Chapter 18 - Stacks and Queues

CS 202

Objectives (1 of 2)

In this chapter, you will:

- Learn about stacks
- Examine various stack operations
- Learn how to implement a stack as an array
- Learn how to implement a stack as a linked list
- Learn about infix, prefix, and postfix expressions, and how to use a stack to evaluate postfix expressions

Objectives (2 of 2)

- Learn how to use a stack to remove recursion
- Learn about queues
- Examine various queue operations
- Learn how to implement a queue as an array
- Learn how to implement a queue as a linked list
- Discover how to use queues to solve simulation problems

Stacks (1 of 4)

- **Stack**: a data structure in which elements are added and removed from one end only
    - Addition/deletion occur only at the top of the stack
    - **Last in first out (LIFO)** data structure
- Operations:
    - **Push**: to add an element onto the stack
    - **Pop**: to remove an element from the stack
    - **Top**: retrieves the current top element without removing it

Stacks (2 of 4)

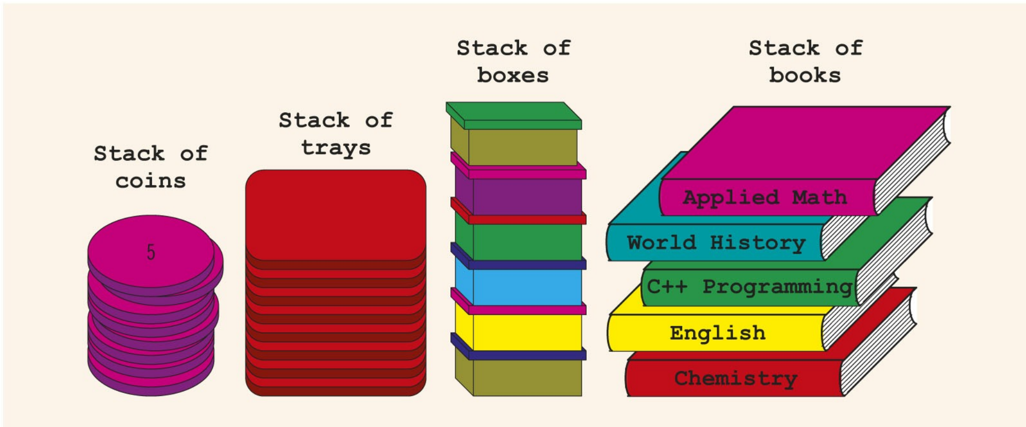

FIGURE 18-1 Various types of stacks

Stacks (3 of 4)

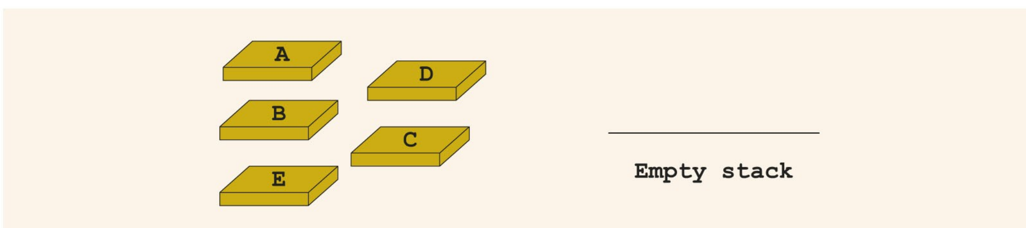

FIGURE 18-2 Empty stack

Stacks (4 of 4)

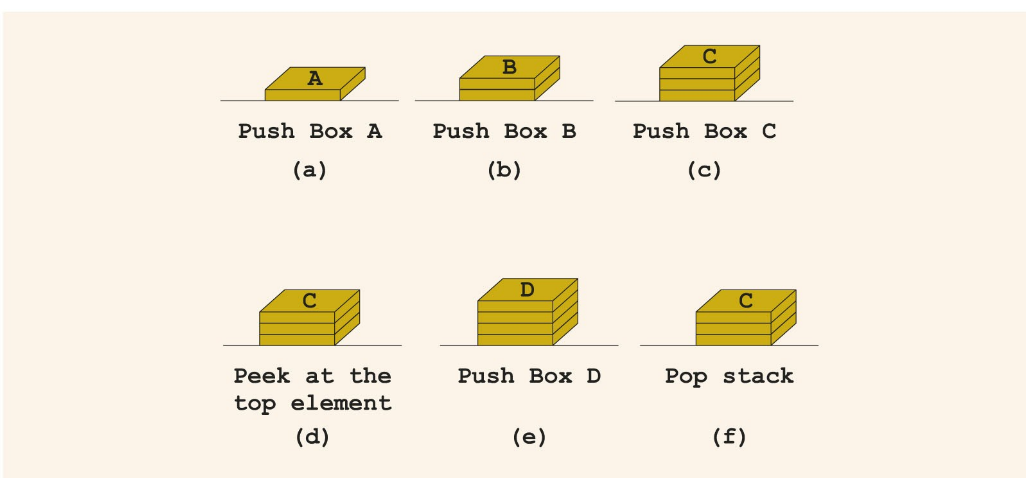

FIGURE 18-3 Stack operations

Stack Operations

- In the abstract class **stackADT**:
    - initializeStack
    - isEmptyStack
    - isFullStack
    - push
    - top
    - pop

UML Class Diagram of the Class stackADT

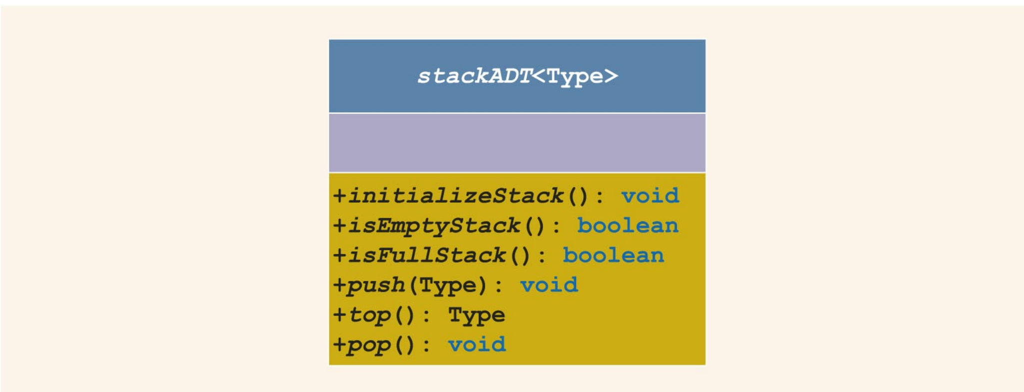

FIGURE 18-4 UML class diagram of the `class stackADT`

Implementation of Stacks as Arrays (1 of 5)

- First element goes in first array position, second in the second position, etc.
- Top of the stack is index of the last element added to the stack
- Stack elements are stored in an array, which is a random access data structure
    - Stack element is accessed only through the **stackTop**
- To track the top position, use a variable called **stackTop**

Implementation of Stacks as Arrays (2 of 5)

- Can dynamically allocate the array
    - Enables the user to specify the size of the array
- The class **stackType** implements the functions of the abstract class **stackADT**

Implementation of Stacks as Arrays (3 of 5)

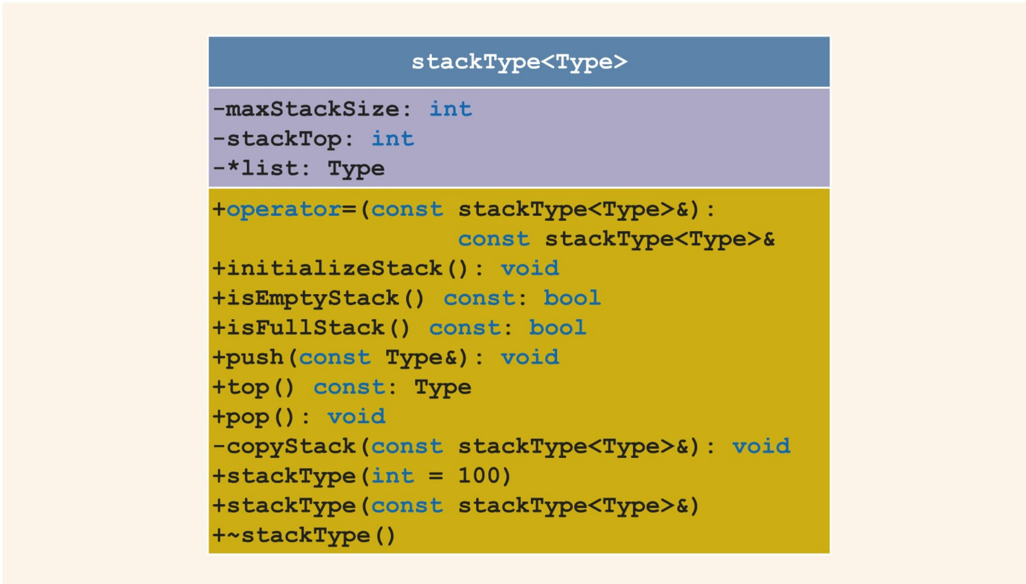

FIGURE 18-5 UML class diagram of the `class stackType`

Implementation of Stacks as Arrays (4 of 5)

- C++ arrays begin with the index 0
    - Must distinguish between:
        - Value of **stackTop**
        - Array position indicated by **stackTop**
- If **stackTop** is 0, the stack is empty
- If **stackTop** is nonzero, the stack is not empty
    - Top element is given by **stackTop - 1**

Implementation of Stacks as Arrays (5 of 5)

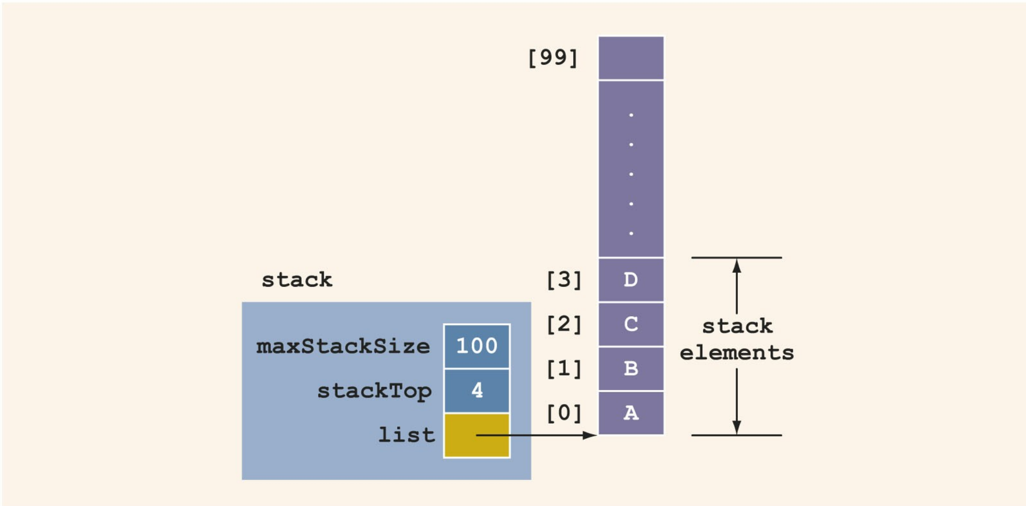

FIGURE 18-6 Example of a stack

Initialize Stack

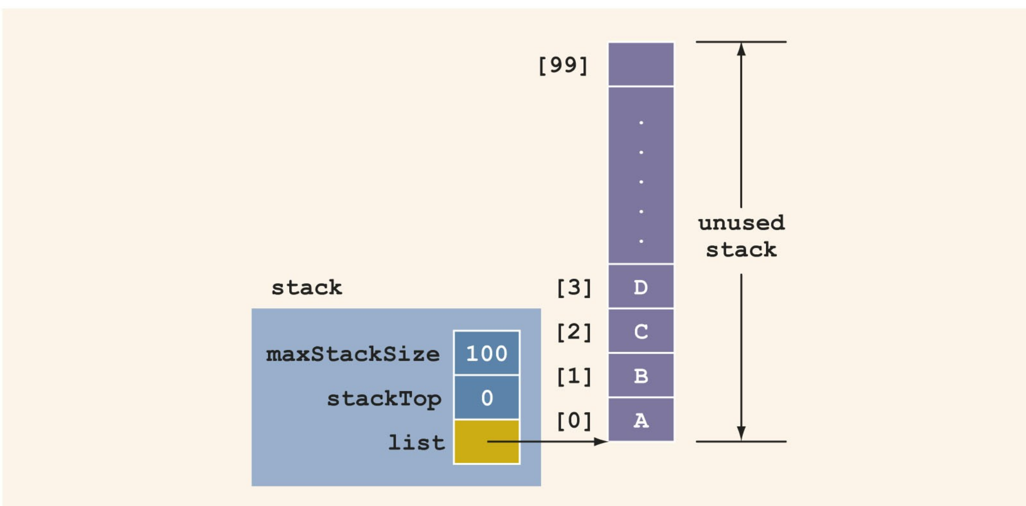

FIGURE 18-7 Empty stack

Empty Stack/Full Stack

- Stack is empty if `stackTop == 0`

```
template <class T>
bool stackType<T>::isEmptyStack() const {
    return stackTop == 0;
}
```

- Stack is full if `stackTop == maxStackSize`

```
template <class T>
bool stackType<T>::isFullStack() const {
    return stackTop == maxStackSize;
}
```

Push (1 of 3)

- Store the **newItem** in the array component indicated by **stackTop**
- Increment **stackTop**
- **Overflow** occurs if we try to add a new item to a full stack

Push (2 of 3)

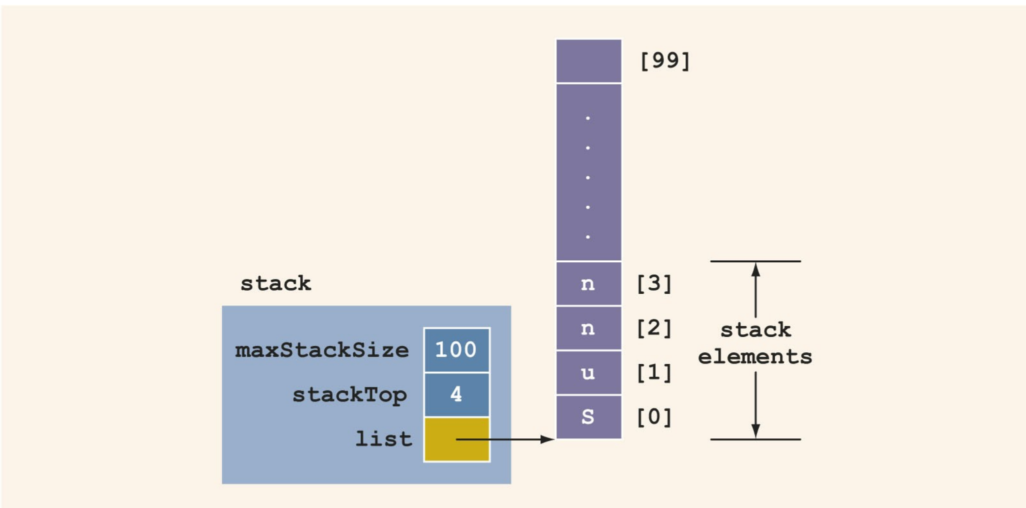

FIGURE 18-8 Stack before pushing y

Push (3 of 3)

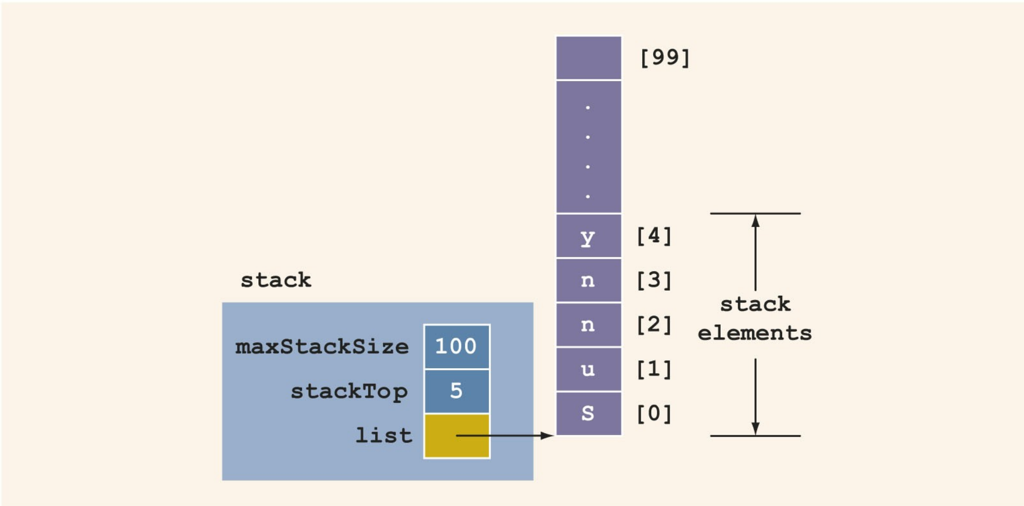

FIGURE 18-9 Stack after pushing y

Return the Top Element

- **Top** operation:
  - Returns the top element of the stack

```
template <class T>
T stackType<T>::top() const {
    assert(stackTop != 0);
    return list[stackTop - 1];
}
```

Pop (1 of 3)

- To remove an element from the stack, decrement **stackTop** by 1
- **Underflow** condition: trying to remove an item from an empty stack

```
template <class T>
void stackType<T>::pop() {
    if (!isEmptyStack()) {
        --stackTop;
    }
}
```

Pop (2 of 3)

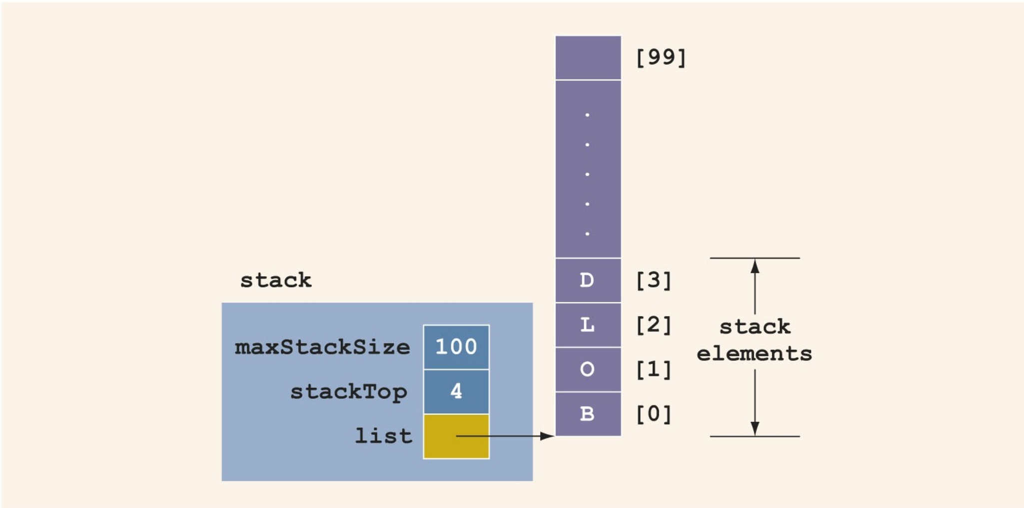

FIGURE 18-10 Stack before popping D

Pop (3 of 3)

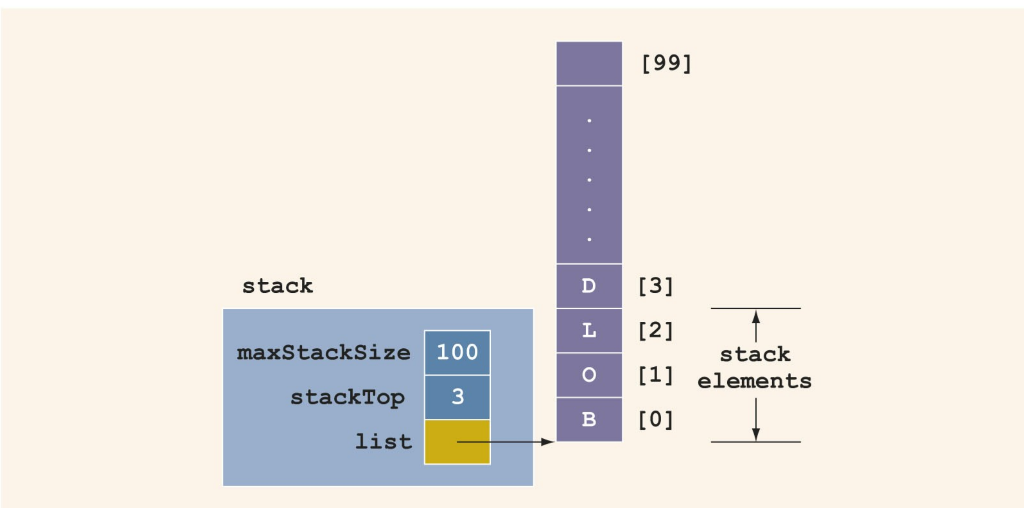

FIGURE 18-11 Stack after popping D

Copy Stack

- **copyStack function**: copies a stack

```
template <class T>
void stackType<T>::copyStack(const stackType<T>& other) {
    delete[] list;
    maxStackSize = other.maxStackSize;
    stackTop = other.stackTop;
```

```
        list = new T[maxStackSize];

        for (int i = 0; i < stackTop; ++i) {
            list[i] = other.list[i];
        }
    }
```

Constructor and Destructor

- Constructor:
  - Sets stack size to parameter value (or default value if not specified)
  - Sets **stackTop** to 0
  - Creates array to store stack elements
- Destructor:
  - Deallocates memory occupied by the array
  - Sets **stackTop** to 0

Copy Constructor

- Copy constructor:
  - Called when a stack object is passed as a (value) parameter to a function
  - Copies values of member variables from the actual parameter to the formal parameter

Overloading the Assignment Operator (=) (1 of 2)

- Assignment operator must be explicitly overloaded because of pointer member variables

```
template <class T>
stackType<T>& stackType<T>::operator=(const stackType<T>& other) {
    if (this != &other) {
        copyStack(other);
    }
    return *this;
}
```

Stack Header File

- Place definitions of the class and functions (stack operations) together in a file
  - Called `myStack.h`

Linked Implementation of Stacks (1 of 2)

- Array only allows a fixed number of elements
- If the number of elements to be pushed exceeds array size, the program may terminate
- Linked lists can dynamically organize data
- In a linked representation, **stackTop** is a pointer to the memory address of the top element in the stack

Linked Implementation of Stacks (2 of 2)

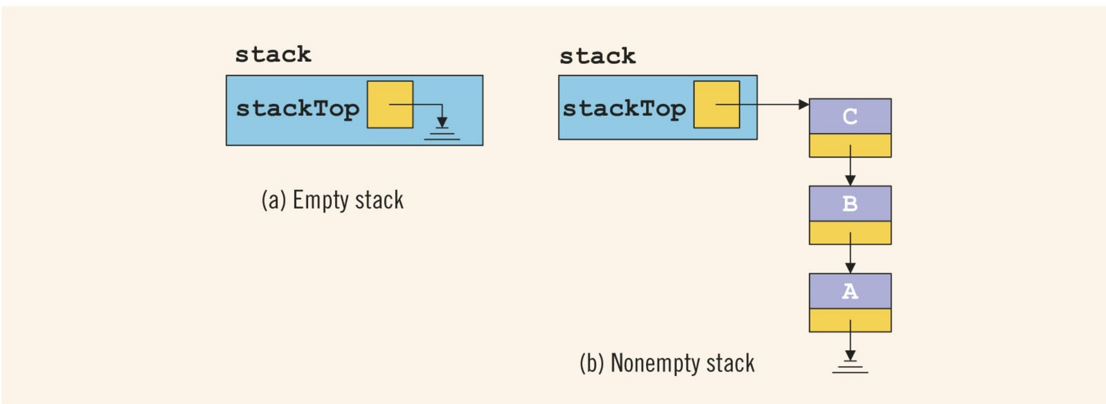

FIGURE 18-12 Empty and nonempty linked stack

Empty and nonempty linked stack

Default Constructor

- Initializes the stack to an empty state when a stack object is declared
  - Sets **stackTop** to `nullptr`

```
template <class T>
linkedStackType<T>::linkedStackType() {
     stackTop = nullptr;
}
```

Empty Stack and Full Stack

- In a linked implementation of stacks, function `isFullStack` does not apply
  - Logically, the stack is never full
- Stack is empty if `stackTop` is `nullptr`

Linked Stack: Initialize Stack

- `initializeStack`: reinitializes the stack to an empty state
  - Must deallocate memory occupied by the current elements

- ○ Sets `stackTop` to `nullptr`

Push (1 of 2)

- **newNode** is added at the beginning of the linked list pointed to by **stackTop**

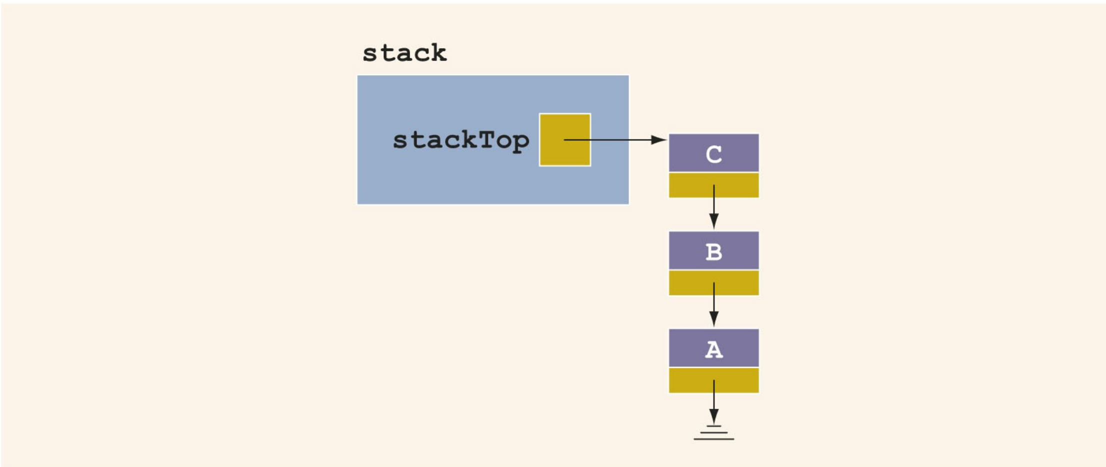

FIGURE 18-13 Stack before the push operation

Stack before the push operation

Push (2 of 2)

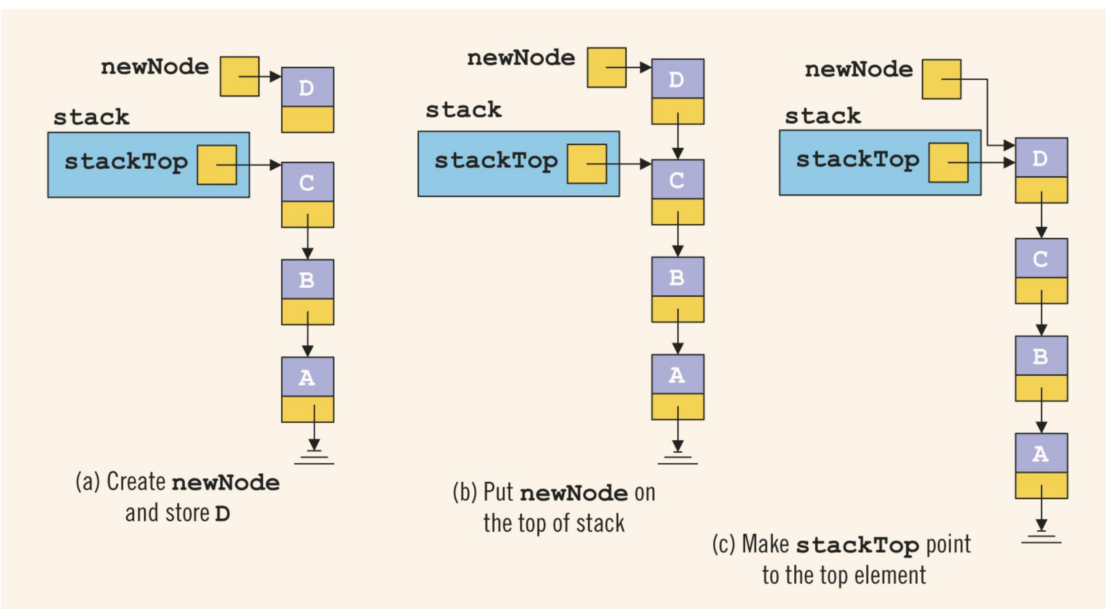

FIGURE 18-14 Push operation

Push operation

Linked Stack: Return the Top Element

```
template <class T>
T& linkedStackType<T>::top() {
    assert(stackTop != nullptr);
    return stackTop->info;
}
```

Pop (1 of 2)

- Node pointed to by **stackTop** is removed
  - The second element becomes the top element

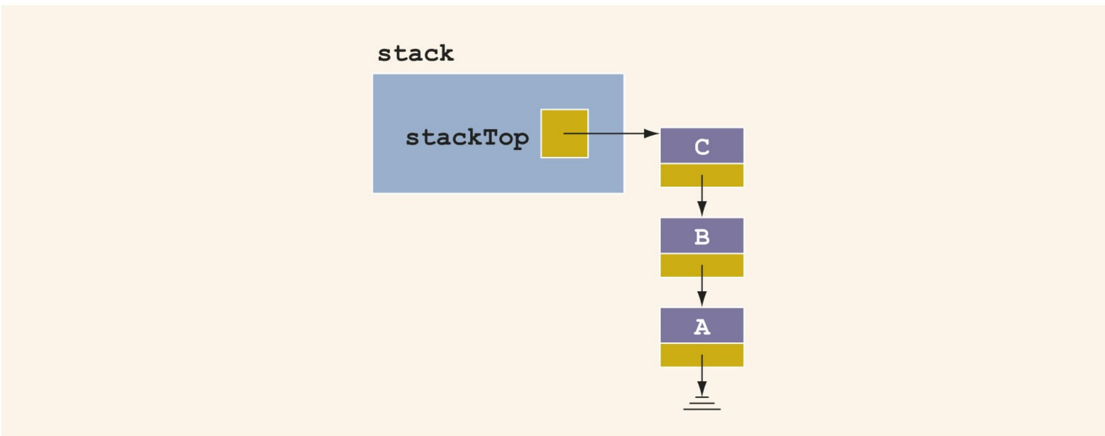

FIGURE 18-15 Stack before the pop operation

Stack before the pop operation

Pop (2 of 2)

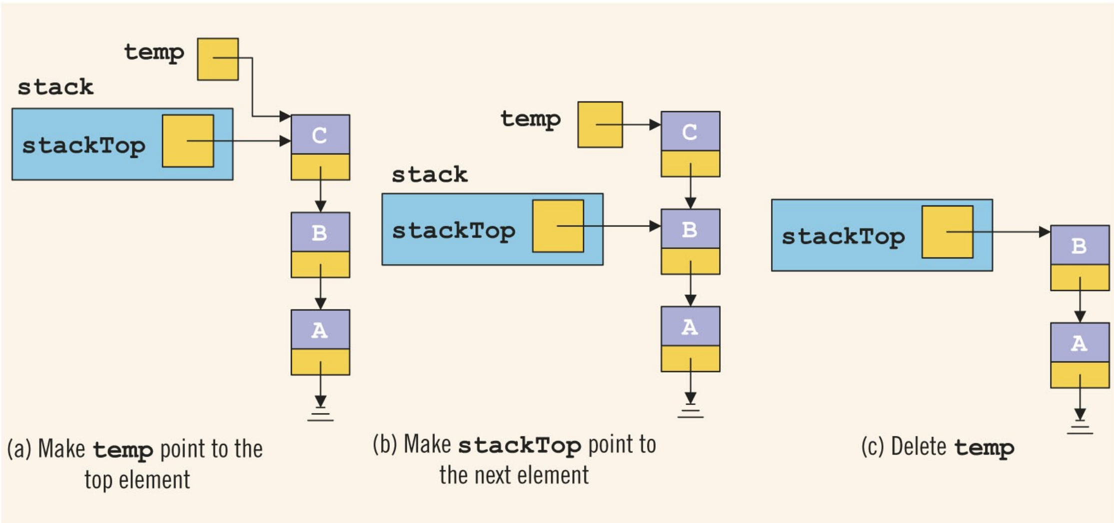

FIGURE 18-16 Pop operation

Pop operation

Linked Stack: Copy Stack

- **copyStack function**: makes an identical copy of a stack - Similar definition to `copyList` for linked lists

Constructors and Destructors

- Copy constructor and destructor:
    - Similar to those for linked lists

```
// copy constructor
template <class T>
linkedStackType<T>::linked

StackType(const linkedStackType<T>& other) {
    stackTop = nullptr;
    copyStack(other);
}

// destructor
template <class T>
linkedStackType<T>::~linkedStackType() {
    initializeStack();
}
```

Overloading the Assignment Operator (=) (2 of 2)

- Overloading the assignment operator:

```
template <class T>
linkedStackType<T>& linkedStackType<T>::operator=(const
linkedStackType<T>& other) {
    if (this != &other) {
        copyStack(other);
    }
    return *this;
}
```

Stack Derived from the Class unorderedLinkedList

- Implementation of push is similar to `insertFirst` for general lists
- Other similar functions:
  - `initializeStack` and `initializeList`
  - `isEmptyList` and `isEmptyStack`
- `linkedStackType` can be derived from `linkedListType`
  - Class `linkedListType` is abstract
- `unorderedLinkedListType` is derived from `linkedListType`
  - Provides the definitions of the abstract functions of the class `linkedListType`
- `linkedStackType` is derived from `unorderedLinkedListType`

Application of Stacks: Postfix Expressions Calculator (1 of 8)

- **Infix notation**: usual notation for writing arithmetic expressions
  - Operator is written between the operands
  - Example: `a + b`
  - Evaluates from left to right
  - Operators have precedence
  - Parentheses can be used to override precedence

Application of Stacks: Postfix Expressions Calculator (2 of 8)

- **Prefix (Polish) notation**: operators are written before the operands
  - Introduced by the Polish mathematician Jan Lukasiewicz in the early 1920s
  - Parentheses can be omitted
  - Example: `+ a b`

Application of Stacks: Postfix Expressions Calculator (3 of 8)

- **Reverse Polish notation**: operators *follow* the operands (postfix operators)
  - Proposed by Australian philosopher and early computer scientist Charles L. Hamblin in the late 1950s
  - Advantage: operators appear in the order required for computation
  - Example: `a + b * c` becomes `a b c * +`

Application of Stacks: Postfix Expressions Calculator (4 of 8)

EXAMPLE 18-4

Infix Expression	Equivalent Postfix Expression
a + b	a b +
a + b * c	a b c * +
a * b + c	a b * c +
(a + b) * c	a b + c *
(a – b) * (c + d)	a b – c d + *
(a + b) * (c – d / e) + f	a b + c d e / – * f +

UnTable18-01

Application of Stacks: Postfix Expressions Calculator (5 of 8)

- Postfix notation has important applications in computer science
  - Many compilers first translate arithmetic expressions into postfix notation and then translate this expression into machine code
- Evaluation algorithm:
  - Scan expression from left to right
  - When an operator is found, back up to get operands, perform the operation, and continue

Application of Stacks: Postfix Expressions Calculator (6 of 8)

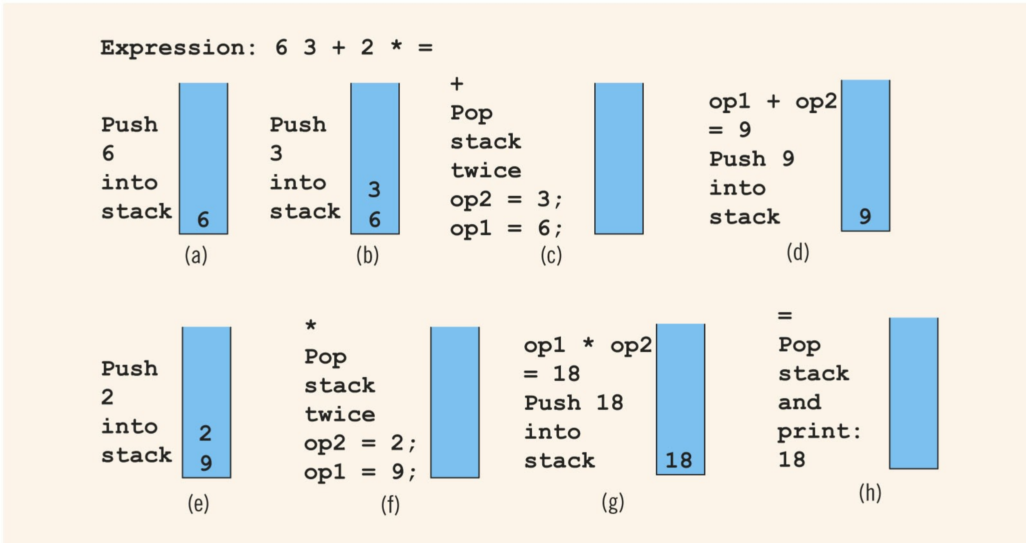

FIGURE 18-17 Evaluating the postfix expression: 6 3 + 2 * =

*Evaluating the postfix expression: 6 3 + 2 * =*

Application of Stacks: Postfix Expressions Calculator (7 of 8)

- Symbols can be numbers or anything else:
  - +, -, *, and / are operators, requiring two operands
    - Pop the stack twice and evaluate the expression
    - If the stack has less than two elements, error
  - If the symbol is =, the expression ends
    - Pop and print answer from stack
    - If the stack has more than one element, error
- If symbol is anything else:
  - Expression contains an illegal operator

Application of Stacks: Postfix Expressions Calculator (8 of 8)

- Assume postfix expressions are in this form: #6 #3 + #2 * =
  - If the symbol scanned is #, the next input is a number
  - If the symbol scanned is not #, then it is:
    - An operator (may be illegal) or
    - An equal sign (end of expression)
- Assume expressions contain only +, -, *, and / operators

Main Algorithm

- Pseudocode:

```
Read the first character
while not the end of input data; do
    a. initialize the stack
    b. process the expression
    c. output result
    d. get the next expression
done
```

- Four functions are needed:
    - `evaluateExpression`, `evaluateOpr`, `discardExp`, and `printResult`

evaluateExpression

- Function `evaluateExpression`:
    - Evaluates each postfix expression
    - Each expression ends with = symbol

```
void evaluateExpression(ifstream& inF,
                        ofstream& outF,
                        stackType<double>& stack,
                        char& ch,
                        bool& isValid)
```

evaluateOpr

- Function `evaluateOpr`:
    - Evaluates an expression
    - Needs two operands saved in the stack
        - If less than two, then error
    - Also checks for illegal operations

discardExp

- Function `discardExp`:
    - Called when an error is discovered in the expression
    - Reads and writes input data until the '='

printResult

- The function `printResult`: If the postfix expression contains no errors, it prints the result
    - Otherwise, it outputs an appropriate message

- Result of the expression is in the stack, and output is sent to a file

Nonrecursive Algorithm to Print a Linked List Backward (1 of 7)

- To print a list backward non-recursively, first get to the last node of the list
  - Problem: Links go in only one direction
  - Solution: Save a pointer to each node in a stack
    - Uses the LIFO principle
- Since the number of nodes is usually unknown, use the linked implementation of a stack

Nonrecursive Algorithm to Print a Linked List Backward (2 of 7)

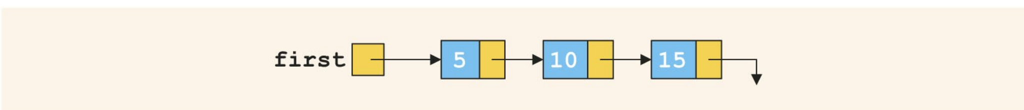

FIGURE 18-18 Linked list

Linked list

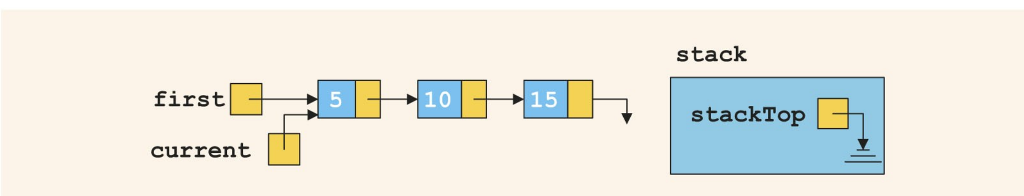

FIGURE 18-19 List after the statement `current = first;` executes

List after the statement `current = first;` *executes*

Nonrecursive Algorithm to Print a Linked List Backward (3 of 7)

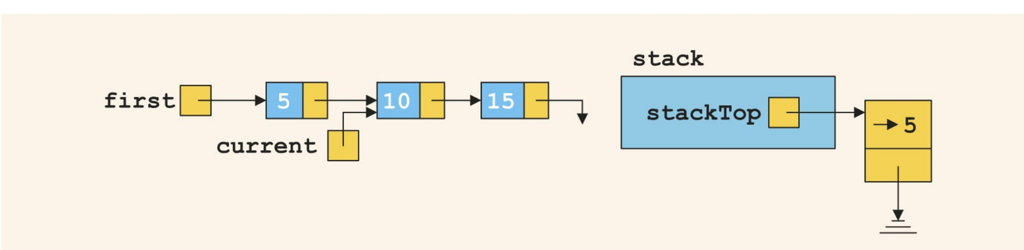

FIGURE 18-20 List and stack after the statements `stack.push(current);` and `current = current->link;` execute

Nonrecursive Algorithm to Print a Linked List Backward (4 of 7)

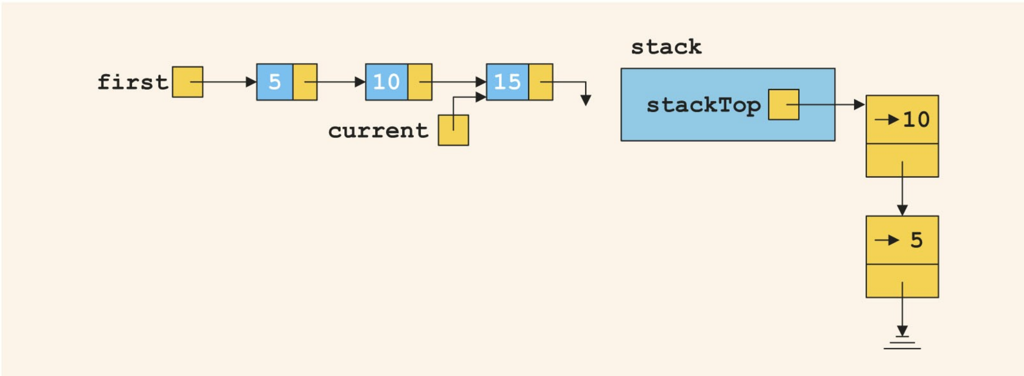

FIGURE 18-21 List and stack after the statements `stack.push(current);` and `current = current->link;` execute

Nonrecursive Algorithm to Print a Linked List Backward (5 of 7)

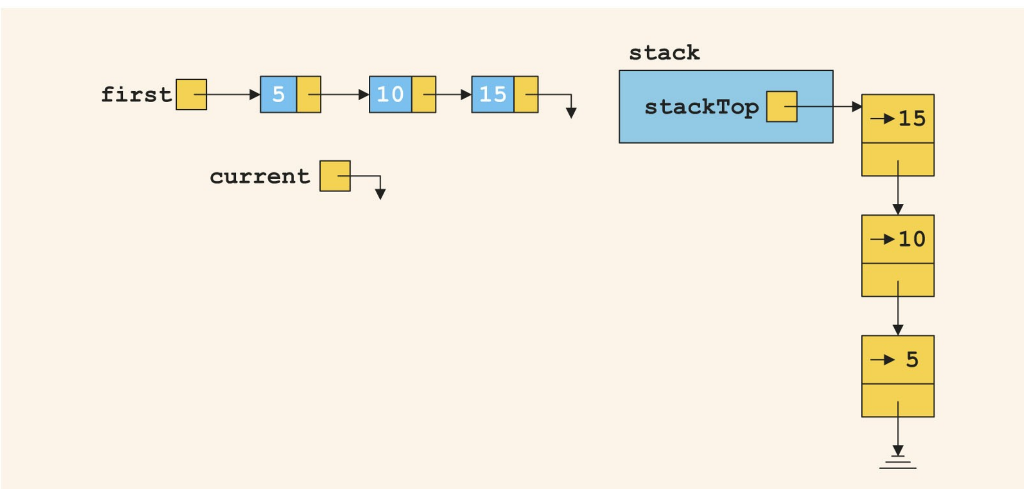

FIGURE 18-22 List and stack after the statements `stack.push(current);` and `current =current->link;` execute

Nonrecursive Algorithm to Print a Linked List Backward (6 of 7)

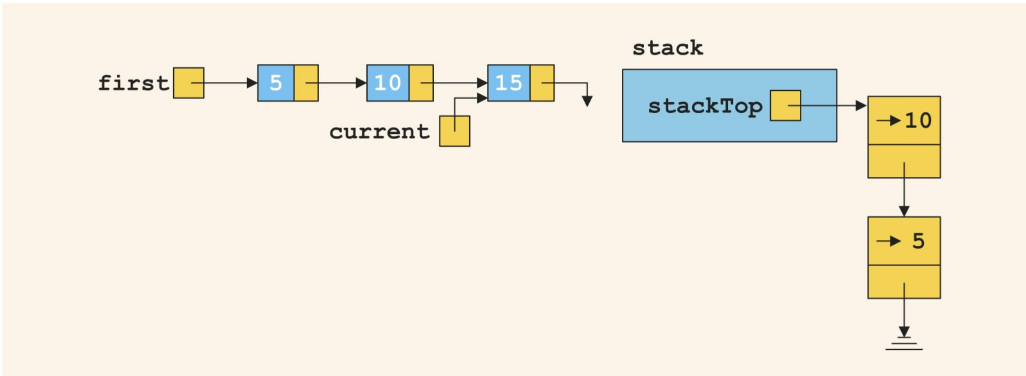

FIGURE 18-23 List and stack after the statements `current = stack.top();` and `stack. pop();` execute

Fig18-23

Nonrecursive Algorithm to Print a Linked List Backward (7 of 7)

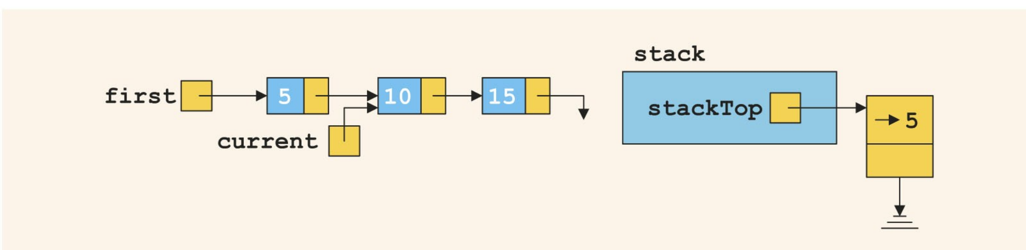

FIGURE 18-24 List and stack after the statements `current=stack.top();` and `stack.pop();` execute

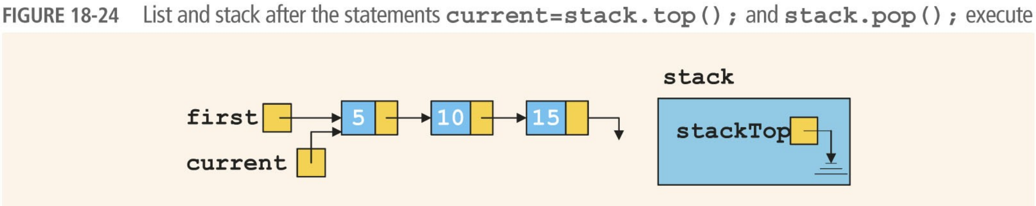

FIGURE 18-25 List and stack after the statements `current=stack.top();` and `stack.pop();` execute

Stack derived from LList

```
#include "LList.hpp"

template <class T>
class Stack : protected LList<T> {
public:
    // Type aliases
```

```
    using value_type = T;
    using size_type  = std::size_t;

    // Member functions
    bool empty() const;          ///< Checks if stack is empty
    size_type size() const;      ///< Returns number of elements in
stack
    T& top();                    ///< Returns a reference to top
element
    void push(const T& value);  ///< Pushes an element onto stack
    void pop();                  ///< Removes top element
};  // class Stack
```

Queues

- **Queue**: set of elements of the same type
- Elements are:
    - Added at one end (the **back** or **rear**)
    - Deleted from the other end (the **front**)
- **First In First Out (FIFO)** data structure
    - Middle elements are inaccessible
- Example:
    - Waiting line in a bank

Queue Operations

- Queue operations include: – **initializeQueue** – **isEmptyQueue** – **isFullQueue** – **front** – **back** – **addQueue** – **deleteQueue**
- Abstract class **queueADT** defines these operations

Implementation of Queues as Arrays (1 of 15)

- Need at least four (member) variables:
    - Array to store queue elements
    - **queueFront** and **queueRear**
        - To track first and last elements
    - **maxQueueSize**
        - To specify the maximum size of the queue

Implementation of Queues as Arrays (2 of 15)

- To add an element to the queue:
    - Advance **queueRear** to the next array position

- Add element to position pointed by **queueRear**

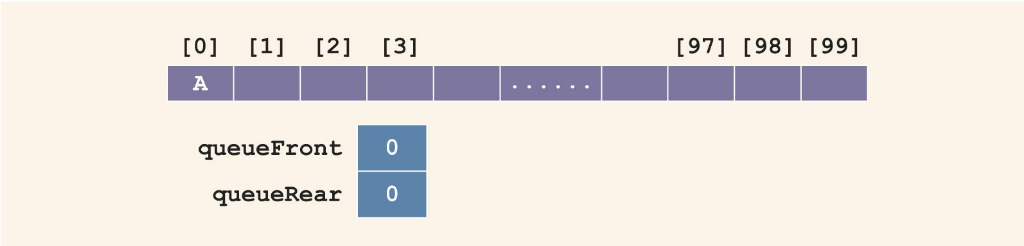

FIGURE 18-26 Queue after the first `addQueue` operation

Implementation of Queues as Arrays (3 of 15)

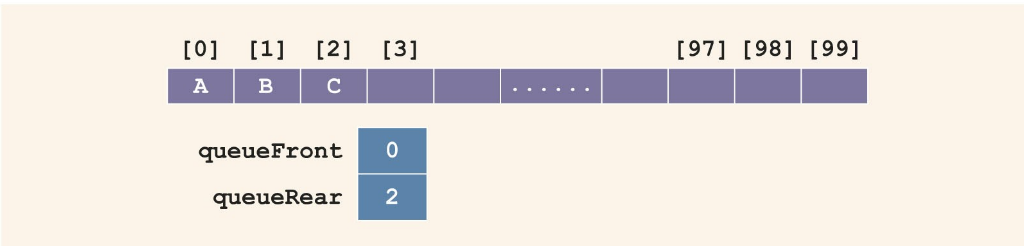

FIGURE 18-27 Queue after two more `addQueue` operations

Implementation of Queues as Arrays (4 of 15)

- To delete an element from the queue:
  - Retrieve the element pointed to by **queueFront**
  - Advance **queueFront** to the next queue element

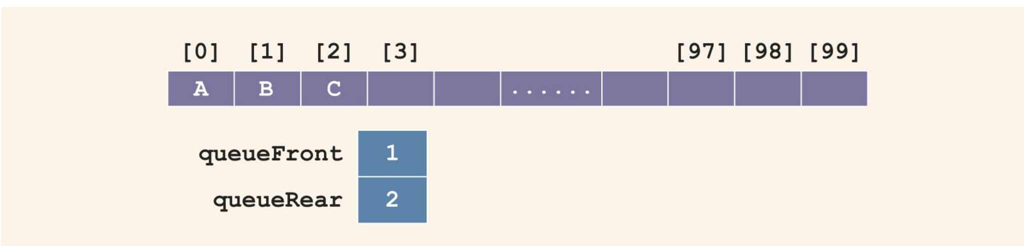

FIGURE 18-28 Queue after the `deleteQueue` operation

Implementation of Queues as Arrays (5 of 15)

- Will this queue design work?
  - Let **A** represent adding an element to the queue
  - Let **D** represent deleting an element from the queue
  - Consider the following sequence of operations:
    - **AAADADADADADADADA**…

Implementation of Queues as Arrays (6 of 15)

- This would eventually set **queueRear** to point to the last array position
  - Giving the impression that the queue is full

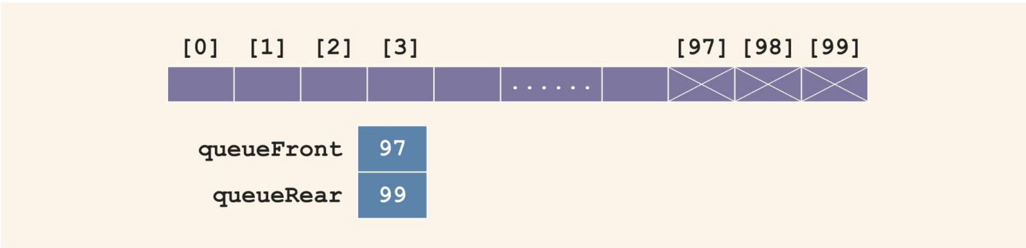

FIGURE 18-29 Queue after the sequence of operations AAADADADADADA . . .

Implementation of Queues as Arrays (7 of 15)

- Solution 1: When the queue overflows at the rear (**queueRear** points to the last array position):
  - Check the value of **queueFront**
  - If **queueFront** indicates there is room at the front of the array, slide all queue elements toward the first array position
  - Problem: too slow for large queues
- Solution 2: Assume

that the array is circular

Implementation of Queues as Arrays (8 of 15)

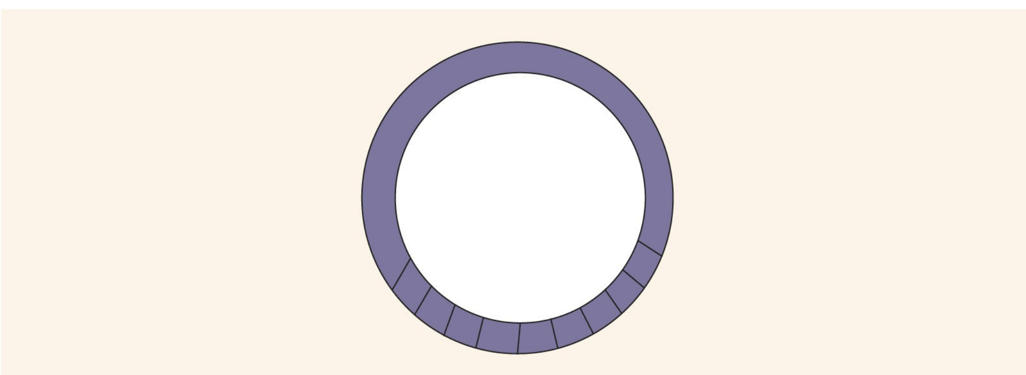

FIGURE 18-30 Circular queue

Linked Implementation of Queues (1 of 2)

- Array implementation has issues:

    - Array size is fixed: only a finite number of queue elements can be stored in it
    - Requires array to be treated in a special way, together with **queueFront** and **queueRear**
- Linked implementation of a queue simplifies many issues
    - Queue is never full because memory is allocated dynamically

Linked Implementation of Queues (2 of 2)

- Elements are added at one end and removed from the other
    - Need only two pointers to maintain the queue: **queueFront** and **queueRear**

Empty and Full Queue

- Queue is empty if `queueFront` is nullptr
- Queue is never full
    - Unless the system runs out of memory

Note: must provide `isFullQueue` function definition because it is an abstract function in parent class **queueADT**

Linked Queue: Initialize Queue

- Initializes queue to an empty state
    - Must remove all existing elements, if any
    - Deallocates memory occupied by elements

addQueue, front, back, and deleteQueue operations

- `addQueue` operation: adds new element to end of queue
- `front` operation: returns first element of queue
- `back` operation: returns last element of queue
- `deleteQueue` operation: removes first element of queue

Linked Queue: Constructors and Destructors

- Constructor
    - Accesses **maxQueueSize**, **queueFront**, and **queueRear**
- Destructor: destroys the queue
    - Deallocates memory occupied by elements
- Copy constructor and overloading assignment operator:
    - Similar to corresponding functions for stack

Queue Derived from the Class unorderedLinkedListType

- Linked implementation of queue: similar to implementation of a linked list created in a forward manner
    - `addQueue` similar to `insertFirst`
    - `initializeQueue` is like `initializeList`
    - `isEmptyQueue` similar to `isEmptyList`
    - `deleteQueue` can be implemented as before
    - **queueFront** is same as **first**
    - **queueRear** is same as **last**

Queue Derived from LList

```
#include "LList.hpp"

template <class T>
class Queue : protected LList<T> {
public:
    // Type aliases
    using value_type = T;
    using size_type  = std::size_t;

    // Member functions
    bool empty() const;           ///< Checks if queue is empty
    size_type size() const;       ///< Returns number of elements in
queue
    T& front();                   ///< Returns a reference to front
element
    T& back();                    ///< Returns a reference to back
element
    void push(const T& value);    ///< Pushes an element to back of
queue
    void pop();                   ///< Removes front element
};  // class Queue
```

Application of Queues: Simulation

- **Simulation**: a technique in which one system models the behavior of another system
- Computer models are used to study the behavior of real systems
- **Queuing systems**: computer simulations using queues as the data structure
    - Queues of objects are waiting to be served

Designing a Queuing System (1 of 3)

- **Server**: object that provides the service
- **Customer**: object receiving the service
- **Transaction time**: service time, or the time it takes to serve a customer
- **Model**: system that consists of a list of servers and a waiting queue holding the customers to be served
    - Customer at the front of the queue waits for the next available server

Designing a Queuing System (2 of 3)

- Need to know:
    - Number of servers
    - Expected arrival time of a customer
    - Time between the arrivals of customers
    - Number of events affecting the system
- Performance of the system depends on:
    - How many servers are available
    - How long it takes to serve a customer
    - How often a customer arrives

Designing a Queuing System (3 of 3)

- If it takes too long to serve a customer and customers arrive frequently, then more servers are needed
    - System can be modeled as a time-driven simulation
- **Time-driven simulation**: the clock is a counter
    - The passage of one unit of time can be implemented by incrementing a counter by 1
    - Simulation is run for a fixed amount of time

Customer

```
customerType
-customerNumber: int
-arrivalTime: int
-waitingTime: int
-transactionTime: int
+setCustomerInfo(int = 0, int = 0, int = 0,
                 int = 0): void
+getWaitingTime() const: int
+setWaitingTime(int): void
+incrementWaitingTime(): void
+getArrivalTime() const: int
+getTransactionTime() const: int
+getCustomerNumber() const: int
+customerType(int = 0, int = 0, int = 0,
              int = 0)
```

FIGURE 18-36 UML class diagram of the class customerType

Server

```
serverType
-currentCustomer: customerType
-status: string
-transactionTime: int
+isFree() const: bool
+setBusy(): void
+setFree(): void
+setTransactionTime(int): void
+setTransactionTime(): void
+getRemainingTransactionTime() const: int
+decreaseTransactionTime(): void
+setCurrentCustomer(customerType): void
+getCurrentCustomerNumber() const: int
+getCurrentCustomerArrivalTime() const: int
+getCurrentCustomerWaitingTime() const: int
+getCurrentCustomerTransactionTime() const: int
+serverType()
```

FIGURE 18-37 UML class diagram of the class serverType

Server List

- Server list: a set of servers
    - At any given time, a server is either free or busy

```
serverListType
-----------------------------------------------
-numOfServers: int
-*servers: serverType
-----------------------------------------------
+getFreeServerID() const: int
+getNumberOfBusyServers() const: int
+setServerBusy(int, customerType, int): void
+setServerBusy(int, customerType): void
+updateServers(ostream&): void
+serverListType(int = 1)
+~serverListType()
```

FIGURE 18-38 UML class diagram of the `class serverListType`

Waiting Customers Queue

- When a customer arrives, they go to the end of the queue
- When a server becomes available, the customer at the front of the queue leaves to conduct the transaction
- After each time unit, the waiting time of each customer in the queue is incremented by 1
- Can use queueType but must add an operation to increment waiting time

Main Program

- Algorithm for main loop:

```
for (clock = 1; clock <= simulationTime; clock++)
{
```

2.1. Update the server list to decrement the transaction time of each busy server by one time unit.

2.2. If the customer's queue is nonempty, increment the waiting time of each customer by one time unit.

2.3. If a customer arrives, increment the number of customers by 1 and add the new customer to the queue.

2.4. If a server is free and the customer's queue is nonempty, remove a customer from the front of the queue and send the customer to the free server.

```
}
```

Summary (1 of 2)

- **Stack**: items are added/deleted from one end

- Last In First Out (LIFO) data structure
- Operations: push, pop, initialize, destroy, check for empty/full stack
- Can be implemented as an array
- Middle elements should not be accessed directly

- **Postfix notation**: operators are written after the operands (no parentheses needed)

Summary (2 of 2)

- **Queue**: items are added at one end and removed from the other end
    - First In First Out (FIFO) data structure
    - Operations: add, remove, initialize, destroy, check if queue is empty/full
    - Can be implemented as an array
    - Middle elements should not be accessed directly
    - Is a restricted version of an array or linked list

Questions?