Lecture 3

STL, Abstract datatype (ADT), vector, grid, stack, queue, list

TDDD86: DALP

Utskriftsversion av Lecture in *Data Structures, Algorithms and Programming Paradigms*
September 9th, 2024

IDA, Linköping University

3.1

Content

Contents

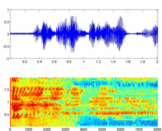
3.2

1 Containers and STL

Motivation

- Almost any interesting problem manipulates **data**.
- Data comes from many sources:
  - files from your local hard drive
  - database
  - downloaded from the internet
  - input from hardware sensors, microphones, controllers...
  - ...






- To store and manage data efficiently and perform interesting calculations on it, we need to learn about some useful *data structures*.

3.3

Why not field/array?

- C++ does not know the size of a C array "`int a[100]`"
- You can use an array as in "`std::array<int,100> a`" which supplies some members (e.g., size(), at, etc)
  - Still fixed size, no insertion/deletion.
- You can access memory in C++ without systematically raising errors. You might silently get "random" content.
- Arrays lack many operations we would like to have:
  - insertion/deletion of elements in the beginning/middle/end of the array
  - sort element
  - search in the array for the given value
  - remove/prevent if duplicate is added 3.4

Container

- **container**: an object that stores the data of a certain type, it is a data structure
  - stored objects are called elements
  - some containers store elements in order, some allow duplicates
  - typical operations are: add, remove, clear, find/search, size, is empty
- To use effectively a container, it is important to understand how they work 3.5

STL — overview

STL is organized in several components, each consisting of generic components that interact with the rest of the library.

- **Containers**. At the heart of the STL are the container classes such as `vector`.
- **Iterators**. Each STL-container exports iterators that allow to access and modify the content of the data. Iterators have a common interface, this allow to write algorithms that work with arbitrary containers.
- **Algorithms**. STL algorithms are functions that use an interval of data specified with iterators.
- **Adaptors**. Adaptors in STL are object which can transform an object from a form to an other. For example, the "stack" adaptor transform a `vector` into a LIFO-container.
- **Function object**. Many of the STL algorithms and functions depend on user-defined callback. The STL helps us to create those functions.
- . . . 3.6

Container — `vector`

- `vector` is a collection of elements with 0-based index
- `vector` has dynamic capacity that can be extended when needed (for instance when adding elements `push_back()`)
- has support for iterators, items that point to an element of the `vector` and knows how to access previous/next elements
- similar to `ArrayList` in Java

```
vector<int> v1;       // default constructor - v1 is empty
vector<int> v2(100);  // initialise v2 with a 100 value-initialized elements
vector<int> v3(v2);   // copy constructor - initialise v3 as a copy of v2
```

3.7

Container — `vector`

```
#include <vector>

vector<int> v; //  initialy empty
int         x;

while (cin >> x)
    v.push_back(x); // insert at the end capacity will be increased if needed

for (size_t i = 0; i < v.size(); ++i) // loop with usual indexation
    v[i] = v[i] + 1;

for (auto it = v.begin(); it != v.end(); ++it) // loop with iterator, here of
    *it = *it + 1;                             // type vector<int>::iterator

for (auto x : v) // C++11, range loop, here x of type int, so
    cout << x << '\n'; // elements cannot be changed

for (auto& x: v) // C++11, range loop, here x of type int&,
    x = 2 * x + 1; //so elements can be changed
```

3.8

Example of operation on `vector`

`v.front()`	returnerar det första elementet i *v*
`v.back()`	returnerar det sista elementet i *v*
`v[i]`	returnerar elementet i indexposition *i*
`v.pop_back()`	tar bort sista elementet i *v*, returnerar inget
`v.insert(it, x)`	sätter in *x* direkt före iteratorpositionen *it*, returnerar interator till det nya elementet
`v.erase(it)`	raderar elementet i iteratorpositionen *it*, returnerar interator till elementet som följer efter det raderade elementet
`v.clear()`	raderar samtliga element i *v*
`v.empty()`	returnerar **true** om *v* är tom
`v.size()`	returnerar antalet element som lagras i *v* (storleken)
v1 = *v2*	tilldelning, innehållet i *v1* ersätts av en kopia av innehållet i *v2*
v1 == *v2*	returnerar **true** om innehållet i *v1* är lika med innehållet i *v2*; != för att jämföra med avseende på olikhet
v1 < *v2*	returnerar **true** om elementen i *v1* är lexikografiskt mindre än *v2*; <=, > och >= för analoga jämförelser
`v1.swap(v2)`	byter innehåll på *v1* och *v2*
`swap(v1, v2)`	byter innehåll på *v1* och *v2*
`begin(v)`	returnerar iterator till första elementet i *v* eller, om *v* är tom, en ”förbi-slutet-iterator”
`end(v)`	returnerar en ”förbi-slutet-iterator”

3.9

`vector` — “mystery”

```
vector<int> v;
for (int i = 0; i <= 10; ++i) {
    v.push_back(10 * i);    // {0, 10, 20, 30, 40, ..., 100}
}
```

- What would this code write out?

```
for (size_t i = 0; i < v.size(); i++) {
    v.erase(v.begin() + i);
}

for (auto x : v) {
    cout << x << endl;
}
```

3.10

Example of algorithm: sort

```
std::vector<int> v = {...};
std::sort(v.begin(), v.end());
```

3.11

Grid

- **grid**: is a container with two dimension index
  - like a two dimension array but easier to use
  - `#include"grid.h"` in lab 1

```
Grid<type> name;                // empty grid
Grid<type> name(nRows, nCols);  // specify the size
```

- The initial size can be specified
  - If empty, the grid object is useless, use `resize`
  - Access/assignment using the `[r][c]`-notation

```
Grid<string> chessBoard(8, 8);
chessBoard[2][3] = "knight";
chessBoard[1][6] = "queen";
```

3.12

2 Abstrakta datatyper

DALG – basic concepts

Abstract datatype (ADT)
machine-independent high-level description of the data and operations on data, such as stack, queue,...

Data structure
logical organisation of computer memory to store data

Algorithm
high level description of concrete operations on data structures

ADT
implemented with appropriate data structures and algorithms

Program
implementation of algorithms and data structures in a particular programming language 3.13

ADT

- Seperate implementation from specification
  - ADT: specify the operations, similar to an interface
  - Implementation: provide the code
  - Client: program that use operations
- Abstract Datatype
  - ADT hides the data representation
  - Client does not manipulate the data and control which operations are allowed
- Principle of least privilege
  - In computer security, require an abstract layer to restrict access to resources to those that are necesserary and legitimate 3.14

Benefits of ADT

- Based on Object-Orientation principles
- Provides reusable and robust data structure
- Encapsulation reduce the risk of data corruption
- Formal definition allow to write generic algorithms 3.15

3 Lists

Lists

A *list* L is a sequence of elements $\langle x_0, \ldots, x_{n-1} \rangle$

- *size* or *length* $|L| = n$
- *empty* list $\langle\rangle$ with length 0
- Two ways of accessing elements:
  - Selection using *index* i (sometimes called *rank*): select the i:th element, x_i, where $0 \le i \le n-1$
  - Selection using current *position*, f.ex. *first* element in L, or *last*, *previous*, *next*, ...
- *position* abstract away from indexing

  $\rightarrow$ ADT arraylist: using *index*

  $\rightarrow$ ADT nodelist: using *position* 3.16

ADT arraylist

Domain: list

Operations on a vector S

- *size*() return $|S|$
- *isempty*() return *true* if $|S| = 0$
- *elemAtIndex*(i) returns $S[i]$; fails for $i < 0$ or $i >$*size*$() - 1$
- *setAtIndex*(i,x) replace the value of the ith element with x; fails for $i < 0$ or $i >$*size*$() - 1$
- *insertAtIndex*(i,x) insert x as a new element with index i: increases the size; fails with $i < 0$ or $i >$*size*()
- *removeAtIndex*(i) remove the i:th element in S: decrease the size; fails with $i < 0$ or $i >$*size*$() - 1$ 3.17

Example: many operations on an initially empty arraylist S

operation	output	S
insertAtIndex(0,7)	-	(7)
insertAtIndex(0,4)	-	(4,7)
elemAtIndex(1)	7	(4,7)
insertAtIndex(2,2)	-	(4,7,2)
elemAtIndex(3)	"error"	(4,7,2)
remove(1)	7	(4,2)
insertAtIndex(1,5)	-	(4,5,2)
insertAtIndex(1,3)	-	(4,3,5,2)
insertAtIndex(4,9)	-	(4,3,5,2,9)
elemAtIndex(2)	5	(4,3,5,2,9)
setAtIndex(3,8)	2	(4,3,5,8,9)

3.18

ADT nodelist

Domain: list Operations on a list L, in addition to *size*() and *isempty*()

- *first*() returns *position* of the first element in L; fails if L is empty
- *last*() returns *position* of the last element in L; fails if L is empty
- *prev*(p) returns *position* for the element before p in L; fails if p is the first position
- *next*(p) returns *position* for the element after p in L; fails if p is the last position
- *set*(p,x) set the element at position p to the value x, return the value that used to be at position p
- *insertFirst*(x) insert new element x as the first element in L, returns the position for x
- *insertLast*(x) insert a new element x as the last element in L, returns the position for x
- *insertBefore*(p,x) insert a new element x before position p in L, returns the position for x
- *insertAfter*(p,x) insert a new element x after position p in L, returns the position for x
- *remove*(p) remove and return the element at position p from L

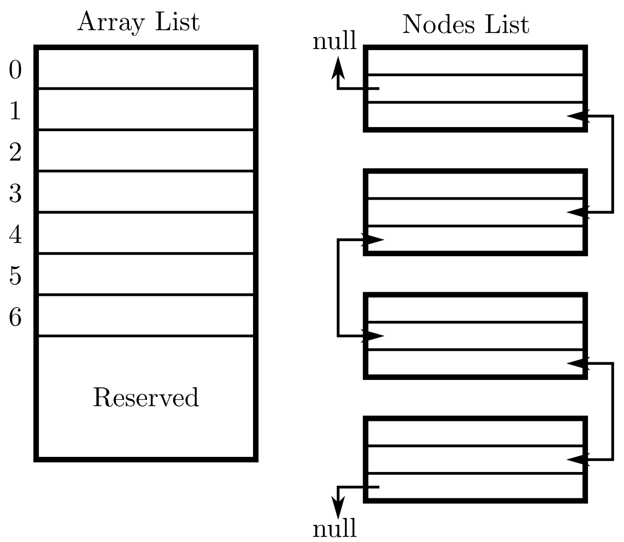
3.19

Example: many operations on an initially on node list L

operation	output	L
insertFirst(8)	-	(8)
first()	$p_1(8)$	(8)
insertAfter(p_1,5)	-	(8,5)
next(p_1)	$p_2(5)$	(8,5)
insertBefore(p_2,3)	-	(8,3,5)
prev(p_2)	$p_3(3)$	(8,3,5)
insertFirst(9)	-	(9,8,3,5)
last()	$p_2(5)$	(9,8,3,5)
remove(first())	9	(8,3,5)
set(p_3,7)	3	(8,7,5)
insertAfter(first(),2)	-	(8,2,7,5)

3.20

Arraylist vs Nodeslist

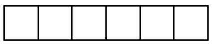


3.21

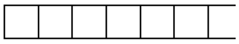
Sequence containers — five base types

Elementen lagras i en strikt sekventiell ordning.

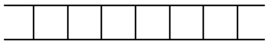
`array`



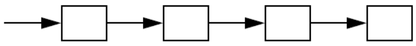
`vector`



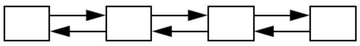
`deque`



`forward_list`



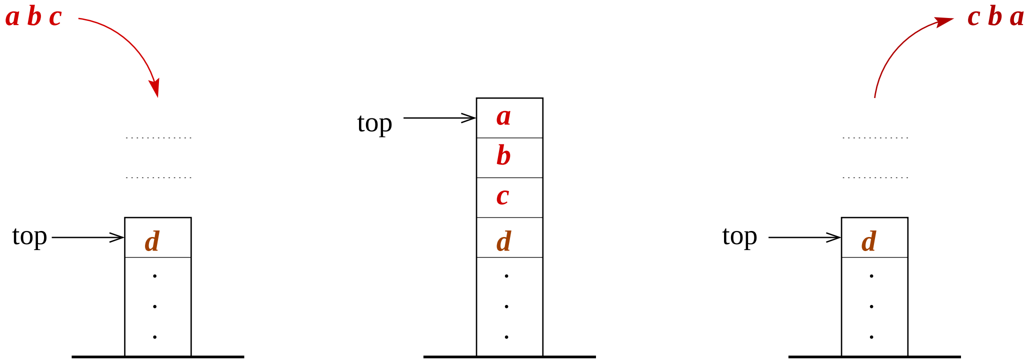
`list`



3.22

4 ADT stack

ADT stack (last in first out, LIFO)



Operationer:

- *Top*(S) returns the top element of the stack S[1]
- *Pop*(S) remove and return the tar bort och returnerar det översta elementet i stack S[1]
- *Push*(S,x) add x at the top of the stack S
- *MakeEmptyStack*() create a new empty stack
- *IsEmptyStack*(S) returns *true* if S is empty

3.23

Typical applications for the ADT stack

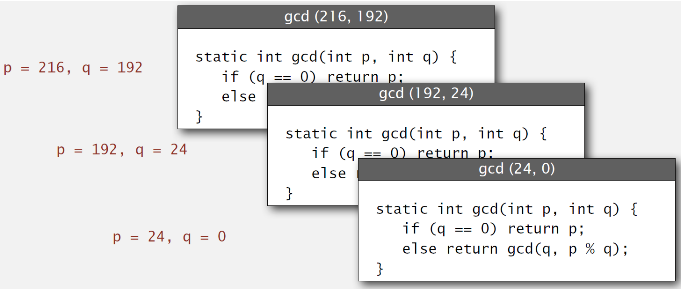
- Programming languages and compilers
  – implementation of function calls
  – compilers use stack for evaluating an expression
- Match up pairs of related things
  – examine whether a string is palindrome
  – examine a file to check if the brackets match
  – convert expressions from prefix to postfix
- Sophisticated algorithms
  – search in a labyrinth with “backtracking”
  – many programs have a “undo-stack” with previous operations

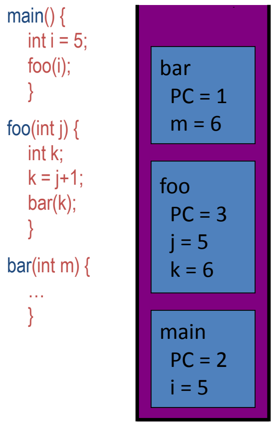
[1] or an error message if S is empty

3.24

Application of stack: function calls

- Compilers implement functions
  - Function call: *push:a* local environment and return address
  - Return: *pop:a* get the return address and the local environment
  - This allow for recursion.





3.25

Sequence adapter

- adapter classes that represent classic data structures
  - `stack`
- internally, there is a container which holds the elements
  - `stack` — `deque`
- it has a simplified interface
  - the number of operations is significantly reduced compared to the sequence container used internally
  - example of operations:

```
n = s.size();
b = s.empty();
x = s.top();
s.push(x);
s.pop();
```

  - no iterators

3.26

Why there is no `.clear()`-function?

- Purely conceptually, there is no cleaning/emptying definition in the interface/ADT
- It is easy to write one yourself:

```
// stack<int> s = ...
while (!s.empty()) {
    s.pop();
}
```

3.27

Why does `.pop()` not return the deleted value?

- The caller may not need the value, and then it would be a waste of resources to return it
- It is easy to write code that pops and saves the value:

```
// stack<int> s = ...
int value = s.top();
s.pop();
```

3.28

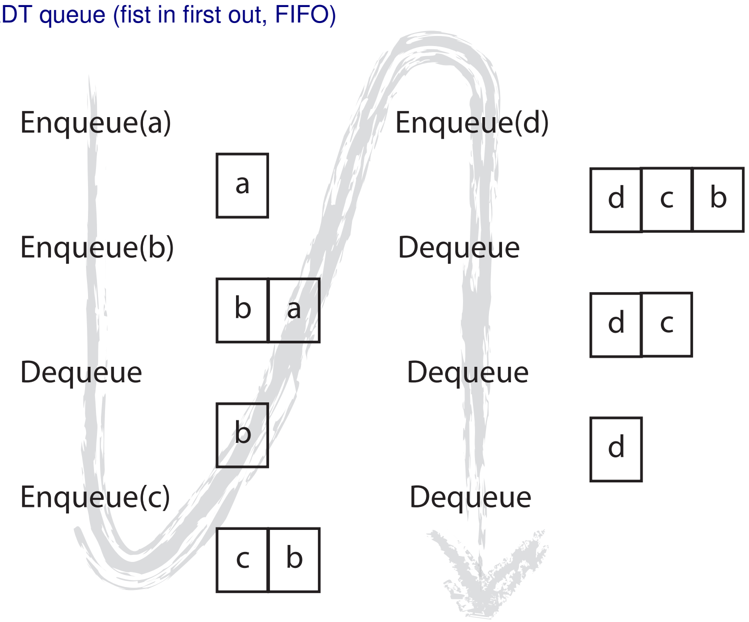
Stack — exercise

- Write a function `checkBalance` which takes a string corresponding to some source code and check that all parentheses are balanced.
  - For each `(` or `{` there must be a corresponding `}` or `)` in reverse order.
  - Returns the index where the imbalance occurs or `-1` if it is balanced.

3.29

5 ADT queue

ADT queue (fist in first out, FIFO)



3.30

ADT queue (fist in first out, FIFO)

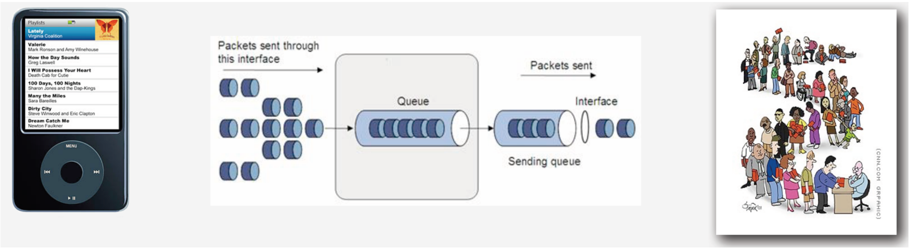
Operations:

- *Front*(Q) returns the first element in the queue Q
- *Dequeue*(Q) remove and return the first element in the queue Q
- *Enqueue*(Q,x) add x last in queue Q
- *MakeEmptyQueue*() create a new empty queue
- *IsEmptyQueue*(Q) returns *true* if Q is empty

3.31

Typical applications for the ADT queue

- Familiar applications
  - Playlist in your phone
  - Asynchronous data transfer (fill-I/O, pipes, sockets)
  - Take care of requests to share services (such as printers)
- Simulation
  - Traffic analysis
  - Waiting times of customers
  - Determines how many clerks/cashiers are needed in a supermarket



3.32

Sequence adapter

- adapter class that represent the data structure:
  - `queue`
- use a container to store the elements
  - `queue` — `deque`
- it offers a simplified interface
  - the number of operations is significantly reduced compared to the internal sequence container
  - example of operations:

```
n = q.size();
b = q.empty();
x = q.front();
q.push(x);
q.pop();
```

  - no iterators

3.33

Queue — exercise

- Write a function `ends` taking a queue of integers and replaces each element with two copies of itself.
  - `{1, 2, 3}` becomes `{1, 1, 2, 2, 3, 3}`
- Write a function `mirror` which takes a queue of strings and concatenate itself in reverse order
  - `{a, b, c}` becomes `{a, b, c, c, b, a}`

3.34