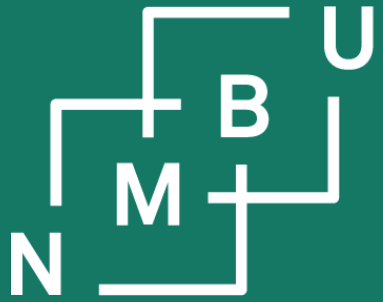
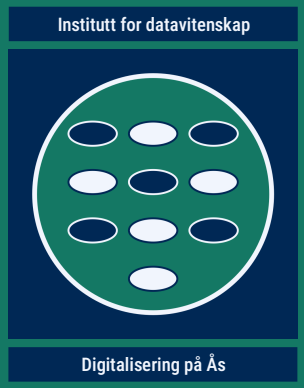


Norges miljø- og biovitenskapelige universitet



INF205
Resource-efficient programming

3 Data structures and libraries

3.1 Co-design data & code
3.2 Shared libraries
3.3 CMake
3.4 Containers
3.5 Linked data
3.6 Graphs and trees

Gauss circle problem

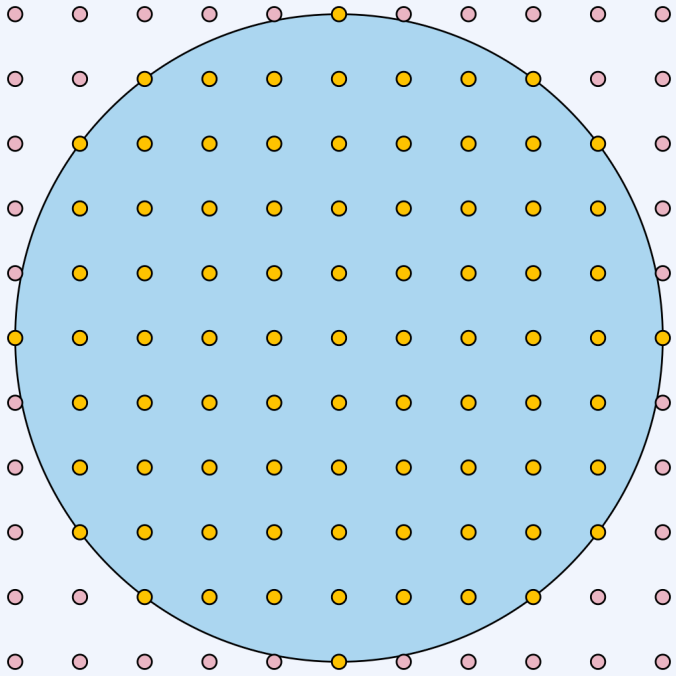
When checking the submitted codes, I also measured their performance (for r = 7,777 and where possible r = 7,777,777).

For equal conditions for all, I used the same **optimization flag** -O2, in all cases, and ran them on the same machine (present laptop).

```
int main(int argc, char** argv)
{
        int r = std::atoi(argv[1]);
        long N = 0;

        for(int x = -r; x <= r; x++)
                for(int y = -r; y <= r; y++)
                        if(x*x + y*y <= r*r) N++;
        std::cout << N << "\n";
}
```

Graded as **8½** points out of 10.



(Diagram under CC0 license, Wikipedia.)

Simple standard solution, did the r = 7,777 case in about 240 ms.

O(r^2) scaling, *i.e.*, quadratic time.

Gauss circle problem

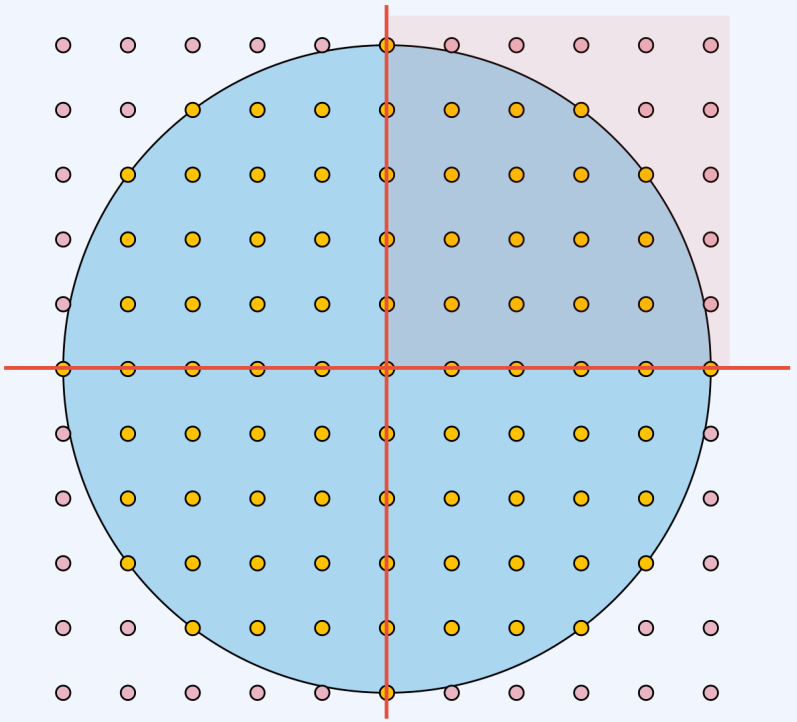
When checking the submitted codes, I also measured their performance (for r = 7,777 and where possible r = 7,777,777).

For equal conditions for all, I used the same **optimization flag** -O2, in all cases, and ran them on the same machine (present laptop).

By exploiting the symmetry and only counting the points in one quadrant, a speedup of about 4 is reached.

Speedup = Slow / Fast code runtime.

This was also a frequent solution and was graded with 9 points out of 10.



(Diagram under CC0 license, Wikipedia.)

This variant of the first solution does r = 7,777 in about 60 ms.

O(r^2) scaling, *i.e.*, quadratic time.

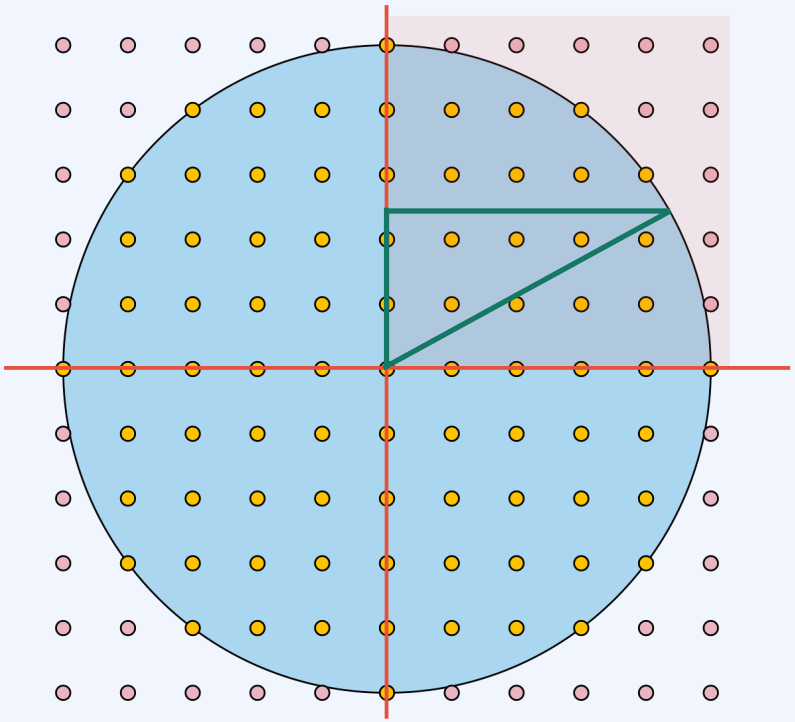
Gauss circle problem

The standard "simple and efficient" solution, which was submitted by several people as well, counts points line by line.

In each line, the number of points is determined from floor(sqrt($r^2 - y^2$)).

```
int main(int argc, char** argv)
{
	long r = std::atoi(argv[1]);

	long long N = 2*r + 1;
	for(long y = 1; y <= r; y++)
	{
		long Nx = std::floor(std::sqrt(r*r - y*y));
		N += 4*Nx + 2;
	}
	std::cout << N << "\n";
}
```



This code solves the problem in linear time, *i.e.*, O(*r*).

About 50 ms for r = 7,777,777.

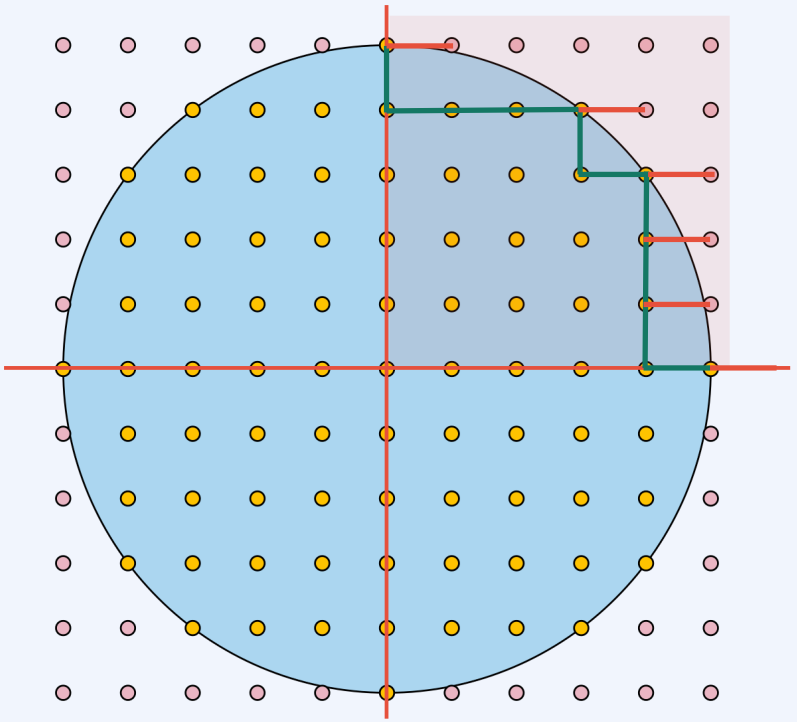
Gauss circle problem

Generally, it is most efficient if we can rely on operations that are directly implemented on hardware, in the arithmetic-logical unit (ALU) of the CPU.

So the code will become even faster if we can avoid computing the square root.

Idea by Marius K. Huseby:

Walk along the contour, only compare the values $x^2 + y^2$ to r^2, no sqrt() call.



This code solves the problem in linear time, *i.e.*, O(*r*).

About 15 ms for r = 7,777,777.

Gauss circle problem: The winning algorithm

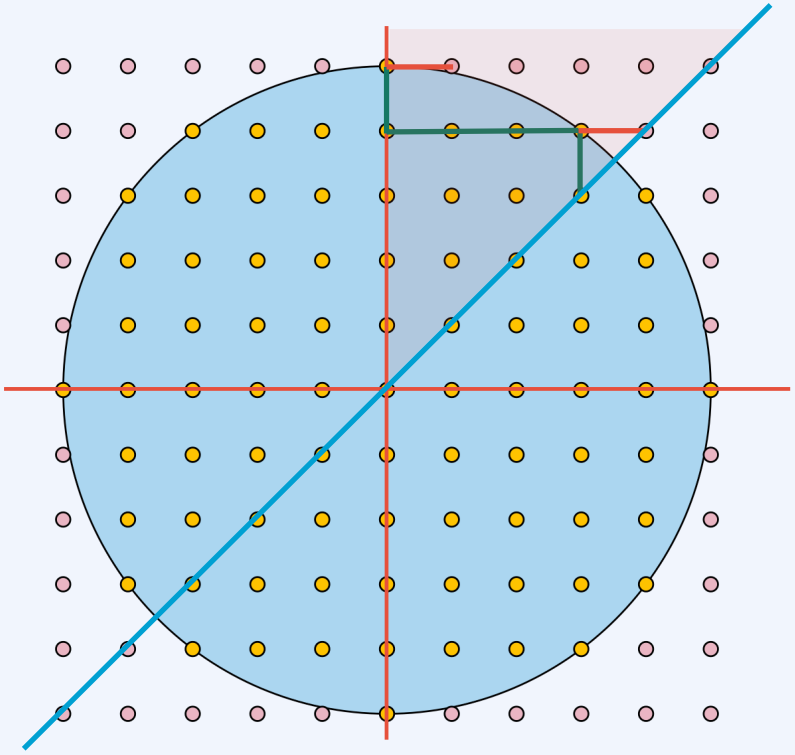
Generally, it is most efficient if we can rely on operations that are directly implemented on hardware, in the arithmetic-logical unit (ALU) of the CPU.

So the code will become even faster if we can avoid computing the square root.

Idea by Marius K. Huseby:

Walk along the contour, only compare the values $x^2 + y^2$ to r^2, no sqrt() call.

Fredrik G. Haugen introduced one more symmetry argument over the diagonal.



This code solves the problem in linear time, *i.e.*, O(*r*).

About 11 ms for r = 7,777,777.

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;
  …
};
```

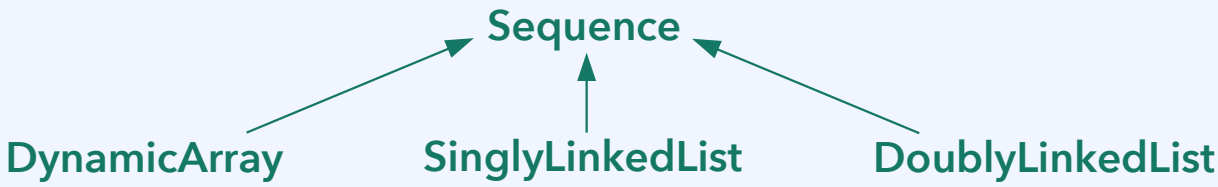
JournalArticleIndex has the property **volume**, but it also **inherits the property year**.

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

Abstract classes, 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**.

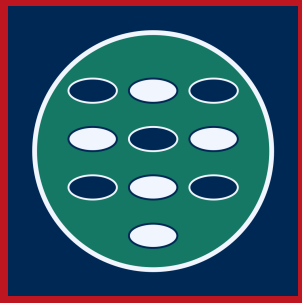
- C.120: Use class hierarchies to represent concepts with inherent hierarchical structure (only).
- C.121: If a base class is used as an interface, make it a pure abstract class.
- C.122: Use abstract classes as interfaces when complete separation of interface and implementation is needed.

Concerning virtual methods and overriding:

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

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap



Digitalisering på Ås

3 Data structures

3.1 Co-design data and code

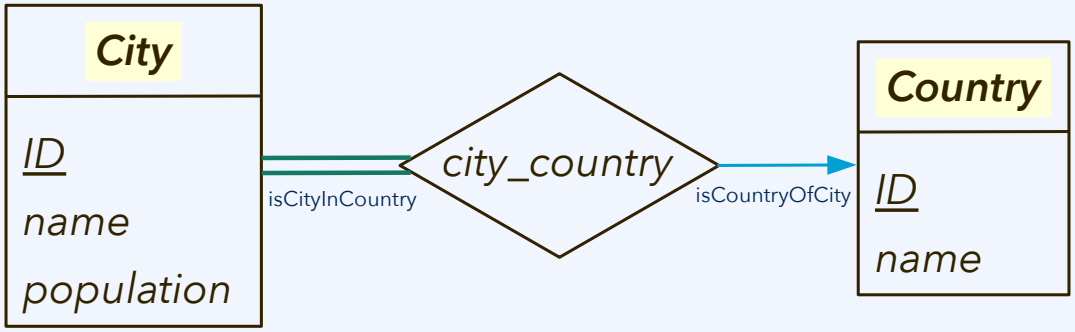
3.2 Shared libraries

INF205

18th February 2025

Designing classes: Entity-relationship diagrams

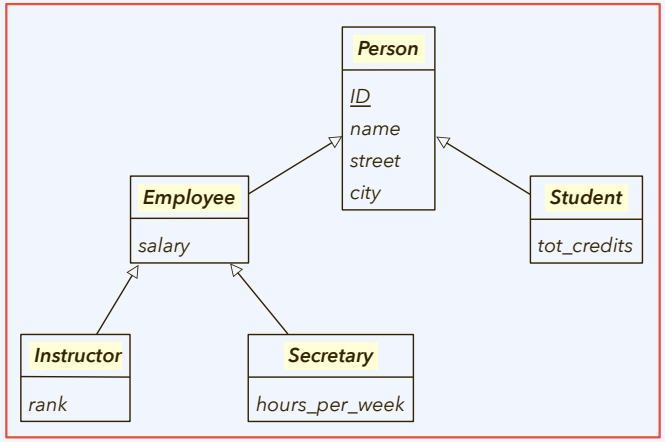
particular:	**object** (entity, individual)	**relationship**	**property** (sometimes: attribute)
universal:	**class** (entity type, concept)	**relation** (relationship type) (in OWL: ObjectProperty)	**attribute** (sometimes: attribute type) (in OWL: DatatypeProperty)



"every City is in such a relationship"

"it is an *N*-to-1 relation from Cities to Countries"

This was also an entity-relationship diagram:



More on entity-relationship diagrams:

- Silberschatz *et al.*, *Database System Concepts*, Chapter 6
- https://en.wikipedia.org/wiki/Entity-relationship_model

Implement relations using non-owning pointers

By storing a pointer to object B as a property of object A, we can encode the relationship between A and B, so that methods from A can access B.

This can go both ways, if needed. Then B also has a pointer to A as a property:

```
class City
{
 public:
  City(string in_name, int in_population, Country* in_country);
  …

 private:
  long ID;
  string name;
  long population;

  Country* country;
};
```

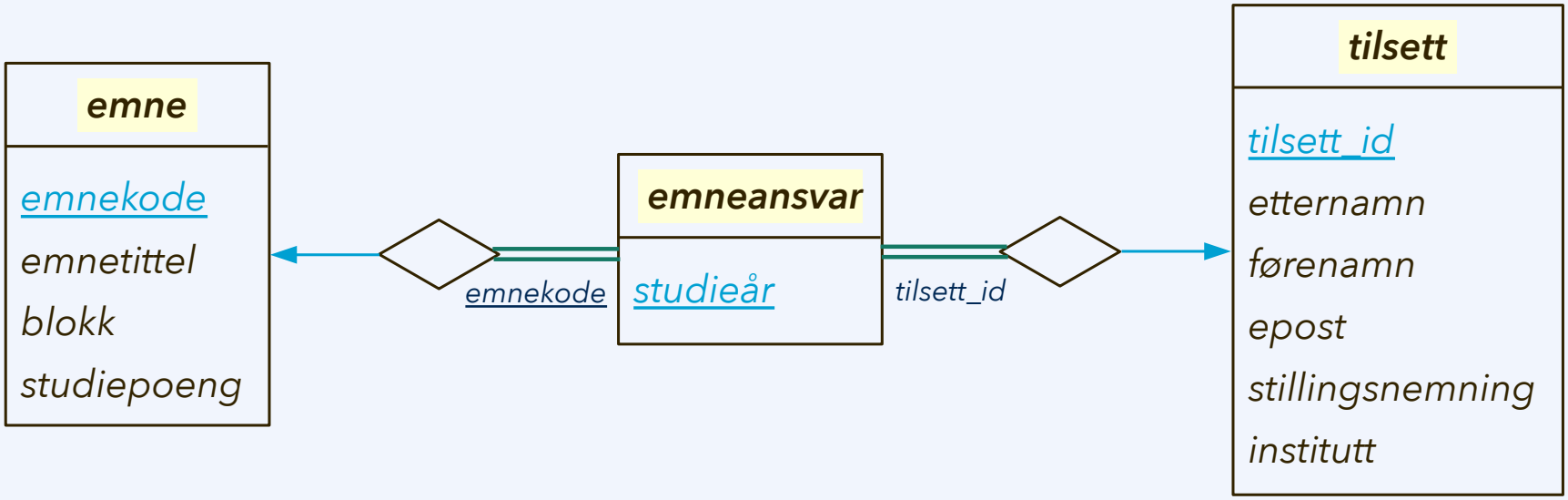
```
class Country
{
 public:
  Country(string in_name);

  void add_city(City* c);
  ...

 private:
  int ID;
  string name;
  vector<City*> cities;
};
```

Example file: city-country.zip

Difference between E-R use in databases and OOP



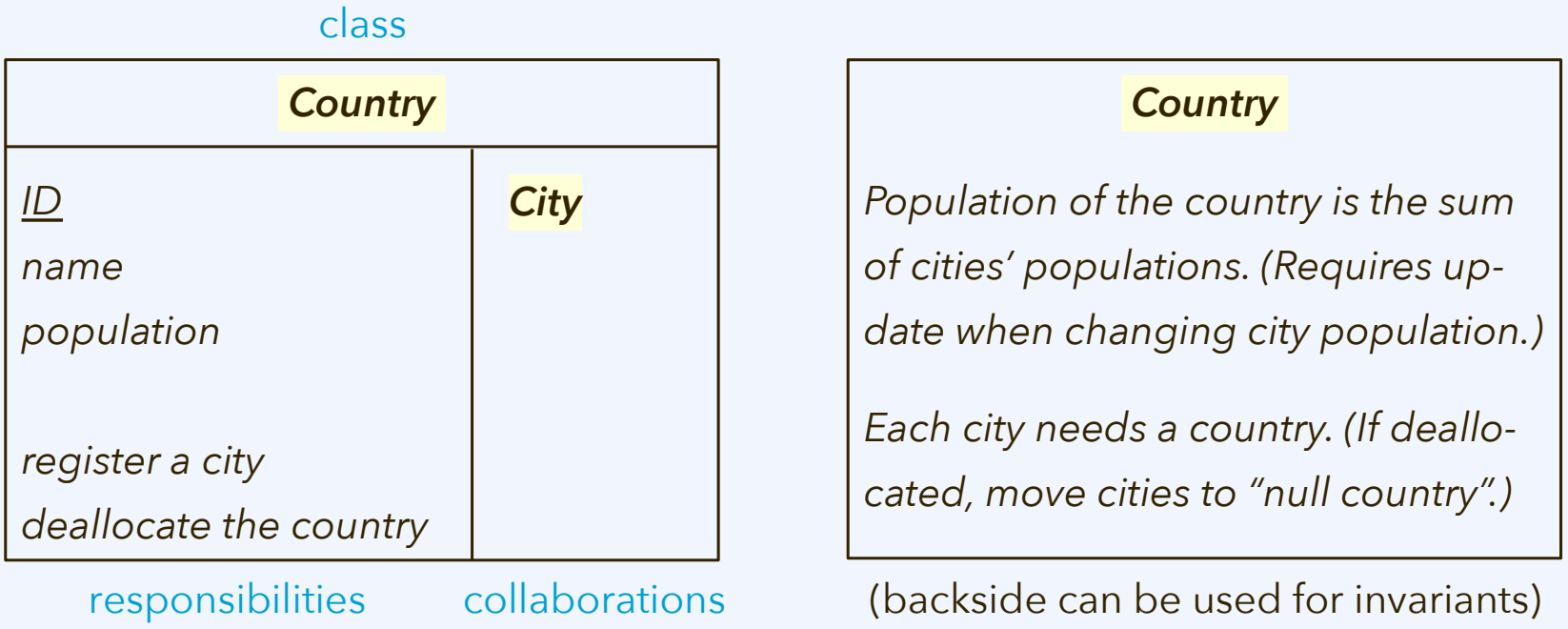
In a *database schema*, objects are identified through *primary keys* (IDs), and their relationship to others are give by *foreign keys*, *i.e.*, other objects' IDs. For example, above, an "emneansvar" entry contains a "tilsett_id" number.

In OOP, we *can* do this in exactly the same way, with a registry of objects, for example using a std::map from the STL (see later). But the standard way of doing this is by instead including a pointer to the object as a property.

CRC cards: Class – responsibilities – collaborations

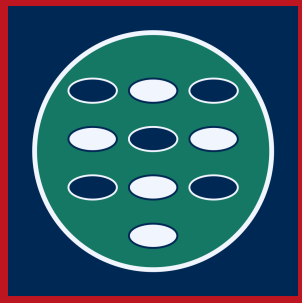
When discussing formal analysis of procedural programming codes, we encountered invariants for a loop. These must hold at every iteration.

Classes in object oriented programming have a much more flexible control flow, as their methods can be called at any time and in any order. It can help to make the invariants explicit during design, *e.g.*, using **CRC cards** as a tool.



Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap



Digitalisering på Ås

3 Data structures

3.1 Co-design data and code

3.2 Shared libraries

INF205

18th February 2025

Standard libraries (and other common libraries)

In programming with C/C++ libraries, we have already seen:

- How to work with the C standard library, *e.g.*, …
- … with the C++ standard library, *e.g.*, …

1. What library includes were we using in today's examples? What for?

   … (discuss)

2. What other C/C++ libraries have you been using? Do you recommend them?

   … (discuss)

Standard libraries (and other common libraries)

In programming with C/C++ libraries, we have already seen:

- How to work with the C standard library, *e.g.*, <cassert>, <cstring>, …
- … with the C++ standard library, *e.g.*, <iostream>, <string>, …

Technically, libraries are pre-compiled object code that can be reused.

The library needs to be accessed at three stages:

- At **compile time**, we need to **include** the **library headers**.
  - The complete source code for the libraries is unnecessary.
  - It is even possible for the library to be coded in another language.
- During **linking**, the object code is **dynamically linked** against the library.
  - At this stage, the library "**static object**" (*.so file) is needed.
  - The executable does not contain the library's object code!
- At **execution time**, the executable and the library are **loaded jointly**.
  - If the library's **static object** code is gone now, the code will not run!

C++ standard template library

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 std::string objects
std::list<std::string> my_list();
```

- **vector<T>** is a **dynamic array** for type **T** elements, similar to Python lists.
- **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**.

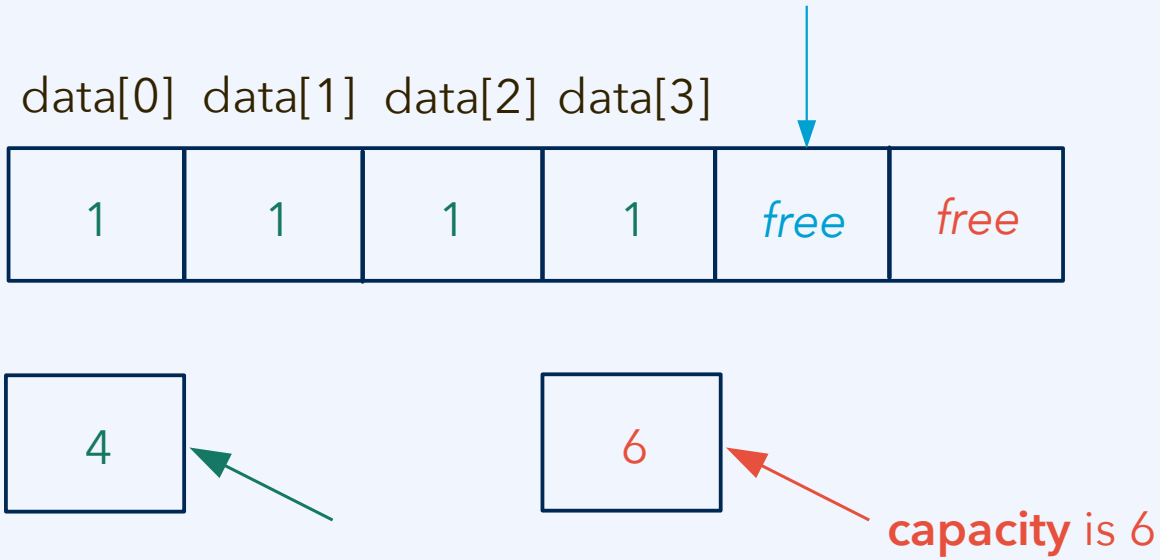
- **set<T>** is a container where each **key** (element) occurs only (at most) once.
- **map<T, V>** contains **key-value** pairs, which each key occurring at most once.
- **multimap<T, V>** contains **key-value** pairs; keys may occur multiple times.

- **array<T, n>** is a **static array** for type **T**, with array size **n**, similar to **T[]** arrays.

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 explicit addressing with "[index]" notation, and many more.

STL, object orientation, containers, and templates

It is **good style** to **use the STL containers** (vectors, lists, *etc.*).
They are implemented to deal with memory safely.

We will look more into them, and how to build such data structures ourselves, once all the required concepts have been introduced.

- **vector**<**T**> is a **dynamic array** for type **T** elements, similar to Python lists.
- **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.
- **multimap**<**T**, **V**> contains **key**-**value** pairs; keys may occur multiple times.

- **array**<**T**, **n**> is a **static array** for type **T**, with array size **n**, similar to **T[]** arrays.

Dynamic (shared) library use: Example

The **convert-to-bmp** code uses the Magick++ API of ImageMagick.[1, 2]

```
#include <Magick++.h>

int main(int argc, char** argv)
{
  // charmap input
  std::ifstream pixistrm(argv[1]);
  diskgraphics::Charmap cm;
  pixistrm >> cm;
  pixistrm.close();

  // image object setup
  Magick::InitializeMagick(*argv);
  Magick::Image img(Magick::Geometry(cm.get_sizex(), cm.get_sizey()), "white");
  img.magick("BMP");
  img.monochrome();
  img.type(Magick::BilevelType);

  // pixel-by-pixel transfer of content
  for(int x = 0; x < cm.get_sizex(); x++)
    for(int y = 0; y < cm.get_sizey(); y++)
      img.pixelColor(x, y, Magick::Color(cm.get_pixel(x, y) == 0? "black": "white"));

  // output in BMP format using one bit per pixel
  img.quantize(2);
  img.write(argv[2]);
}
```

read a "Charmap" object from a pixel graphics file, using an ad-hoc format

ImageMagick can deal with many file formats; we need an uncompressed pixel graphics format such as BMP

copy colour value of pixels

with quantize(2) we get one bit per pixel

[1]Magick++ API documentation: https://imagemagick.org/Magick++/Documentation.html

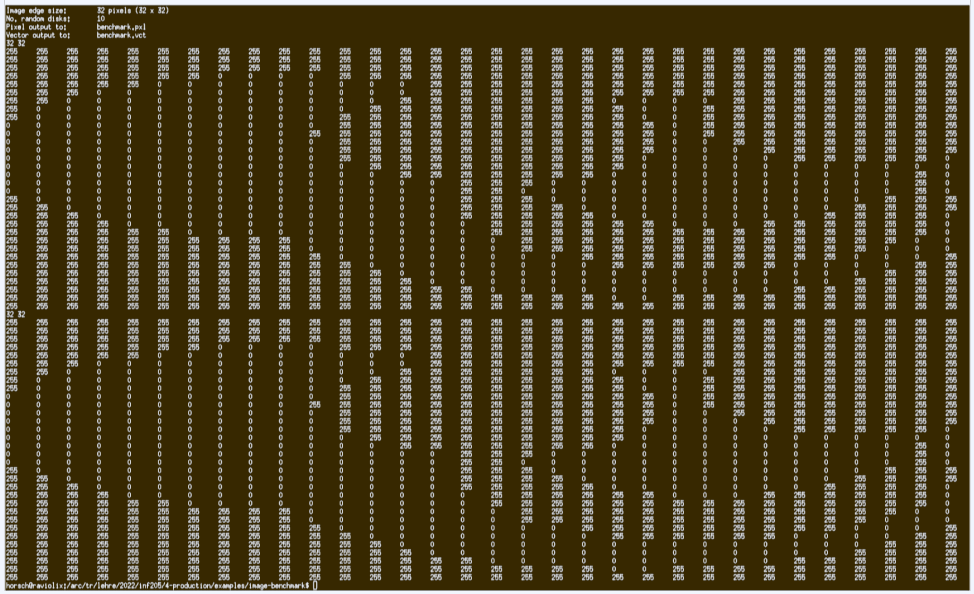
[2]Magick++ Tutorial: https://www.imagemagick.org/Magick++/tutorial/Magick++_tutorial.pdf

Dynamic (shared) library use: Example

The **convert-to-bmp** code uses the Magick++ API of ImageMagick.[1, 2]

```
#include <Magick++.h>
```

How does the compiler know where to look for this file?



[1]Magick++ API documentation: https://imagemagick.org/Magick++/Documentation.html

[2]Magick++ Tutorial: https://www.imagemagick.org/Magick++/tutorial/Magick++_tutorial.pdf

Compiling and linking with libraries

The **convert-to-bmp** code uses the Magick++ API of ImageMagick.[1, 2]

```
#include <Magick++.h>
```

How does the compiler know where to look for this file?

For this particular library, there is a tool that helps call g++ with the right flags:

```
g++ -c -std=c++17 -o <name>.o <name>.cpp `Magick++-config --cppflags`
```

```
-fopenmp -DMAGICKCORE_HDRI_ENABLE=1
-DMAGICKCORE_QUANTUM_DEPTH=16 -I/usr/local/include/ImageMagick-7
```

```
g++ -std=c++17 -o <name> *.o `Magick++-config --libs`
```

```
-L/usr/local/lib -lMagick++-7.Q16HDRI -lMagickWand-7.Q16HDRI -lMagickCore-7.Q16HDRI
```

More typically, you need to provide this information to the compiler by hand.

[1]Magick++ API documentation: https://imagemagick.org/Magick++/Documentation.html

[2]Magick++ Tutorial: https://www.imagemagick.org/Magick++/tutorial/Magick++_tutorial.pdf

Creating a dynamic (shared) library

A **shared object file** can be created from an object file using g++ **-shared**:

```
g++ -c -o first.o first.cpp
g++ -c -o second.o second.cpp
g++ -shared -o libname.so first.o second.o
```

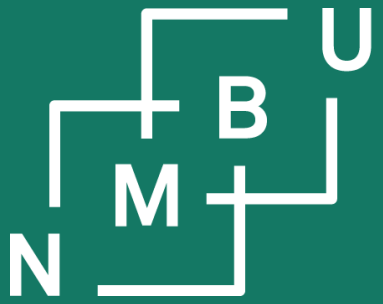
this is a capital i, not a lower-case L

The library header location can be passed to g++ at compile time with -I…, and the shared object is found by the linker with the -L and -l options.

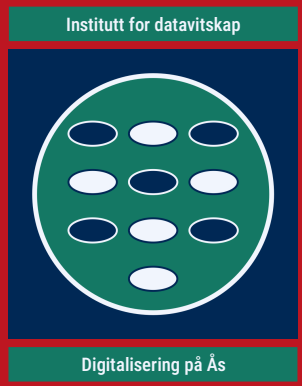
this time it is a lower-case L

But the library also needs to be found at execution time. For that to work, it must be in the appropriate path, or one of the environment variables for library paths must be set to include the location of the shared object.

this can be $LD_LIBRARY_PATH



Noregs miljø- og biovitskaplege universitet



Looking into the implementation of a linked list

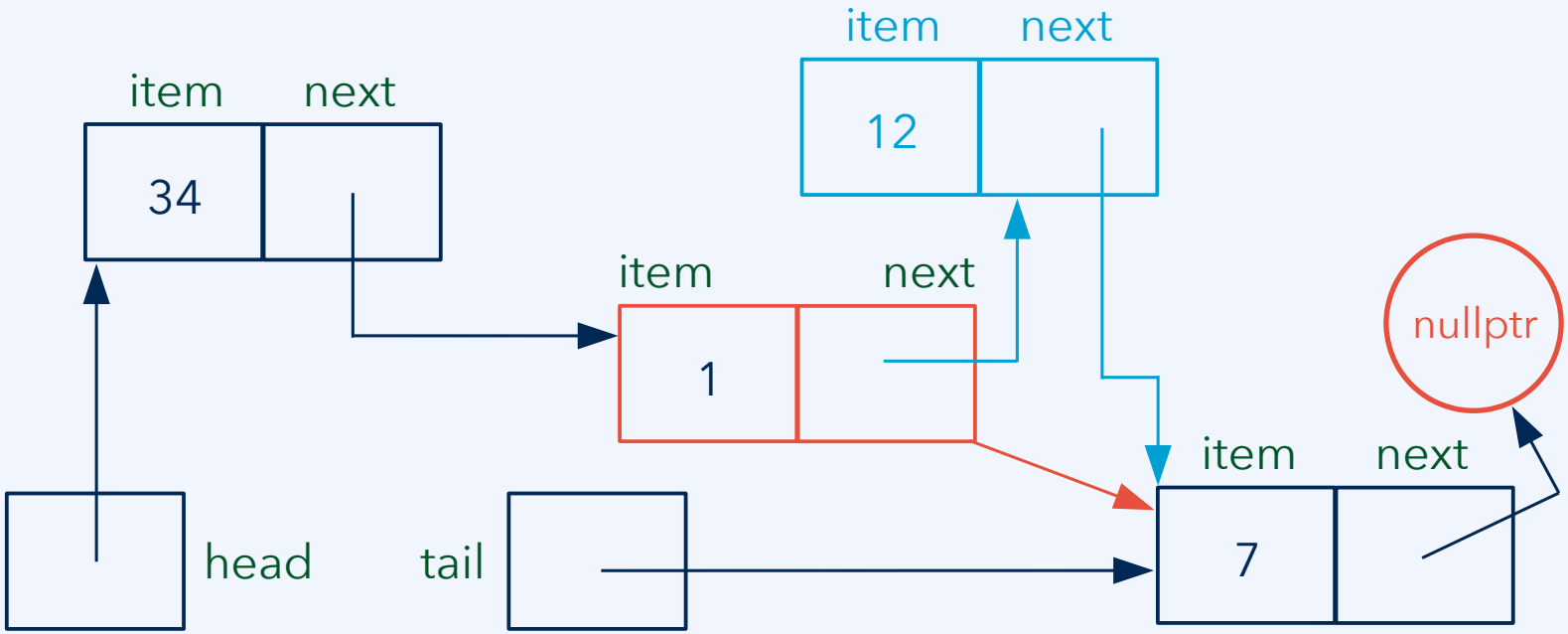
INF205

18th February 2025

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.



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**:

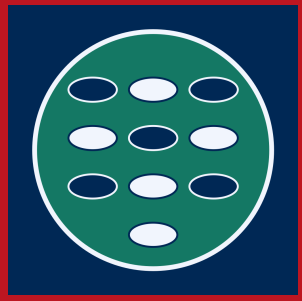
- bool empty() const;
- size_t size() const;
- int& front();
- int& back();
- int& at(int i);
- void push_front(const int& in);
- void push_back(const int& in);
- void push(const int& in);
- void insert_at(int idx, const int& in);
- void pop_front();
- void pop_back();
- void pop(const int& in);
- void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

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap



Digitalisering på Ås

Remark on templates

INF205

18th February 2025

Templates: Parameterized classes

We have already seen the STL templates: The **same container implementation** can be used for **different types of contained objects**, such as **list<float>** and **list<double>**. We can define our own class templates in this way:

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

private:
  T item;
  SinglyLinkedListNode<T>* next = nullptr;
  void set_next(SinglyLinkedListNode<T>* in_next) { this->next = in_next; }
};
```

attention with split between header and object file; think about "what the compiler will do"

attention with initializations

While there is only one **source code** for each template, **object code** is normally generated separately for each concrete version of it. (But not for the template!)

Example file: list-template.zip

Templates for functions and methods

Norwegian University of Life Sciences

The same sort of syntax applies for parameterized function and method declarations and definitions. This includes cases with multiple parameters.

```
template<typename T>
  void SinglyLinkedList<T>::push_front(
    const T& pushed_item
){
  SinglyLinkedListNode<T>* new_node
              = new SinglyLinkedListNode<T>;
  new_node->set_item(pushed_item);

  if(this->empty()) this->tail = new_node;
  else new_node->set_next(this->head);
  this->head = new_node;
}
```

```
template<typename SeqnT, typename ElmnT>
  void test_sequence(
    SeqnT* sqn, int n, int m,
    ElmnT a, ElmnT b, ostream* os
){
  …
}

template<typename SeqnT, typename ElmnT>
  float test_with_time_measurement(
    SeqnT* sqn, int iterations, ElmnT a, ElmnT b
){
    int sequence_length = 1000001;
    int deletions = 10;
    test_sequence(sqn, 100000, 10, a, b, &cout);
}
```


Example file: list-template.zip

Templates: Case distinctions

The standard library header <**type_traits**> includes parameterized flags that can be used to make case distinctions, *e.g.*,

- **is_arithmetic**<T>::value, **is_signed**<T>::value, *etc.*;
- **is_pointer**<T>::value, **is_class**<T>::value, **is_array**<T>::value, *etc.*;
- **is_same**<T, S>::value, to check whether T and S are the same type.

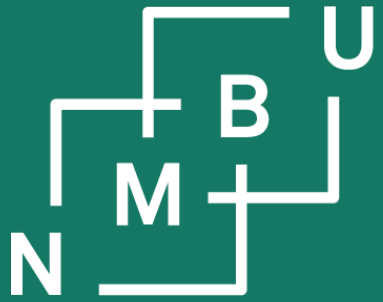
In **list-template**, solutions for initializing the property "**T** item" would include:

```
T item = T();
```

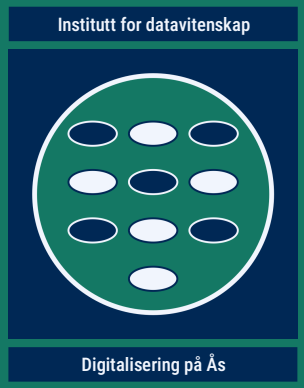
```
template<typename T>
    const T initial_value = T();
…
T item = initial_value<T>;
```

```
SinglyLinkedListNode<T>() {
  if constexpr(is_arithmetic<T>::value) this->item = 3;
  else if constexpr(is_pointer<T>::value) this->item == nullptr;
  else if constexpr(is_same<T, string>::value) this->item = "uninitialized";
  else this->item = T();
}
```

Only with the solution on the right we can make more high-level design distinctions depending on the nature of the type T used for parameterizing.



Norges miljø- og biovitenskapelige universitet



INF205
Resource-efficient programming

3 Data structures and libraries

3.1 Co-design data & code
3.2 Shared libraries
3.3 CMake
3.4 Containers
3.5 Linked data
3.6 Graphs and trees