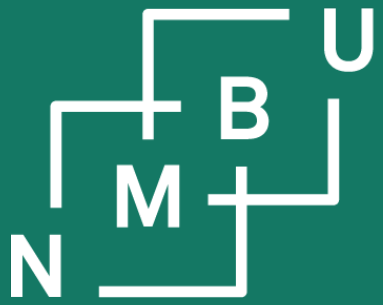
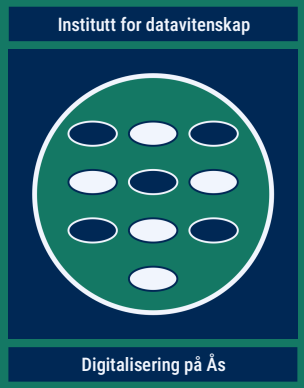


Norges miljø- og biovitenskapelige universitet



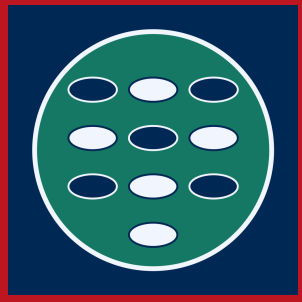
INF205
Resource-efficient programming

2 Data structures

2.7 Tailored containers

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap



Digitalisering på Ås

2 Data structures

2.7 Bespoke containers

INF205

19th October 2022

"Rule of five" or "rule of three"

Container objects take **ownership** (*i.e.*, lifetime and deallocation responsibility) for all or some of the data that they contain. Ownership of data in memory is unique: When the container is deallocated, its owned data are deallocated.

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 five:**" Implement

- **(1)** destructor,
- **(2)** copy constructor,
- **(3)** copy assignment operator,
- **(4)** move constructor,
- **(5)** move assignment operator.

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

"**Rule of three:**"

- **(1)** destructor,
- **(2)** copy constructor,
- **(3)** copy assignment operator.

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](); }
  …
private:
  S* p = nullptr;
  …
}
```

T tobject;

T* tpointer = new T;

If you create a constructor with arguments, also implement the default constructor.

Destructor

General rule: For every "new" there must be a matching "delete".
For containers, this almost always needs to be *at least in the **destructor***.

The destructor **T::~T()** is called when an object of type T is deallocated.

This is the case both for objects on the stack and on the heap:

```
void function_name(…)
{
  // constructor is called
  T tobject;
  …
  // destructor is called
  return;
}
```

```
{
  …
  // constructor is called
  T* tpointer = new T;
  …
  // destructor is called
  delete T;
}
```

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

without this delete[], there would be a **memory leak**!

There might by other properties that do *not* need to be deallocated manually. (*Why?*)

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
    );
  }
  …
}
```

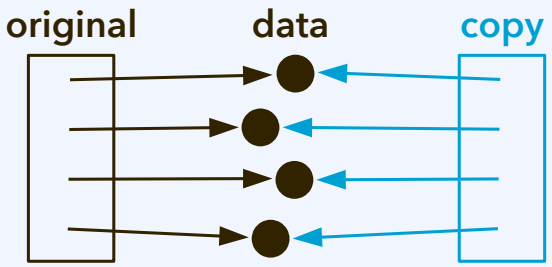
std::copy can be used for data that are contiguous in memory

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

“Deep copy” vs “shallow copy”

Shallow copy:

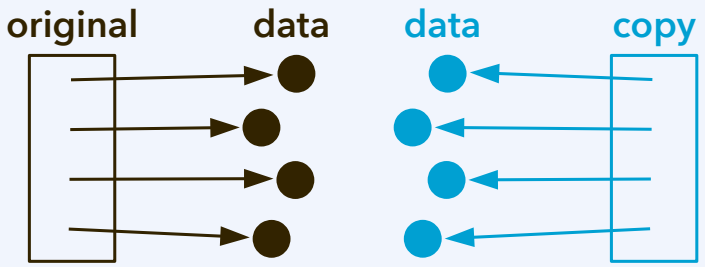
Standard copying, such as if there is no handwritten copy constructor or copy assignment operator, will simply **copy the value of pointers**, *not the content* to which they point.



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.

Copy assignment operator

Norwegian University of Life Sciences

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**.

```
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.

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

Move constructor

The **move constructor** is called when the content of an *old object* can be shifted to a *new object* that is *allocated and initialized* (*e.g.*, *before we deallocate the old object*).

```
T::T(T&& old) { … }
```

```
T func(…){
  T tfirst;
  …
  return tfirst;
  // the destructor will be called
}

int main(){
  // but before, call the move constructor
  T tsecond = std::move( func(…) );
}
```

Typical use case: Efficient **handover of content** returned by a function.

```
class T
{
public:
  T(){ this->p = new S[1000](); }

  T(T&& old){
    this->p = old.p;
    old.p = nullptr;
  }
  …

private:
  S* p …
}
```

A **shallow copy** of the pointer to the content is good enough; after the action, the content *exists in memory only once*!

Attention: Right after the move constructor for "this", the destructor of "old" might be called.

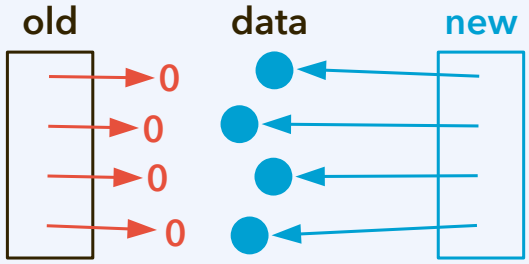
Remove all pointers to the content from old, so that it does not get deallocated!

Move: Why is it advantageous?

Norwegian University of Life Sciences

Move constructor + destructor:

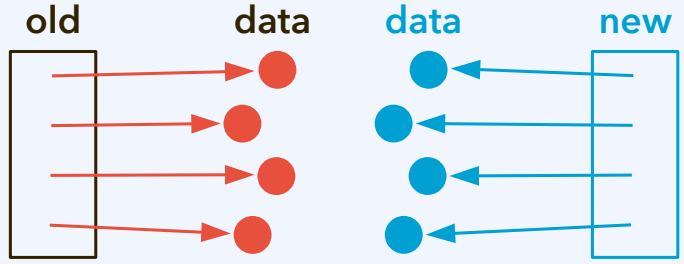
The move constructor is used to *make a new container own the data* without copying the data. A **shallow copy** is made, and the *data are detached* from the old container.



The shallow copy is an inexpensive operation. If the data exist **once in memory** both before the operation and after, *why copy them* from one place to another?

Copy constructor + destructor:

If there is *no move* constructor, or the compiler does not enforce a move, first all the content is copied (**deep copy**); the old container is probably *deallocated right after*.



This is an expensive operation whenever there is a substantial amount of data. All data are *copied*, *unnecessarily*, since at the end they still exist only **once in memory**.

Move assignment operator

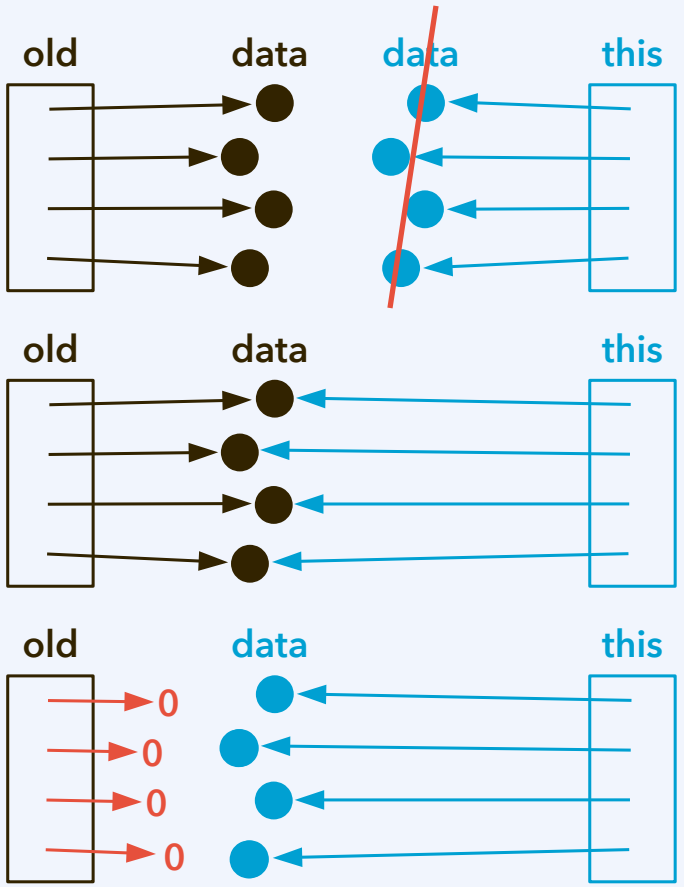
The move assignment operator relates to the move constructor the same way as the copy assignment operator relates to the copy constructor.

```
T& T::operator=(T&& old) { … }
```

```
T func(…) {
  T tfirst;
  …
  return tfirst;
  // the destructor will be called
}

int main() {
  T tsecond;
  …
  // but before, call the move assignment operator
  tsecond = std::move( func(…) );
}
```

constructor called

tsecond exists already



Tutorial week 41 examples

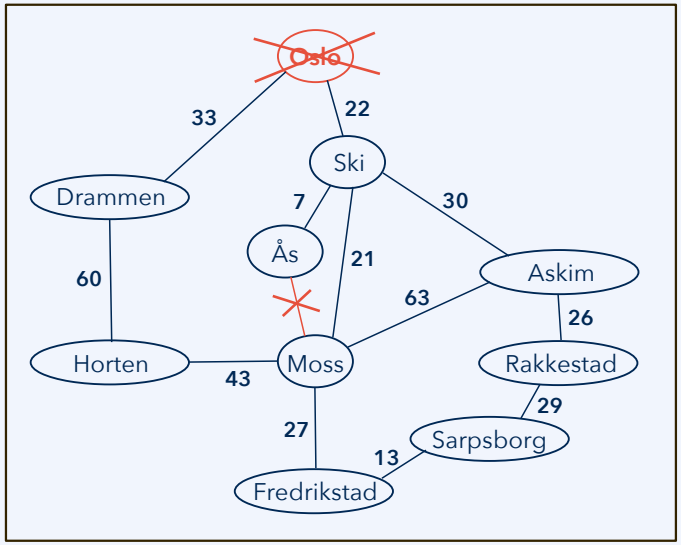
See the **copying-and-moving** code for an implementation and performance comparison for the STL and directly written sequences with int elements.

Below: Copy and move assignment operators for the singly linked list.

```
// copy assignment: clear pre-existing content,
// then make a deep copy of original content

SinglyLinkedList& SinglyLinkedList::operator=(
  const SinglyLinkedList& right_hand_side
){
  if(&right_hand_side == this) return *this;
  this->clear();    // remove pre-existing content

  for(
    auto n = right_hand_side.begin();
    n != nullptr;
    n = n->get_next()
  ) this->push_back(n->get_item());

  return *this;
}
```

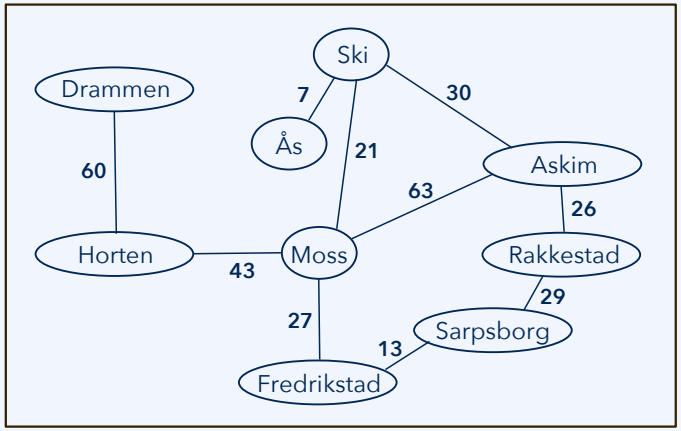
```
// move assignment: clear pre-existing content,
// then shallow-copy pointers to moved content

SinglyLinkedList& SinglyLinkedList::operator=(
  SinglyLinkedList&& old
){
  if(&right_hand_side == this) return *this;
  this->clear();  // remove pre-existing content

  // now proceed as for the move constructor
  this->head = old.head;
  this->tail = old.tail;
  old.head = nullptr;
  old.tail = nullptr;

  return *this;
}
```

Tutorial week 41 examples

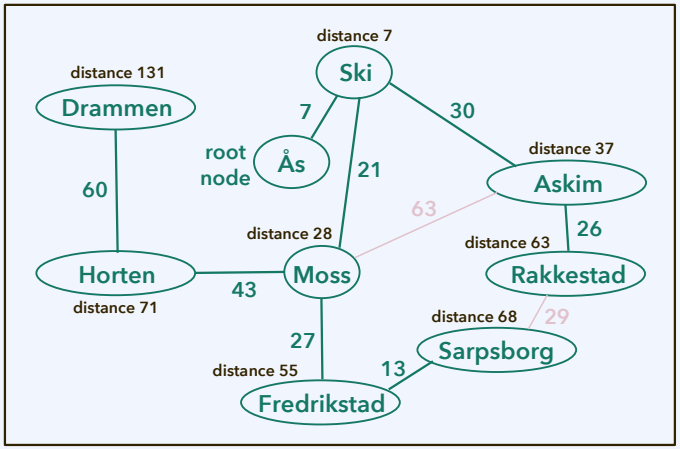


For week 41 problem 3, about deletion of nodes and edges in a graph, see example code **node-edge-deletion**.

This also includes a few other minor improvements of the incidence-list based implementation of an undirected graph.



Dijkstra's algorithm



Modern C++ constructs for ownership

Norwegian University of Life Sciences

By including <**memory**> from the C++ standard library, encapsulated pointer templates (**smart pointers**) can be used that can support secure manual memory management:

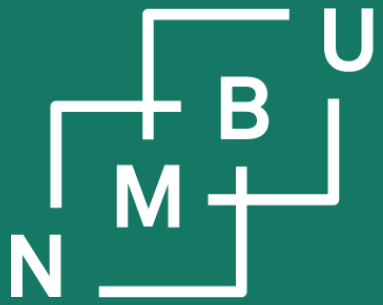
```
std::unique_ptr<T> p = new T();
```

Now the unique_ptr p has taken ownership of the new T object.
When p is deallocated, its destructor ensures that the T object is deallocated.

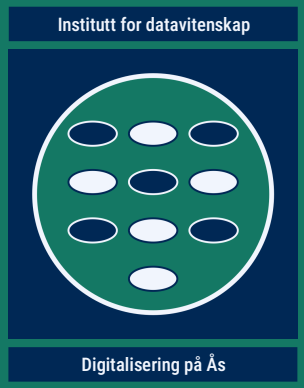
Unique-pointer objects *cannot be copied*, they can only be *moved*!

A smart pointer that can be copied, so that multiple smart pointers can point to the same object and "own it together," is of type **std::shared_ptr**<**T**>. That object is deallocated once all shared pointers to it have been deallocated.

Core Guidelines I.11 recommends against ever passing ownership through "raw" pointers or references. It advises to rely on smart pointers instead.



Norges miljø- og biovitenskapelige universitet



INF205
Resource-efficient programming

2 Data structures

2.7 Tailored containers