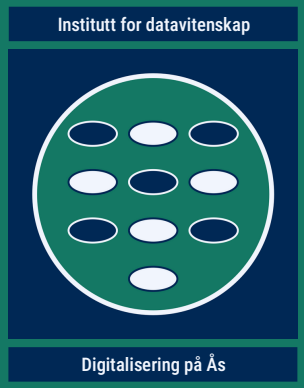
Norges miljø- og biovitenskapelige universitet



INF205
Resource-efficient programming

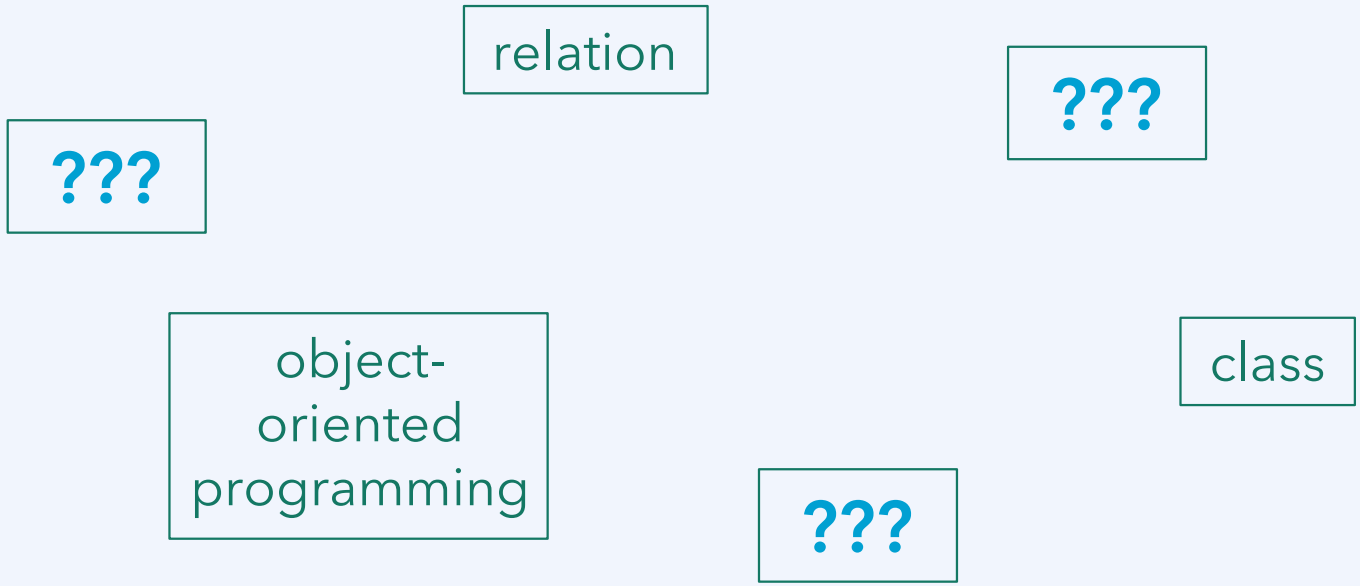
3 Data structures

3.1 Object orientation
3.2 Inheritance
3.3 Linked data structures
3.4 Containers
3.5 Graph data structures
3.6 Streams and file I/O

Weekly glossary concepts

What are essential concepts from the previous lecture?

Let us include them in the **INF205 glossary**.[1]



[1]https://home.bawue.de/~horsch/teaching/inf205/glossary-en.html

Noregs miljø- og biovitskaplege universitet

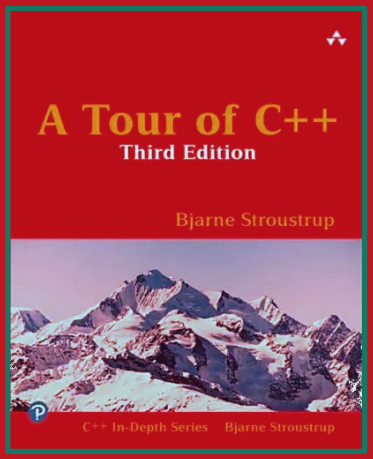
Institutt for datavitskap

Digitalisering på Ås

3 Data structures

3.1 Object orientation

3.2 Inheritance



Sections 5.3 – 5.5

Core Guidelines:

C.120 – C.122
C.146 – C.148

(and more in "C")

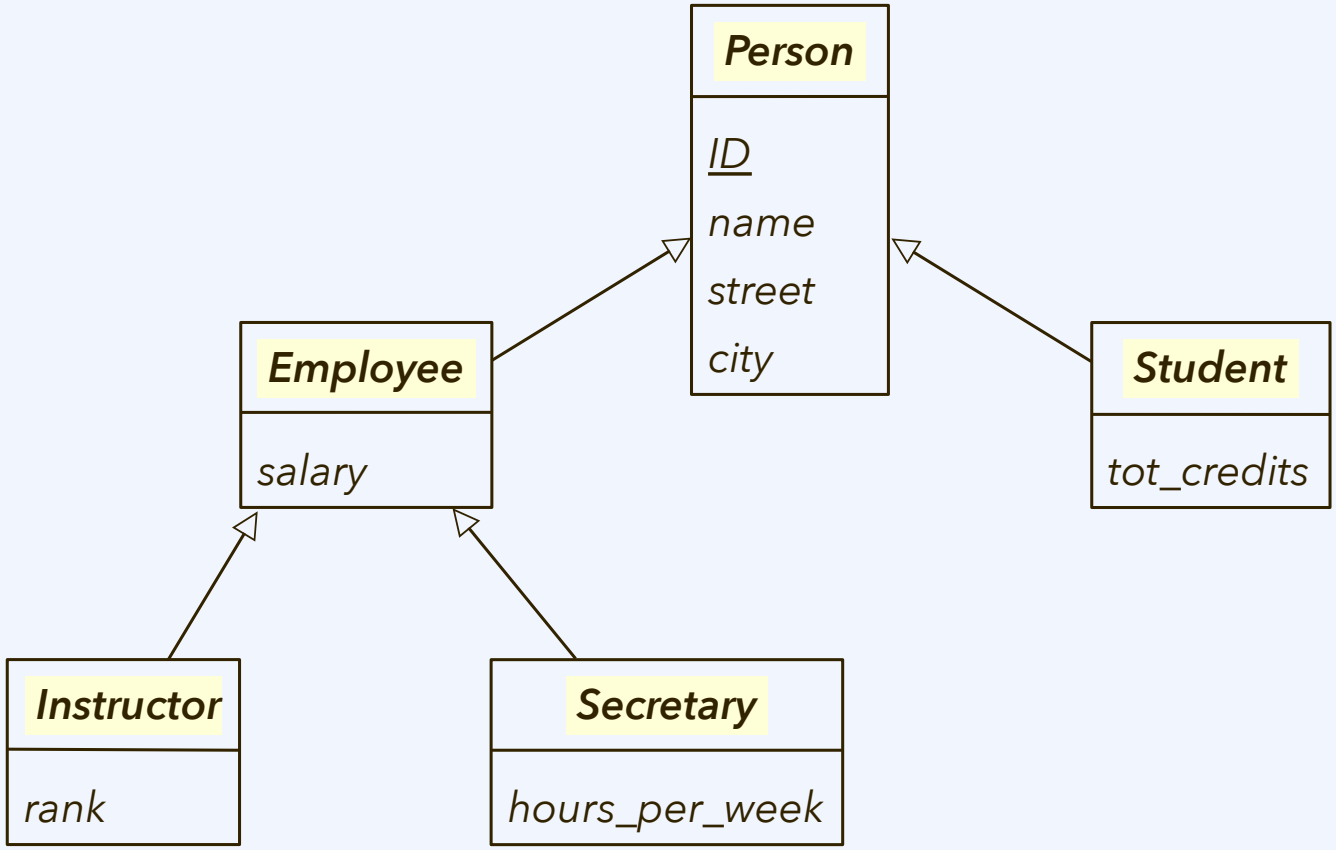
INF205

4th March 2024

E-R notation: Taxonomy/class hierarchy

Norwegian University of Life Sciences

Example from Silberschatz *et al.*[1] (Fig. 6.18):



[1]A. Silberschatz, H. F. Korth, S. Sudarshan, *Database System Concepts*, 7th int. stud. edn., McGraw-Hill, **2019**.

Class hierarchy implementation in C++

Norwegian University of Life Sciences

Classes can stand in a hierarchical relationship: A more general superclass and its more specific subclass (also, "derived class" or "child").

An object of the subclass then (automatically) is **also an object of the superclass**; it has all the members defined in its class definition, but also **inherits the members defined for the superclass**, to which it also belongs.

Example file: literature-indices.zip

```
class LiteratureIndex {
public:
  virtual int next_page();
  …
private:
  int year = 0;
  …
};
```

```
class JournalArticleIndex: public LiteratureIndex {
public:
  int next_page();
  …
private:
  int volume = 0;
  …
};
```

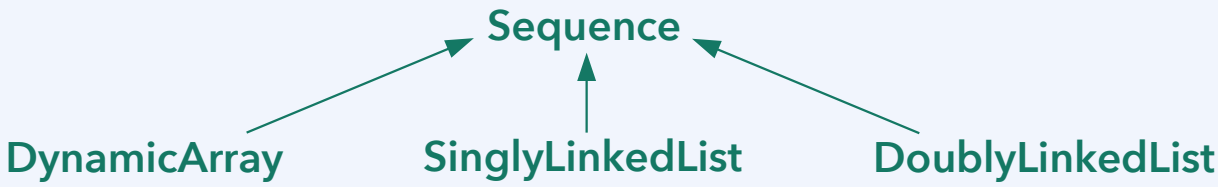
JournalArticleIndex has the property **volume**, but it also **inherits the property year**.

It can override the **next_page** method definition from its superclass, because it is **virtual**.

Abstract classes and concrete subclasses

The code **sequences-int.zip** has an **abstract class** at the top of a class hierarchy.

Such a class has a **pure virtual** method that is only declared, but not defined. The declaration uses the construction "**virtual** … method(…) **= 0;**".




```
class Sequence
{
public:
  virtual bool empty() const = 0;   // whether sequence is empty
  virtual size_t size() const = 0;  // size (number of items)

  virtual int& front() = 0;  // return reference to first item
  virtual int& back() = 0;   // return reference to final item

  virtual int& at(int i) = 0;  // reference to item at index i

  ...
};
```

A class is concrete (*i.e.*, not abstract) if it does *not* have any pure virtual methods.

If it has an abstract superclass, it must **override (define) all** its pure virtual method declarations.

Core guidelines

An abstract class might contain “normal” methods in addition to its pure virtual method(s). If it only has pure virtual methods, it is a pure abstract class. Such classes are used to specify **interfaces**.

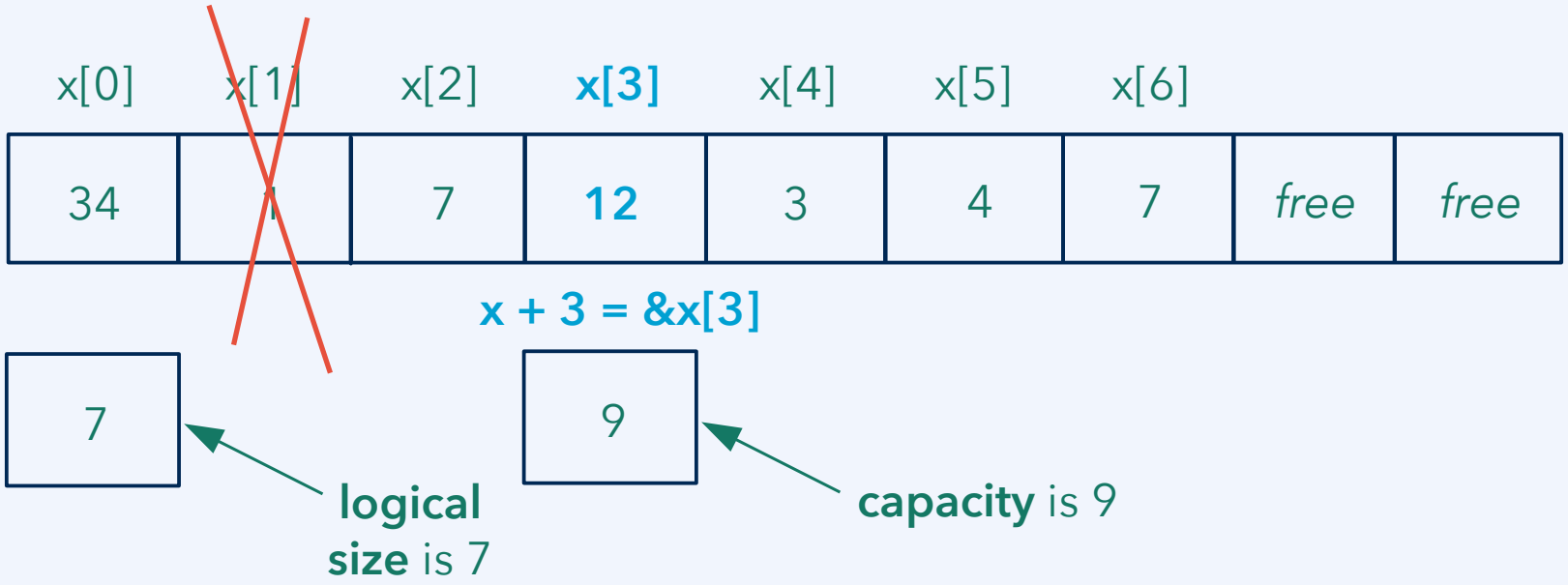
- C.120: Use class hierarchies to represent concepts with inherent hierarchical structure (only).
- C.121: If a base class is used as an interface, make it a pure abstract class.
- C.122: Use abstract classes as interfaces when complete separation of interface and implementation is needed.

Concerning virtual methods and overriding:

- C.128: Virtual functions should specify exactly one of virtual, override, or final.

Dynamic array implementation

- **Read/write access to an array element: O(1) time.**
  Address of the i-th element computable by pointer arithmetics.
- **Deleting an element from the array?** O(1) at the end, O(*n*) elsewhere.
  All the elements with greater indices need to be shifted.
- **Extending the array by one element?** O(1) at the end, if there is capacity.
  O(*n*) elsewhere, or if the capacity of the dynamic array is exhausted.



Abstract classes and concrete subclasses

The **sequences-int.zip** archive contains an example implementation.

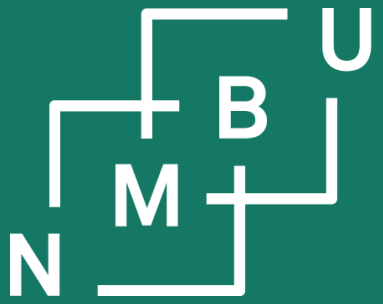
Interface (abstract class) from **Sequence**:

- bool empty() const;
- size_t size() const;
- int& front();
- int& back();
- int& at(int i);
- void push_front(const int& in);
- void push_back(const int& in);
- void push(const int& in);
- void insert_at(int idx, const int& in);
- void pop_front();
- void pop_back();
- void pop(const int& in);
- void erase_at(int idx);
- void clear();

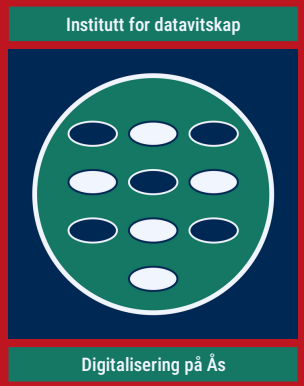
```
class DynamicArray: public Sequence {
public: … // implement all the interface from Sequence
  ~DynamicArray() { this->clear(); }

private:
  int* values = nullptr;

  size_t logical_size = 0;  // how many data items are we storing?
  size_t capacity = 0;  // how much memory did we allocate?

  // shift to static array with increased/decreased capacity
  void resize(size_t new_capacity);
};
```

Capacity takes values 0, 1, 2, 4, 8, 16, …

For several tasks we need to copy data that are contiguous in memory. For this, we use std::copy(init, end, target) from <algorithm>.



Noregs miljø- og biovitskaplege universitet



3 Data structures

3.1 Object orientation

3.2 Inheritance

3.3 Linked data structures

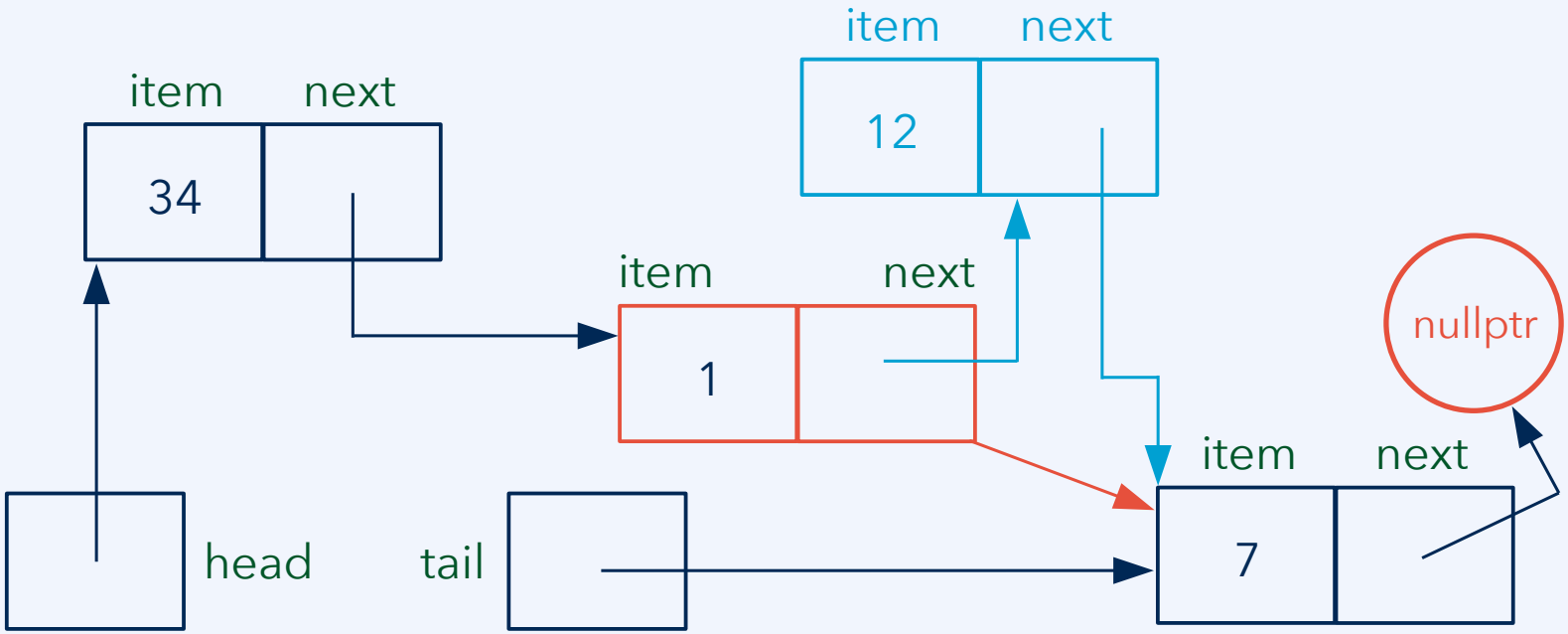
INF205

4th March 2024

Linked lists (singly linked)

Linked lists are **dynamic data structures**. Their elements are **not contiguous in memory**. Therefore, pointer arithmetics and increments (p++) cannot be used. Instead, the linked list consists of **nodes**.

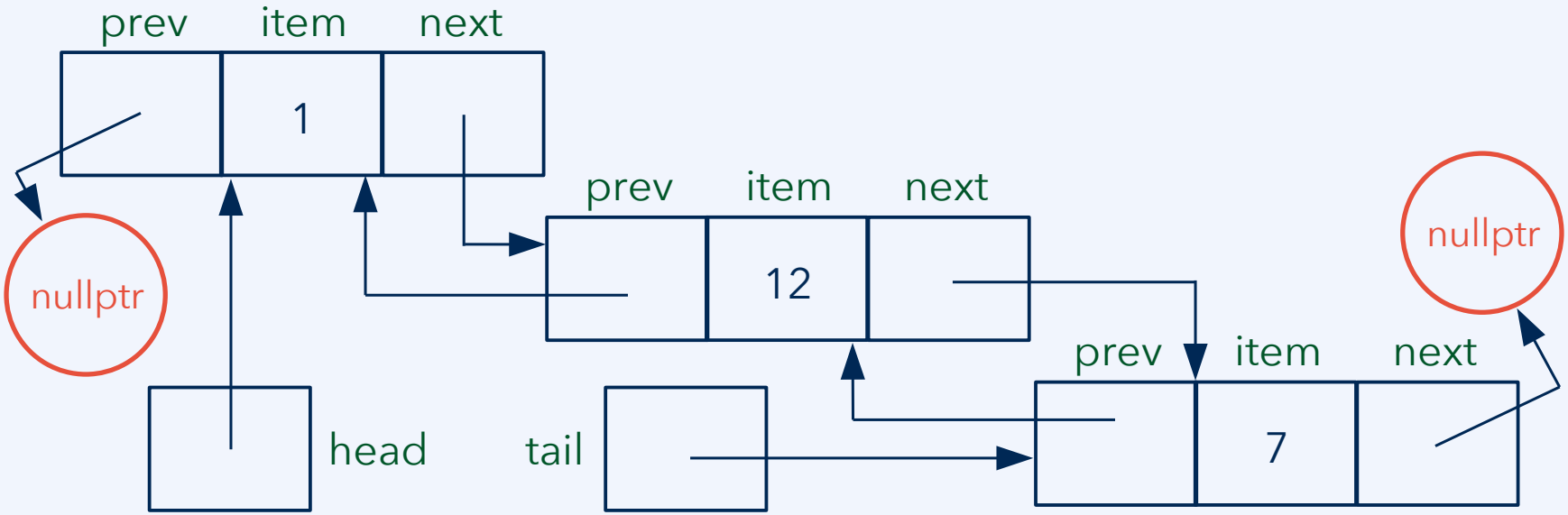
Example task: Insert 12 after node x, to which we already have a reference.



Linked lists (doubly linked)

In a **doubly linked list**, each node also contains a reference (or pointer) to the **previous node**. This facilitates traversal in **both directions and inserting** a new data item **before** any given node (rather than only after it), all in constant time.

Singly linked lists require two variables per data item (item and next).
Doubly linked lists require three variables per data item (prev, item, and next).



Discussion: Deleting or copying a linked list

Say we have a **linked-list** object x, consisting of the fields “head” and “tail”.

- The object runs out of scope and is getting deallocated. We did not define a constructor, so the default constructor is used.
  - What will this do? Is it what we want?
- We assign the value of x to another list object y, writing “y = x”. By default, this means that the property values “head” and “tail” are copied.
  - Is this reasonable? What could the user be expecting instead?

Or: We have a **singly-linked-list node** object, consisting of “item” and “next”.

- What will happen upon deallocation by default? What should happen?
- What will happen upon copying by default? What should happen?

Example implementation

Singly linked list of integers as in the **sequences-int.zip** example:

Interface (abstract class) from **Sequence**:

- bool empty() const;
- size_t size() const;
- int& front();
- int& back();
- int& at(int i);
- void push_front(const int& in);
- void push_back(const int& in);
- void push(const int& in);
- void insert_at(int idx, const int& in);
- void pop_front();
- void pop_back();
- void pop(const int& in);
- void erase_at(int idx);
- void clear();

```
class SinglyLinkedList: public Sequence {
public: … // implement all the interface from Sequence
  ~SinglyLinkedList() { this->clear(); }
private:
  SinglyLinkedListNode* head = nullptr;
  SinglyLinkedListNode* tail = nullptr;
};
```

```
class SinglyLinkedListNode {
public:
  int& get_item() { return this->item; }
  SinglyLinkedListNode* get_next() const { return this->next; }
  void set_item(int in_item) { this->item = in_item; }
private:
  int item = 0;
  SinglyLinkedListNode* next = nullptr;
  void set_next(SinglyLinkedListNode* in) { this->next = in; }
  friend class SinglyLinkedList;
};
```

Overview: Sequential data structures

- **Read/write** access to a data item at position k
  - For a dynamic array, $O(1)$ time; fast access by pointer arithmetics
  - For a singly linked list, $O(k)$ time, *i.e.*, $O(n)$ in the average/worst case
  - For a doubly linked list, $O(\min(k, n - k))$, which is still effectively $O(n)$

- **Iterating** over the data, *i.e.*, proceeding from one item to the next one
  - $O(1)$ both for dynamic arrays and for linked lists

- **Deleting** a data item at position k
  - For a dynamic array, $O(1)$ at the end, $O(n - k)$ in general
  - For a singly linked list, $O(1)$ at the head, or if we have a reference to the element at position k–1; otherwise, in general, $O(k)$
  - For a doubly linked list, $O(1)$ at the head or tail, or if we have a reference to that region of the list; in general, $O(\min(k, n - k))$

Remark: For linked lists, insertion/deletion as such takes constant time, once the node has been localized. However, getting to the node can take $O(n)$ time.

Overview: Sequential data structures

- **Read/write** access to a data item at position k
  - For a dynamic array, $O(1)$ time; fast access by pointer arithmetics
  - For a singly linked list, $O(k)$ time, *i.e.*, $O(n)$ in the average/worst case
  - For a doubly linked list, $O(\min(k, n - k))$, which is still effectively $O(n)$

- **Iterating** over the data, *i.e.*, proceeding from one item to the next one
  - $O(1)$ both for dynamic arrays and for linked lists

- **Inserting** an additional data item at position k
  - For a dynamic array, $O(n)$ in the worst case, *i.e.*, whenever the capacity is exhausted; with free capacity, $O(1)$ at the end, $O(n - k)$ elsewhere
  - For a singly linked list, $O(1)$ at the head or tail, or if we have a reference to the element at position k–1; Otherwise, in general, $O(k)$
  - For a doubly linked list, $O(1)$ at the head or tail, or if we have a reference to that region of the list; in general, $O(\min(k, n - k))$

Remark: For linked lists, insertion/deletion as such takes constant time, once the node has been localized. However, getting to the node can take $O(n)$ time.

Stacks and queues

Queue

- nn, nb kø *m.*

Definition: A queue is a sequential (list-like) dynamic data structure that functions by the principle *first in, first out (FIFO)*.

- A queue class must provide a method for appending an element, usually called ***push*** and done on one end of the queue (*e.g.*, ***enqueue*** or ***push_back***), and a method for detaching an element, usually called ***pop*** and done at the other end of the queue (*e.g.*, ***dequeue*** or ***pop_front***).
- Singly and doubly linked lists are well suitable for implementing a queue, since both ***push*** and ***pop*** can be realized in constant time. However, a singly linked list should only be used if the data structure includes an explicit reference to the tail node; otherwise, the whole list needs to be traversed just to reach the tail, taking O(*n*) time.
- Dynamic arrays are less suitable for this purpose, requiring O(*n*) time for the ***push*** and ***pop*** operations in the long run (as their capacity gets exhausted).

- **Queues** function by the principle **"first in, first out" (FIFO)**
  - Can be implemented using a singly linked list (with a tail reference):
    - Attach (enqueue / push) new elements at the tail of the list only
    - Detach (dequeue / pop) elements from the head of the list only

Stacks and queues

- **Stacks** function by the principle **"last in, first out" (LIFO)**
  - Can be implemented using a singly linked list:
    - Attach (push) new elements at the head of the list only
    - Detach (pop) elements from the head of the list only
  - Can be implemented using a dynamic array:
    - Attach (push) new elements at the end of the array only
    - Detach (pop) elements from the end of the array only
- **Queues** function by the principle **"first in, first out" (FIFO)**
  - Can be implemented using a singly linked list (with a tail reference):
    - Attach (enqueue / push) new elements at the tail of the list only
    - Detach (dequeue / pop) elements from the head of the list only

All these operations can be carried out in constant time;
in case of the push operation for the dynamic array, subject to capacity.

Linked data structures as containers

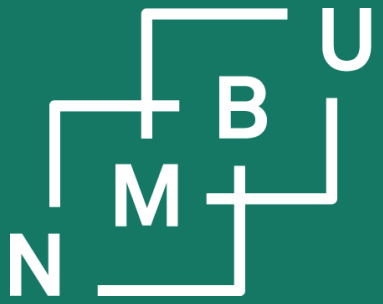
Container

- nn konteinar, container *m.*
- nb konteiner, container *m.*

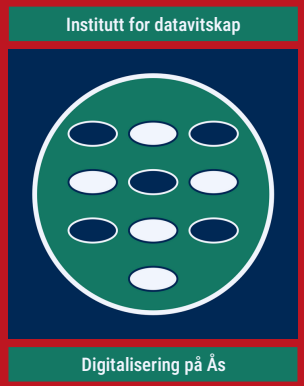
Definition (Stroustrup): "A class with the main purpose of holding objects is commonly called a *container*."

- Container objects take *ownership*, *i.e.*, responsibility for allocating and deallocating any contained data. The programmer needs to take care of this whenever there are data subject to manual memory management (new and delete) in a self-designed container.
- Examples include the standard template library (STL) containers (list, map, set, vector, *etc.*). Other than for educational purposes as an exercise, it does not make sense to reimplement these standard data structures by hand.
- Many problems require special, tailored container data structures in order to be solved efficiently. It is then part of the development work to both design and implement the required data structure.

See also: Object-oriented programming, pointer, queue.



Noregs miljø- og biovitskaplege universitet



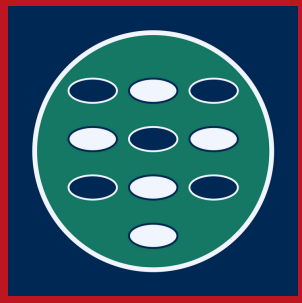
Third worksheet

INF205

4th March 2024

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap



Digitalisering på Ås

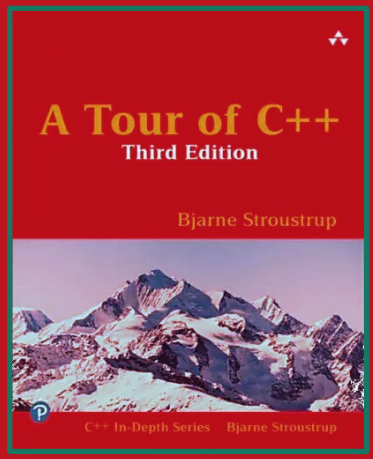
3 Data structures

3.1 Object orientation

3.2 Inheritance

3.3 Linked data structures

3.4 Containers



Sections 6.1 – 7.2

Core Guidelines:

C.31 – C.33, T.1, T.2, T.62, T.83

(+ more "C" & "T")

INF205

4th March 2024

Rule of three

Container objects take **ownership**, *i.e.*, lifetime and deallocation responsibility. The programmer needs to take care of this whenever there are data subject to **manual memory management** (*new* and *delete*) in a **self-designed container**.

The programmer needs to take care of this whenever there are data subject to **manual memory management** (*new* and *delete*) in a **self-designed container**.

"**Rule of three:**" For a container, implement at least
(1) destructor,
(2) copy constructor,
(3) copy assignment operator.

Most often you will then also need to implement **(0)** a constructor.

At least implement **(1) the destructor**!
If **(2)** and **(3)** are not there, **forbid copying**.

Ownership

Container objects take **ownership**, *i.e.*, lifetime and deallocation responsibility. The programmer needs to take care of this whenever there are data subject to **manual memory management** (*new* and *delete*) in a **self-designed container**.

Example: Let us assume that **class T** has one property for which it has ownership, a pointer p to class S that points to an array of 1000 S elements.

It is typical for the owned content, if manual memory management needs to be done, to be allocated in the constructor, **T::T()** and/or **T::T(…)**.

```
class T
{
public:
  T() { this->p = new S[1000](); }
  ~T() { delete[] this->p; }
  …
private:
  S* p = nullptr;
  …
}
```

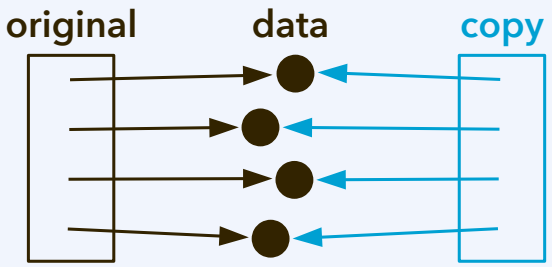
T tobject;

T* tpointer = new T;

Copying an object

Shallow copy:

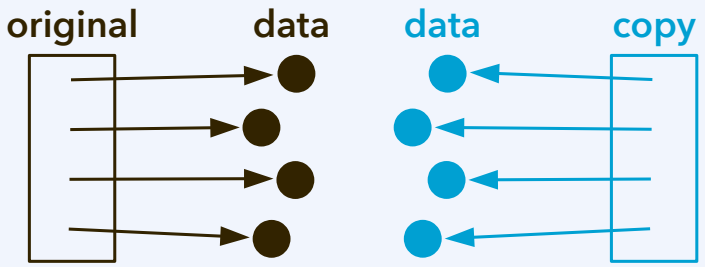
Standard copying, such as if there is no handwritten copy constructor or copy assignment operator, will simply **copy the value of pointers**, *not the content* to which they point.



After shallow copying, the content will exist **once in memory**. This can be appropriate when the content is **not owned** but just pointed at.

Deep copy:

Standard copying, such as if there is no handwritten copy constructor or copy assignment operator, will simply **copy the value of pointers**, *not the content* to which they point.



After deep copying, content exists **twice in memory**. Design following the concept of a "container" that **uniquely "owns"** its content requires deep copying.

Fast copying (to implement deep copying)

Element-wise copying

```
for(int i = 0;  i < num_copy;  i++)  target[i] = source[idx_start + i];
```

This is slow! Don't do this for large numbers of elements adjacent in memory. Also, note that Core Guidelines recommend "size_t" instead of int.

C-style fast copying (apply this only to a traditional C/C++ array)

```
#include <cstring>
…
std::memcpy(target,  source + idx_start,  num_copy * sizeof(element_type));
```

Modern C++ style fast copying (can also be used for STL containers)

```
#include <algorithm>
…
std::copy(source + idx_start,  source + idx_start + num_copy,  target);
```

Copy constructor

The **copy constructor T::T(const T&** orig**)** is called when the following two are done at the same time: **(1) allocation** of an object, so that a constructor needs to be called, and its **(2) initialization** to the value of a pre-existing object that continues to exist.

Examples for when the **copy constructor** is called:

```
// default constructor
T tfirst;
…
// copy constructor
T tsecond = tfirst;
```

```
void func(T param) { … }

int main() {
  T tobject;
  …
  // copy constructor
  func(tobject);
}
```

after running the copy constructor, the same content must exist in memory twice!

```
class T
{
public:
  T() { this->p = new S[1000](); }
  T(const T& original) {
    this->p = new S[1000]();
    std::copy(
      original.p, original.p+1000,
      this->p
    );
  }
  …
}
```

std::copy can be used for data that are contiguous in memory

1. Create space for the duplicate.
2. Now write the duplicate into it.

Copy assignment operator

The **copy assignment operator** technically is an overloaded "=" operator:

```
T& T::operator=(const T& rhs) { … }
```

Difference from the copy constructor:

- Object already exists, hence *no initial allocation* of memory for content.
- But *deallocate pre-existing content*.

```
// default constructor
T tfirst, tsecond;
…
// copy assignment
tsecond = tfirst;
```

A **copy assignment** is done whenever we copy the value of one variable to another, **both existed** before, and **both continue to exist**.

after running the copy assignment, the same content must exist in memory twice!

```
class T
{
public:
  T() { this->p = new S[1000](); }
  T& operator=(const T& rhs) {
    if(&rhs == this) return *this;
    delete this->p;
    this->p = new S[1000];
    std::copy(
      rhs.p, rhs.p+1000, this->p
    );
    return *this;
  }
  …
}
```

Note that a reference to *this needs to be returned.

Copy assignment operator

The **copy assignment operator** technically is an overloaded "=" operator:

```
T& T::operator=(const T& rhs) { … }
```

Difference from the copy constructor:

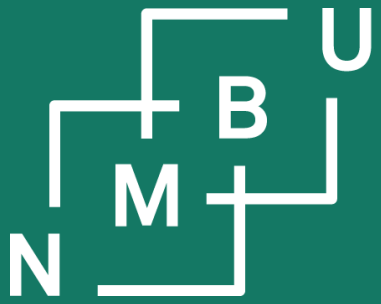
- Object already exists, hence *no initial allocation* of memory for content.
- But *deallocate pre-existing content if necessary*.

```
// default constructor
T tfirst, tsecond;
…
// copy assignment
tsecond = tfirst;
```

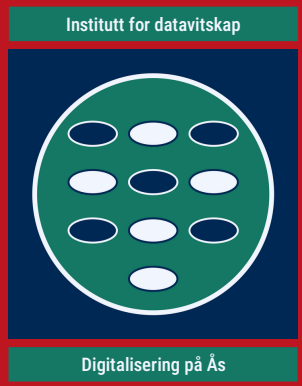
A **copy assignment** is done whenever we copy the value of one variable to another, **both existed** before, and **both continue to exist**.

after running the copy assignment, the same content must exist in memory twice!

```
class T
{
public:
  T() { this->p = new S[1000](); }
  T& operator=(const T& rhs) {
    if(&rhs == this) return *this;

    std::copy(
      rhs.p, rhs.p+1000, this->p
    );
    return *this;
  }
  …
}
```



Noregs miljø- og biovitskaplege universitet



Conclusion

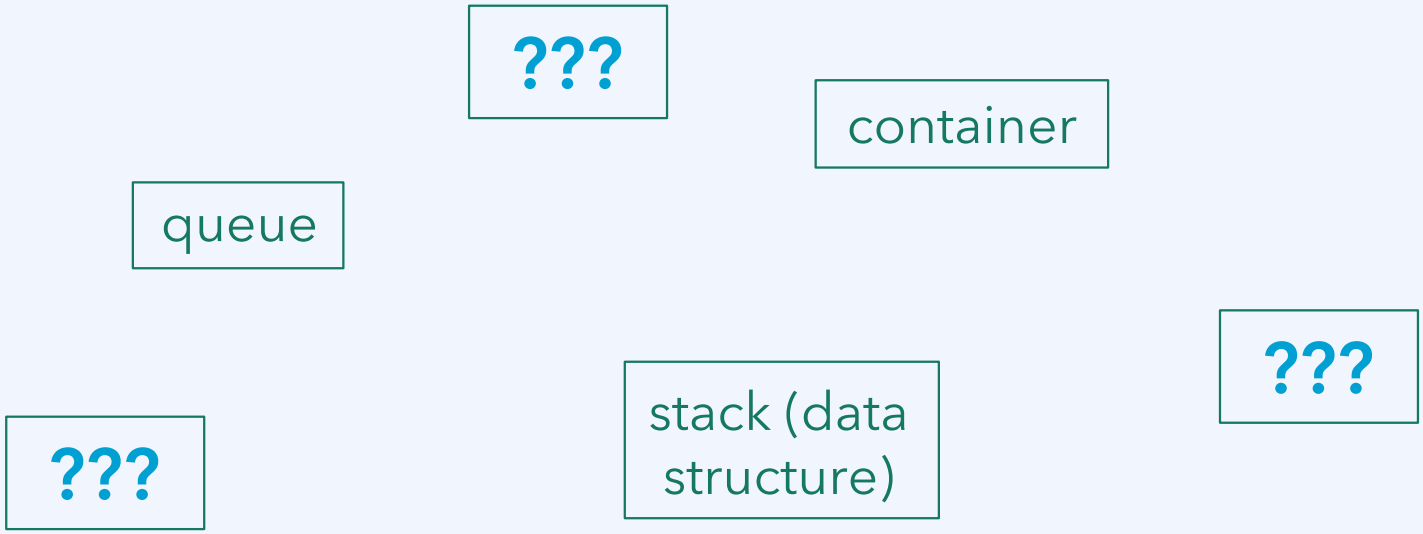
INF205

4th March 2024

Weekly glossary concepts

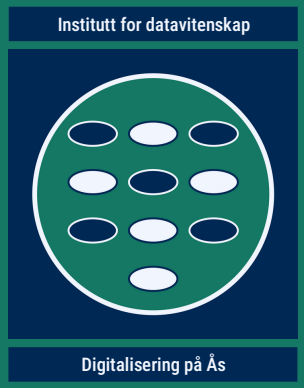
What are essential concepts from this lecture?

Let us include them in the **INF205 glossary**.[1]



[1]https://home.bawue.de/~horsch/teaching/inf205/glossary-en.html

Norges miljø- og biovitenskapelige universitet



INF205
Resource-efficient programming

3 Data structures

3.1 Object orientation
3.2 Inheritance
3.3 Linked data structures
3.4 Containers
3.5 Graph data structures
3.6 Streams and file I/O