Stacks and Queues

Modified from section 13.2

A Linked List Application

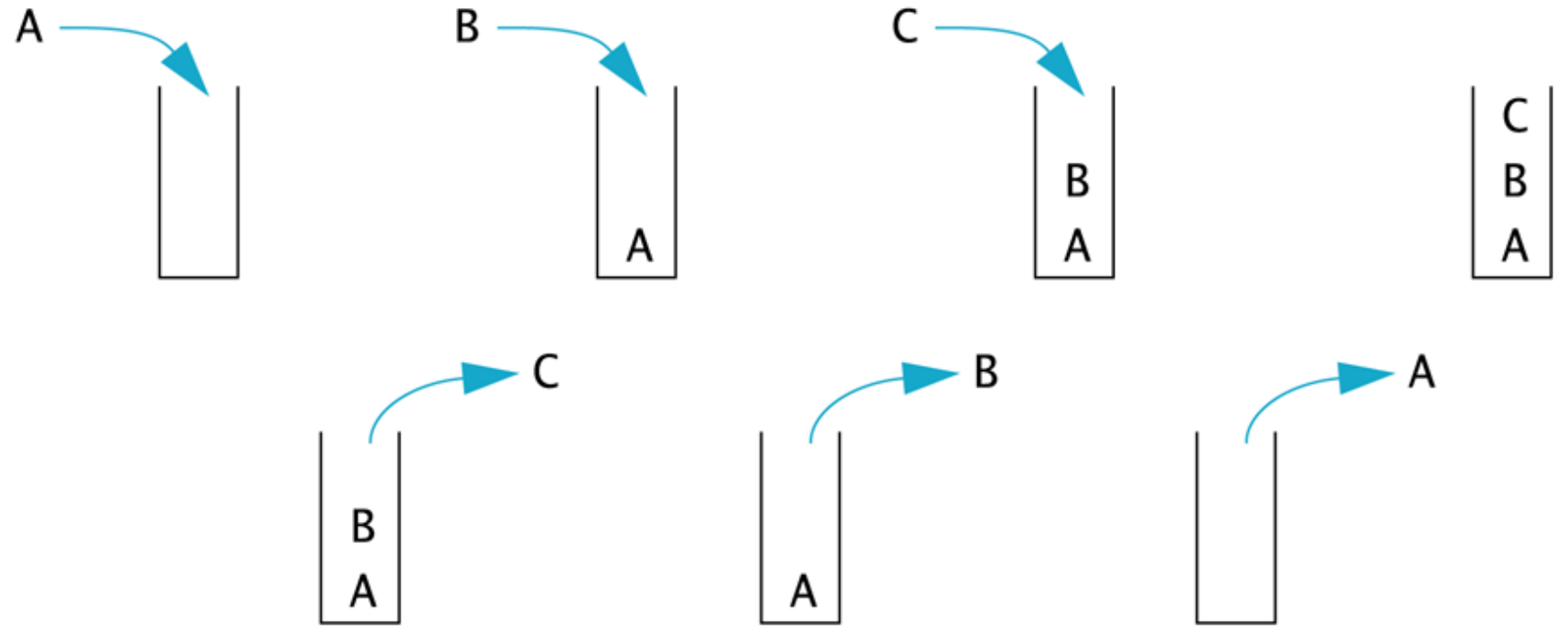
- A stack is a data structure that retrieves data in the reverse order the data was stored
  - If 'A', 'B', and then 'C' are placed in a stack, they will be removed in the order 'C', 'B', and then 'A'
- A stack is a last-in/first-out data structure like the stack of plates in a cafeteria; adding a plate pushes down the stack and the top plate is the first one removed

Display 13.16

Display 13.16

Back Next

A Stack



Program Example: A Stack Class

- We will create a stack class to store characters
  - Adding an item to a stack is pushing onto the stack
  - Member function push will perform this task
  - Removing an item from the stack is popping the item off the stack
  - Member function pop will perform this task
- **Display 13.17** contains the stack class interface

Display 13.17

Back Next

Interface File for a Stack Class

```
//This is the header file stack.h. This is the interface for the class Stack,
//which is a class for a stack of symbols.
#ifndef STACK_H
#define STACK_H
namespace stacksavitch
{
    struct StackFrame
    {
        char data;
        StackFrame *link;
    };

    typedef StackFrame* StackFramePtr;

    class Stack
    {
    public:
        Stack( );
        //Initializes the object to an empty stack.

        Stack(const Stack& a_stack);
        //Copy constructor.

        ~Stack( );
        //Destroys the stack and returns all the memory to the freestore.

        void push(char the_symbol);
        //Postcondition: the_symbol has been added to the stack.

        char pop( );
        //Precondition: The stack is not empty.
        //Returns the top symbol on the stack and removes that
        //top symbol from the stack.

        bool empty( ) const;
        //Returns true if the stack is empty. Returns false otherwise.
    private:
        StackFramePtr top;
    };
}//stacksavitch

#endif //STACK_H
```

Display 13.18 (1/2)

Program Using the Stack Class (*part 1 of 2*)

```
//Program to demonstrate use of the Stack class.
#include <iostream>
#include "stack.h"
using namespace std;
using namespace stacksavitch;

int main( )
{
    Stack s;
    char next, ans;

    do
    {
        cout << "Enter a word: ";
        cin.get(next);
        while (next != '\n')
        {
            s.push(next);
            cin.get(next);
        }

        cout << "Written backward that is: ";
        while ( ! s.empty( ) )
            cout << s.pop( );
        cout << endl;

        cout << "Again?(y/n): ";
        cin >> ans;
        cin.ignore(10000, '\n');
    }while (ans != 'n' && ans != 'N');

    return 0;
}
```

The `ignore` *member of* `cin` *is discussed in Chapter 11. It discards input remaining on the current input line up to 10,000 characters or until a return is entered. It also discards the return (*`'\n'`*) at the end of the line.*

Display 13.18 (2/2)

Program Using the Stack Class (*part 2 of 2*)

Sample Dialogue

```
Enter a word: straw
Written backward that is: warts
Again?(y/n): y
Enter a word: C++
Written backward that is: ++C
Again?(y/n): n
```

Function push

- The push function adds an item to the stack
  - It uses a parameter of the type stored in the stack

```
void push(char the_symbol);
```

  - Pushing an item onto the stack is precisely the same task accomplished by function head_insert of the linked list
  - In the stack class, a pointer named top is used instead of a pointer named head

Function pop

- The pop function returns the item that was at the top of the stack
  - Declaration: char pop();
  - Before popping an item from a stack, pop checks that the stack is not empty
    - An empty stack is identified by setting the top pointer to NULL,
      top = NULL;
  - pop stores the top item in a local variable result, and the item is "popped" by: top = top->link;
    - A temporary pointer must point to the old top item so it can be "deleted" to prevent a memory leak
  - pop then returns variable result

The Copy Constructor

- Because the stack class uses a pointer and creates new nodes using new, a copy constructor is needed
  - The copy constructor must make a copy of each item in the stack and store the copies in a new stack
    - Items in the new stack must be in the same position in the stack as in the original

The stack destructor

- Because function pop calls delete each time an item is popped off the stack, ~stack only needs to call pop until the stack is empty

```
char next;
while( ! empty ( ) )
{
      next = pop( );
}
```

stack Class Implementation

- The stack class implementation is found in **Display 13.19**

Display 13.19 (1/3)

Implementation of the Stack Class (*part 1 of 2*)

```
//This is the implementation file stack.cpp.
//This is the implementation of the class Stack.
//The interface for the class Stack is in the header file stack.h.
#include <iostream>
#include <cstddef>
#include "stack.h"
using namespace std;

namespace stacksavitch
{
    //Uses cstddef:
    Stack::Stack( ) : top(NULL)
    {
        //Body intentionally empty.
    }
```

Display 13.19 (2/3)

Implementation of the Stack Class (*part 2 of 2*)

```
Stack::~Stack( )
{
    char next;
    while (! empty( ))
       next = pop( );//pop calls delete.
}


//Uses cstddef:
bool Stack::empty( ) const
{
    return (top == NULL);
}

//Uses iostream:
char Stack::pop( )
{
    if (empty( ))
    {
        cout << "Error: popping an empty stack.\n";
        exit(1);
    }

    char result = top->data;

    StackFramePtr temp_ptr;
    temp_ptr = top;
    top = top->link;

    delete temp_ptr;

    return result;
}
```

Display 13.19 (3/3)

Back Next

```
    //Uses cstddef:
    Stack::Stack(const Stack& a_stack)
    {
        if (a_stack.top == NULL)
            top = NULL;
        else
        {
            StackFramePtr temp = a_stack.top;//temp moves
                //through the nodes from top to bottom of
                //a_stack.
            StackFramePtr end;//Points to end of the new stack.

            end = new StackFrame;
            end->data = temp->data;
            top = end;
            //First node created and filled with data.
            //New nodes are now added AFTER this first node.

            temp = temp->link;
            while (temp != NULL)
            {
                end->link = new StackFrame;
                end = end->link;
                end->data = temp->data;
                temp = temp->link;
            }
            end->link = NULL;
        }
    }

    //Uses cstddef:
    void Stack::push(char the_symbol)
    {
        StackFramePtr temp_ptr;
        temp_ptr = new StackFrame;
        temp_ptr->data = the_symbol;

        temp_ptr->link = top;
        top = temp_ptr;
    }
}//stacksavitch
```

A Queue

- A queue is a data structure that retrieves data in the same order the data was stored
  - If 'A', 'B', and then 'C' are placed in a queue, they will be removed in the order ‘A', 'B', and then ‘C'
- A queue is a first-in/first-out data structure like the checkout line in a supermarket

Display 13.20

Display 13.20

Back Next

DISPLAY 13.20 A Queue