# Datastructuren

2009-10

André Deutz

# Datastructuren 2009-10

Docent: André Deutz; kamer 116; tel.: 071 527 7071; deutz@...

Werkgroep/practicum: Minh Tran Ngod; kamer 142; tel.: 071 527 7037 minhtn@...

Werkgroep & programming: Simon Zaaijer; szaaijer@...

Book: Adam Drozdek

*Data Structures and Algorithms in C++*

Third Edition, 2005; isbn: 0-534-49182-0

Handouts

# Datastructuren 2009-10

Th. H. Cormen, Ch. E. Leiserson, R. L. Rivest, C. Stein *Introduction to Algorithms* , The MIT Press, second edition (possibly already *third edition!*), 2002 (Third Edition: September 30, 2009)

S. Dasgupta, Ch. Papadimitriou, U. Vazirani, *Algorithms,* McGraw Hill of Higher Education, 2008

Kurt Mehlhorn, Peter Sanders, *Algorithms and Data Structures,* Springer, 2008

Steven S. Skiena, *The Algorithm Design Manual,* Springer,Second Edition, 2008

# Datastructuren 2009-10

Tentamen

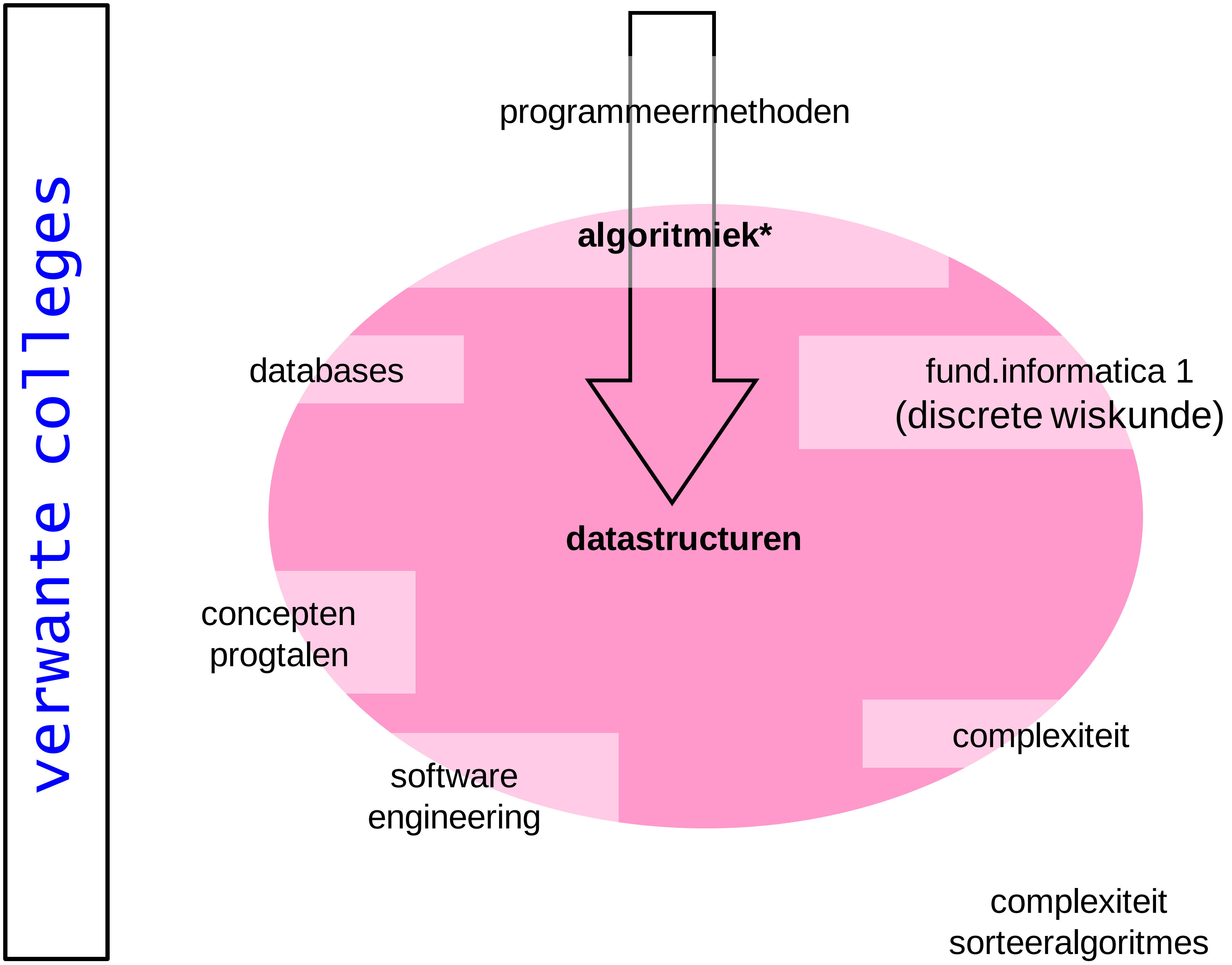
Programmeeropdrachten: ±4 kleine en 1 grote (eerste programmeeropdracht wordt vandaag uitgedeeld)

Werkgroep/practicum: eerste vandaag (31 aug; 13.45)

Programmeertaal

Website: *will be up today*

Meer info: *watch site and emails*


verwante colleges
programmeermethoden
algoritmiek*
databases
fund.informatica 1
(discrete wiskunde)
datastructuren
concepten
progtalen
complexiteit
software
engineering
complexiteit
sorteeralgoritmes

# Data Structures

What will we learn in this course?

- Datastructures *and* Algorithms (so the course name is a little bit of a misnomer)
- Continue to learn to design and understand algorithms – at least the standard algorithms which should be in any computing expert's baggage
- Data/Datastructures and Algorithms work in tandem: so we devote also considerable time to datastructures (again at least the standard ones)

# Data Structures

What will we learn in this course? Continued:

- We will also experience the interplay between datastructures and algorithms:
  - often carefully chosen data structures will allow the use of the most efficient algorithm
  - After the data structures are chosen, the algorithms to be used often become relatively obvious.
  - Sometimes things work in the opposite direction data structures are chosen because certain key tasks have algorithms that work best with particular data structures.
-

# Data Structures

What will we learn in this course? Continued:

- The interplay between datastructures and algorithms:
  - How much intelligence do you put into the data and how much in the algorithm?
  - the choice of the datastructure is often done via the choice of an abstract data type (ADT) : we will learn this way of thinking
  - A well-designed data structure allows a variety of critical operations to be performed, using as few resources, both execution time and memory space, as possible : we are also going after efficiency!

## Data Structures

Summary of what we will learn:

Understanding and use of standard data structures and associated algorithms, some understanding of efficiency issues, learn to think abstractly (that is, distinguish use and implementation)

# From Problem to Abstract Data Type

How can you implement the “undo” for the following situation.
When you type a  line of text at a keyboard, you are likely to make mistakes.
We assume that you can use the usual ascii characters to enter lines of
text except the asterisk ‘*’.  With this character you can announce the wish
for the undo in case you made a typo.  The understanding is that you
use the asterisk key to correct these mistakes, each asterisk erases
the previous character entered. Consecutive asterisks are applied in
sequence and so erase several characters.

For instance, if you type the line

```
abcc*ddde***ef*fg
```

the  corrected input would be

```
abcdefg
```

How can a program read the original line and get the corrected input?

## Design of Solution

- Eventually must decide how to store the input line
- ADT approach: postpone this decision until you have a better idea what the operations you will need to perform on the data.
- People who are accostomed to use ADTs naturally adhere to this approach

# First Attempt

```
//read the line, correcting mistakes along the way
while (not end of line) {
    read a new character  ch
    if (ch is not ‘*’) {
        add ch to ADT
    } else {
        remove from the ADT the item that was
          added most recently
    }
}
```

## First attempt

From this pseudo code we see that we need two operations to work on the data:

1) Add an item to the ADT
2) Remove the item most recently added to the ADT

## Second Attempt

- What happens when you type ‘*’ and the ADT is empty?
  - Terminate program with error message? No we will not do this
  - We let the program ignore ‘*’ in this case and continue

We modify our first attempt:

## Second Attempt

```
while (not end of line) {
    read a new character  ch
    if (ch is not ‘*’)
        add ch to ADT
     } else {
        if (ADT is not empty) {
            remove from the ADT the item that was added
            most recently
         } else {
            ignore the ‘*’
         }
     }
}
```

## Second Attempt

From this pseudo code we identify a third operation:

3) **Determine whether the ADT is empty**

This solution places the correct input line in the ADT

Suppose that you want to display the corrected line or process it (in order entered)??

## Writing or Processsing the Line

```
// write the line (or process it in some other way)
While (the ADT not empty) {
    remove from the ADT the item that was added most recently
    write(or process) .... Nou-nee!
}
```

Wrong: 1) *remove* the item from the ADT, it is gone! Cannot write it. Should have done: *retrieve*

*2)* the most recently added item is the last character of input line; don’t want to write it first

# Writing (or Processsing) the Line in Reverse Order

```
While (the ADT not empty) {
    retrieve from the ADT the item that was added most recently and put it in ch
    write/process   ch
    remove from the ADT the item that was added most recently    }
```

A fourth operation is required by the ADT:

return from the ADT the item that was added most recently *without removing it* (retrieve *== return without removing)*

## Operations required for our ADT:

- Determine whether ADT is empty
- Add new item to the ADT
- Remove from the ADT the item that was added most recently
- Retrieve from the ADT the item that was added most recently (Retrieve == return *without removing* it from the ADT)
- It is customary to include *initialization* and *destruction* operations for an ADT

# ADT Stack:

- **createStack()** // creates an empty stack
- **destroyStack()** // destroys a stack
- **stackIsEmpty()** // determines whether the stack is empty
- **push(newItem)** // Adds NewItem to a stack.
- **pop()** // Removes from a stack the item that was added most recently.
  **getStackTop(stackTop)**// Retrieves into var *stackTop* the item that was added most recently to a stack, leaving the stack unchanged.

## Can use the ADT stack without knowing the implementation of the ops

```
S.createStack() ; //in C++ declare S as instance of the stack class, since
    //createStack() is implemented as the class's constructor
 Read newChar;
While (newChar not eoln){
   if (newChar is not '*'){
        S.push(newChar);
    } else {
         if (! S.stackIsEmpty){
              S.pop();
         }
    }
    read newChar
} //end while
```

Program Read input line and
correct along
The way

## Can use the ADT stack without knowing the implementation of the operations:

```
tempS.createStack();
While (!S.isEmptyStack()) {
    S.getTopStack(ch);
    S.pop();
    tempS.push(ch);
}
// top of the stack   tempS   holds the first char of the line
// entered !!
```

Program Read input line and
correct along
The way continued

## ADT Stack spec in C++

```
class StackClass {
public:
        StackClass();
        StackClass(const StackClass & S); //copy constructor
        ~StackClass();
// stack operations
         bool stackIsEmpty();
        // determines whether the stack is empty.
        // precondition: the constructor has been called
        // postcondition: Returns TRUE if the stack was empty, otherwise returns FALSE
        void push(stackItemType newItem);
        // adds an item to the top of the stack
        // precondition: the constructor has been called. newItem is the item to be
        // added.
        // postcondition: if insertion was successful, newItem is on top of the
        // stack.
       void pop();
        // Removes the top of stack.
        // precondition: the constructor has been called.
        // postcondition: if the stack was not empty, the item that was added
        // MOST RECENTLY is removed.
        void getStackTop(stackItemType & topItem);
         // Retrieves the top of the stack.
        // If the stack was not empty, topItem contains the item
        // that was added MOST RECENTLY.
private:
// belongs to implementation!
};
```

Resembles too much C programming, since *typedef* is used.
Use templates (generic programming) instead

## Implemting ADT Stack in C++,Part 1

```
// header file;    array based implementation

#ifndef _ARRAYIMPLSTACK_      // or use the directive  #pragma once
#define _ARRAYIMPLSTACK_
#include <iostream>
using namespace std;

const int MAX_STACK = 100;

typedef char stackItemType;

class StackClass {
public:
        StackClass();
        StackClass(const StackClass & S); //copy constructor
        ~StackClass();
// stack operations
         bool stackIsEmpty();
        void push(stackItemType newItem);
        void pop();
        void getStackTop(stackItemType & topItem);
private:
        stackItemType items[MAX_STACK];     // information hiding, only accessible throuhg public interface/contract
         int top;                           //  information hiding
};
#endif
```

Resembles too much C programming, since *typedef* is used.
Use templates (generic programming) instead

## Implmenting ADT Stack in C++,Part 2

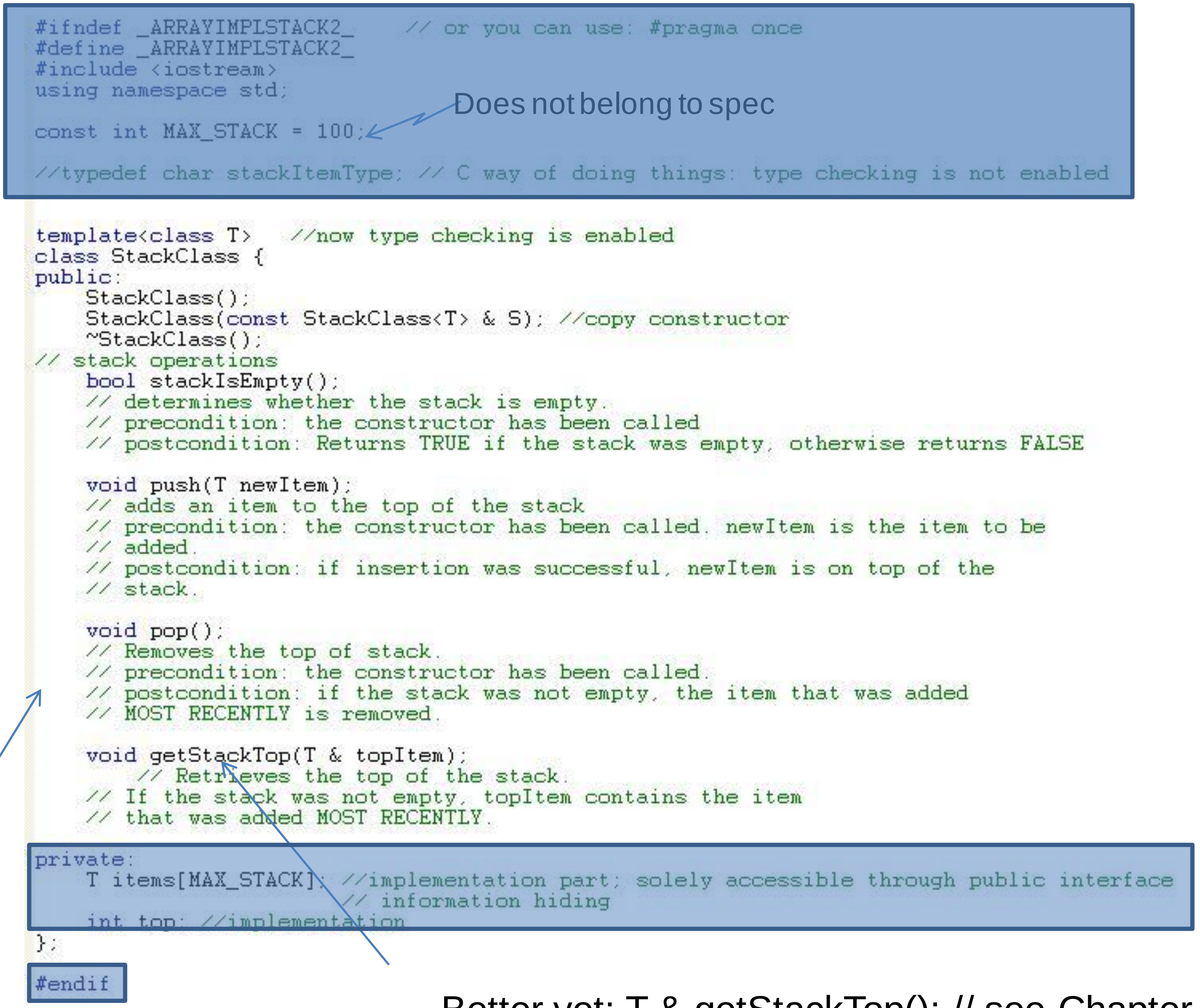
```
//  implementation file arrayImplStack.cpp for the ADT Stack; array-based
    //implementation
#include  "arrayImplStack.h"
StackClass::StackClass(): top(-1) {}
StackClass::StackClass(const StackClass & S): top(S.top) {
    for(int i=0; i<=S.top; i++){
        items[i] = S.items[i];
    }
}
StackClass::~StackClass() {}
bool StackClass::stackIsEmpty() {return bool (top<0);}
void StackClass::push(stackItemType newItem) {
    if (top<(MAX_STACK-1)){
        ++top;
        items[top]=newItem;
    }
}
void StackClass::pop() {
    if (!stackIsEmpty()) {
           --top;
    }
}
void StackClass::getStackTop(stackItemType& topItem){
    if (!stackIsEmpty()){
        topItem=items[top];
    }
}
```

## Client program of Stack, i.e. Algorith which uses Stack

```
//  a client program  that uses stack(s)
    // (i.e., an algorithm
// that uses a stack, actually two
    //stacks)
#include  “arrayImplStack.h”
#include <iostream>
using namespace std;
int main (){
    stackItemType anItem;
    StackClass  S;
    cin.get(anItem);
    while (anItem!='\n'){
        if (anItem!='*'){
        S.push(anItem);
    }else{
        if (!S.stackIsEmpty()){
        S.pop();
    }
    cin.get(anItem);
    cout << "\n"<<"\n";
    StackClass tempS;
    while(!S.stackIsEmpty()){
    S.getStackTop(anItem);
    S.pop();
    tempS.push(anItem);
}
```

```
// top of the stack tempS now contains
    //the first char of the entered line
// you can print, for instance to the
    //screen
    while (!tempS.stackIsEmpty()){
        tempS.getStackTop(anItem);
        cout << anItem;
        tempS.pop();
    }
    cout << "\n";
    cout << "\n";
    return 1;
}
```

## ADT Stack Spec in C++, using templates



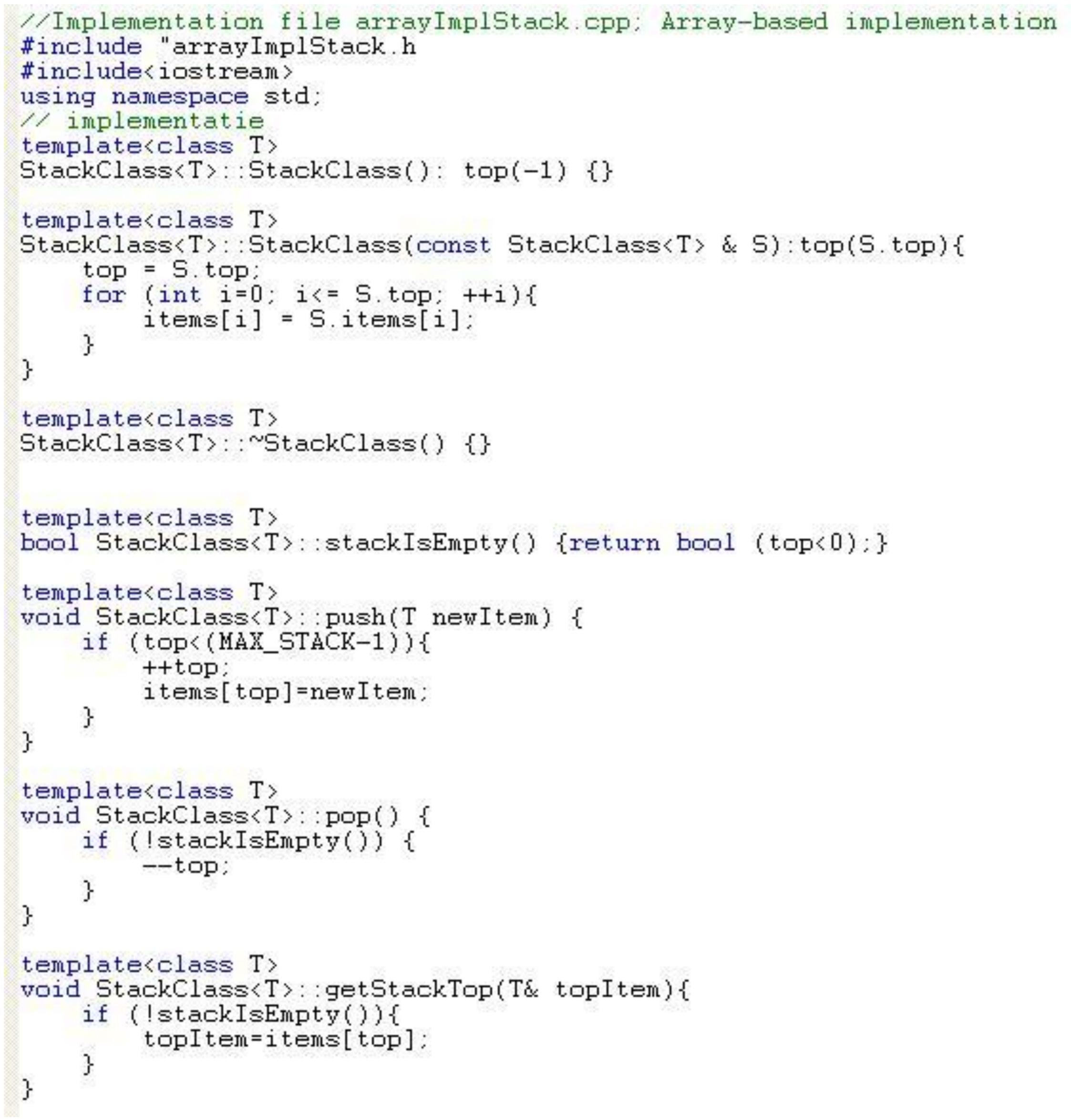
```
#ifndef _ARRAYIMPLSTACK2_    // or you can use: #pragma once
#define _ARRAYIMPLSTACK2_
#include <iostream>
using namespace std;

const int MAX_STACK = 100;

//typedef char stackItemType; // C way of doing things: type checking is not enabled

template<class T>   //now type checking is enabled
class StackClass {
public:
    StackClass();
    StackClass(const StackClass<T> & S); //copy constructor
    ~StackClass();
// stack operations
    bool stackIsEmpty();
    // determines whether the stack is empty.
    // precondition: the constructor has been called
    // postcondition: Returns TRUE if the stack was empty, otherwise returns FALSE

    void push(T newItem);
    // adds an item to the top of the stack
    // precondition: the constructor has been called. newItem is the item to be
    // added.
    // postcondition: if insertion was successful, newItem is on top of the
    // stack.

    void pop();
    // Removes the top of stack.
    // precondition: the constructor has been called.
    // postcondition: if the stack was not empty, the item that was added
    // MOST RECENTLY is removed.

    void getStackTop(T & topItem);
        // Retrieves the top of the stack.
    // If the stack was not empty, topItem contains the item
    // that was added MOST RECENTLY.

private:
    T items[MAX_STACK]; //implementation part; solely accessible through public interface
                        // information hiding
    int top; //implementation
};

#endif
```

Does not belong to spec

spec

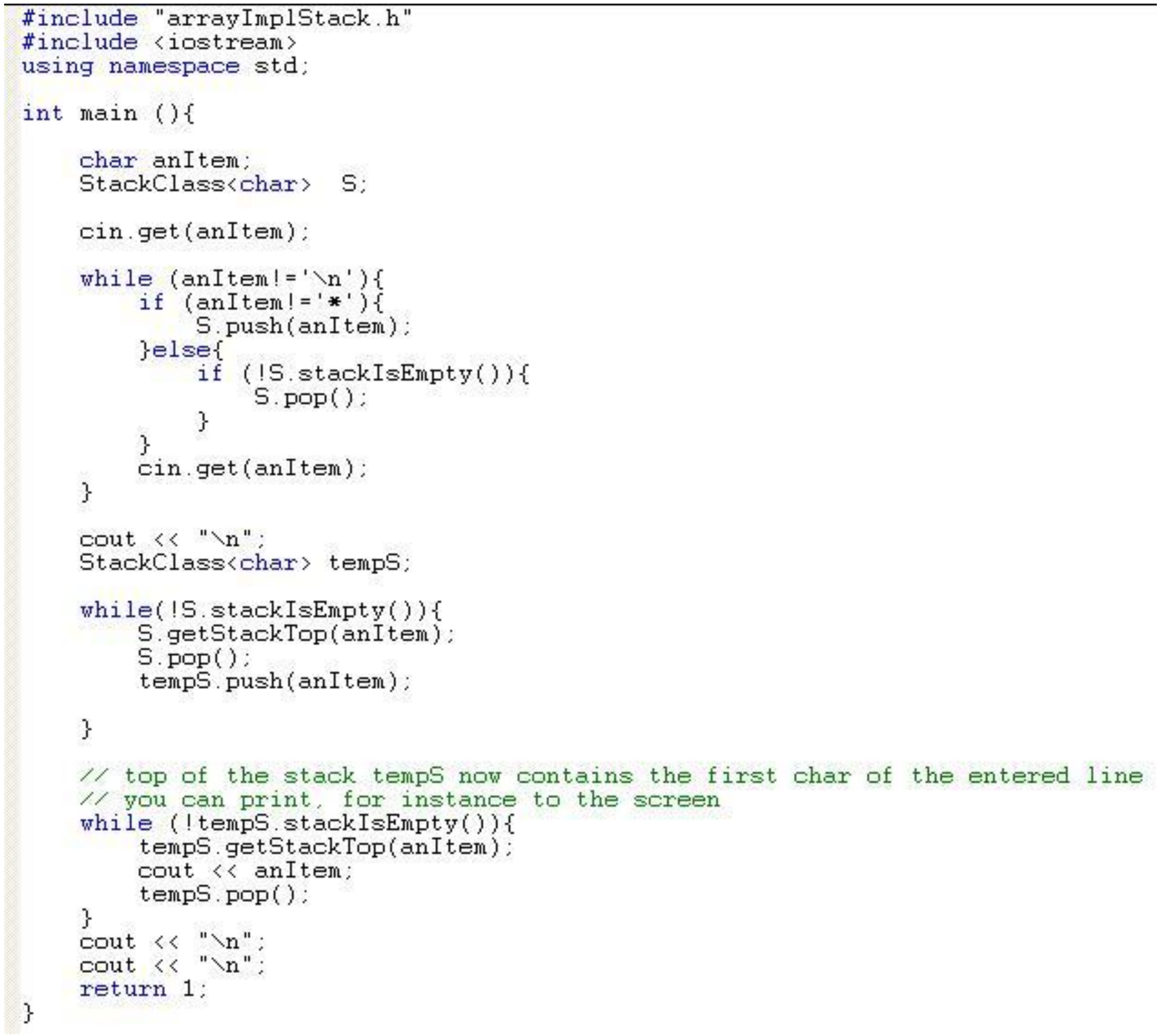
Better yet: T & getStackTop(); // see Chapter 4 in Drozdek

# Implementing ADT Stack in C++



```
//Implementation file arrayImplStack.cpp; Array-based implementation
#include "arrayImplStack.h
#include<iostream>
using namespace std;
// implementatie
template<class T>
StackClass<T>::StackClass(): top(-1) {}

template<class T>
StackClass<T>::StackClass(const StackClass<T> & S):top(S.top){
    top = S.top;
    for (int i=0; i<= S.top; ++i){
        items[i] = S.items[i];
    }
}

template<class T>
StackClass<T>::~StackClass() {}


template<class T>
bool StackClass<T>::stackIsEmpty() {return bool (top<0);}

template<class T>
void StackClass<T>::push(T newItem) {
    if (top<(MAX_STACK-1)){
        ++top;
        items[top]=newItem;
    }
}

template<class T>
void StackClass<T>::pop() {
    if (!stackIsEmpty()) {
        --top;
    }
}

template<class T>
void StackClass<T>::getStackTop(T& topItem){
    if (!stackIsEmpty()){
        topItem=items[top];
    }
}
```

# Client program of Stack,
# i.e. Algorithm which uses Stack



```
#include "arrayImplStack.h"
#include <iostream>
using namespace std;

int main (){

    char anItem;
    StackClass<char>  S;

    cin.get(anItem);

    while (anItem!='\n'){
        if (anItem!='*'){
            S.push(anItem);
        }else{
            if (!S.stackIsEmpty()){
                S.pop();
            }
        }
        cin.get(anItem);
    }

    cout << "\n";
    StackClass<char> tempS;

    while(!S.stackIsEmpty()){
        S.getStackTop(anItem);
        S.pop();
        tempS.push(anItem);

    }

    // top of the stack tempS now contains the first char of the entered line
    // you can print, for instance to the screen
    while (!tempS.stackIsEmpty()){
        tempS.getStackTop(anItem);
        cout << anItem;
        tempS.pop();
    }
    cout << "\n";
    cout << "\n";
    return 1;
}
```

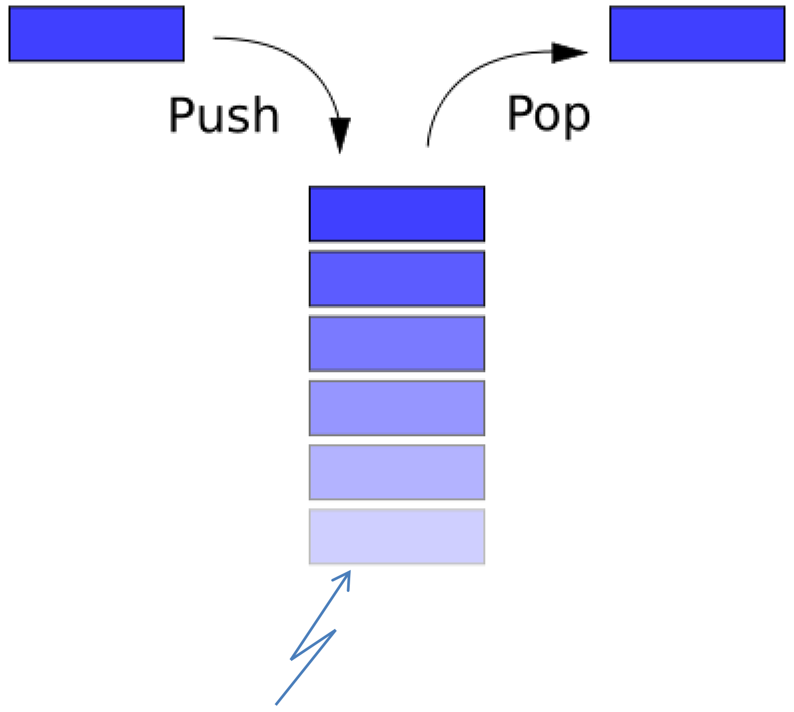
## Discussion of Uses of Stacks

edgser dijkstra:  invented stacks to implement recursively defined functions/programs

CA: stack frames used in function calls

visiting nodes in a tree

Ubiquitous! (see also Chapter 4 for more apps)



The end ☺. This means read and study
Chapter 4 (emphasize pages 137-144)  in Drozdek

(Review Chapter 1 in Drozdek)