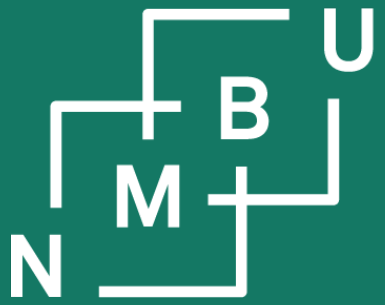


Norges miljø- og biovitenskapelige universitet

INF205
Resource-efficient programming

I C++ basics

I.1 Scope of the module
I.2 C/C++: Getting started
I.3 Discussion: From Python to C and C++
I.4 Good practice and style

Why resource-efficient programming?

Norwegian University of Life Sciences

Figure 1. Moore's original prediction graph showed component count followed a straight line when plotted on log paper.[26]

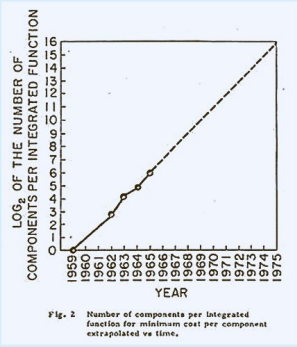
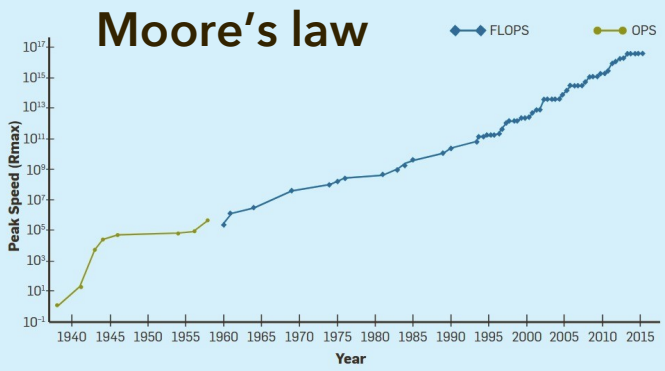
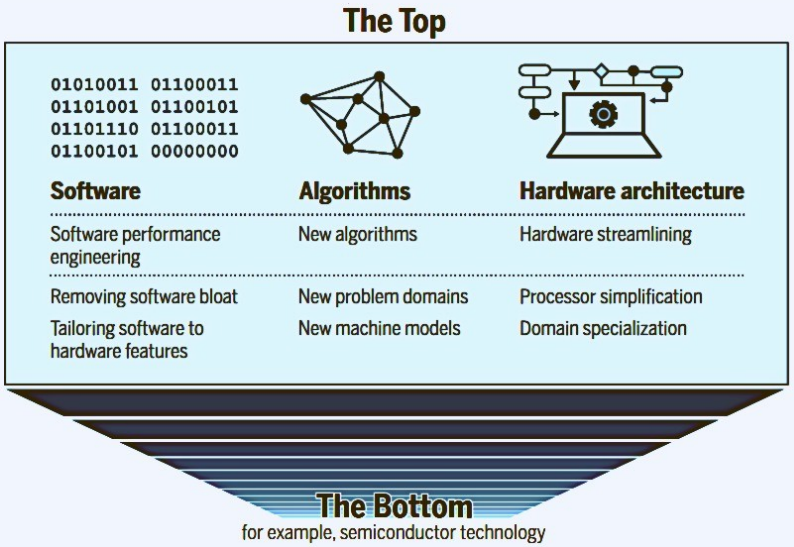


Figure 2. Speeds of the fastest computers from 1940 show an exponential rise in speed. From 1965 to 2015, the growth was a factor of 12 orders of 10 over 50 years, or a doubling approximately every 1.3 years.



P. J. Denning, T. G. Lewis, doi:10.1145/2976758, **2017**.

"What comes after Moore's law?"



Performance gains after Moore's law ends. In the post-Moore era, improvements in computing power will increasingly come from technologies at the "Top" of the computing stack, not from those at the "Bottom", reversing the historical trend.

C. E. Leiserson *et al.*, doi:10.1126/science.aam9744, **2020**.

Embedded systems

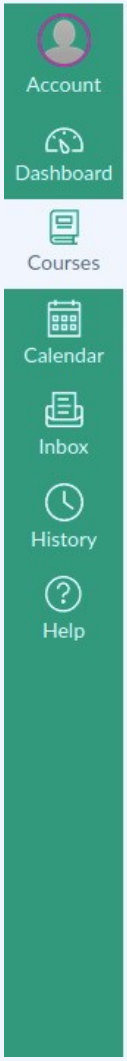
Digitalization entails pervasive computing, including at nodes or components without a great amount of computational resources.

therein, see Tab. 1

Module room on Canvas

Norwegian University of Life Sciences

What features on Canvas are you using most?



2022 HØST

- Home
- **Syllabus**
- People
- Office 365
- Grades
- Discussions
- Zoom
- Collaborations
- BigBlueButton
- Panopto Video

Course syllabus

NMBU, INF205, H2022: Resource-efficient programming

- Lecture: Wednesday, 14.00 - 16.00, TF1-102.
- Datalab: Thursday, 10.00 - 12.00, TF1-105 (group #1); Thursday, 12.00 - 14.00, TF1-105 (group #2).

Team: Martin Horsch (office: TF2-303A), Jorge Hermoso.

Module resources:

- INF205 course website
- NMBU course catalogue entry in English and in Norsk Bokmål

Literature:

Stroustrup's "Tour of C++" is a compact book presenting modern C++ to readers with a solid background in programming. It is the main literature source for the module.

- B. Stroustrup, *A Tour of C++*, 2nd edn., Addison-Wesley (ISBN 978-0-134-99783-4), **2018**.

Course summary:

Date	Details	Due
Wed, 7 Sep 2022	Ressurseffektiv programmering - Forelesning	14:00 to 16:00
Thu, 8 Sep 2022	Ressurseffektiv programmering - Øving	10:00 to 12:00
	Ressurseffektiv programmering - Øving	12:00 to 14:00

Norwegian University of Life Sciences

Course website

NMBU, INF205: Resource-efficient programming (autumn 2022)

- Lecture: Wednesday, 14.00 - 16.00, TF1-102.
- Datalab: Thursday, 10.00 - 12.00, TF1-105 (group #1); Thursday, 12.00 - 14.00, TF1-105 (group #2).

Team: Martin Horsch (office: TF2-303A), Jorge Hermoso.

University resources:

- INF205 page on Canvas/Instructure
- NMBU course catalogue entry in English and in Norsk

https://home.bawue.de/~horsch/teaching/inf205/

Literature:

Stroustrup's "Tour of C++" is a compact book presenting modern C++ to readers with a solid background in programming. It is the main literature source for the module.

- B. Stroustrup, *A Tour of C++*, 2nd edn., Addison-Wesley (ISBN 978-0-134-99783-4), **2018**.

Supporting literature/other books that could be of interest:

- R. Grimm, *C++ Core Guidelines Explained: Best Practices for Modern C++*, Addison-Wesley (ISBN 978-0-136-87567-3), **2022**.
- P. Sanders *et al.*, *Sequential and Parallel Algorithms and Data Structures*, Springer (ISBN 978-3-030-25209-0), **2019**.
- B. Stroustrup, *Programming: Principles and Practice Using C++*, 2nd edn., Addison-Wesley (ISBN 978-0-321-99278-9), **2014**.

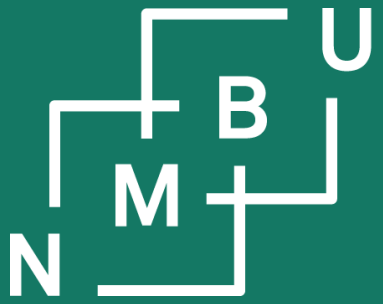
Structure:

1. C++ basics
    - Calendar week 36
        - Literature: Stroustrup (2018) Sections 1.2, 1.3, and 1.8
        - Software: Eclipse IDE C/C++ version (install this or the previous version, but not the eclipse rpm package, which is for Java only)
    - Calendar week 37
        - Literature: Stroustrup (2018) Sections 1.4 to 1.6, 2.2, and 3.2 to 3.4

C++ versus Python

C++	Python

"What differences between C++ and Python are the most important (to you)?"



Noregs miljø- og biovitskaplege universitet

1 C++ basics

1.1 Scope of the module

Learning outcomes

After completing the course you will be able to

- implement solutions in modern C++;
- manage memory correctly in larger projects;
- make use of capabilities provided by the C++ Standard Library and third-party libraries;
- implement data types from "first principles;"
- write code suitable for embedded systems;
- create interfaces allowing your code to interact with other software.

We speak of "**modern C++**" because of the long history of C++, *e.g.*, retaining all of the C programming language. C++ is like several languages in one.

Focus: Develop solutions that work both reliably and efficiently.

Structure

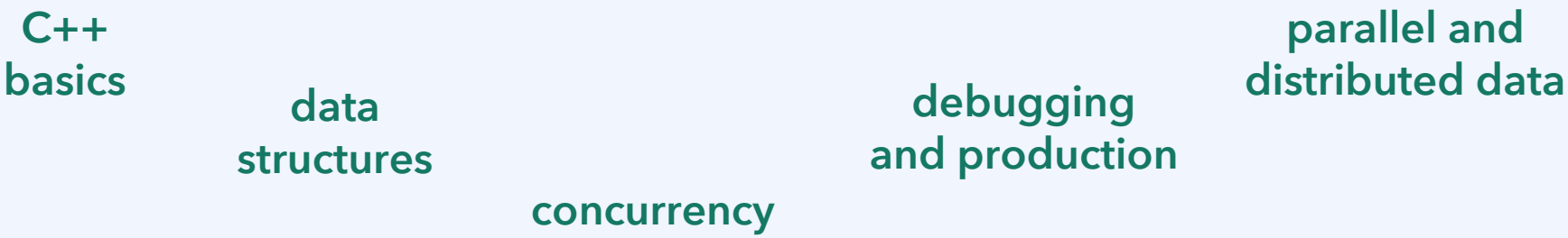
1) C++ basics: Intro into “modern C++” as a programming language.

2) Data structures: C++ standard template library (lists, maps, *etc.*). How to build data structures in a language that gives you control over memory management.

3) Concurrency: MPI and parallelization in scientific computing; ROS in C++.

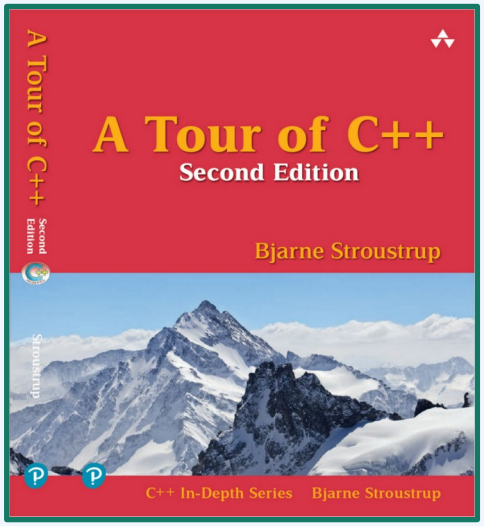
4) Debugging and production: Tools, good practice, and optimization.

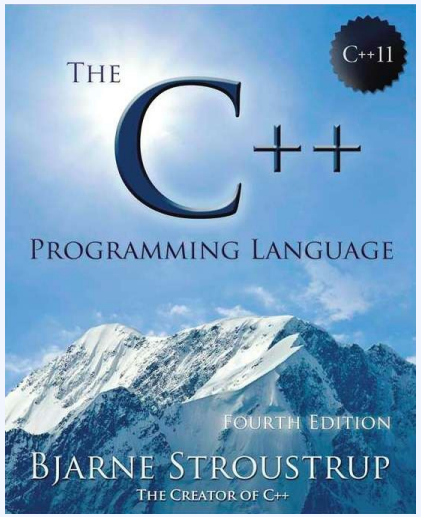
5) Parallel and distributed data: Concurrency and efficiency in dealing with data.

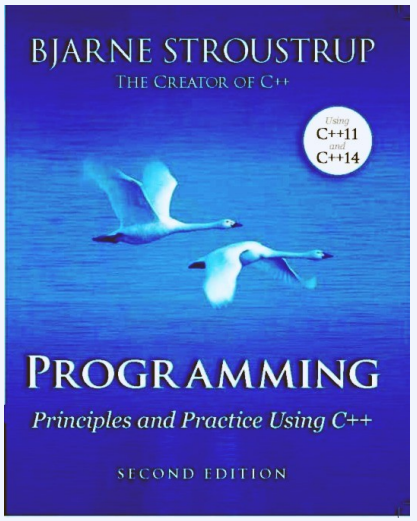


Literature

Stroustrup's C++ books:







compact, best for people with programming knowledge

Resources on modern C++ programming style:

CORE GUIDELINES

- *C++ Core Guidelines*, https://github.com/isocpp/CppCoreGuidelines
- R. Grimm, *C++ Core Guidelines Explained*, Pearson, **2022**

Graded programming project

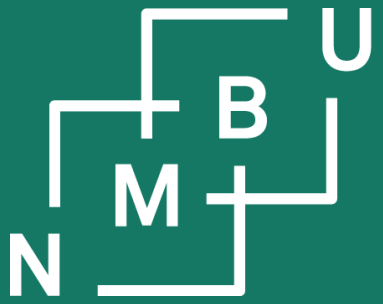
Choice between a **robotics-related** and a **scientific computing** problem.

The programming-project group work is evaluated and graded in two parts:

- Handed-in source code and documentation (70%);
- Presentation of the project with discussion (30%).

Projects should be done by **groups of three** participants jointly; **groups of two** are also possible. Grades will be individualized based on a clearly designated split of responsibilities between group members.

Coding group work is carried out from **week 43 to week 48**. Presentations and discussions are held in week 49; depending on the number of submissions and scheduling, this period may extend into week 50.



Noregs miljø- og biovitskaplege universitet

1 C++ basics

1.1 Scope of this module

1.2 C/C++: Getting started

The “main” function

```
#include <iostream>
using namespace std;

bool is_prime(int n)
{
  for(int i = 2; n >= i*i; i++)
  {
    if((n % i) == 0) return false;
  }
  return true;
}

int main()
{
  int x = 900;
  if(is_prime(x)) cout << x << " is prime.\n";
  else cout << x << " is not prime.\n";
}
```

```
def is_prime(n):
    for i in range(2, 1 + int(n**0.5)):
        if n%i == 0:
            return False
    return True

x = 900
if is_prime(x):
    print(x, "is prime.")
else:
    print(x, "is not prime.")
```

What does the program do?

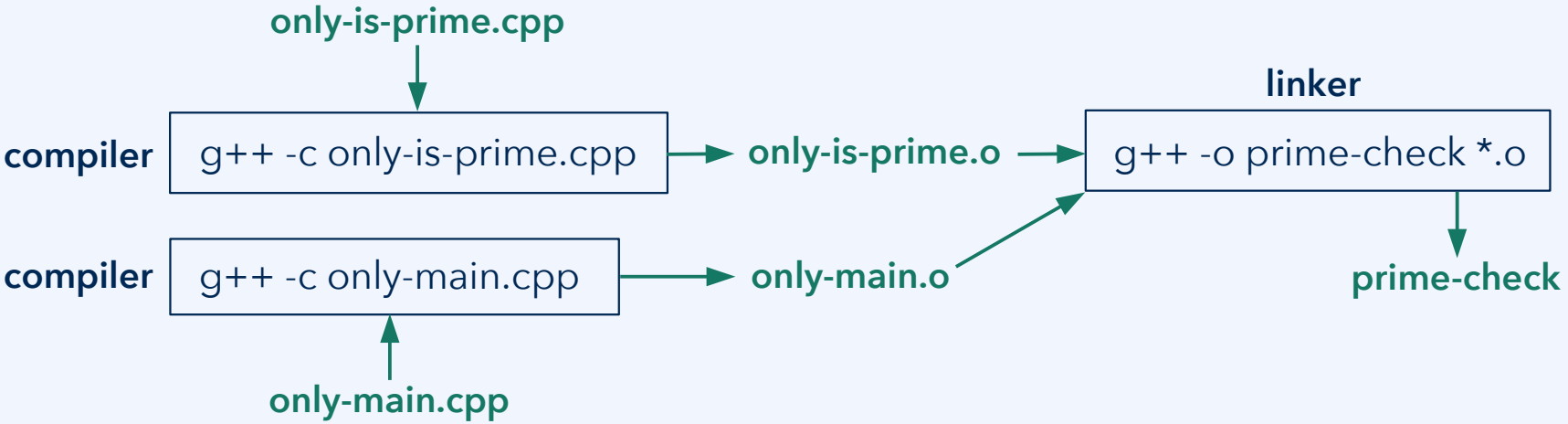
What is the role of “main”?

C++ as a compiled language

Compile the code from the previous slide (file name: prime-check.cpp), using the GNU C++ compiler: **g++ prime-check.cpp -o prime-check**

Alternatively, in a Linux environment, we have GNU make: **make prime-check**

Normally, codes comprise **multiple code files**. They are compiled separately (creating object files), and then linked. Only after linking there is an executable file. With the GNU C++ compiler, g++ is called both as **compiler** and **linker**:



Makefiles and GNU make

GNU make operates on instructions in a file that must be called Makefile.

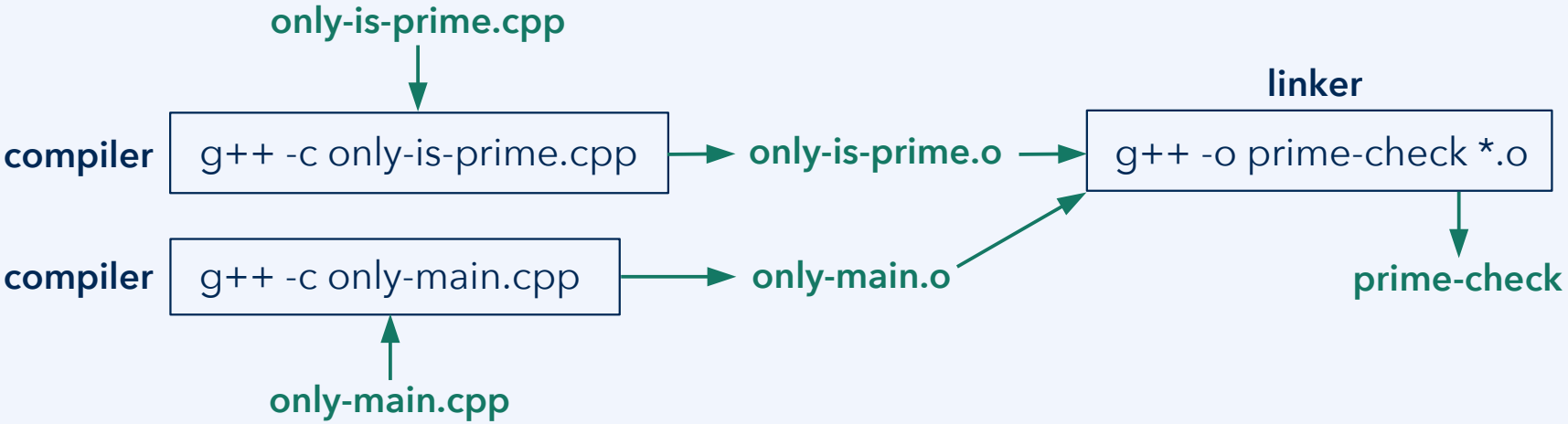
```
target-1: requirements
	instruction

target-2: requirements
	instruction
…
```

call *e.g.* "**make target-2**"

Makefile

use **tab** for indenting

```
prime-check: only-main.o only-is-prime.o
	g++ -o prime-check *.o

clean:
	rm *.o prime-check
```



A brief demo

```
#include <iostream>
using namespace std;

bool is_prime(int n)
{
  for(int i = 2; n >= i*i; i++)
  {
    if((n % i) == 0) return false;
  }
  return true;
}

int main()
{
  int x = 900;
  if(is_prime(x)) cout << x << " is prime.\n";
  else cout << x << " is not prime.\n";
}
```

Let us split this code into two code files, one for each of the functions.

How does main then know about is_prime at compile time? The **declaration**

bool is_prime(int n);

can be split from the **definition**:

bool is_prime(int n) { … }

The code file only_main.cpp only needs to contain the declaration. Then is_prime() can be called from within main().

A brief demo: Header files

```
#include <iostream>
using namespace std;

bool is_prime(int n)
{
  for(int i = 2; n >= i*i; i++)
  {
    if((n % i) == 0) return false;
  }
  return true;
}

int main()
{
  int x = 900;
  if(is_prime(x)) cout << x << " is prime.\n";
  else cout << x << " is not prime.\n";
}
```

Let us split this code into two code files, one for each of the functions.

How does main then know about is_prime at compile time? The **declaration**

bool is_prime(int n);

can be split from the **definition**:

bool is_prime(int n) { … }

This sort of declarations are commonly stored in separate "interface" or **"header" files** with the ending ".h". In this way, the header can be included by all external code that requires the same declarations.

Discussion: Resource efficiency

Usually we are not interested in the resource requirements of a single execution, but in understanding how the requirements behave as a function of a characteristic quantity, the **problem size *n***, that describes the magnitude of the task.

We distinguish between:

- **Time requirements**, describing the computing time. Where possible, this should be expressed in terms of actual CPU time (+ I/O time); the operating system will usually distribute CPU time between multiple processes.
- **Memory (or space) requirements**, describing the memory allocated to the program; depending on definition, this may include I/O size.
- **Worst-case performance**, which for any given problem size n corresponds to the input/special case of size n with the greatest requirements.
- **Average-case performance**, over many representative cases of size *n*.

Metrics closer to the hardware (*e.g.*, energy consumption) can also be relevant.

Discussion: Resource efficiency

Observations:

- **Performance analysis** is carried out by measurements; it is usually very hard to determine the worst case, therefore it is common to describe the **average-case performance**, *e.g.*, from random input.
- **Algorithm efficiency** can consider both the average and the worst case, but the average case usually requires a statistical analysis. Statements on the **worst case** can be very straightforward.
- There is no universal rule for how the **problem size *n*** should be defined. It is up to the person analysing an algorithm to define it appropriately. It should describe how complicated the task is.
- Distinguish between the efficiency of a program (or algorithm) and the complexity of the problem. The **complexity of the problem** is given by the **efficiency of the best possible program** (or algorithm).

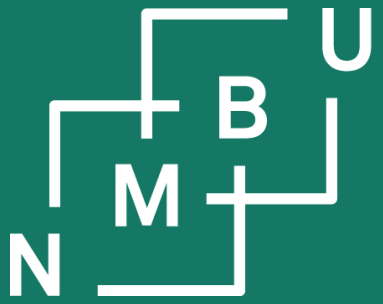
Discussion: Resource efficiency

```
#include <iostream>
using namespace std;

bool is_prime(int n)
{
  for(int i = 2; n >= i*i; i++)
  {
    if((n % i) == 0) return false;
  }
  return true;
}

int main()
{
  int x = 900;
  if(is_prime(x)) cout << x << " is prime.\n";
  else cout << x << " is not prime.\n";
}
```

```
def is_prime(n):
    for i in range(2, 1 + int(n**0.5)):
        if n%i == 0:
            return False
    return True

x = 900
if is_prime(x):
    print(x, "is prime.")
else:
    print(x, "is not prime.")
```

How would you describe the **time efficiency** (or time requirements, or performance) of the function is_prime(n), as a function of n?

Compare: What would you say about the complexity of the problem?



Noregs miljø- og biovitskaplege universitet

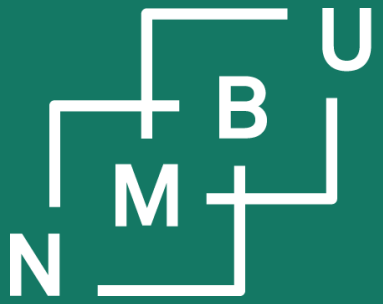
1 C++ basics

Analogies and differences

```
#include <iostream>
using namespace std;

bool is_prime(int n)
{
  for(int i = 2; n >= i*i; i++)
  {
    if((n % i) == 0) return false;
  }
  return true;
}

int main()
{
  int x = 900;
  if(is_prime(x)) cout << x << " is prime.\n";
  else cout << x << " is not prime.\n";
}
```

```
def is_prime(n):
    for i in range(2, 1 + int(n**0.5)):
        if n%i == 0:
            return False
    return True

x = 900
if is_prime(x):
    print(x, "is prime.")
else:
    print(x, "is not prime.")
```

Let us gather as much as we can from our simple example: **What do C++ and Python syntax have in common? What is different?**



Noregs miljø- og biovitskaplege universitet

1 C++ basics

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

Function syntax

```
// declaration:
ret_type function_name(argtype_a argname_a, argtype_b argname_b, ...);

// definition:
ret_type function_name(argtype_a argname_a, argtype_b argname_b, ...)
{
    ...
    return return_value; // must be of type ret_type
}
```

Function overloading:

Multiple versions of a function (named equally) with different argument types:

```
// takes an integer argument
//
void print(int n) { ... }
```

```
// takes a string argument
//
void print(string str) { ... }
```

Function syntax

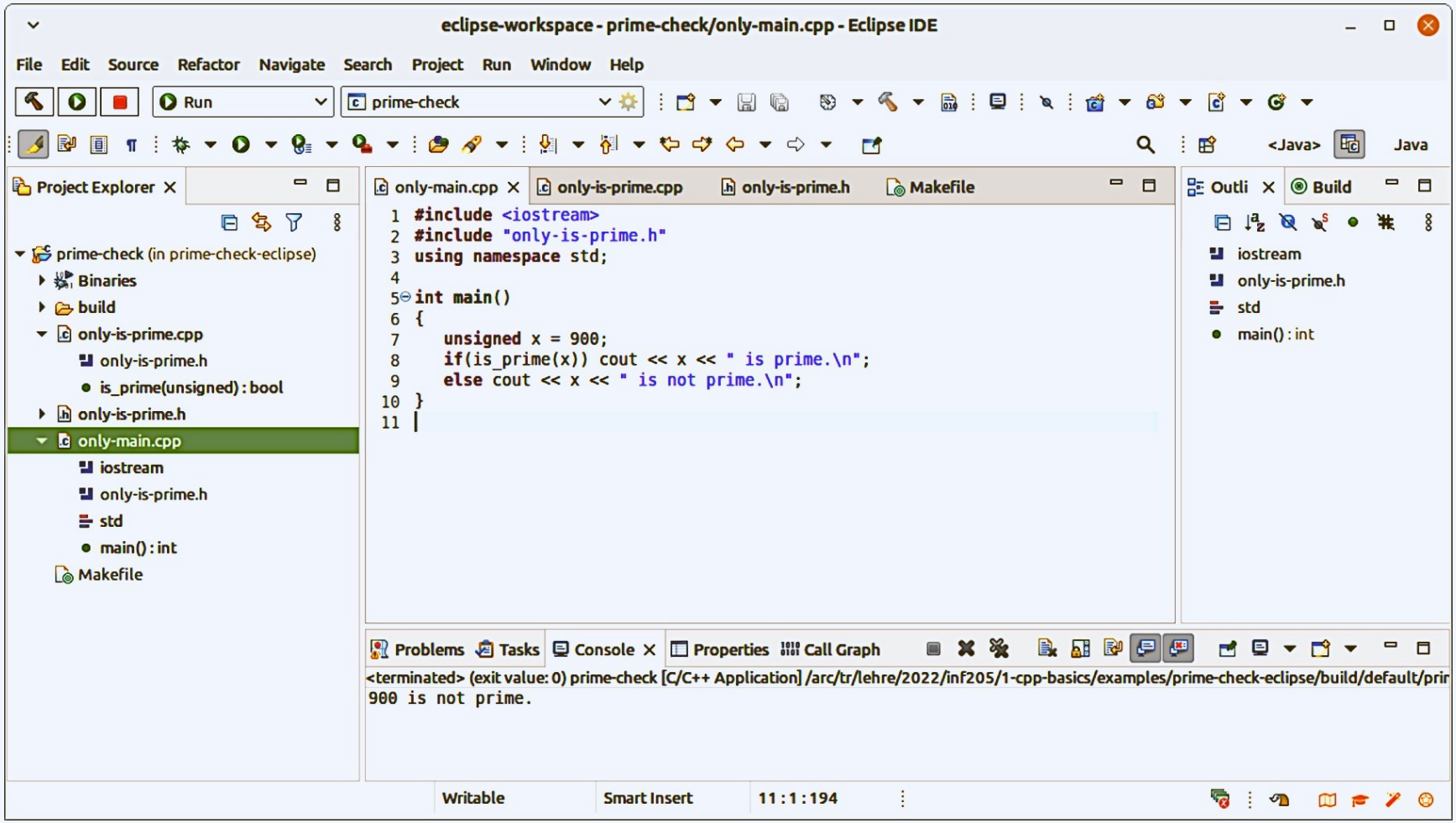
```
// declaration:
ret_type function_name(argtype_a argname_a, argtype_b argname_b, ...);

// definition:
ret_type function_name(argtype_a argname_a, argtype_b argname_b, ...)
{
   ...
   return return_value; // must be of type ret_type
}
```

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()

Eclipse IDE



https://www.eclipse.org/downloads/packages/release/2022-09/rc1/eclipse-ide-cc-developers/

Our practices in INF205

Tutorial/øving sections

- #1, Thursdays, 10.00 – 12.00, in TF1-105
- #2, Thursdays, 12.00 – 14.00, in TF1-105

We have 85 students in INF205, the size of room TF1-105 is limited …

- It is necessary for all to split up fairly evenly into the two sections.

Registration

Use self-signup functionality under "Groups" on Canvas.

Limit: 44 each.

Our practices in INF205

Recordings

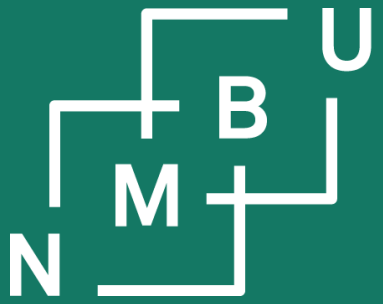
Do we want to have the lecture recorded?
What is your **experience with recorded lectures** at NMBU?

OS and installations

Who is already normally working by default under **Linux**? (Or other Unix.)
Who **cannot** at least work with a dual boot system or using Linux on a VM?
We will figure out the best solutions on a case-by-case basis in the tutorial.

Group formation and robotics vs. HPC

Who has a robotics background and has worked with ROS? Who has not?
Who prefers a project task from **robotics**, who prefers **scientific computing**?
Best already start working together in groups that are aligned on this question.

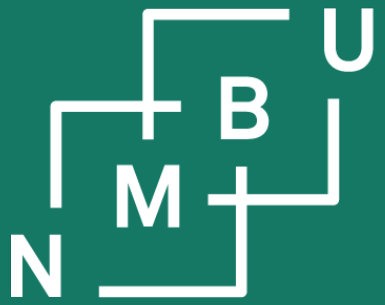


Noregs miljø- og biovitskaplege universitet

Conclusion

INF205

7th September 2022



Norges miljø- og biovitenskapelige universitet

INF205
Resource-efficient programming

I C++ basics

I.1 Scope of the module
I.2 C/C++: Getting started
I.3 Discussion: From Python to C and C++
I.4 Good practice and style