Norges miljø- og biovitenskapelige universitet

INF205
Resource-efficient programming

2 Data structures

2.1 Strings
2.2 Dynamic data structures
2.3 Standard template library
2.4 Linked lists

Use of the "const" keyword

1. Variables are declared as const if we do not plan to modify them after initialization.

2. We declare pointers to const (of type T) as const T*. (References as const T&.)

3. The above is used for pass-by-reference:

```
int do_something(const T& s);
int do_something(const T* s);
```

const variables may only be passed by reference if the parameter is also const.

4. Read-only method: T::method() **const**.

Examples:

1) "**const-array**" bug, see right.
2) Next let us look at the "**sphere-collisions-struct**" code. Where is it possible to write "const"?

const-array.cpp – mistake fixed as follows:

```
int second_largest_of(int N, const int x[]) {
  int largest = numeric_limits<int>::min();
  int second_largest = numeric_limits<int>::min();

  for(int i = 0; i < N; i++)
    if(x[i] > largest) {
      second_largest = largest;
      largest = x[i];
    }
    else if(x[i] > second_largest)
      second_largest = x[i];
  return second_largest;
}

int main() {
  constexpr int fixed_array_size = 5;
  const int x[fixed_array_size] = {4, 0, 6, 5, 2};
  cout << second_largest_of(fixed_array_size, x);
}
```

From structures to classes

u39 tutorial problem 2:
Develop the "Sphere" (collisions) example into a class where the properties are private.

1) Use "class" instead of "struct".

2) Split class definition into a public part and a private part.

3) Introduce "get" and "set" methods to access the properties.

4) Adjust the remaining code so that direct access to private properties is replaced with calling the "get" and "set" methods.

```
class Sphere
{
public:
   float get_size() const { return this->size; }
   float get_coordinate(int axis) const;

   void set_size(float in_size);
   void set_coordinate(int axis, float in_coord);

   // is there a collision between "this" and "other"?
   bool check_collision(const Sphere* other) const;

private:
   float size = 0.0;
   float coords[3] = {0.0, 0.0, 0.0};
};
```

```
…
spheres[i].set_size(size);
…
```

Norwegian University
of Life Sciences

Virtual methods and abstract classes

u39 tutorial problem 3 used the keyword "virtual" for method declarations in the superclass (parent) to specify that they are overridden in all cases if an object instantiates the subclass (child, derived class).

This can be made more explicit by the (optional) keyword "override", as below:

$$\begin{tikzcd}
 & \text{LiteratureIndex} & \\
\text{BookIndex} \arrow[ur] & & \text{JournalArticleIndex} \arrow[ul]
\end{tikzcd}$$

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

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

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

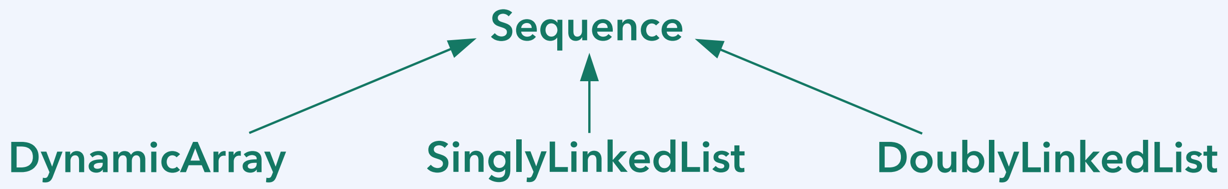
It has the property **volume**, but it also **inherits the property year**.

Virtual methods and abstract classes

u40 tutorial problem 2 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;**".

(See code example "**sequential-data-structures**".)



```
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)** **<u>all</u>** its pure virtual method declarations.

Noregs miljø- og biovitskaplege universitet

2 Data structures

2.1 Strings

Strings in C and in C++

The C++ language only prescribes what functionalities a **std::string** should provide, not how it is realized at the memory level, which is up to the compiler.

Most implementations remain close to that from the C language, where character arrays terminated by the null character '\0' are employed. (If you want to enforce this, you can also still use all the C style constructs explicitly.)

string s = "INF205"; or **char s[] = "INF205";** produce the following in memory:

'I'	'N'	'F'	'2'	'0'	'5'	'\0'
73	78	70	50	48	53	0

Also to ensure backwards compatibility with C, **string literals** between double quotation marks such as "INF205" are of the type **const char*** (not **std::string**). Between single quotation marks there is always a **char**, such as **char x = 'a';**

Strings in C and in C++

To remember about **C++ strings** as opposed to **C strings**: Even if they are usually realized as arrays at the memory level, they are not arrays to the programmer. As a consequence, **it is possible to pass C++ strings by value**.

C strings, however, **can never be passed by value** because they are arrays.

```
void increment_at(int p, char* str) {
   str[p]++;
}

int main()
{
   char c_style_str[] = "INF205";
   increment_at(5, c_style_str);
   cout << c_style_str << "\n";
}
```

```
void increment_at(int p, string str) {
   str[p]++;
}

int main()
{
   string cpp_style_str = "INF205";
   increment_at(5, cpp_style_str);
   cout << cpp_style_str << "\n";
}
```

I/O operator overloading

See example code **io-operator-overloading** for the following.

Assume that for some **class C**, we have (as discussed before) defined methods that write content to a stream, or that analogously read from a stream.

```
void C::out(ostream* target) const {
  *target << … ;
}
```

```
void C::in(istream* source) {
  *source >> … ;
}
```

You can convert this to overloaded I/O operator definitions:

```
ostream& operator<<(
  ostream& str, const C& x
) const {
  x.out(&str);
  return str;
}
```

```
istream& operator>>(istream& str, C& x)
{
  x.in(&str);
  return str;
}
```

Now you can use the operator << and the operator >> on objects of type C just like for numbers, *etc.*

The overloaded operators must be positioned outside a namespace. (Like the original << and >> operators.)

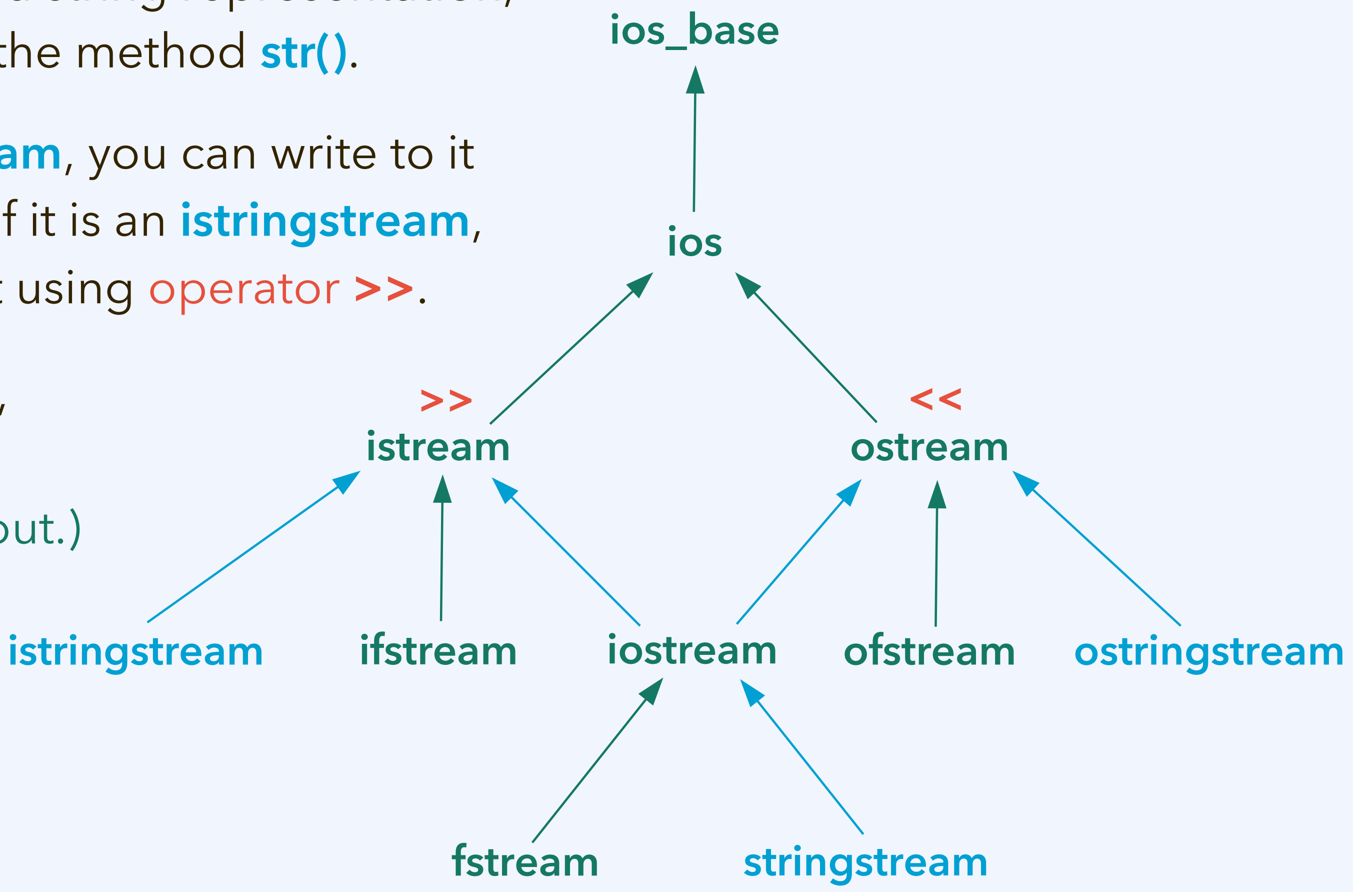
String streams

See example code **io-operator-overloading** for the following.

String streams have a string representation, accessible through the method **str()**.

If it is an **ostringstream**, you can write to it using operator **<<**, if it is an **istringstream**, you can read from it using operator **>>**.

With a **stringstream**, you can do both.
(FIFO: First-in, first-out.)



Noregs miljø- og biovitskaplege universitet

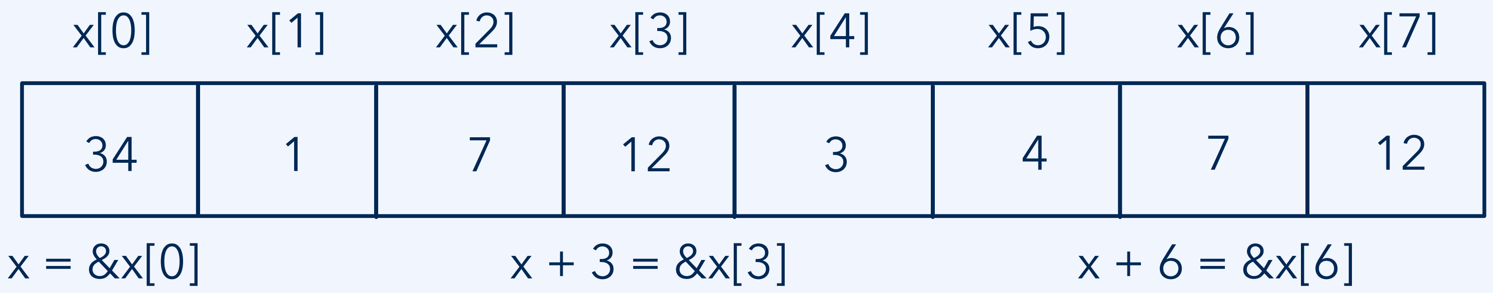
2 Data structures

2.1 Strings

2.2 Dynamic data structures

C/C++ arrays (static arrays)

An array contains a sequence of elements of the same type, arranged **contiguously in memory**. This supports fast access using **pointer arithmetics**. Once created, the size of a C/C++ array is fixed; we cannot append elements.



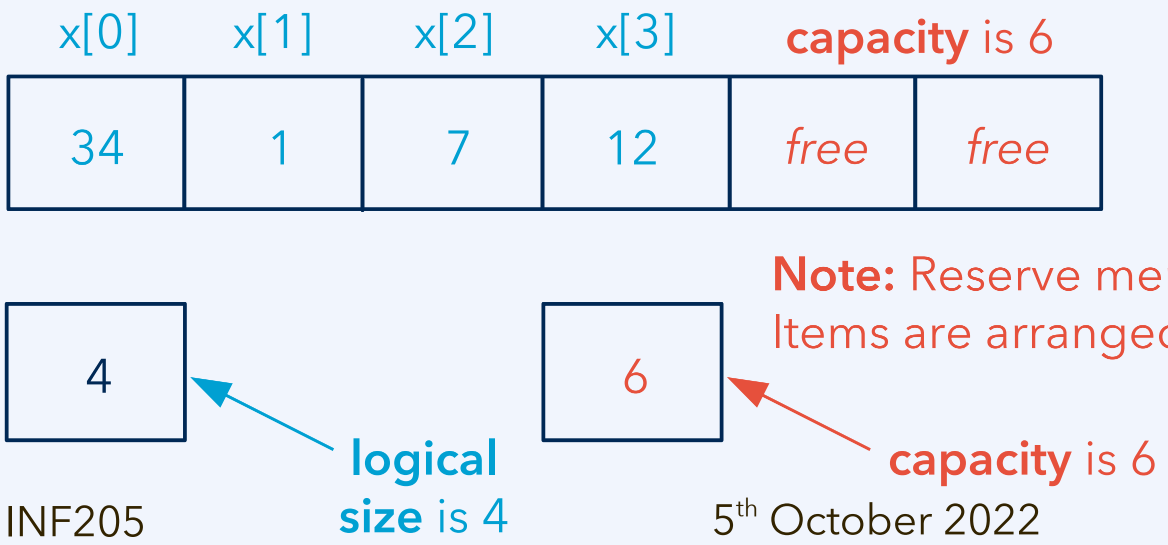
In C/C++, the type of an array such as **int[] is the same as the corresponding pointer type int***, *i.e.*, **the array actually is a pointer**. Its value is an address at which an integer is stored, namely, the memory **address of the first element**.

This is highly efficient since when x[i] is accessed, the compiler transforms this into accessing the memory address x + sizeof(int) * i.

Lists in Python (dynamic arrays)

Conventional arrays are **static data structures**. Their size in memory is constant, and memory needs to be allocated only once, *e.g.*, at declaration time. (Details depend on programming language, compiler, flags/optimization level, *etc.*).

Dynamic data structures can change in size and/or structure at runtime. For an array, this can be implemented by **allocating reserve memory** for any elements that may be appended in the future. When the capacity of the dynamic array is exhausted, all of its contents need to be shifted to another position in memory.



in Python:

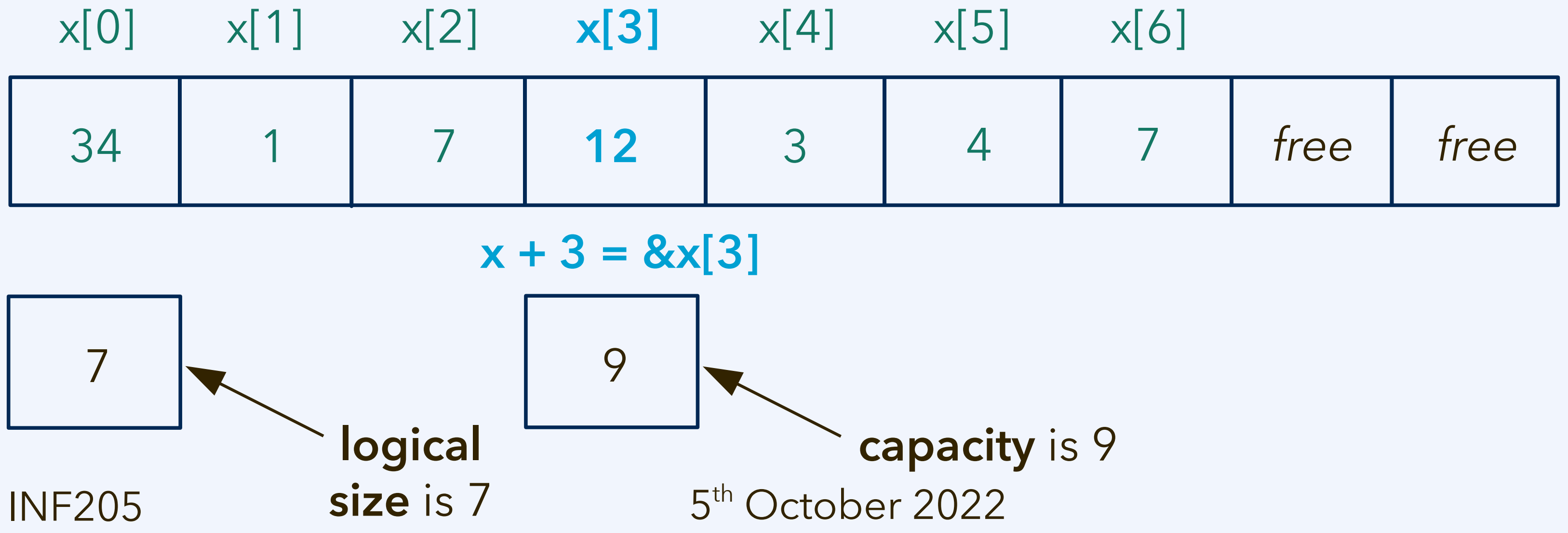
x = [34, 1, 7, 12]

Note: Reserve memory capacity is allocated. Items are arranged contiguously in memory.

Dynamic arrays: Efficiency analysis

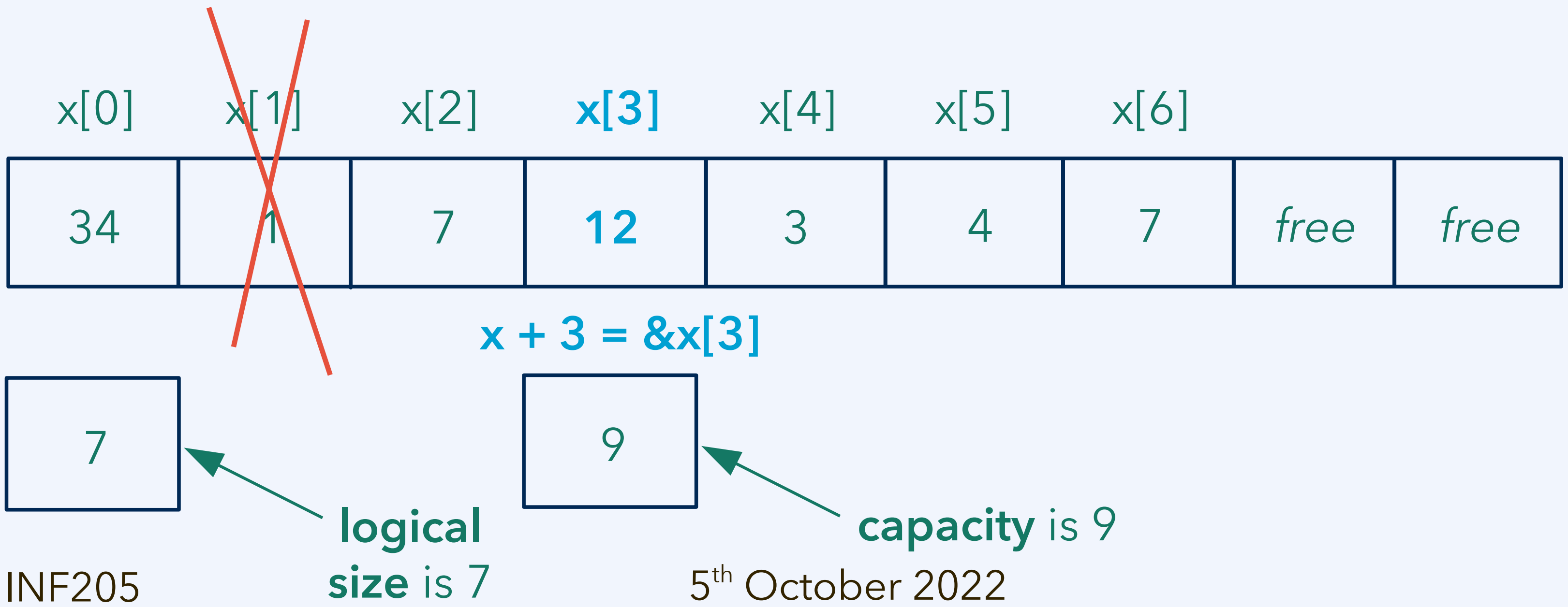
- **Read/write access to an array element: O(1) time.**
  Address of the i-th element computable by pointer arithmetics.

An array contains a sequence of elements of the same type, arranged **contiguously in memory**. The compiler, and also the programmer, can use **pointer arithmetics** for converting indices to addresses.



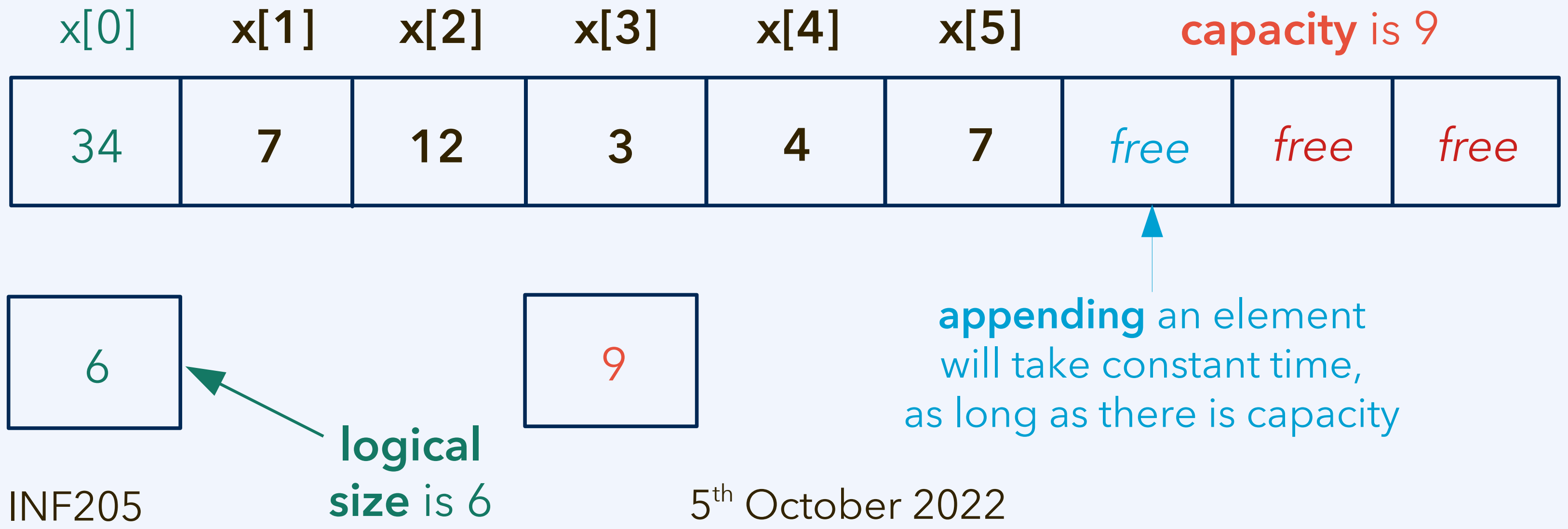
Dynamic arrays: Efficiency analysis

- **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.



Dynamic arrays: Efficiency analysis

- 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.



Example implementation

The **sequential-data-structures** 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);
};
```

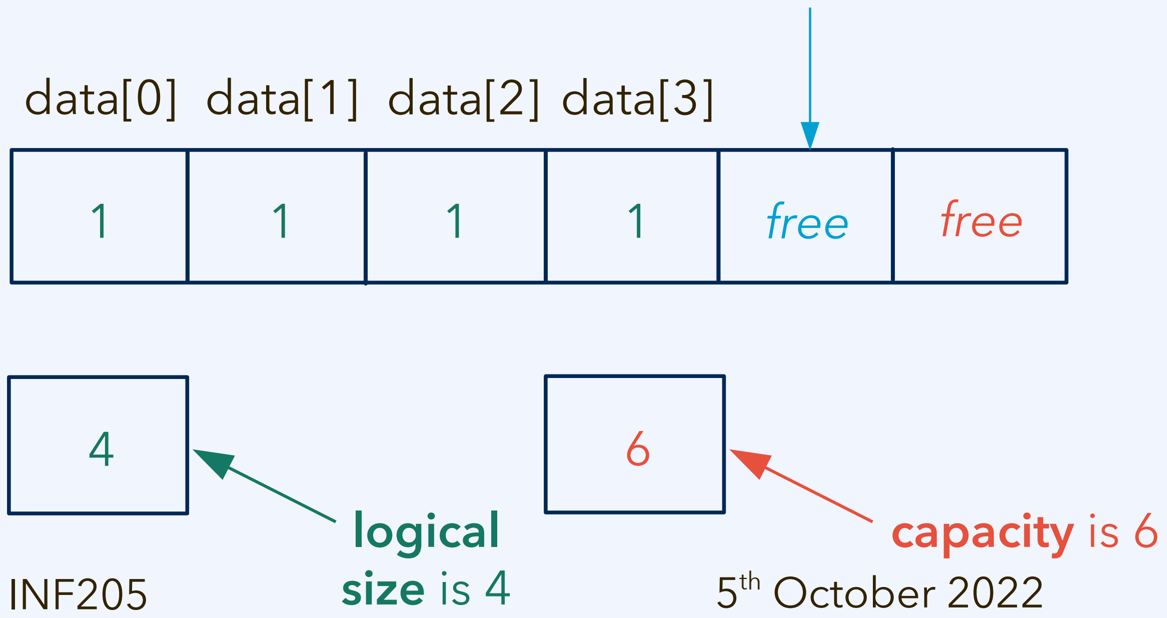
Capacity takes values 0, 1, 2, 4, 8, 16, …

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

STL vector: Dynamic array in C++

The standard template library (STL) provides typical **container** data structures. They are **templates**: They can contain any type of fundamental data items or objects as their elements. The **element type** is specified in angular brackets.

A dynamic array can be declared (with "#include <vector>") as an object of the parameterized class **vector**<**T**>, *e.g.*, "**vector**<**int**> data = {1, 2, 3, 4};".



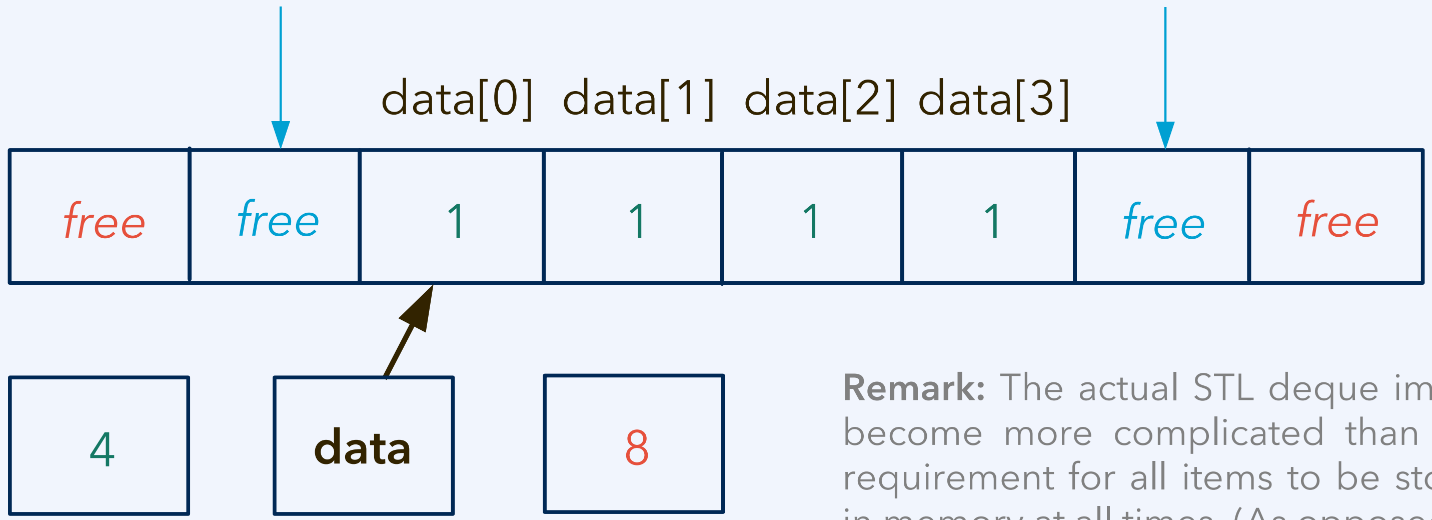
Functionalities of the **STL vector** include the ones from our "Sequence" interface, but also explicit addressing with "[index]" notation and many more.

Symmetric dynamic array data structure

The dynamic array data structure that we introduced is asymmetric. Insertions and deletions at the tail are fast if there is *free capacity at the tail*. At the head end, however, there is never any free capacity.

If needed, this can be improved upon by allocating free capacity at both ends.

A symmetric dynamic array can be declared (with "#include <deque>") as a "double ended queue" object **deque**<T>, *e.g.*, "**deque**<int> data = {1, 2, 3, 4};".



Remark: The actual STL deque implementation can become more complicated than this. There is no requirement for all items to be stored contiguously in memory at all times. (As opposed to vector.)

Noregs miljø- og biovitskaplege universitet

2 Data structures

2.1 Strings

2.2 Dynamic data structures

<u>2.3 Standard template library</u>

The STL containers

The standard template library (STL) provides typical **container** data structures. They are **templates**: They can contain any type of fundamental data items or objects as their elements. The **element type** is specified in angular brackets.

```
// declare a list of int values
std::list<int> my_list();
```

```
// declare a list of BookIndex objects
std::list<BookIndex> my_list();
```

- **vector**<**T**> is a **dynamic array** for type **T** elements. (Free capacity: Tail only.)
- **deque**<**T**> ("double ended queue"): **Dynamic array** with capacity both ends.
- **forward_list**<**T**> is a **singly linked list** data structure for type **T**.
- **list**<**T**> is a **doubly linked list** data structure for type **T**.
- **set**<**T**> is a container where each **key** (element) occurs only (at most) once.
- **map**<T, **V**> contains **key-value** pairs, which each key occurring at most once.
- **array**<T, **n**> is a **static array** for type **T**, with array size **n**, similar to **T[ ]** arrays.

Noregs miljø- og biovitskaplege universitet

2 Data structures

Example implementation

Singly linked list of integers as in the sequential-data-structures 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;
};
```

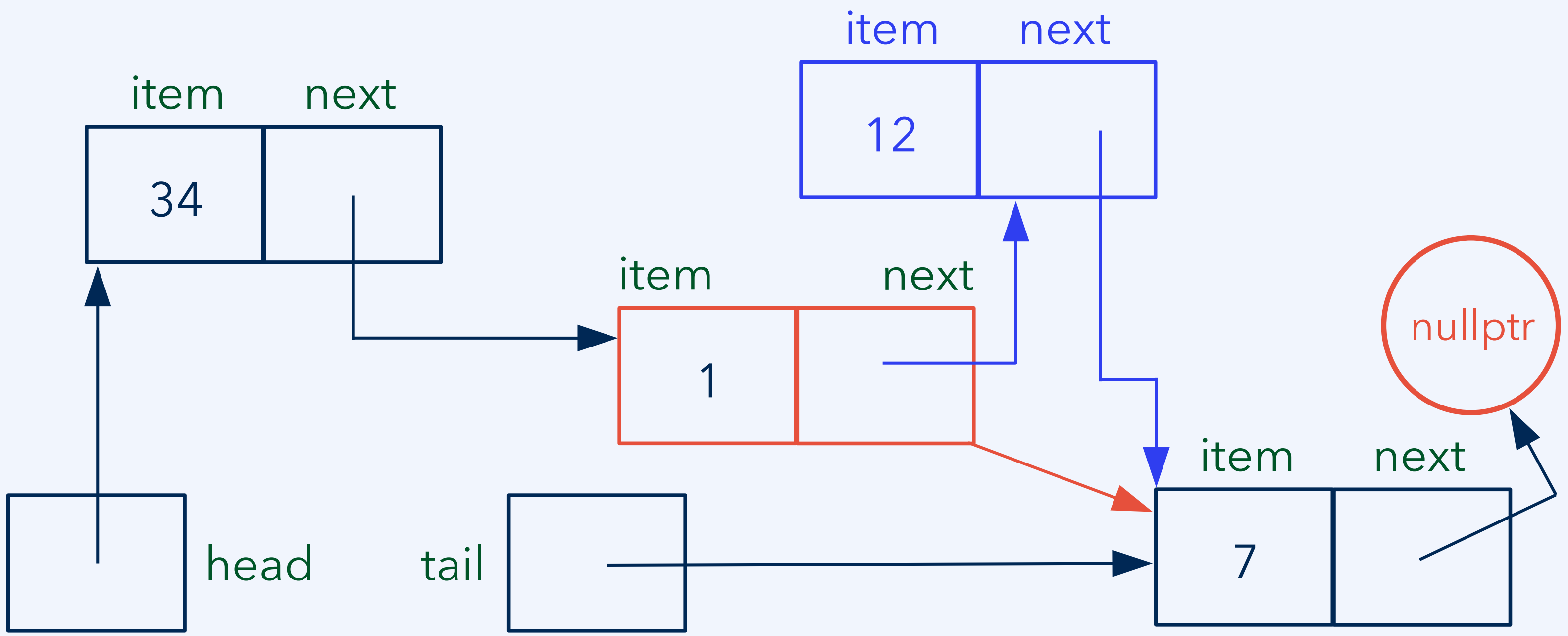
```
class SinglyLinkedListNode {
public:
  int& get_item() { return this->item; }
  SinglyLinkedListNode* get_next() const { return this->next; }
  void set_item(int in_item) { this->item = in_item; }

private:
  int item = 0;
  SinglyLinkedListNode* next = nullptr;
  void set_next(SinglyLinkedListNode* in) { this->next = in; }
  friend class SinglyLinkedList;
};
```

Singly linked list

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.

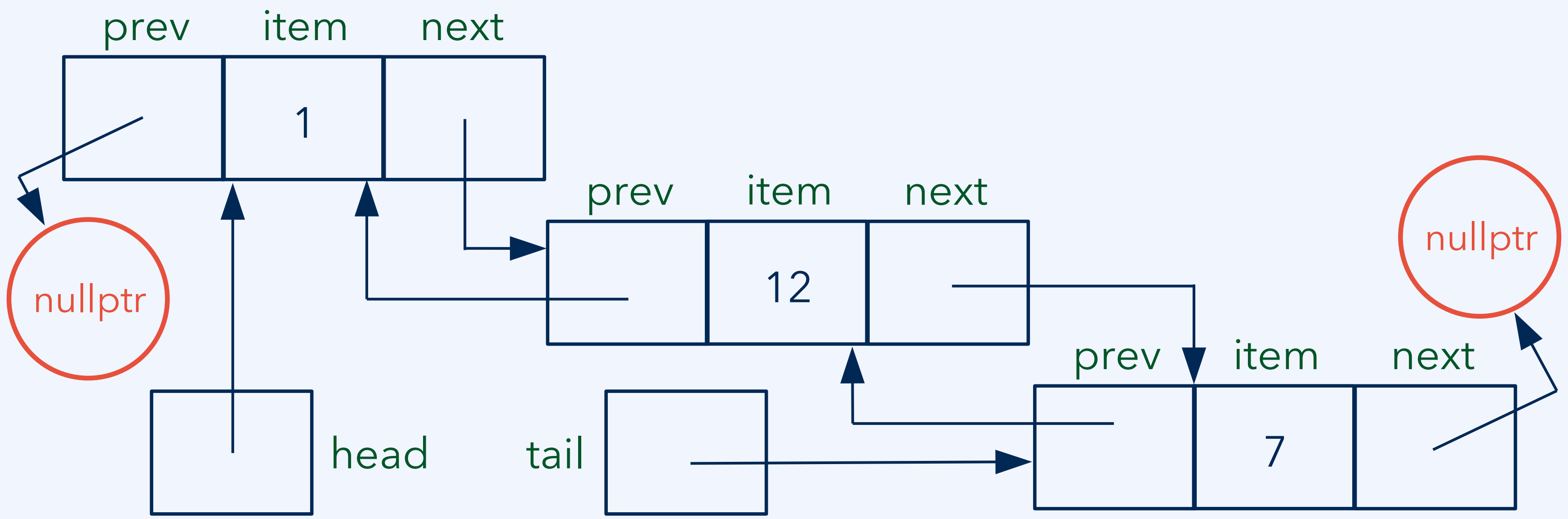


Norwegian University of Life Sciences

Doubly linked list

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).



Summary: Efficiency analysis

- **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.

Norwegian University of Life Sciences

Summary: Efficiency analysis

- **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.

Noregs miljø- og biovitskaplege universitet

Conclusion

Norges miljø- og biovitenskapelige universitet

INF205
Resource-efficient programming