ALGORITHMS AND DATA STRUCTURES

IMPLEMENTATION OF STACK

DEFINING A CHARSTACK CLASS

- Let us write classes *using dynamic memory allocation*
- The class to be implemented will be a stack for `char`'s (has less methods than other DSs)
- Choose a class we know how it works so can focus on the implementation and not on details of operations
- Let us start with a simple implementation using `std::vector`

THE CHARSTACK.H INTERFACE

```
#ifndef _charStack_hpp_
#define _charStack_hpp_
#include <vector>

class CharStack {
public:
    CharStack();
    ~CharStack();

    int size();
    bool isEmpty();
    void clear();
    void push(char ch);
    char pop();
    char peek();

#include "charStackPriv.hxx"
};

#endif /* _charStack_hpp_ */
```

Roberts, Eric. (2013). *Programming Abstractions in C++*. Pearson.

THE CHARSTACK.H PRIVATE INTERFACE WITH STD::VECTOR

```
#ifndef _charStackPriv_h_
#define _charStackPriv_h_

private:
    std::vector<char> elements;

#endif /* _charStackPriv_h_ */
```

Choosing a representation for the data goes in this file

Roberts, Eric. (2013). *Programming Abstractions in C++*. Pearson.

USING VECTOR TO IMPLEMENT CHARSTACK

```
#include <vector>
#include "charStack.hpp"
using namespace std;

CharStack::CharStack() {
    /* Empty */
}

CharStack::~CharStack() {
    /* Empty */
}

int CharStack::size() {
    return elements.size();
}

bool CharStack::isEmpty() {
    return elements.empty();
}
```

Roberts, Eric. (2013). *Programming Abstractions in C++*. Pearson.

USING VECTOR TO IMPLEMENT CHARSTACK

```
void CharStack::clear() {
    elements.clear();
}

void CharStack::push(char ch) {
    elements.push_back(ch);
}

char CharStack::pop() {
    char result = elements[elements.size() - 1];
    elements.pop_back();
    return result;
}

char CharStack::peek() {
    return elements[elements.size() - 1];
}
```

Roberts, Eric. (2013). *Programming Abstractions in C++*. Pearson.

USING VECTOR TO IMPLEMENT CHARSTACK

- Choosing `std::vector` shows the right instincts
- **Good**: Not bad to solve a problem in terms of one you've already solved
- **Bad**: Using vector makes it harder to analyze the implementation
- Could use instead a built-in type to make it more transparent: *What type would be a good option?* We saw this not long ago…

USING DYNAMIC ARRAYS TO IMPLEMENT CHARSTACK

- Array vs. dynamic arrays: none of them allow for expansion once they are defined
- What is the best strategy to solve this issue?
- Should we go back to what we had before?

USING DYNAMIC ARRAYS TO IMPLEMENT CHARSTACK

- Array vs. dynamic arrays: none of them allow for expansion once they are defined
- What is the best strategy to solve this issue?
- Should we go back to what we had before? ***NO!***
  - Allocate fixed-size array
  - Replace it with new array whenever the first array runs out of space

USING DYNAMIC ARRAYS TO IMPLEMENT CHARSTACK

- What programming ingredients do we need to succeed?

USING DYNAMIC ARRAYS TO IMPLEMENT CHARSTACK

- What programming ingredients do we need to succeed?
- Using this strategy, we need to keep track of:

USING DYNAMIC ARRAYS TO IMPLEMENT CHARSTACK

- What programming ingredients do we need to succeed?
- Using this strategy, we need to keep track of:

1. A pointer to a dynamic array that contains all the characters in stack [array]

USING DYNAMIC ARRAYS TO IMPLEMENT CHARSTACK

- What programming ingredients do we need to succeed?
- Using this strategy, we need to keep track of:

1. A pointer to a dynamic array that contains all the characters in stack [array]
2. How much space for potential elements has been allocated [capacity]

USING DYNAMIC ARRAYS TO IMPLEMENT CHARSTACK

- What programming ingredients do we need to succeed?
- Using this strategy, we need to keep track of:

1. A pointer to a dynamic array that contains all the characters in stack [array]
2. How much space for potential elements has been allocated [capacity]
3. The actual size of the stack [amount of char's stored]

THE CHARSTACK.H PRIVATE INTERFACE WITH DYNAMIC ARRAYS

```
#ifndef _charStackPriv_h_
#define _charStackPriv_h_

private:

/* Instance variables */
    char *array;  /* Dynamic array of characters   */
    int capacity; /* Allocated size of that array  */
    int count;    /* Current count of chars pushed */

/* Private function prototype */
    void expandCapacity();

#endif /* _charStackPriv_h_ */
```

Roberts, Eric. (2013). *Programming Abstractions in C++*. Pearson.

USING DYNAMIC ARRAYS TO IMPLEMENT CHARSTACK

```
#include "charstack.h"
using namespace std;

const int INITIAL_CAPACITY = 10;

CharStack::CharStack() {
    capacity = INITIAL_CAPACITY;
    array = new char[capacity];
    count = 0;
}

CharStack::~CharStack() {
    delete[] array;
}

int CharStack::size() {
    return count;
}
```

Roberts, Eric. (2013). *Programming Abstractions in C++*. Pearson.

USING DYNAMIC ARRAYS TO IMPLEMENT CHARSTACK

```
bool CharStack::isEmpty() {
    return count == 0;
}

void CharStack::clear() {
    count = 0;
}

void CharStack::push(char ch) {
    if (count == capacity) expandCapacity();
    array[count++] = ch;
}

char CharStack::pop() {
    return array[--count];
}

char CharStack::peek() {
    return array[count - 1];
}
```

Roberts, Eric. (2013). *Programming Abstractions in C++*. Pearson.

USING DYNAMIC ARRAYS TO IMPLEMENT CHARSTACK

```
void CharStack::expandCapacity() {
    char *oldArray = array;
    capacity *= 2;
    array = new char[capacity];
    for (int i = 0; i < count; i++) {
        array[i] = oldArray[i];
    }
    delete[] oldArray;
}
```

Pay really close attention to what is going on up there…

Roberts, Eric. (2013). *Programming Abstractions in C++*. Pearson.

EXTRA READING

- From the book Programming Abstractions in C++:
  - 12.5 *Heap-stack diagrams*
  - 12.6 *Unit testing*
  - 12.7 *Copying objects*
  - 12.8 *The uses of* `const`
  - 12.9 *Efficiency of the CharStack class*

Roberts, Eric. (2013). *Programming Abstractions in C++*. Pearson.