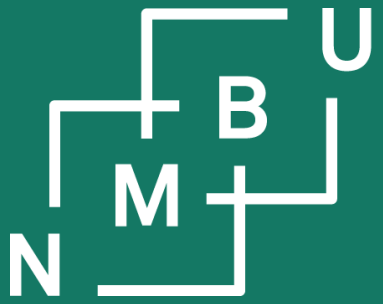
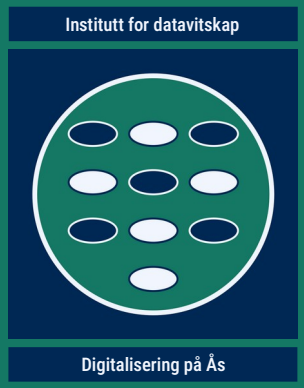


Noregs miljø- og biovitskaplege universitet

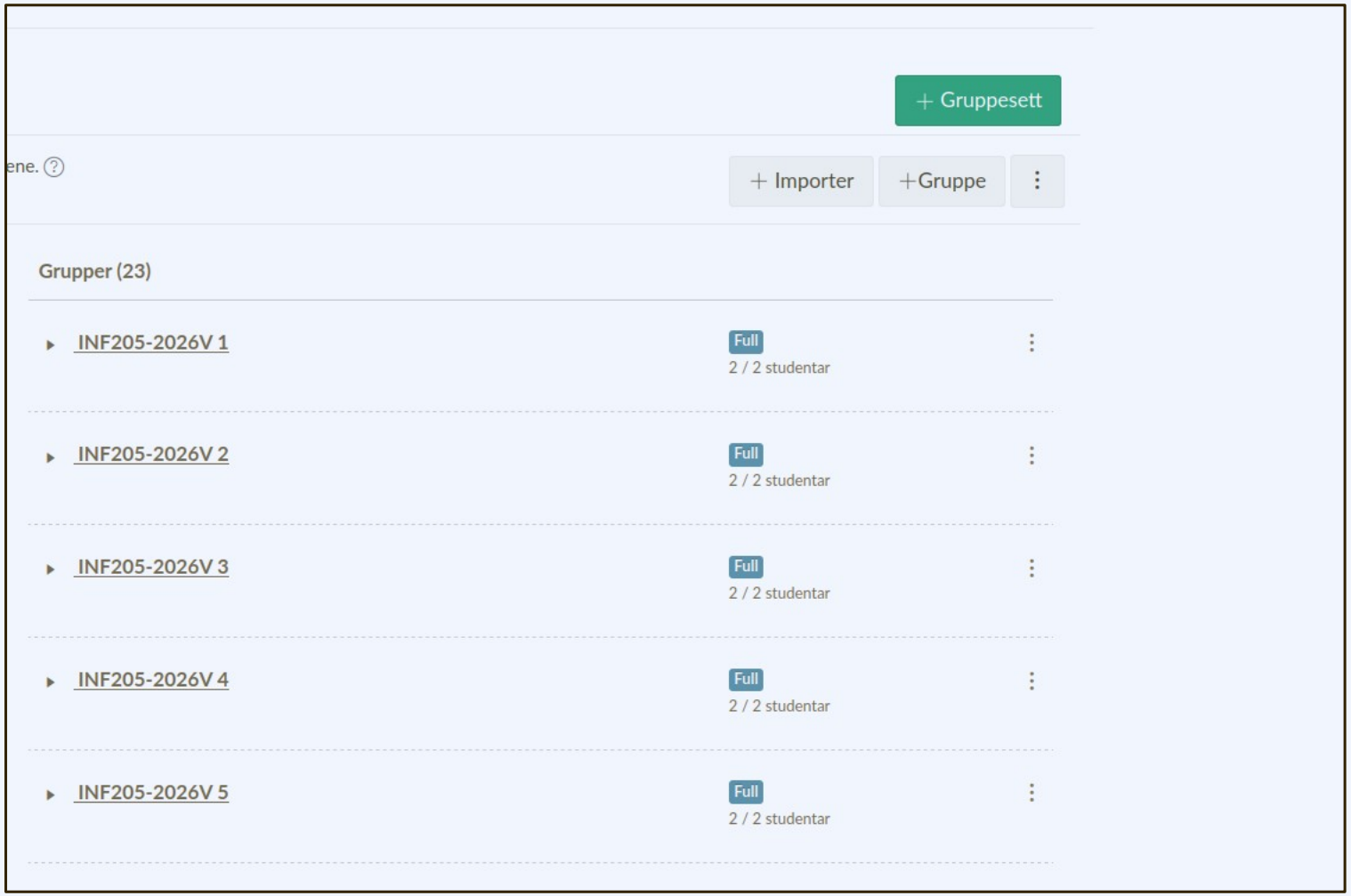


INF205

Ressurseffektiv programmering

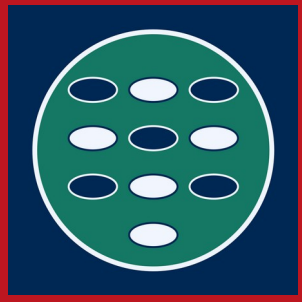
1 Moderne C++

Gruppetilordning



Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap



Digitalisering på Ås

1 Moderne C++

1.1 Frå Python til C++

1.2 Peikarar og datatabellar

1.3 Programmeringsparadigme

INF205

11. februar 2026

Ordliste

Class	Klasse
Object	Objekt
Object-oriented programming	Objektorientert programmering
Postcondition	Ettertilstand (Etterbetingelse)
Precondition	Førtilstand (Førbetingelse)
Procedural programming	Prosedyrisk programmering

Programming paradigms

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

- **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")

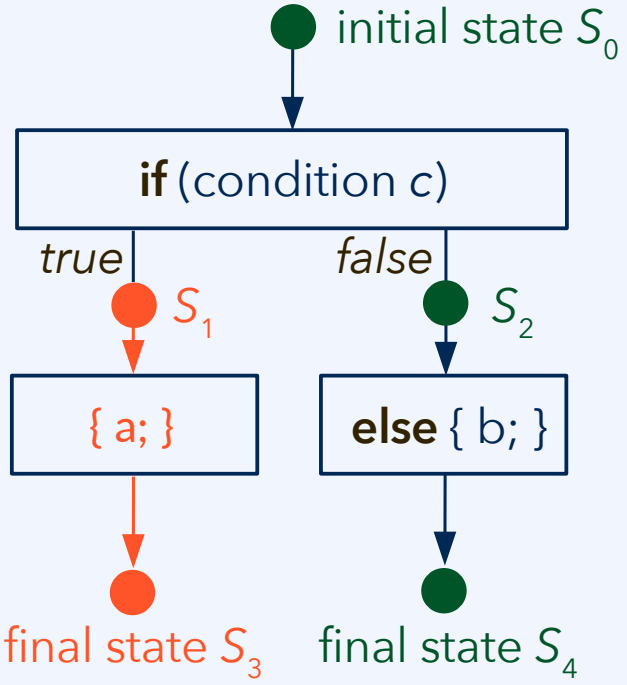
Logic programming

Constraint programming

Preconditions and postconditions

Precondition: State of the program at a point directly before the considered unit. This may include assumptions taken from the specification.

Postcondition: State of the program at a point directly after the considered unit, assuming that the precondition was fulfilled at the point directly before it.



Note

Consider the statement "a" from transition $S_1 \rightarrow S_3$:

- Execution state S_1 fulfils the **precondition** of statement a.
- Execution state S_3 fulfils the **postcondition** of statement a.

Program flow graphs

Formal analysis goes the opposite way, constructing conditions for states.

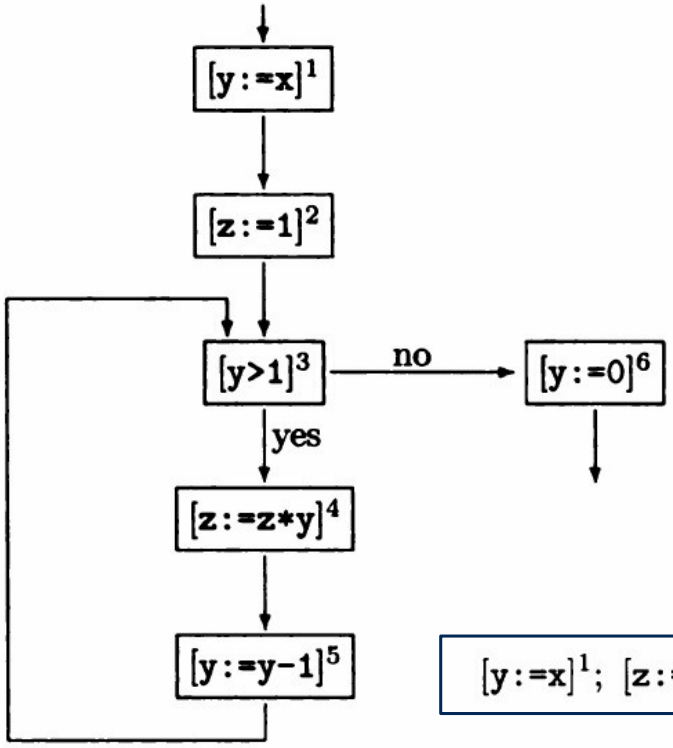


Figure 1.2: Flow graph for the factorial program.

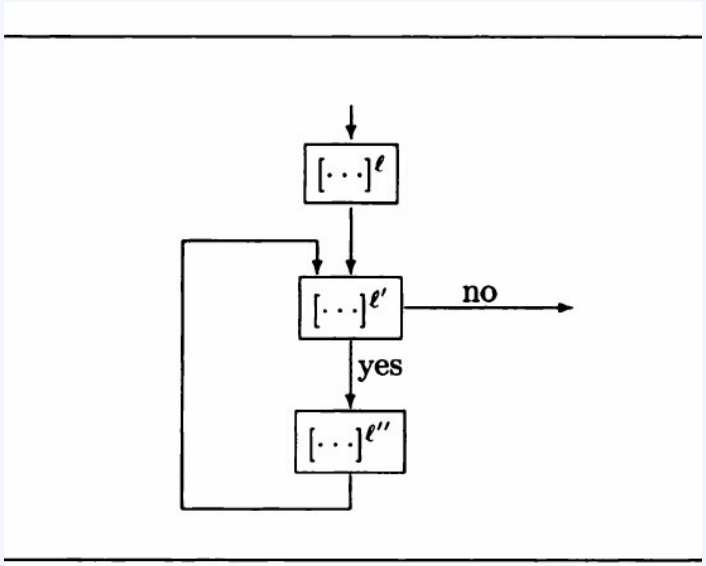


Figure 2.2: A schematic flow graph.

$[y:=x]^1;\ [z:=1]^2;$ `while` $[y>1]^3$ `do` $([z:=z*y]^4;\ [y:=y-1]^5);\ [y:=0]^6$

[1]F. Nielson, H. Riis Nielson, C. Hankin, *Principles of Program Analysis*, Heidelberg: Springer, **2005**.

Functions / procedural programming

In many procedural programming languages, including C/C++ and Python, code blocks that can be called from other blocks are called **functions**. However, do not confuse **procedural programming** (as a programming paradigm) with **functional programming**, a name given to a very different approach (LISP, *etc.*).

- Functions are named
- Each function has a distinct task
- It may have its own variables
- It may call another function, including calls to itself (recursion),
- It may return a value; it must have a return type (which may be **void**)
- It may accept arguments
- Function **parameters** are the variables listed in the function's definition. Function **arguments** are the values passed to the function, which are assigned to the function's parameters at runtime.

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)

Class definitions: From Python to C++

jupyter book-index (autosaved)

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

Run Code

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 [4]:

```
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.»

Class definitions: From Python to C++

```
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()
```

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";
}
```

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**.

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: 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.

Glossar

Klasse

Definisjon: det som «definerer en datatype og beskriver innhold og egenskaper til objekter av denne datatypen»

«Klassens datafelter er lagerenheter for objektets verdier, og klassens metoder beskriver operasjonene som kan gjøres på eller av objektene»

(sitat tekne frå E. H. Vihovde sitt SNL-bidrag om objektorientering)

Glossar

Objekt

Forklaring: «grunnsteinen […] i objektorienterte datasystemer; […] lukket modul bestående av data og behandlingsregler. Sentrale begreper i systemer som baseres på objekter er innkapsling, arv og polymorfi» (E. H. Vihovde, SNL).

- Arv: «Klasser dannes hierarkisk, slik at underordnede klasser arver egenskapene til de overordnede. I en database kan for eksempel underklassen 'sekretær' arve egenskapene til overklassen 'ansatte'.» (ibid.).
- «Innkapsling betyr å avgrense hva slags datatyper og hvilke behandlingsregler som skal gjelde for en bestemt klasse av objekter» (ibid.).
- "Polymorfi betyr at prosedyrer kan gjøres gjeldende for objekter hvis nøyaktige form eller type kan variere" (ibid.).

Glossar

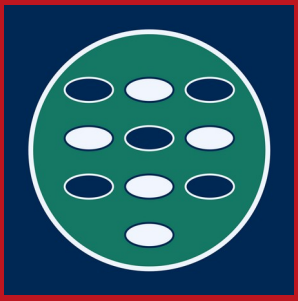
Objektorientert programmering

Definisjon: «form for dataprogrammering der dataprogrammene organiseres i en struktur med klasser og objekter som minner om hvordan vi organiserer virkeligheten rundt oss. Data og metoder som naturlig hører sammen er samlet i enheter, kalt objekter. Ved hjelp av klassedefinisjoner kan man lage nye datatyper» (E. H. Vihovde i SNL).

- «Nordmennene Ole-Johan Dahl (1931–2002) og Kristen Nygaard (1926–2002) var opphavsmennene til objektorientert programmering» (ibid.).

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap



Digitalisering på Ås

1 Moderne C++

1.1 Frå Python til C++

1.2 Peikarar og datatabellar

1.3 Programmeringsparadigme

1.4 God praksis i moderne C++

INF205

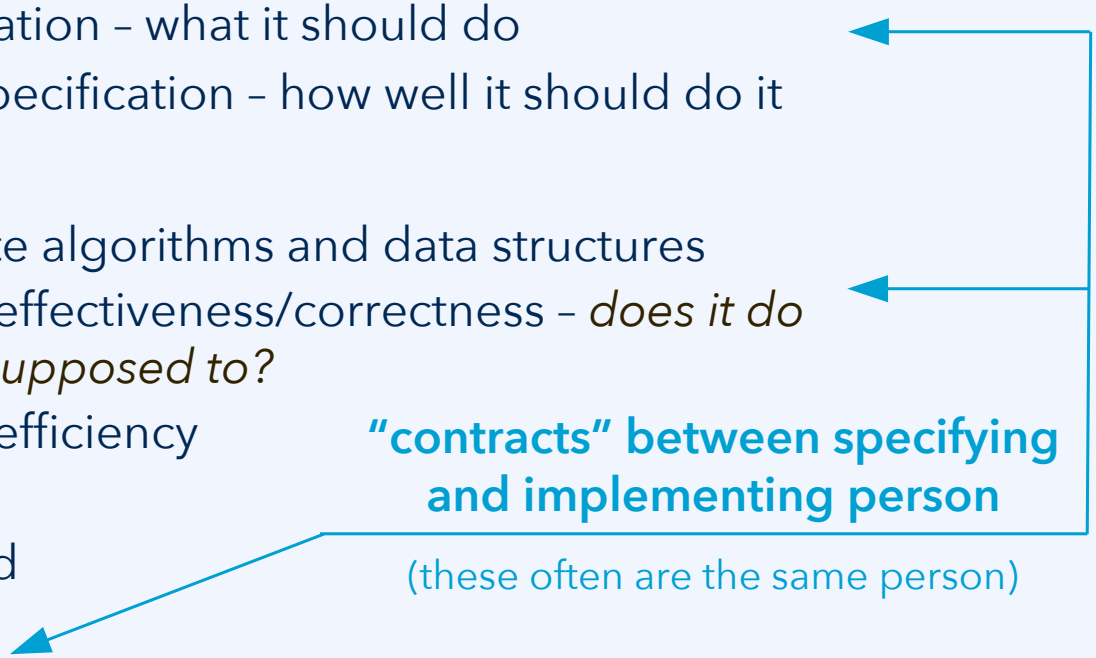
11. februar 2026

Ordliste

Assertion, assert	Forvissing
Argument	Argument
Design by contract	Kontraktbasert programmering
Namespace	Namnerom
Program flow graph	Programflytgraf
Unnamed namespace	Anonymt namnerom

Design by contract

- Specify
  - Function specification – what it should do
  - Non-functional specification – how well it should do it
- Design
  - Select appropriate algorithms and data structures
    - Consider effectiveness/correctness – *does it do what it is supposed to?*
    - Consider efficiency
      - Size
      - Speed
- Implement
  - Create solution at low level
- Evaluate
  - Debug, assess for syntactic & semantic correctness
  - Check performance (i.e., resource requirements)



Program flow graphs: Formal analysis

For purposes of formal analysis, the program flow is analysed step by step, e.g., at the instruction (statement) level, at the level of blocks of code that form a coherent unit, or at the level of functions or methods.

Precondition: State of the program at a point directly before the considered unit. This may include assumptions taken from the design contract or specification.

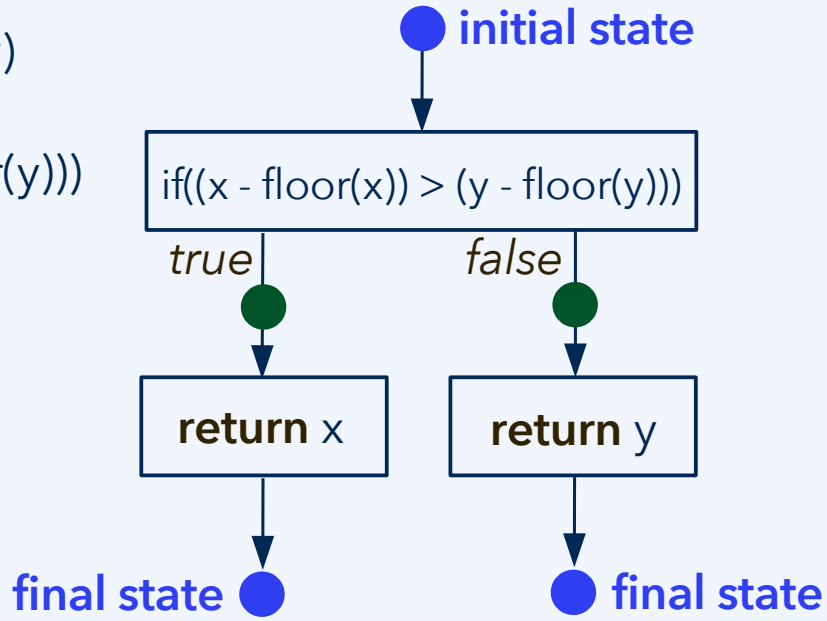
Postcondition: State of the program at a point directly after the considered unit, assuming that the precondition was fulfilled at the point directly before it.

Example

As part of a development project, we need a function grtfrac(x, y) that takes **two floating-point arguments** and returns the one with the greater fractional part; *e.g.*, grtfrac(2.7, 3.6) is to return 2.7, because ".7" is greater than ".6". **In design by contract, the caller, not the called method needs to guarantee the precondition.**

Program flow graphs: Formal analysis

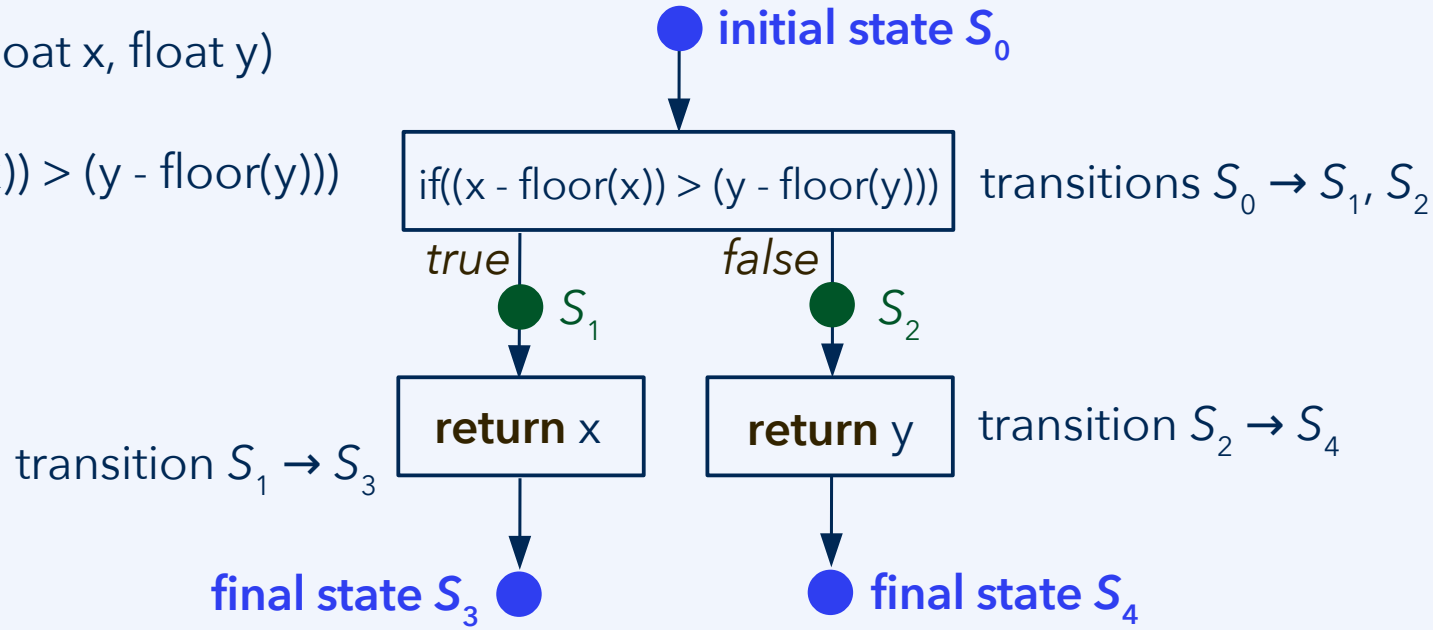
```
float grtfrac(float x, float y)
{
  if((x - floor(x)) > (y - floor(y)))
    return x;
  else
    return y;
}
```



Initial state: x and y are floating-point numbers (by specification).

Program flow graphs: Formal analysis

```
float grtfrac(float x, float y)
{
  if((x - floor(x)) > (y - floor(y)))
    return x;
  else
    return y;
}
```



S_0: x and y are floating-point numbers (by specification).

S_1: x, y as above; the fractional part of x is greater than that of y.

S_2: x, y as above; the fractional part of y is greater than that of x, or equal.

S_3: The fractional part of x is the greater one, and x was returned.

S_4: The fractional part of y is greater (or they are equal); y was returned.

Overloading and namespaces

Function **overloading** (identical name within the **same namespace**, if any) and the use of **multiple namespaces** are technically different mechanisms. However, they become similar if equal names occur in multiple namespaces.

```
namespace task_a
{
  void run(double x, double y);
}
namespace
{
  void run(int x, int y);
}

int main()
{
  using namespace task_a;
  run(1.0, 1.0);
}
```

```
namespace task_b
{
  void run(int x, int y);
  void run(double x, double y);
}

int main()
{
  using namespace task_b;
  run(1.0, 1.0);
}
```

```
namespace task_c
{
  void run(double x, double y);
}
namespace
{
  void run(double x, double y);
}

int main()
{
  run(1.0, 1.0);
  task_c::run(1.0, 1.0);
}
```

In what case are we strictly overloading “run” (within a single namespace)?

In each of the cases, which version of “run” will be executed?

Example file: **namespaces-overloading.zip**

C++ Core Guidelines

- In: Introduction
- P: Philosophy
- I: Interfaces
- F: Functions
- C: Classes and class hierarchies
- Enum: Enumerations
- R: Resource management
- ES: Expressions and statements
- Per: Performance
- CP: Concurrency and parallelism
- E: Error handling
- Con: Constants and immutability
- T: Templates and generic programming
- CPL: C-style programming
- SF: Source files
- SL: The Standard Library

https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md

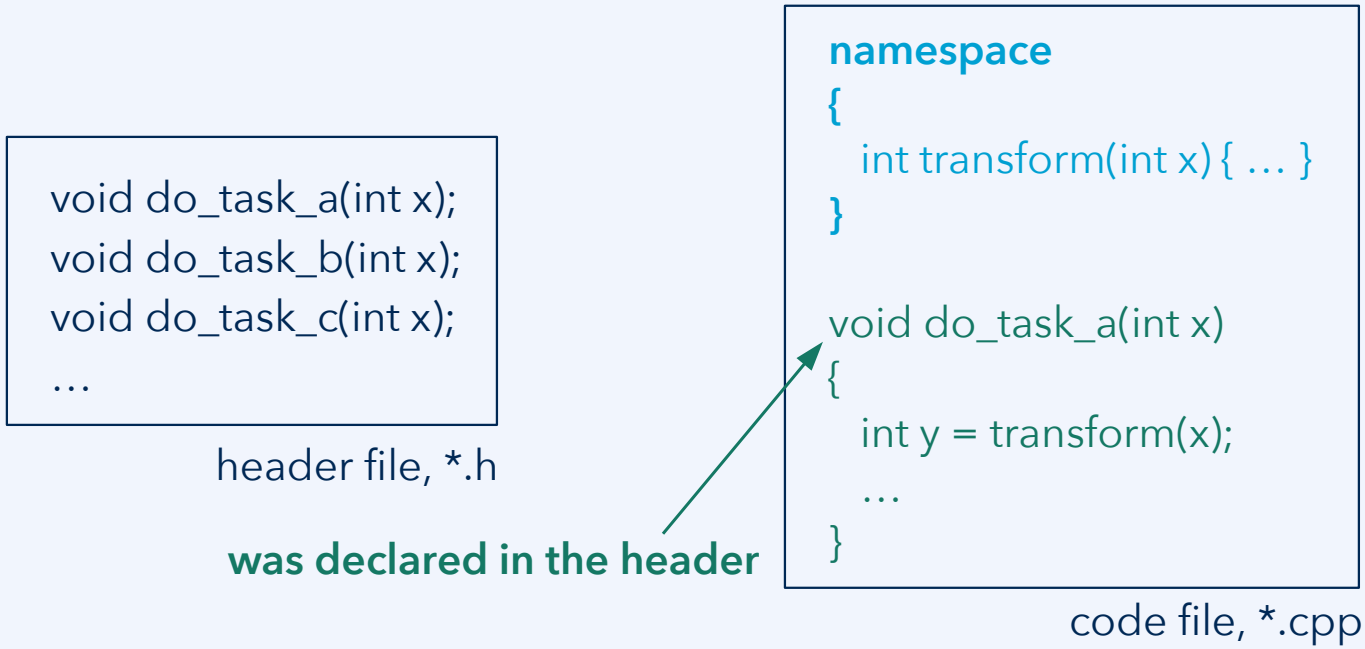
Selected guidelines on namespaces

SF.20: Use namespaces to express logical structure

Use of the "unnamed namespace" construction: **namespace{ … }**

- **SF.21:** Don't use an unnamed namespace in a header
- **SF.22:** Use an unnamed namespace for all internal/non-exported entities

(This makes it easy to distinguish "helper" code from that needed outside.)



Selected guidelines on functions

Core Guidelines on functions:

- F.1: "Package" meaningful operations as carefully named functions
- F.2: A function should perform a single logical operation
- F.3: Keep functions short and simple

  …
- F.46: int is the return type for main()

I.6: Prefer **Expects()** for expressing preconditions

I.7: State postconditions [with **Ensures()**]

example based on Grimm, 2022, p.443:

```
int area(int height, int width)
{
  Expects(height > 0);
  int retv = height*width;
  Ensures(retv > 0);
  return retv;
}
```

More traditional style uses **assert(…)**.

Example files: **conditions-gsl.cpp** (use of GSL) and **conditions-assert.cpp** (typical use).

Selected guidelines: Signed/unsigned

Core Guidelines style rules against "**unsigned**" (same as **unsigned int**).
These rules use elements taken from the **Guidelines Support Library (GSL)**.

ES.102: Use signed types for arithmetic

ES.106: Don't try to avoid negative values by using "unsigned"

ES.107: Don't use unsigned for subscripts [*e.g.*, array indices], prefer **gsl::index**

The reasoning against a normal (signed) integer is that "**int** might not be big enough."

Then rather use **long** (instead of **unsigned int**) …

Remember the pitfall: For arithmetics over unsigned integer variables, the result of the subtraction "**2 – 3**" is the value **4 294 967 295**.

Glossar

Argument

Definisjon: verdi som blir overført til ein funksjon

Kontraktbasert programmering

Definisjon: programutvikling og kodedesign basert på spesifisering av før- og ettertilstandar, der ein viss del av programmet (t.d. ein funksjon) må garantere at ettertilstanden blir nådd, medan brukaren må ta vare på at krava til førtilstanden er innfridde

Namnerom

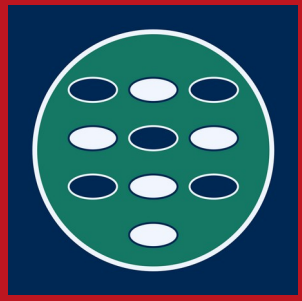
Definisjon: globalt tilgjengeleg mengde av namn, dvs. av faste nemningar for funksjonar, variablar, klasser, osv.

For å få tilgang til namn Y definert i namnerom X, skriv ein X::Y.

Gjennom using namespace X; blir alle namn frå X tilgjengelege utan prefiks.

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap

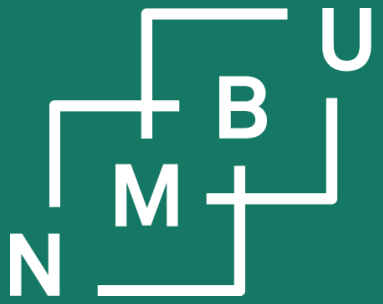


Digitalisering på Ås

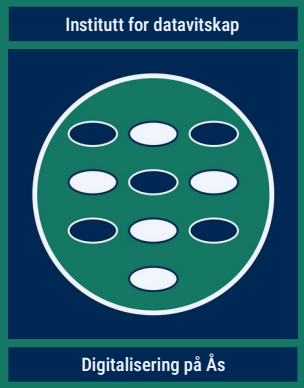
Samandrag / diskusjon

INF205

11. februar 2026



Noregs miljø- og biovitskaplege universitet



INF205

Ressurseffektiv programmering

1 Moderne C++