction. Note, std::vector employs an expansion/contraction technique similar to the one shown in COS30008.

- A naive implementation of priority queue might use priority-value pairs as elements and sorting of queue. A better solution is the use a heap that can be viewed as both a binary tree and an array.

© Dr Markus Lumpe, 2024