AMORTIZED ANALYSIS

AMORTIZED ANALYSIS

- Amortized analysis is a technique in which the time required to perform a sequence of operations is averaged over all operations performed on a given data structure.
- Amortized analysis can show that the average cost of an operation is small, if one calculates the average of a sequence of operations, even though a single operations might be expensive (e.g., if $T_i(n) = 1, 1, 6, 1, 1, 1$, then the average amortized cost is 1.8).
- Amortized analysis, unlike average complexity analysis, does not involve probability. It guarantees an average performance for each operation in the worst case.
- The most common techniques are the aggregate, accounting, and potential methods.

ELEMENTARY DATA STRUCTURE STACK

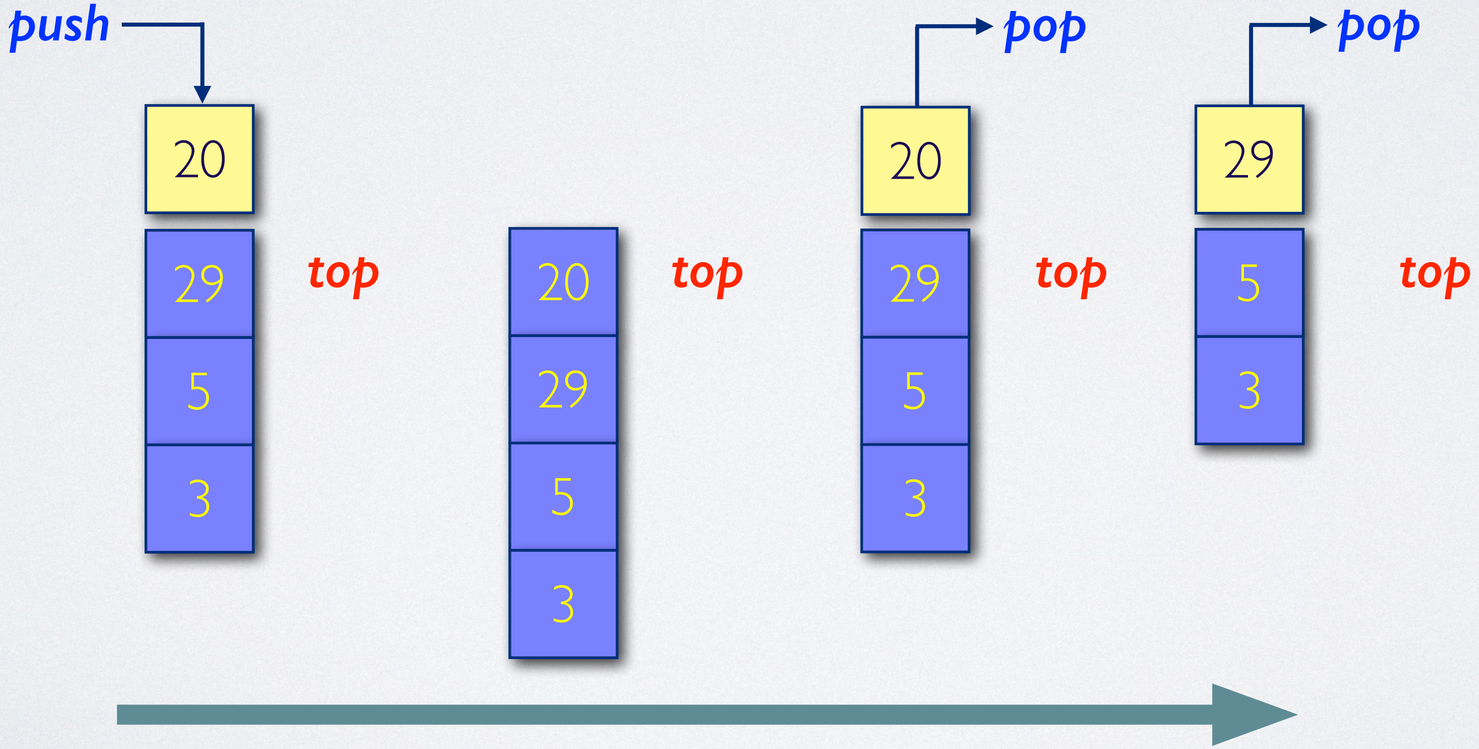
```
template<typename T, size_t N = 32>
class Stack
{
public:
    // member type definitions relevant to Stack

    Stack() noexcept;

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

private:
    T fElements[N];
    size_t fStackPointer;
};
```

C++17 feature

BasicStack is trivially copyable

STACK

- A stack is a dynamic set in which the element removed from the set by a delete operation is pre-specified. The element deleted from the set is the most recently inserted: stack implements a last-in-first-out, or LIFO, policy.



APPLICATIONS OF STACKS

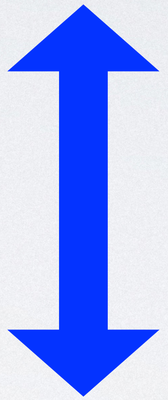
- Reversal of input (e.g., palindrome testings)
- Checking for matching parentheses (e.g., stack automata in compiler implementations)
- Backtracking (e.g., Prolog or graph analysis)
- State of program execution (e.g., storage for parameters and local variables of functions)
- Tree traversal

REVERSE POLISH NOTATION

- Reverse Polish Notation (RPN) is a prefix notation wherein operands come before operators:

Infix: a * b + c

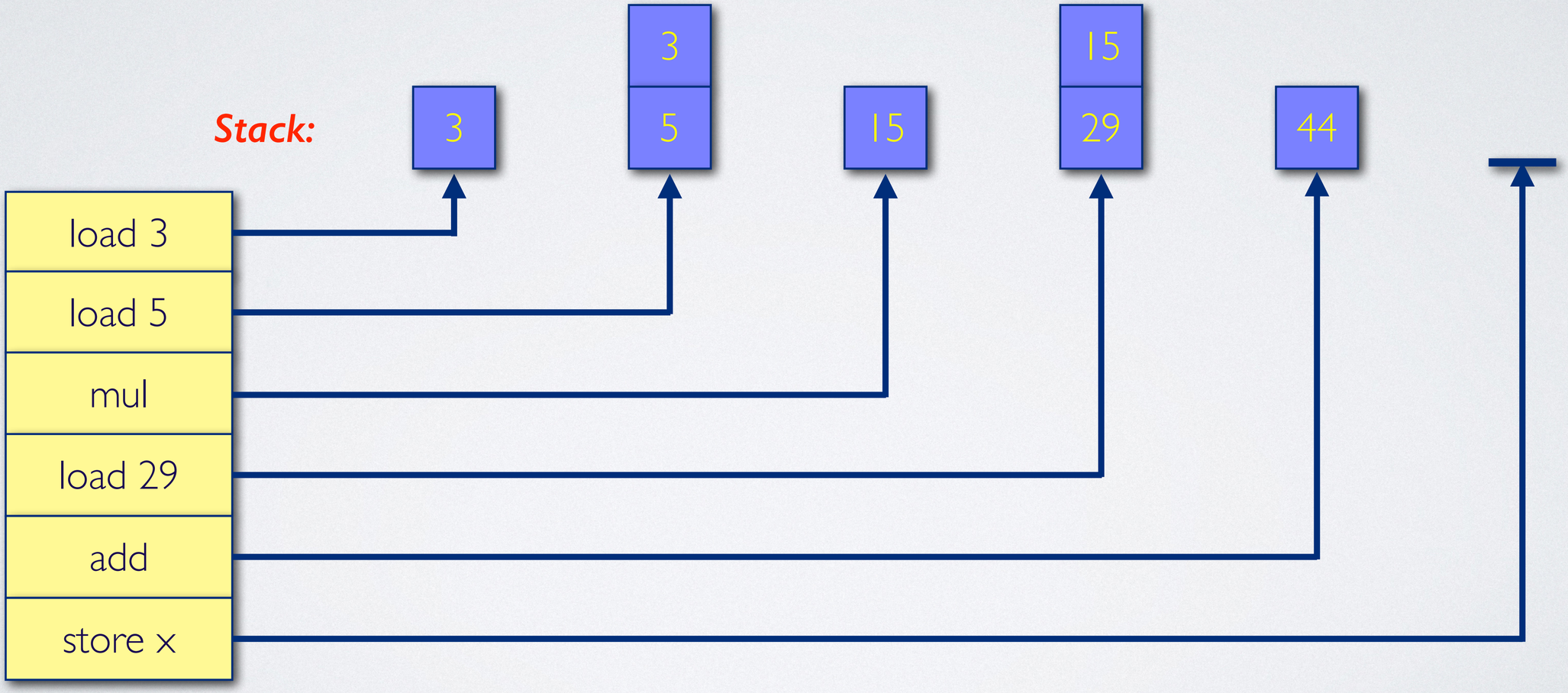
RPN: a b * c +



- RPN is used in stack machines that employ a pushdown stack rather than individual machine registers to evaluate sub-expressions in a program. A stack machine is programmed with an RPN instruction set (sometimes called P-code in reference to Pascal's RPN implementation).

RPN CALCULATION

- x = 3 * 5 + 29;



STACK CONSTRUCTOR

```
Stack() noexcept :
    fElements(),
    fStackPointer(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 stack pointer is set to zero. The stack is empty initially.

STACK PUSH

```
void push( const T& aValue )
{
    assert( fStackPointer < N );

    fElements[fStackPointer++] = aValue;
}
```

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

STACK POP

```
void pop()
{
    assert( fStackPointer > 0 );

    fStackPointer--;
}
```

- The pop operation decreases the stack pointer. The top element will be freed later, on the next push operation or when the stack object goes out of scope.
- The assert statement checks for a stack underflow error. This test is run in Debug mode and suppressed in the Release version.
- The running time of pop() is O(1).

STACK TOP

```
std::optional<T> top()
{
    if ( fStackPointer > 0 )
    {
        return std::optional<T>( fElements[fStackPointer - 1] );
    }
    else
    {
        return std::optional<T>{};
    }
}
```

- The top operation returns the most recently inserted element, if it exists.
- C++17 provides the vocabulary type std::optional<T>. A common use case for optional is the return value of a function that may fail. Any instance of std::optional<T> at any given point in time either contains a value or does not contain a value. When an object of type std::optional<T> is contextually converted to bool (e.g., condition in **if**-statement), the conversion returns true if the object contains a value and false otherwise. 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).

STACK POP(K) - MULTIPOP

```
void pop( size_t k )
{
    while ( fStackPointer > 0 && k != 0 )
    {
        pop();
        k--;
    }
}
```

- The pop(k) operation removes the k top elements from the stack or pops the entire stack if it contains less than k elements.
- The running time of pop(k) is O(n). The number of iterations of the while-loop is min(fStackPointer, k). The running time is a linear function of the cost min(fStackPointer, k).

AMORTIZED ANALYSIS - AGGREGATE METHOD

- In the aggregate method, we show that for all n, a sequence of n operations takes $T(n) = W(n)$, worst-case time, in total.

- In worst-case, the average cost, or amortized cost, per operation is $T(n)/n$. This amortized cost applies to each operation, even if there are varying kinds of operations in the sequence.

FUNDAMENTAL STACK OPERATIONS

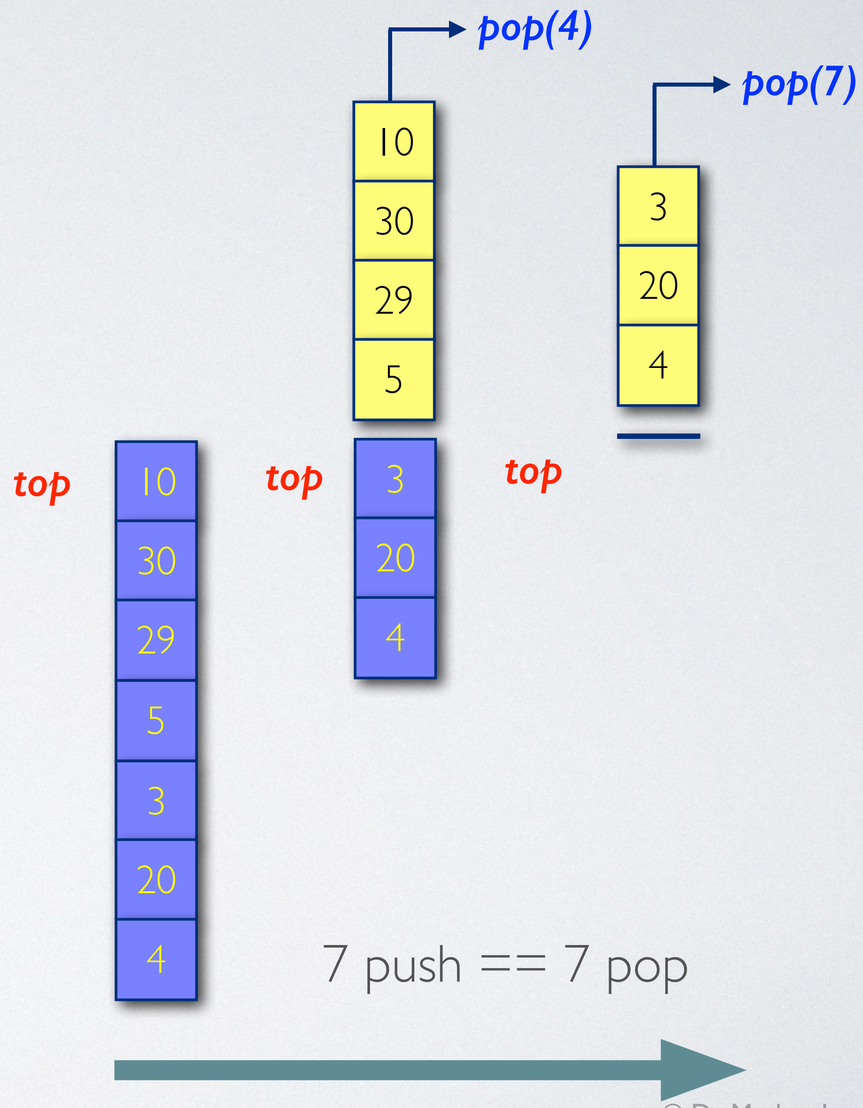
- The fundamental stack operations all take $O(1)$ time:
  - The operation push(aValue) pushes aValue onto the stack
  - The operation pop() pops the top of the stack.
  - The operation top() returns the top element of the stack, if it exists.
- Since each operation runs in $O(1)$ time, let the cost of each operation be 1.
- The total cost of a sequence of n push, pop, and top operations is n, and the actual running time for n operations is $\Theta(n)$. (The amortized cost of an operation is the average $\Theta(n)/n = \Theta(1)$.)

STACK OPERATIONS PLUS POP(K)

- Let |s| be the current number of elements on the stack. Then the total cost of pop(k) is min(|s|, k), and the actual running time is a linear function.
- The worst-case cost of a pop(k) in a sequence of n push, pop, top, and pop(k) on an initially empty stack is O(n) since the stack size is at most n.
- The worst-case time of any stack operation is O(n), and a sequence of n operations costs $O(n^2)$ since we may have O(n) pop(k) operations costing O(n) each.
- But $O(n^2)$ is not tight.

AGGREGATE ANALYSIS POP(K)

- Using the aggregate method of amortized analysis, we obtain a cost of at most O(n) for a sequence of n push, pop, top, and pop(k):
  - Each element can be popped at most once for each time it is pushed. Therefore, the number of pop operations, including pop(k), equals the number of push operations, at most n.
  - For any value of n, any sequence of n push, pop, top, and pop(k) operations takes O(n) time. The amortized cost of an operation is the average: O(n)/n = O(1).



AMORTIZED ANALYSIS - ACCOUNTING METHOD

- The accounting method assigns different charges to various operations; some are charged more or less than others.
- The amount we charge an operation is called its amortized cost.
- Some operations can get charged more than their actual costs. The difference is assigned to other elements in the data structure as credit. Credit can be used later to pay for operations whose amortized costs are lower than their actual cost.
- The accounting method differs from the aggregate method, in which all operations have the same amortized cost.

AMORTIZED COST DESIGN

- The goal of the accounting method is to show that in the worst case, the average cost per operation is small. The amortized cost of a sequence of operations must be an upper bound on the total costs of the sequence.

- This relationship must hold for all sequences of operations.

- **Note**: The total credit associated with the data structure must be non-negative. Otherwise, the amortized cost would not be an upper bound on the total costs.

STACK OPERATIONS

	Actual Costs	Amortized Costs
push	1	2
pop	1	0
top	1	1
pop(k)	k' = min(|s|, k)	0

All amortized costs are O(1).

ACCOUNTING METHOD EXAMPLE

	# Push	# Pop	Cost	Credit
push	1		2	1
push	1		2	1
top			1	
pop		1	0	
push	1		2	1
push	1		2	1
push	1		2	1
pop(3)		3	0	
top			1	
pop		1	0	
10(12)	5	5	12	5

The total amortized cost is O(n).

AMORTIZED ANALYSIS - POTENTIAL METHOD

- Instead of representing prepaid work as credit, the potential method of amortized analysis models prepaid work as "potential energy" or just "potential" that can be released to pay for future operations.

- The potential is associated with the whole data structure rather than specific elements/operations.

POTENTIAL METHOD - APPROACH

- The potential method starts with an initial data structure D_0 on which n operations are performed. For each $i = 1, 2, \ldots, n$, we let c_i be the actual cost of the ith operation and D_i be the data structure that results after applying the ith operation to data structure D_{i-1}. A potential function Φ maps each data structure D_i to a real number $\Phi(D_i)$, which is the potential associated with data structure D_i.

- The amortized cost $\underline{c}_i$ of the ith operation is given by

$$\underline{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

- The amortized cost of each operation is its actual cost plus the change in potential due to the operation. The total amortized cost of n operations is sum all $\underline{c}_i$, $i = 1, 2, \ldots, n$.

- If we can define a potential function Φ so that $\Phi(D_n) \geq \Phi(D_0)$, then the total amortized cost is an upper bound on the total. In practice, we do not always know how many operations might be performed, so we require $\Phi(D_i) \geq \Phi(D_0)$ for all i, which means we pay in advance.

STACK OPERATIONS

- We can define the potential function Φ on a stack as the number of elements stored.
- We start with the empty stack D_0 and have $\Phi(D_0) = 0$.
- Since the number of elements on the stack is never negative, the stack D_i that results after the ith operation has nonnegative potential: $\Phi(D_i) \geq 0 = \Phi(D_0)$.
- The total amortized cost of n operations concerning Φ represents an upper bound of the actual.

Φ - STACK OPERATIONS

	$\Phi(D_i) - \Phi(D_{i-1})$	Actual Cost	Amortized Cost
push	$\|s\| + 1 - \|s\| = 1$	1	2
pop	$\|s\| - 1 - \|s\| = -1$	1	0
top	$\|s\| - \|s\| = 0$	1	1
pop(k)	$k' = \min(\|s\|, k)$ $\|s\| - k' - \|s\| = -k'$	k'	0

All amortized costs are O(1), and since $\Phi(D_i) \geq \Phi(D_0)$ for all i, the amortized cost of n operations is an upper bound of the total cost. The worst-case cost of n operations is therefore O(n).

AMORTIZED COST OF INSERTION AND DELETION

- When designing a stack, we may not know how many elements will be stored on the stack.
- If the original stack size is too small, it must be reallocated with a larger size, and all elements stored in the original stack must be copied onto the new, larger stack.
- Similarly, if elements have been removed from the stack, it can be worthwhile to reallocate the stack with a smaller size and copy all elements stored in the original stack onto the new, smaller stack.
- Using amortized analysis, we can show that the amortized cost of insertion and deletion is only O(1), even though the actual cost of an operation is large when it triggers an expansion or a contraction.

LOAD FACTOR (STACK)

- To control expansion and contraction, we use a concept called load factor. The load factor is the number of elements stored divided by the size of slots available in the data structure (e.g., stack).
- An empty stack has size 0, and the corresponding load factor is 1.
- If the load factor of a stack is bounded below by a constant, the unused space in the stack is never more than a constant fraction of the total amount of space.
- For expansion, a common heuristic is to double the space each time the stack reaches a load factor of 1. Consequently, the load factor is always at least 1/2, and the space wasted, due to expansion, never exceeds half the total space available.
- For contraction, we allow the load factor to drop below 1/2. Specifically, we half the stack size when the load factor becomes 1/4. After resizing, the smaller stack has a load factor of at least 1/2 again.

DYNAMIC STACK

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

   DynamicStack();
   ~DynamicStack();

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

   std::optional<T> top() const 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();
};
```

DynamicStack is not copyable.

We require a pointer to stack array as it is dynamically allocated.

Reallocation features

DYNAMICSTACK CONSTRUCTOR

```
DynamicStack() :
        fElements(new T[1]),
        fStackPointer(0),
        fCapacity(1)
    {}
```

- The default constructor initializes all member variables with sensible values:
  - We allocate at least one slot for a dynamic stack. Hence, the load factor is initially 0. Starting with zero slots initially requires more logic later.
  - The stack pointer is set to zero. The stack is empty initially.
  - Memory allocation can fail. For this reason, the default constructor is not marked **noexcept**.

DYNAMICSTACK DESTRUCTOR

```
~DynamicStack()
{
    delete[] fElements;
}
```

- When a stack-based object goes out of scope, C++ automatically calls the destructor, a special member function responsible for freeing resources.
- DynamicStack objects maintain a heap array of type T. The array has at least one element.
- **Delete[]** first deletes all objects stored in the array fElements and second frees the heap array fElements. This sequence is essential in avoiding memory leaks in C++.

DISABLE COPY SEMANTICS

```
DynamicStack( const DynamicStack& ) = delete;
DynamicStack& operator=( const DynamicStack& ) = delete;
```

- An important part of class design in C++ is copy control.
- Copy control defines what has to happen if an object is assigned to a variable, passed as a parameter to a function, or returned from a function as a result.
- If the user does not explicitly define copy control, the C++ compiler synthesizes the necessary operations (e.g., copy constructor and assignment operator). Synthesized operations only perform shallow copy, which is not a viable approach when an object maintains heap memory.
- Rather than explicitly defining copy control, we disable it for class DynamicStack. This renders DynamicStack objects non-copyable in the same way stream objects cannot be copied.

DYNAMICSTACK PUSH

```
void push( const T& aValue )
{
    ensure_capacity();

    fElements[fStackPointer++] = aValue;
}
```

- The push operation takes a constant reference to a value of type T and copies the referenced value into the array fElements at the current stack pointer. Afterwards, the stack pointer points to the next free element.
- At the start, we ensure the stack has sufficient space to store aValue. This may result in an expansion of the stack.

DYNAMICSTACK POP

```
void pop()
{
    assert( fStackPointer > 0 );

    fStackPointer—;
    adjust_capacity();
}
```

- The pop operation decreases the stack pointer. The top element will be freed later, on the next push operation or when the stack object goes out of scope.
- The assert statement checks for a stack underflow error. This test is run in Debug mode and suppressed in the Release version.
- Finally, we adjust the stack size if necessary. This may result in a contraction of the stack.

DYNAMICSTACKTOP

```
std::optional<T> top() const noexcept
{
    if ( fStackPointer > 0 )
    {
        return std::optional<T>( fElements[fStackPointer - 1] );
    }
    else
    {
        return std::optional<T>();
    }
}
```

- The top operation returns the most recently inserted element, if it exists.
- Changing the storage to a dynamically allocated heap object requires no code changes. In C++, we can use array indexing even if the target is a pointer.
- The running time of top() is O(1).

TRIGGERING RESIZING

```
void ensure_capacity()
{
    // Is load factor 1?
    if ( fStackPointer == fCapacity )
    {
        resize( fCapacity * 2 );
    }
}

void adjust_capacity()
{
    // Is load factor 1/4?
    if ( fStackPointer <= fCapacity / 4 )
    {
        resize( fCapacity / 2 );
    }
}
```

- The method ensure_capacity() tests whether the stack has reached a load factor 1. This happens when the stack pointer equals the capacity of the stack. In this case, we reallocate the stack to double the space. The new, larger stack has a load factor of at least 1/2.

- The method adjust_capacity() tests if the stack has reached a load factor 1/4. This happens when the stack pointer is less than or equal to a quarter of the stack size. In this case, we reallocate the stack to half the space. The new, smaller stack has a load factor of at least 1/2.

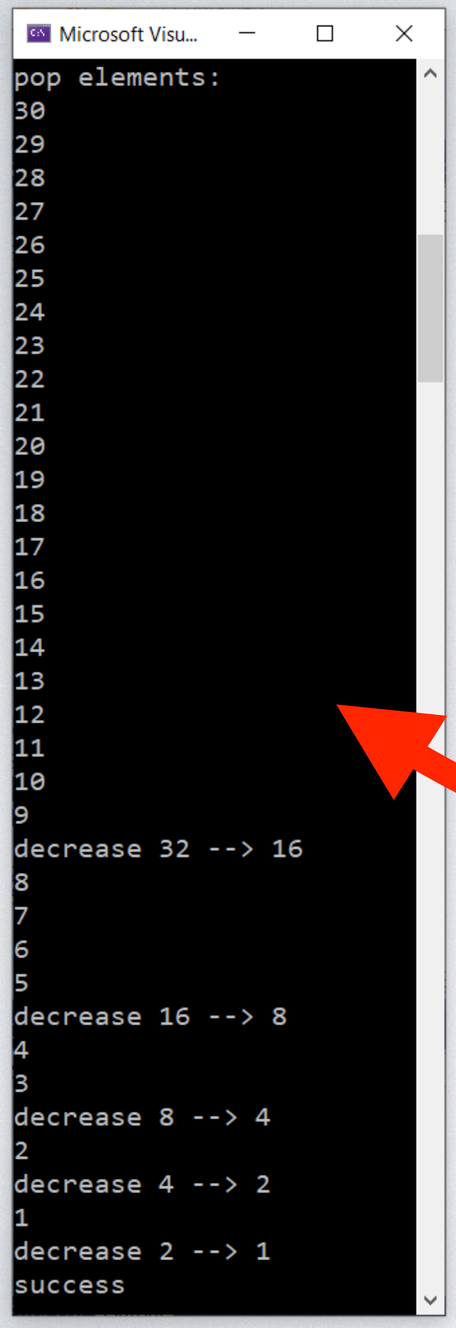
DYNAMICSTACK RESIZE

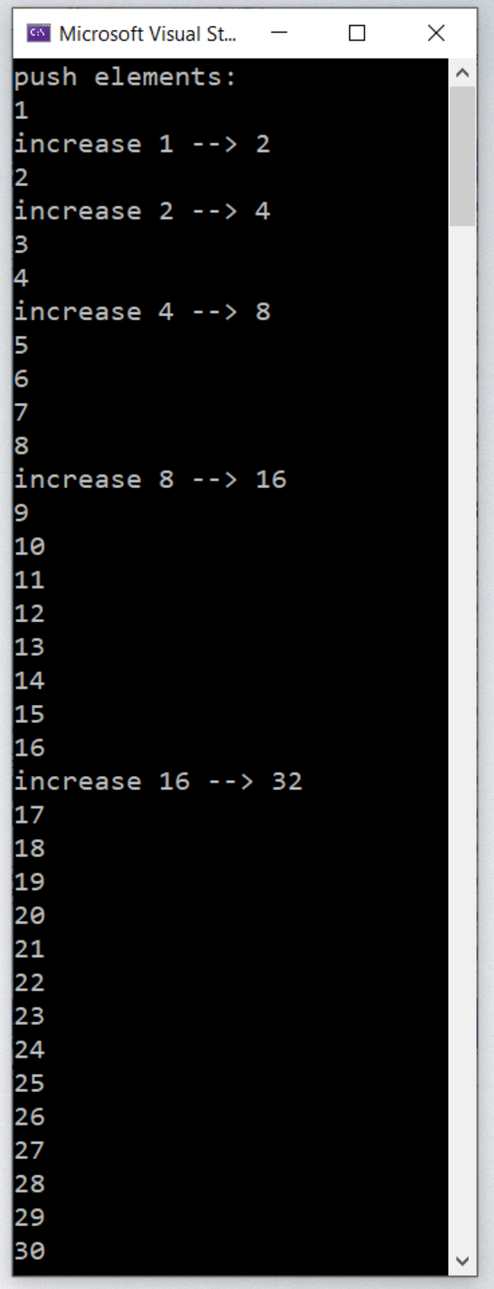
```
void resize( size_t aNewCapacity )
{
    assert( fStackPointer <= aNewCapacity );

    T* lNewElements = new T[aNewCapacity];

    for ( size_t i = 0; i < fStackPointer; i++ )
    {
        lNewElements[i] = std::move( fElements[i] );
    }

    delete[] fElements;

    fElements = lNewElements;
    fCapacity = aNewCapacity;
}
```

- The method resize() starts with a safety check. It verifies the constraint that the new stack space must be large enough to accommodate the current stack content.
- The current content is moved to the new stack space. The template function std::move forces move semantics. Move semantics allows objects to be moved, rather than copied, when their lifetime expires. The data is transferred from one object to another. In most cases, the data is not moved physically in memory.
- After moving the current content to the new stack space, the old stack space is freed, and the member variables are updated accordingly.
- The running time of method resize() is O(n). It is the **for**-loop, moving elements to their new location, that dominates the running time.

DYNAMICSTACK TEST

```
DynamicStack<int> IStack;

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

for ( int i = 1; i <= 30; i++ )
    IStack.push( i );

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

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

while ( IValue )
{
    std::cout << *Ivalue << std::endl;
    IStack.pop();
    IValue = IStack.top();
}

if ( IValue )
    std::cout << "failure" << std::endl;
else
    std::cout << "success" << std::endl;
```





DYNAMICSTACK PUSH- AMORTIZED ANALYSIS

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
table size	1	2	4	4	8	8	8	8	16	16	16	16	16	16	16	16	32	32
insert cost	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
resize cost		1	2		4				8								16	
total cost	1	2	3	1	5	1	1	1	9	1	1	1	1	1	1	1	17	1

- What is the cost c_i of the *i*th push operation?
  - If there is room in the stack, then $c_i = 1$. We only perform one insert operation.
  - If the stack is full, $c_i = i$, an expansion occurs. We perform i-1 copy operations plus one insert operation.
- If n push operations are performed, the worst-case cost of a push operation is O(n), which leads to an upper bound of $O(n^2)$. But this is not tight because the stack is only so often expanded. Most of the time, we require one insert operation.

AMORTIZED ANALYSIS - AGGREGATE METHOD

- We first observe that expansion only occurs for the *i*th push operation if *i* - 1 is an exact power of 2.
- The cost of the *i*th push operation is

$$c_i = \begin{cases} 2^k + 1 & \text{if } i = 2^k + 1 \\ 1 & \text{otherwise} \end{cases}$$

- The total cost of n push operations is, therefore

$$\sum_{i=1}^{n} c_i = n + \sum_{k=0}^{\lfloor log_2 n \rfloor} 2^k = n + 2n - 1 < 3n$$

since there are n insert operations that cost 1 and the cost of the copy operations form a geometric series. Since the total cost of n stack push is 3n, the amortized cost of a single push operation is 3, that is, O(1).

- The analysis of n pop operations yields a similar result. The amortized cost of a single pop is O(1).