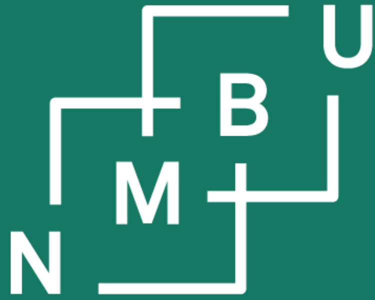
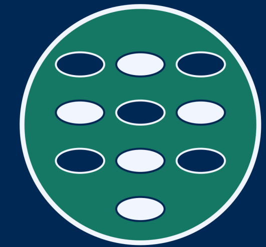


Norges miljø- og
biovitenskapelige
universitet



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

26th February 2024

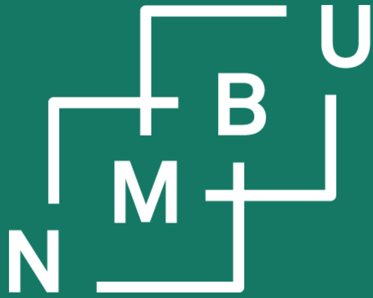
Weekly glossary concepts

Norwegian University of Life Sciences

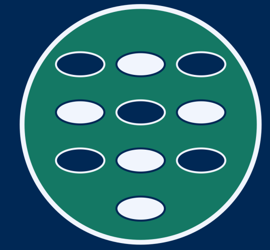
What are essential concepts from the previous lecture?

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



Recapitulation and looking into the second worksheet

On Fagpersonweb, there are now 35 registered course participants.
Out of these registered participants, 29 submitted the first worksheet.
Out of these submissions, 26 passed the first worksheet, 3 did not.

INF205

26th February 2024

Mandatory activities

- There will be five **lab worksheets**, of which you are **required to pass at least three**. A worksheet is passed if the majority of its problems have been solved (more or less) correctly.
- Collaboration between two people is allowed; then, submit on Canvas twice. Write explicitly when there is a collaboration or joint submission.
- **Solutions to the problems** are presented by students in the tutorial (data lab) sessions. **Everybody needs to present once.** Other than this, attendance at the tutorial (data lab) or lecture is in no way mandatory.
- **Programming projects** are also **presented**, individually or as a group.

On Fagpersonweb, there are now 35 registered course participants.
Out of these registered participants, 29 submitted the first worksheet.
Out of these submissions, 26 passed the first worksheet, 3 did not.

Allocate: new. Deallocate: delete.

Allocation: Reserve memory to store data.
Deallocation: Release the memory.

On the stack

The stack is already handled completely and safely by the compiler. **Memory on the stack** (local variables of functions) is **allocated** as part of a **stack frame** **when the function is called**. It is **deallocated** again **when the function returns**.

On the heap

Memory on the heap is managed independent of the stack, at runtime, subject to **explicit allocation and deallocation** instructions that must come from the programmer. There is no garbage collection in C++!

- **Allocation** is done with **new**. Example: **int* i = new int;**
- **Deallocation** is done with **delete**. Example: **delete i;**

Manual memory management: 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.

Example: Fixing a memory leak

The code in is-prime-memory-leak.zip is an extreme example of poor coding. As a consequence, it will cause a memory leak.

```
bool is_prime(long* n) {
  if((*n%2 == 0) || (*n%3 == 0)) {
    delete n;  return false;
  }

  for(long i = 5; *n >= i*i; i += 6) {
    if((*n % i == 0) || (*n % (i+2) == 0)) {
      delete n;  return false;
    }
  }

  return true;
}
```

```
double time_measurement(long n) {
  auto t0 = high_resolution_clock::now();
  for(int i = 0; i < num_tests; i++) {
    is_prime(new long{n});
  }
  auto t1 = high_resolution_clock::now();

  return duration_cast<nanoseconds>(t1-t0).count()
    / (double)num_tests;
}

int main() {
  for(long x = xmin; xmax >= x; x += xstep)
    cout << x << "\t" << time_measurement(x) << "\n";
}
```

We can try both strategies, (a) clear responsibility for deallocating the item on the heap, and (b) avoiding the heap. Which of the two makes more sense?

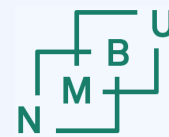
Second worksheet: Questions and discussion

```
int main(int argc, char** argv) {
    [...]
    // configuration: an array of float numbers
    float* present_configuration
        = new float[size]();
    [...]

    for(long i = 0; i < steps; i++) {
        [...]
        // do the random walk step
        present_configuration
            = crw::step(size,
                present_configuration);
        float present_elongation
            = crw::elongation(size,
                present_configuration);
        [...]
    }
    delete[] present_configuration;
}
```

```
float* crw::step(long size, float previous[])
{
    // allocate the next configuration
    float* config = new float[size]();

    // first, let the chain contract:
    // each element is attracted
    // by its neighbours
    for(long i = 0; i < size; i++)
        config[i] = 0.5*previous[i]
            + 0.25*previous[(i-1) % size]
            + 0.25*previous[(i+1) % size];

    // actual random walk step
    stochastic_unit_step(size, config);

    // shift such that the average is zero
    shift_centre_to_origin(size, config);

    return config;
}
```

Example file: memleak.zip

Structure of the course



Norwegian University of Life Sciences

1) Introduction (week 6)

- Getting started – the lecture last week.

2) The C/C++ programming language(s) (weeks 7 and 8)

- Essential features that make C/C++ different from Python; *e.g.*, dealing with memory allocation and deallocation explicitly, using pointers.

3) Data structures (weeks 9 to 11)

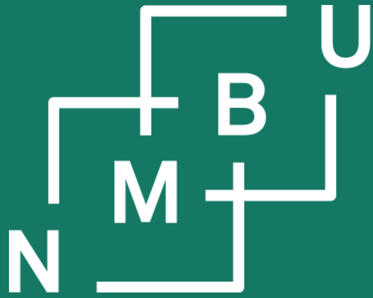
- Linked data structures, containers, C++ standard template library.
- Memory management for container data structures.

4) Concurrency (week 12 to 17)

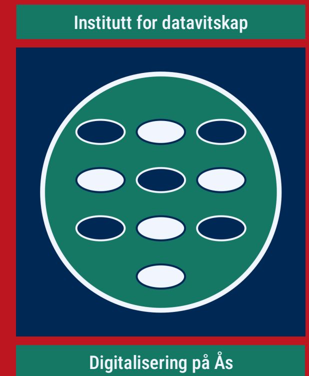
- MPI and ROS2 for parallel programming and concurrent processes.

5) Production and optimization (week 18 and 19)

- Good practices and useful tools for programming projects.

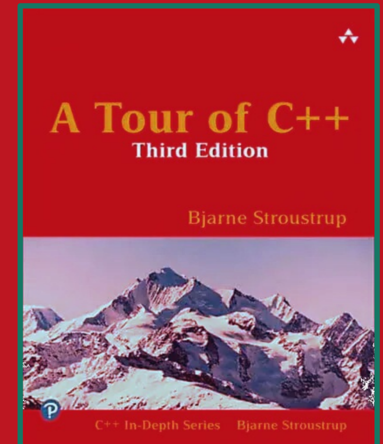


Noregs miljø- og biovitskaplege universitet



3 Data structures

3.1 Object orientation



Sections 5.1, 5.2

Core Guidelines:

C.3 – C.11
C.43 – C.51
(and more in "C")

Class definitions: From Python to C++

jupyter book-index (autosaved)

File Edit View Insert Cell Kernel Widgets Help   Not Trusted   Python 3

In [1]:
```python
class BookIndex:
    def __init__(self):
        self._chapter = 1
        self._section = 1
        self._page = 1

    def next_chapter(self):
        self._chapter += 1
        self._section = 1
        self._page += 1
        return self._chapter

    def next_section(self):
        self._section += 1
        return self._section

    def next_page(self):
        self._page += 1
        return self._page

    def out(self):
        print("Section ", self._chapter, \
              ".", self._section, ", p. ", \
              self._page, sep="", end="\n")
```

In [4]:
```python
idx = BookIndex()
idx._chapter = 1
idx._section = 8
idx._page = 8

idx.out()
```

```
Section 1.8, p. 8
```

Why is it **bad practice** to do this?
What should we do instead?

Python tutorial, Section 9.6:

«"**private**" instance variables that cannot be accessed except from inside an object **don't exist in Python**.

However, there is a **convention** that is followed by most Python code: a name **prefixed with an underscore** (*e.g.* _spam) should be treated as a **non-public** part of the API (whether it is a function, a method or a data member). It should be considered an implementation detail and subject to change without notice.»

Example file: book-index-python.ipynb

Class definitions: From Python to C++

Example file: book-index.zip

```
In [1]: class BookIndex:
            def __init__(self):
                self._chapter = 1
                self._section = 1
                self._page = 1

            def next_chapter(self):
                self._chapter += 1
                self._section = 1
                self._page += 1
                return self._chapter

            def next_section(self):
                self._section += 1
                return self._section

            def next_page(self):
                self._page += 1
                return self._page

            def out(self):
                print("Section ", self._chapter, \
                      ".", self._section, ", p. ", \
                      self._page, sep="", end="\n")

In [2]: idx = BookIndex()
        idx._chapter = 1
        idx._section = 8
        idx._page = 25

        idx.out()

        Section 1.8, p. 25

In [ ]: def start_chapter(b):
            b.next_chapter()
            b.out()

In [ ]: start_chapter(idx)
        idx.out()
```

```cpp
class BookIndex
{
  int chapter = 1;
  int section = 1;
  int page = 1;

  int next_chapter();

  int next_section();

  int next_page();

  void out() const;
}
```

```cpp
int BookIndex::next_chapter( )  {
  this->chapter++;
  this->section = 1;
  this->page++;
  return this->chapter;
}

int BookIndex::next_section( )  {
  this->section++;
  return this->section;
}

int BookIndex::next_page( )  {
  this->page++;
  return this->page;
}

void BookIndex::out( ) const  {
  cout << "Section " << this->chapter
       << "." << this->section
       << ", p. " << this->page << "\n";
}
```

A method is a function that belongs to an object. Methods are declared in the class definition (**header file**) and usually defined in the **code file**.

Class definitions: From Python to C++

Example file: book-index.zip

```
In [1]: class BookIndex:
            def __init__(self):
                self._chapter = 1
                self._section = 1
                self._page = 1

            def next_chapter(self):
                self._chapter += 1
                self._section = 1
                self._page += 1
                return self._chapter

            def next_section(self):
                self._section += 1
                return self._section

            def next_page(self):
                self._page += 1
                return self._page

            def out(self):
                print("Section ", self._chapter, \
                      ".", self._section, ", p. ", \
                      self._page, sep="", end="\n")

In [2]: idx = BookIndex()
        idx._chapter = 1
        idx._section = 8
        idx._page = 25

        idx.out()

        Section 1.8, p. 25

In [ ]: def start_chapter(b):
            b.next_chapter()
            b.out()

In [ ]: start_chapter(idx)
        idx.out()
```

```
class BookIndex
{
  int chapter = 1;
  int section = 1;
  int page = 1;

  int next_chapter();

  int next_section();

  int next_page();

  void out() const;
}
```

```
int BookIndex::next_chapter( )  {
  this->chapter++;
  this->section = 1;
  this->page++;
  return this->chapter;
}

int BookIndex::next_section( )  {
  this->section++;
  return this->section;
}

int BookIndex::next_page( )  {
  this->page++;
  return this->page;
}

void BookIndex::out( ) const  {
  cout << "Section " << this->chapter
       << "." << this->section
       << ", p. " << this->page << "\n";
}
```

How do Python and C++ deal with argument passing?

The pointer **this** is analogous to the object reference "self" from Python. It points to the object itself.

If a method is declared as **const**, it cannot change any of the object's own properties.

Access object members using dot (.) and arrow (->)

Properties: Variables of an object;
Methods: Functions of an object.

The properties and methods are called the members of the object.

Just like in Python, the **dot operator** can be used to access a member:

```
BookIndex b;
b.chapter = 1;
```

Often we deal with pointers to an object. Then we might write:

```
BookIndex* c = &b;
(*c).chapter = 2;
```

The **arrow operator** abbreviates this:

```
c->chapter = 2;
```

Example file: book-index.zip

```
class BookIndex
{
  int chapter = 1;
  int section = 1;
  int page = 1;

  int next_chapter();

  int next_section();

  int next_page();

  void out() const;
}
```

```
int BookIndex::next_chapter( )  {
  this->chapter++;
  this->section = 1;
  this->page++;
  return this->chapter;
}

int BookIndex::next_section( )  {
  this->section++;
  return this->section;
}

int BookIndex::next_page( )  {
  this->page++;
  return this->page;
}

void BookIndex::out( ) const  {
  cout << "Section " << this->chapter
       << "." << this->section
       << ", p. " << this->page << "\n";
}
```

The pointer **this** is analogous to the object reference "self" from Python. It points to the object itself.

If a method is declared as **const**, it cannot change any of the object's own properties.

Private members cannot be accessed from outside

The **private and public status of class members** (*i.e.*, properties and methods) is stated in the class definition, where properties and methods are declared:

```
class ExampleClass {
public:
  TypeA getPropertyA() const {return this->propertyA;}
  TypeB* getPropertyB() const {return this->propertyB;}
  void setPropertyA(TypeA a) {this->propertyA = a;}
  void setPropertyA(TypeB* b) {this->propertyB = b;}
  void do_something();

private:
  TypeA propertyA;
  TypeB* propertyB;

  void helper_method();
};
```

Only the public part of the class definition is the interface accessible to code outside the scope of the class.

Typical object-oriented design makes all properties (objects' variables) private. They are read using public "get" methods and modified using public "set" methods.

Methods that are only called by other methods of the same class, but not from outside, are also declared to be private.

Constructors and destructors

Constructor: A method that is called when an object is **allocated**.
Destructor: A method that is (implicitly) called when an object is **deallocated**.

They are not mandatory (as we have seen); use them if you need to specify some functionality for this purpose. Most typically:

- Provide a **constructor** if you want to give the user control over how the private properties of an object are initialized.
- There are also special “copy constructors” and “move constructors”. (Not to be discussed right now.)
- Provide a **destructor** if your memory management strategy requires it; there might be properties stored as pointers that need to be deleted.

```
class BookIndex {
public:
  BookIndex(int c, int s, int p);
  ~BookIndex();
  ...
};
```

```
BookIndex::BookIndex(int c, int s, int p) {
  this->chapter = c; this->section = s; this->page = p;
}
BookIndex::~BookIndex() {
  cout << "Deleting a BookIndex object.\n";
}
```

Constructors and destructors

General rule: For every "new" there must be a matching "delete".

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

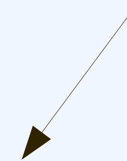
This is the case both for objects on the stack and on the heap:

```
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 …
}
```

If T has ownership over p, this must be done! Without it, there would be a **memory leak**!



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

OOP as a programming paradigm

Imperative programming

- It is stated, instruction by instruction, what the processor should do
- Control flow implemented by jumps (**goto**)

Structured programming

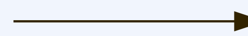
- Same, but with **higher-level control flow**
- Contains "instruction by instruction" code

Procedural programming

- **Functions** (procedures) as **highest-level structural unit** of code
- Still contains loops, *etc.*, for control flow within a function

Object-oriented programming (OOP)

- **Classes** as **highest-level structural unit** of code; objects instantiate classes
- Still contains functions, *e.g.*, as methods



Programming paradigms based on **describing the solution** rather than computational steps:

Functional programming
(also: "declarative programming")

Constraint programming

Logic programming

Generic programming
(introduces ideas from declarative and logical methods into OOP)

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.

Wikidata Query Service

Eksempler | Spørringsbygger | Hjelp | Flere verktøy | norsk (bokmål)

```
1  #Largest cities per country
2  SELECT DISTINCT ?city ?cityLabel ?population ?country ?countryLabel ?loc WHERE {
3    {
4      SELECT (MAX(?population_) AS ?population) ?country WHERE {
5        ?city wdt:P31/wdt:P279* wd:Q515 .
6        ?city wdt:P1082 ?population_ .
7        ?city wdt:P17 ?country .
8      }
9      GROUP BY ?country
10     ORDER BY DESC(?population)
11   }
12   ?city wdt:P31/wdt:P279* wd:Q515 .
13   ?city wdt:P1082 ?population .
14   ?city wdt:P17 ?country .
15   ?city wdt:P625 ?loc .
16   SERVICE wikibase:label {
17     bd:serviceParam wikibase:language "en" .
18   }
19 }
20 ORDER BY DESC(?population)
```

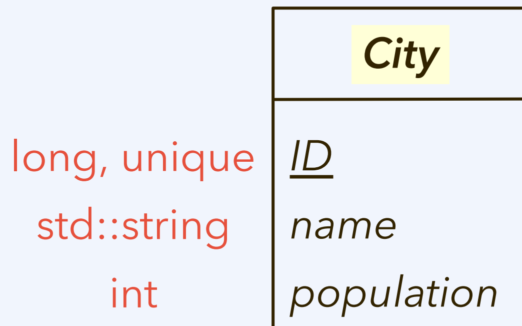
290 resultater i løpet av 4117 ms | Kode | Last ned | Lenke

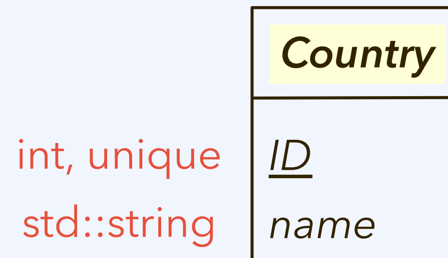
city	cityLabel	population	country	countryLabel	loc
wd:Q172	Toronto	2731571	wd:Q16	Canada	Point(-79.386666666 43.670277777)
wd:Q1490	Tokyo	14047594	wd:Q17	Japan	Point(139.691722222 35.689555555)
wd:Q585	Oslo	693494	wd:Q20	Norway	Point(10.738888888 59.913333333)
wd:Q1761	Dublin	553165	wd:Q27	Republic of 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)

Designing classes: Entity-relationship diagrams

particular: **object** (entity, individual) — **property** (sometimes: attribute)

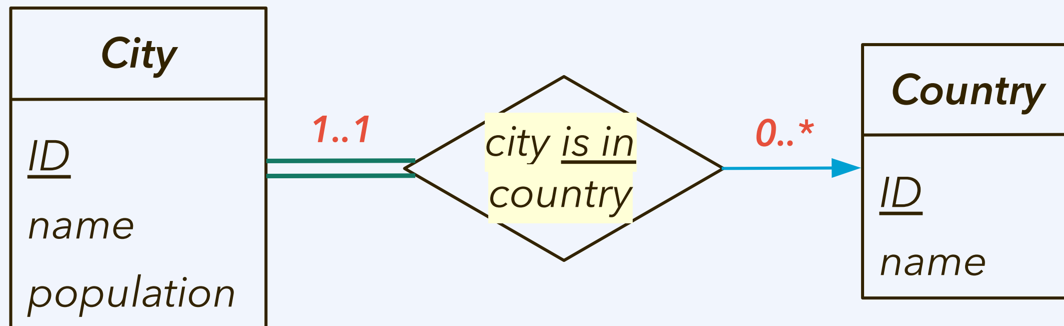
universal: **class** (entity type, concept) — **attribute** (sometimes: attribute type) (in OWL: DatatypeProperty)





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"

(or use cardinality constraints, as in red above)

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;
};
```

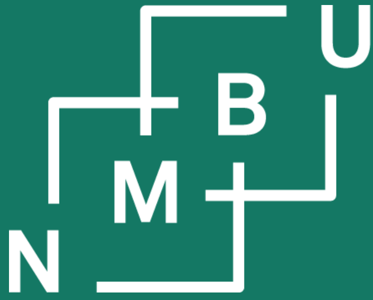
Example file: city-country.zip

Constructors and destructors: Do I need them?

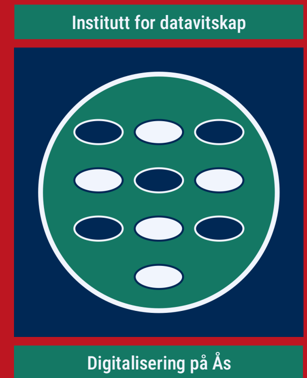
Core Guidelines:

- C.20: “If you can avoid defining default operations, do.”
  - “This is known as “the **rule of zero**.” Define zero constructors or destructors if it can be done without creating an inconsistent state.
  - A simple and good reason for defining a constructor is to force the user to provide some information that is required.
  - Define a constructor in cases where it does not make sense to initialize the object’s properties to some specified default value.

- **C.30**: “Define a destructor if a class needs an explicit action at object destruction.” And related, **C.31**: “All resources acquired by a class must be released by the class’s destructor.”
  - If a data structure needs to be built up (memory allocated, *etc.*), this normally requires both a constructor and a destructor.
  - For such cases, we will learn the *rule of three* and the *rule of five*.



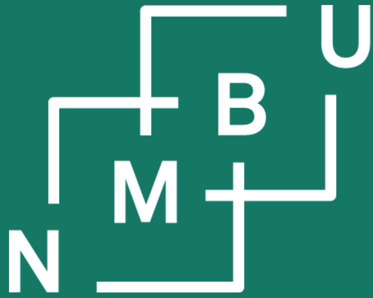
Noregs miljø- og biovitskaplege universitet



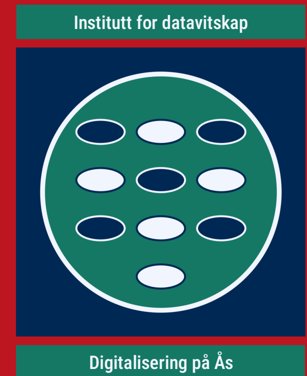
Tutorial scheduling

INF205

26th February 2024



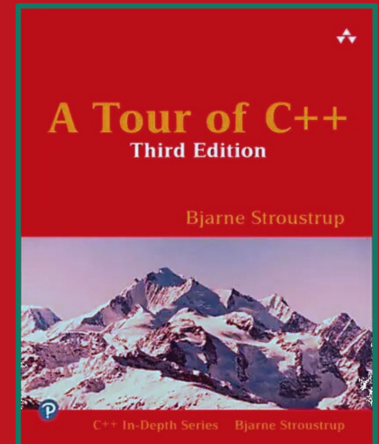
Noregs miljø- og biovitskaplege universitet



3 Data structures

3.1 Object orientation

3.2 Inheritance



Sections 5.3 – 5.5

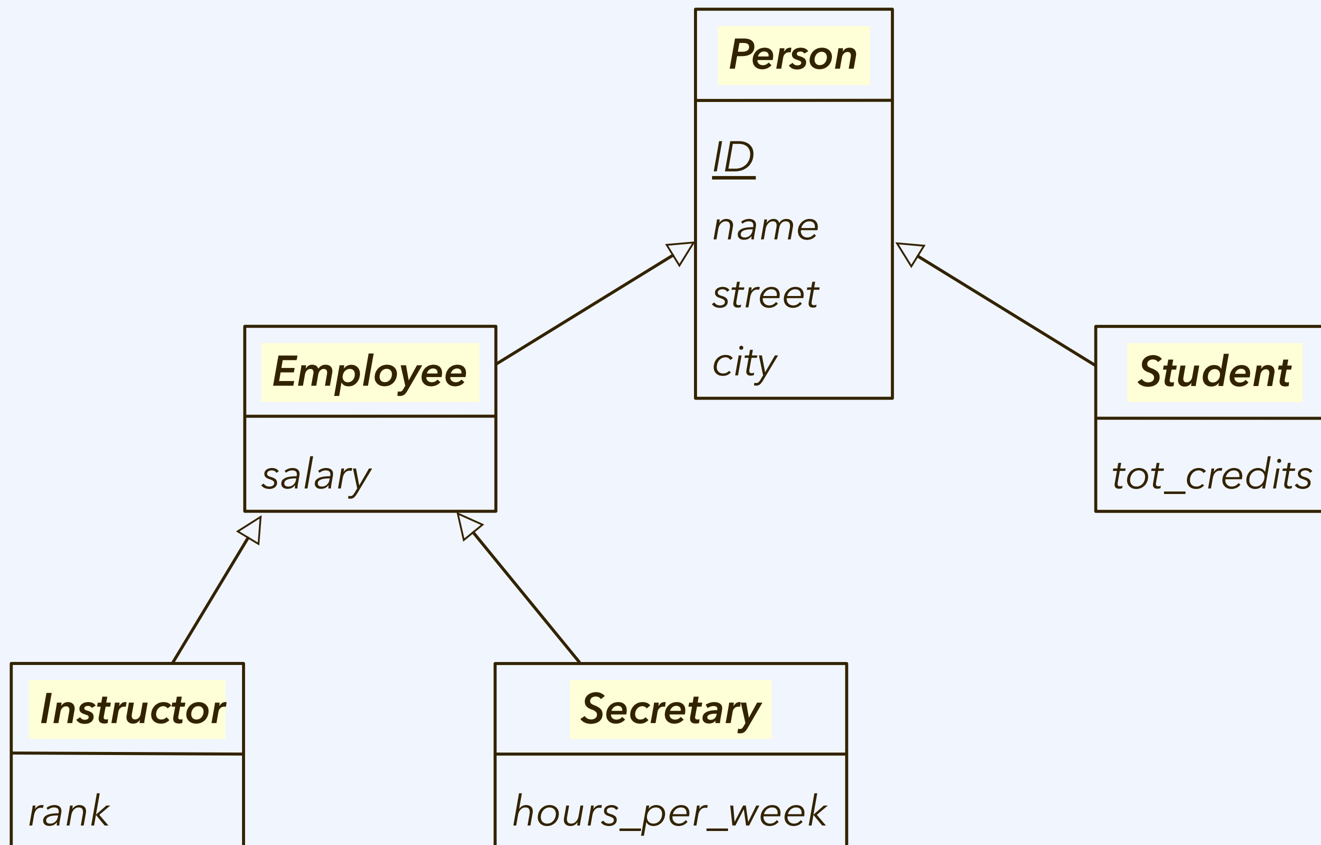
Core Guidelines:

C.120, C.121
C.146 – C.148

(and more in "C")

Taxonomy: Representation in an E-R diagram

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**.

Taxonomy (class hierarchy)

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

Taxonomy (class hierarchy) in C++

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.

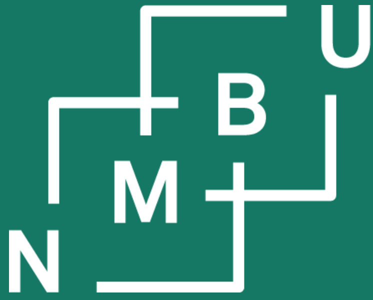
Example file: literature-indices.zip

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

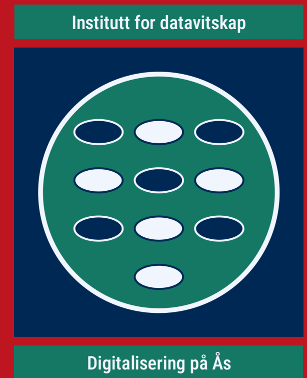
```
class JournalArticleIndex: public LiteratureIndex {
public:
  int next_page();
  …
private:
  int volume = 0;
  …
};
```

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**.



Noregs miljø- og biovitskaplege universitet



Conclusion

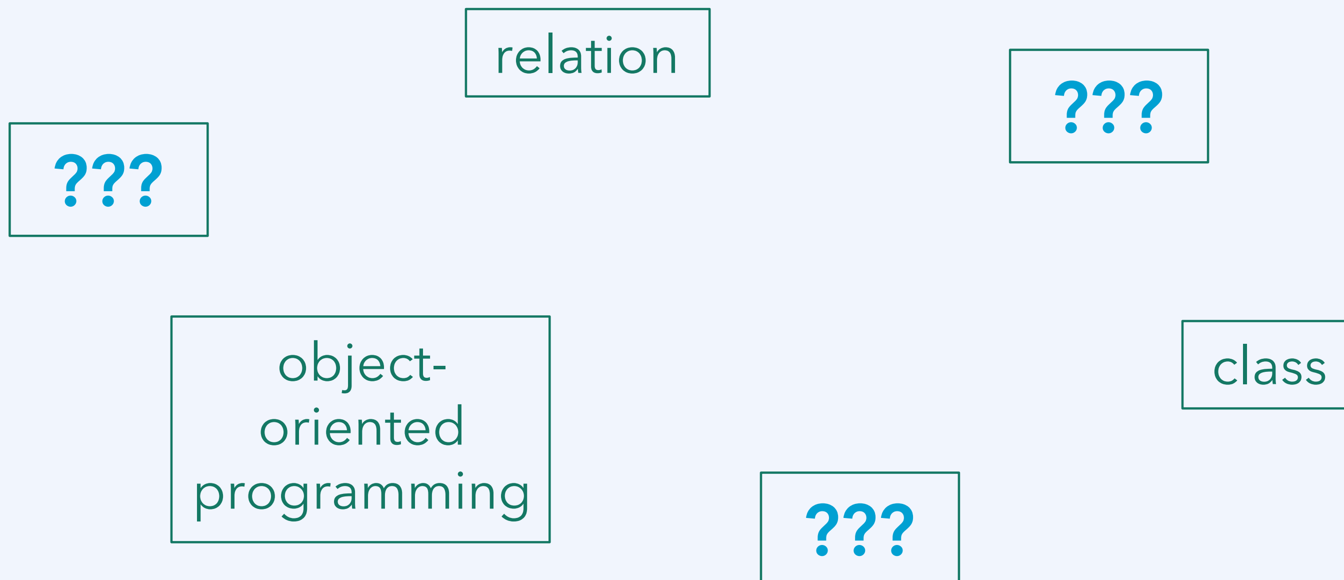
INF205

26th February 2024

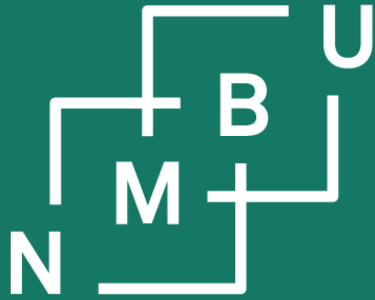
Weekly glossary concepts

What are essential concepts from this lecture?

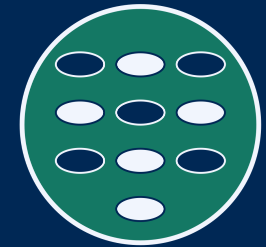
Let us include them in the **INF205 glossary**.[1]



[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