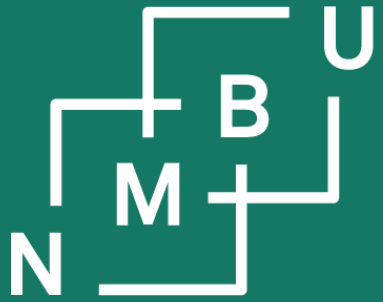
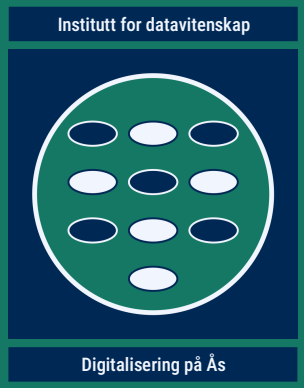


Norges miljø- og biovitenskapelige universitet



INF205
Resource-efficient programming

2 Memory and objects

Common mistakes

Repetition: We talked about three main kinds of mistakes in manual memory management. What were they? (1) … … (2) … … (3) … …

Strategies that we had identified for trying to avoid these mistakes:

1) **Avoid manual memory management** if possible; use the stack, not the heap.
   - Don't work with pointers unless there is a clear advantage. Don't pass by reference without a good reason.

2) **Assign clear responsibilities** for what part of the code is to allocate each data item that you create on the heap.
   - Create a "container" that "owns" it. If suitable, library (*e.g.* STL) containers.

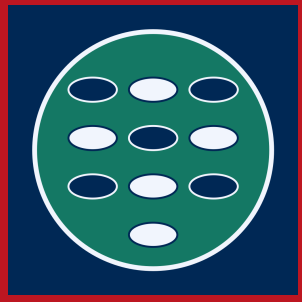
Smart pointers are very elementary containers. They have ownership over the object to which they point.

Instead of T*, where T is the type, we can use std::unique_pointer<T> or std::shared_pointer<T>.

See Core Guidelines I.11.

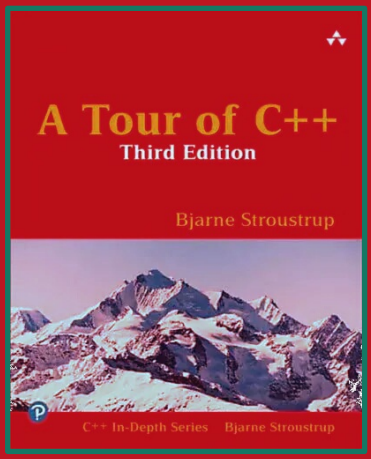
Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap



Digitalisering på Ås

2 Memory and objects



book Section 1.6

2.4 Constants & immutability

2.5 Streams and file I/O

2.6 Class hierarchies

INF205

14th February 2025

The keywords auto, const, constexpr

auto: Leave it to the compiler to determine the type

This requires an initialization.

Remark:

typeid(x).name() can be used to output the type assigned to x.

```
for (auto i = 0; i < 26; i++)
{
  auto c = 'a';
  c += i;
  cout << c;
}
```

should become **int**

should become **char**

const: Used to declare an immutable variable

constexpr: Immutable and, additionally, can be evaluated at compile time

```
constexpr int space_dimension = 3;  int n = 0;  cin >> n;  const int num_coords = n*space_dimension;
```

Con.1: By default, make objects immutable

"make objects non-const only when there is a need to change their value"

Con.4: Use **const** to define objects with values that do not change

Con.5: Use **constexpr** for values that can be computed at compile time

“const” parameters of a function

const-array-broken.cpp – does not compile

If we pass an argument by reference but do not intend to modify it, the parameter should be declared as **const**. Such as:

```
void do_something(const int N);
void do_something(const int* const x);
void do_something(const int x[]);
```

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

1. What is the code supposed to do?
2. Why does it not compile?
3. What should be changed?
4. What more const/-expr can we add?

```
int second_of(int N, int* x) {
        int largest = x[0];
        int second_largest
                = std::numeric_limits<int>::min();

        for(int i = 1; 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() {
   int fixed_array_size = 5;
   const int x[fixed_array_size] = {4, 0, 6, 5, 2};
   int t = second_of(fixed_array_size, x);
}
```

Making proper use of const and constexpr

const-array-fixed.cpp – fixed as follows:

If x[] in main() is a const int array, or even "constexpr" (which is stronger than "const"), we must make the x parameter in second_of() const int*.

1. Variables are declared as const if we do not plan to modify them after initialization. If their value can be determined at compile time, we can even use "constexpr".

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

3. Pointers that are constant (*i.e.*, have as value an address that cannot be changed) are of the type T* const. This can can be combined with the above: const T* const.

```
int second_of(const int N, const int* const x) {
        int largest = x[0];
        int second_largest
                = std::numeric_limits<int>::min();

        for(int i = 1; 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;
  constexpr int x[fixed_array_size] = {4, 0, 6, 5, 2};
  const int t = second_of(fixed_array_size, x);
}
```

“const”, pass by reference, and const pointers

1) If you can pass by value, that is always to be preferred!
2) If you pass an argument by reference, the compiler assumes that the function will modify it. Write “const” whenever that’s not the case.

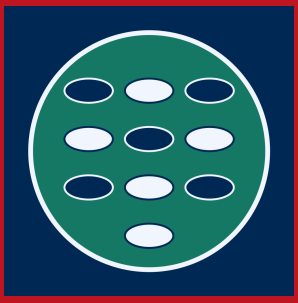
An array is a pointer. Therefore **it is impossible to pass an array by value**. If you don’t intend the function to write to the array, it should be a const parameter.

Pay attention to C++ syntax for combining pointers with “const”. Illustration:

```
int v = 3;
const int x[3] = {1, v, v*v};      // x is an array of constant integers
const int* y = &x[1];              // y is a pointer to a constant integer
int* const pv = &v;                // pv will forever point to address of v
const int* const z = &x[2];        // z will forever point to address of x[2]

(*pv)++;                           // this is legal, we may change *pv, just not pv
y++;                               // this is legal, we may change y, just not *y
```

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap



Digitalisering på Ås

2 Memory and objects

2.4 Constants & immutability

2.5 Streams and file I/O

2.6 Class hierarchies

INF205

14th February 2025

I/O operator overloading

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

Assume that for some **class C**, we have 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.*

Advice: Input & output methods/operators should use the same serialization.

File input/output

We **must serialize the data** in order to store them in a file!

To transfer data through a communication channel as a message, the data items and their parts need to be serialized (ordered) in a well-defined way that is understood both by the sender and the receiver.

- As a contiguous chunk of memory, *if the exchange is memory-based*.
- As a file, *if file I/O is the mechanism* by which data are exchanged.

File stream objects can be used in order to read or write a file.

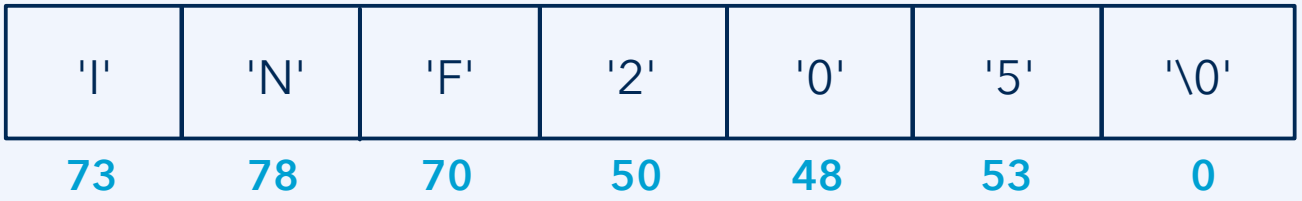
```
// open in-filestream
std::ifstream infile(argv[1]);      // file name given as command-line argument argv[1]
```

Remark: Strings in C and 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:



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';**

Remark: Strings in C and C++

C++ strings may be the same as arrays at the memory level, but they are not arrays to the language. Therefore, **it is possible to pass C++ strings by value**.

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

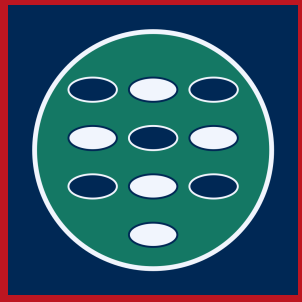
```
void increment_at(int p, char* str)
{
  str[p]++;
}

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

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

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

Example file: string-argument-passing.cpp

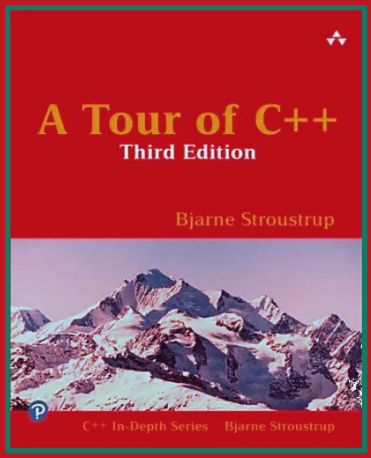
Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap



Digitalisering på Ås

2 Memory and objects



Sections 5.3 – 5.5

2.4 Constants & immutability

2.5 Streams and file I/O

2.6 Class hierarchies

Core Guidelines:

C.120, C.121
C.146 – C.148

(and more in "C")

INF205

14th February 2025

Why object orientation?

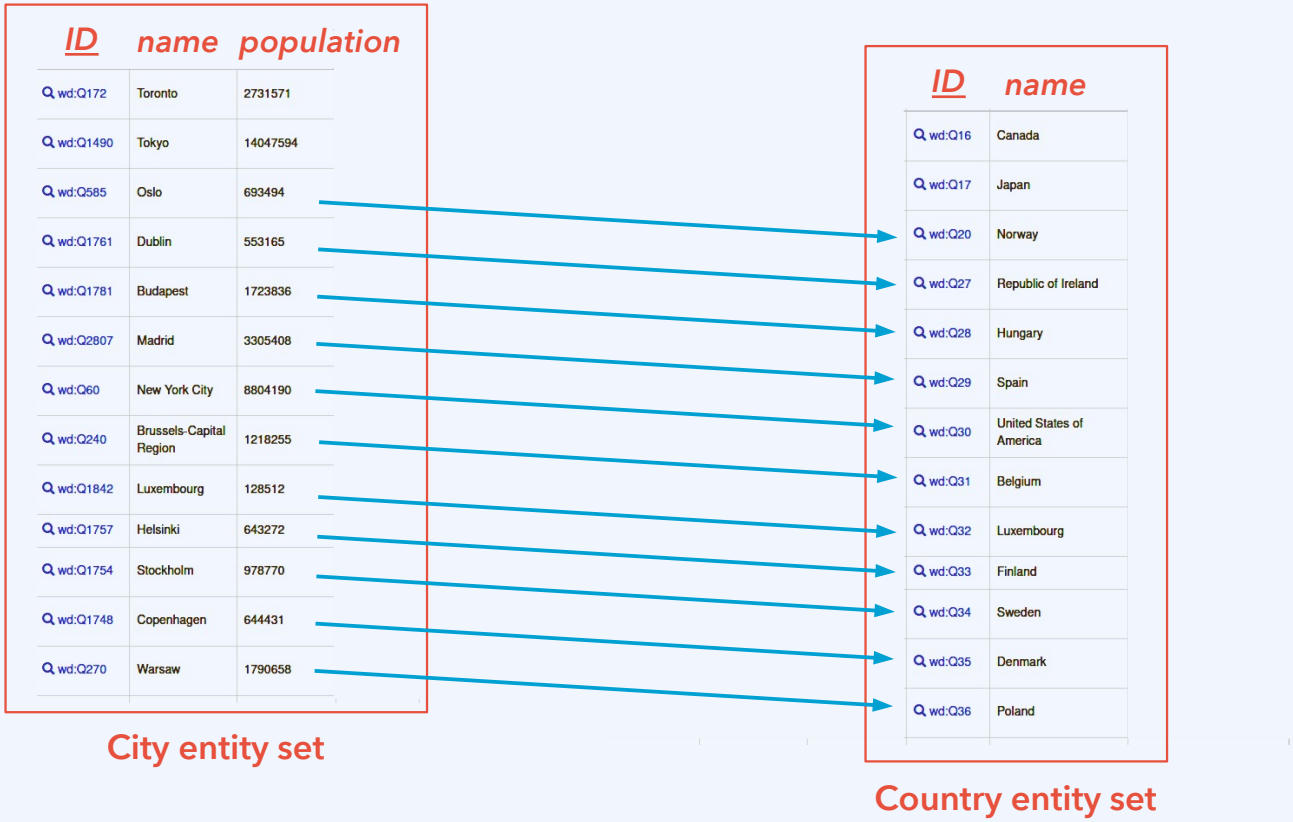
The job of variables is to store data. In object oriented programming (OOP) the focus is on *how data belong together* and how we can facilitate *safe and correct access to data*. How do data-centered tools (DBs, *etc.*) present data?

Example: "Largest cities by country" query on Wikidata.

city	cityLabel	population	country	countryLabel	loc
wd:Q172	Toronto	2731571	wd:Q16	Canada	Point(-79.386666666 43.670277777)
wd:Q1490	Tokyo	14047594	wd:Q17	Japan	Point(139.691722222 35.689555555)
wd:Q585	Oslo	693494	wd:Q20	Norway	Point(10.738888888 59.913333333)
wd:Q1761	Dublin	553165	wd:Q27	Republic of Ireland	Point(-6.260277777 53.349722222)
wd:Q1781	Budapest	1723836	wd:Q28	Hungary	Point(19.040833333 47.498333333)
wd:Q2807	Madrid	3305408	wd:Q29	Spain	Point(-3.7025 40.416666666)
wd:Q60	New York City	8804190	wd:Q30	United States of America	Point(-74.0 40.7)
wd:Q240	Brussels-Capital Region	1218255	wd:Q31	Belgium	Point(4.3525 50.846666666)
wd:Q1842	Luxembourg	128512	wd:Q32	Luxembourg	Point(6.132777777 49.610555555)
wd:Q1757	Helsinki	643272	wd:Q33	Finland	Point(24.93417 60.17556)
wd:Q1754	Stockholm	978770	wd:Q34	Sweden	Point(18.068611111 59.329444444)
wd:Q1748	Copenhagen	644431	wd:Q35	Denmark	Point(12.568888888 55.676111111)
wd:Q270	Warsaw	1790658	wd:Q36	Poland	Point(21.011111111 52.23)

Entity-relationship (E-R) diagrams

particular: object entity individual — relationship — property (sometimes: attribute)



ID	name	population
wd:Q172	Toronto	2731571
wd:Q1490	Tokyo	14047594
wd:Q585	Oslo	693494
wd:Q1761	Dublin	553165
wd:Q1781	Budapest	1723836
wd:Q2807	Madrid	3305408
wd:Q60	New York City	8804190
wd:Q240	Brussels-Capital Region	1218255
wd:Q1842	Luxembourg	128512
wd:Q1757	Helsinki	643272
wd:Q1754	Stockholm	978770
wd:Q1748	Copenhagen	644431
wd:Q270	Warsaw	1790658

City entity set

ID	name
wd:Q16	Canada
wd:Q17	Japan
wd:Q20	Norway
wd:Q27	Republic of Ireland
wd:Q28	Hungary
wd:Q29	Spain
wd:Q30	United States of America
wd:Q31	Belgium
wd:Q32	Luxembourg
wd:Q33	Finland
wd:Q34	Sweden
wd:Q35	Denmark
wd:Q36	Poland

Country entity set

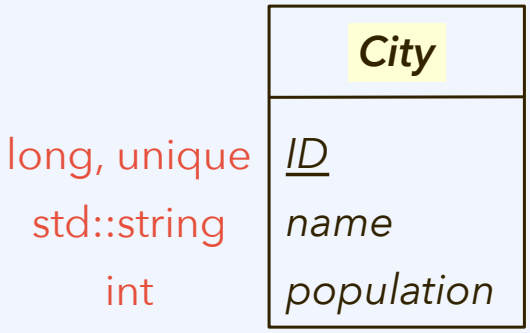
Designing classes: Entity-relationship diagrams

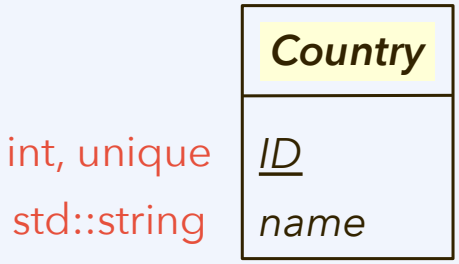
particular: **object** (entity, individual)

universal: **class** (entity type, concept)

(sometimes: attribute)
property

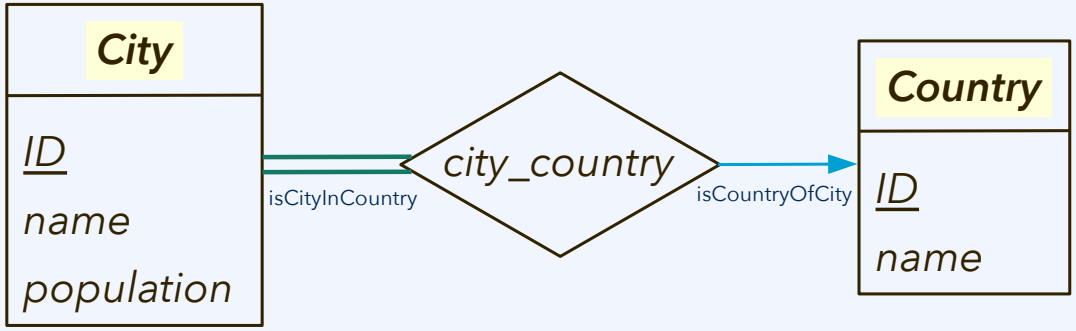
(sometimes: attribute type)
attribute
(in OWL: DatatypeProperty)





Designing classes: Entity-relationship diagrams

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



"every City is in such a relationship"

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

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:

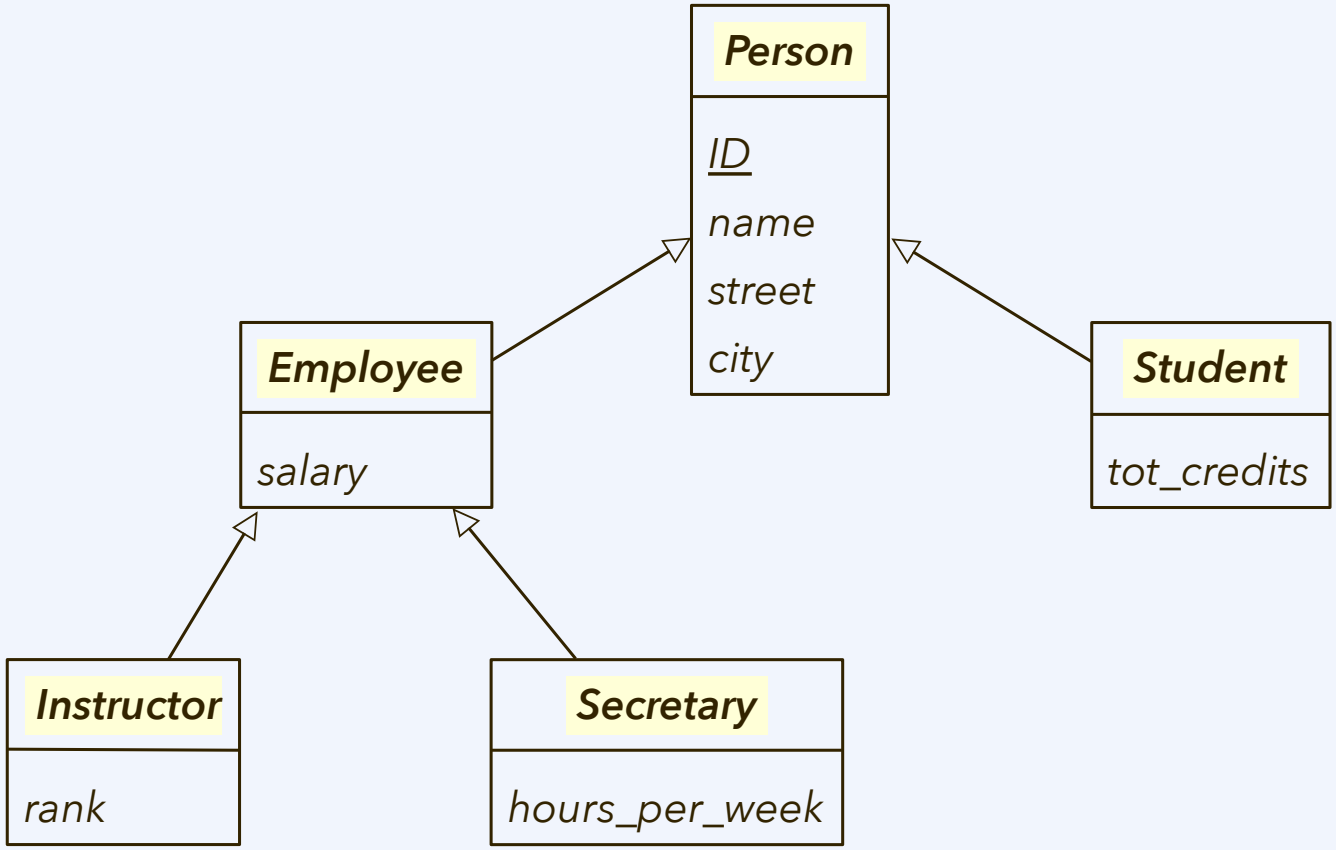
```
class City
{
 public:
  City(string in_name, int in_population, Country* in_country);
  …

 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

E-R notation: Taxonomy/class hierarchy

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

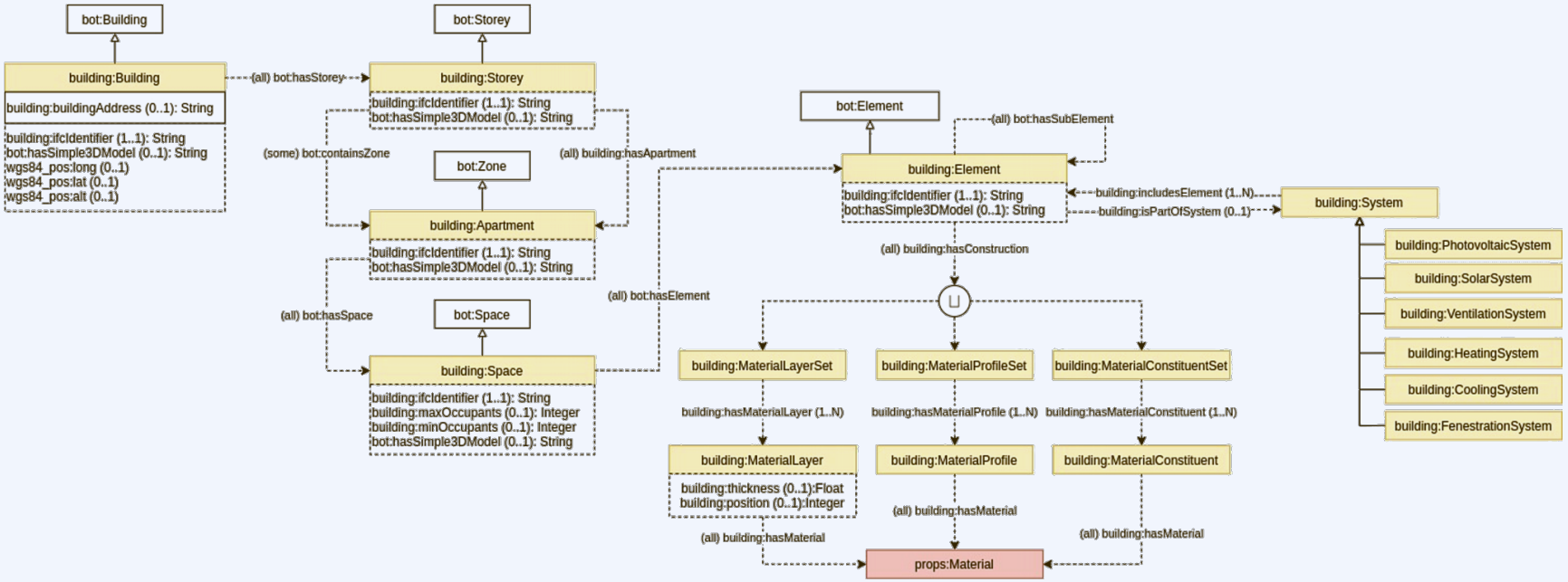


[1]A. Silberschatz, H. F. Korth, S. Sudarshan, *Database System Concepts*, 7th int. stud. edn., McGraw-Hill, **2019**.

E-R diagrams on draw.io and Chowlk[1, 2]

The draw.io tool can be used for E-R diagrams using a variety of conventions.

With Chowlk by Poveda Villalón *et al.*,[1, 2] these can be converted to ontologies.



[1]M. Poveda Villalón *et al.*, in *Proc. VOILA23*, *CEUR Works. Proc.* **3508**: 2 (link to paper), **2023**.

[2]Chowlk template: https://chowlk.linkeddata.es/static/resources/chowlk-library-complete.xml
Lightweight version: https://chowlk.linkeddata.es/static/resources/chowlk-library-lightweight.xml

Taxonomy (class hierarchy)

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 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;
  …
};
```

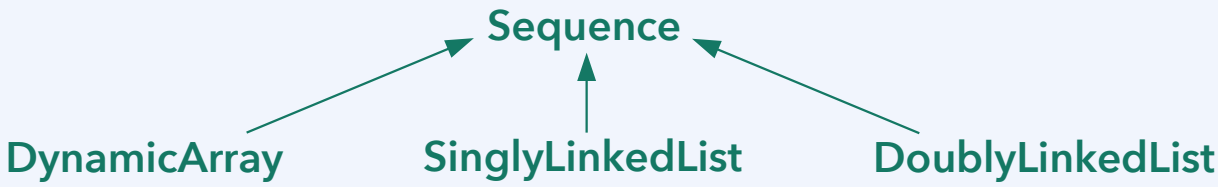
JournalArticleIndex has the property **volume**, but it also **inherits the property year**.

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.

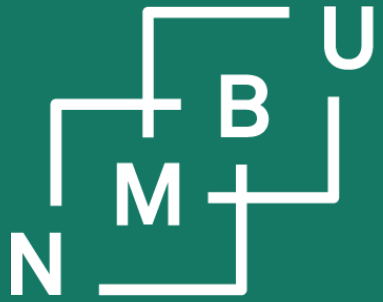
Core guidelines

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**.

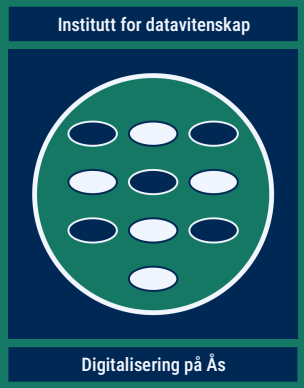
- C.120: Use class hierarchies to represent concepts with inherent hierarchical structure (only).
- C.121: If a base class is used as an interface, make it a pure abstract class.
- C.122: Use abstract classes as interfaces when complete separation of interface and implementation is needed.

Concerning virtual methods and overriding:

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



Norges miljø- og biovitenskapelige universitet



INF205
Resource-efficient programming

2 Memory and objects