Norges miljø- og biovitenskapelige universitet

Institutt for datavitenskap

Digitalisering på Ås

INF205 Resource-efficient programming

1 Introduction

INF205 learning outcomes

After completing the course you will be able to

- implement solutions in modern C++;
- manage memory safely;
- make use of capabilities provided by the C++ Standard Library and third-party libraries;
- implement data types from "first principles;"
- assess programs and their use in terms of sustainability metrics;
- write code suitable for embedded systems and high-performance computing;
- 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 of the course

1) **Introduction** (week 6)
   - Getting started – the lecture today.

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

Institutt for datavitskap

Digitalisering på Ås

1 Introduction

1.1 Why C++?

Norwegian University
of Life Sciences

Resource efficient computing: Why?

"What comes after Moore's law?"

[redacted]

P. J. Denning, T. G. Lewis, doi:10.1145/2976758, **20**[redacted]

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

↓

therein, see Tab. 1

Embedded systems

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

C++ vs. Python: Programming language features

C++	Python

"What do you know about language features that are different in C++ and Python?"

C++ vs. Python: Compare the codes

C++ code

```cpp
#include <iostream>

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))
    std::cout << x << " is prime.\n";
  else std::cout << x << " is not prime.\n";
}
```

Python code

```python
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.")
```

These two codes are equivalent.

What does the program do?

Example file: is-prime-900.cpp

Corrigendum: This is what it should have looked like

C++ code

```
#include <iostream>

bool is_prime(int n)
{
  if(i < 2) return false;
  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))
    std::cout << x << " is prime.\n";
  else std::cout << x << " is not prime.\n";
}
```

Python code

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

                           e.")

  print(x, "is not prime.")
```

This should have looked like here, to catch the case where n is smaller than two. Apologies for not thinking of it earlier. Adding this slide now.

Example file: is-prime-900.cpp

Noregs miljø- og biovitskaplege universitet

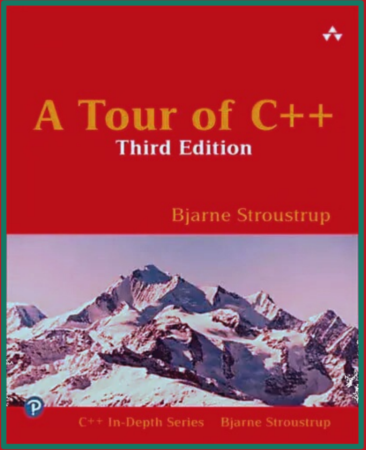
Institutt for datavitskap

Digitalisering på Ås

1 Introduction

1.1 Why C++?

1.2 C/C++ compiler



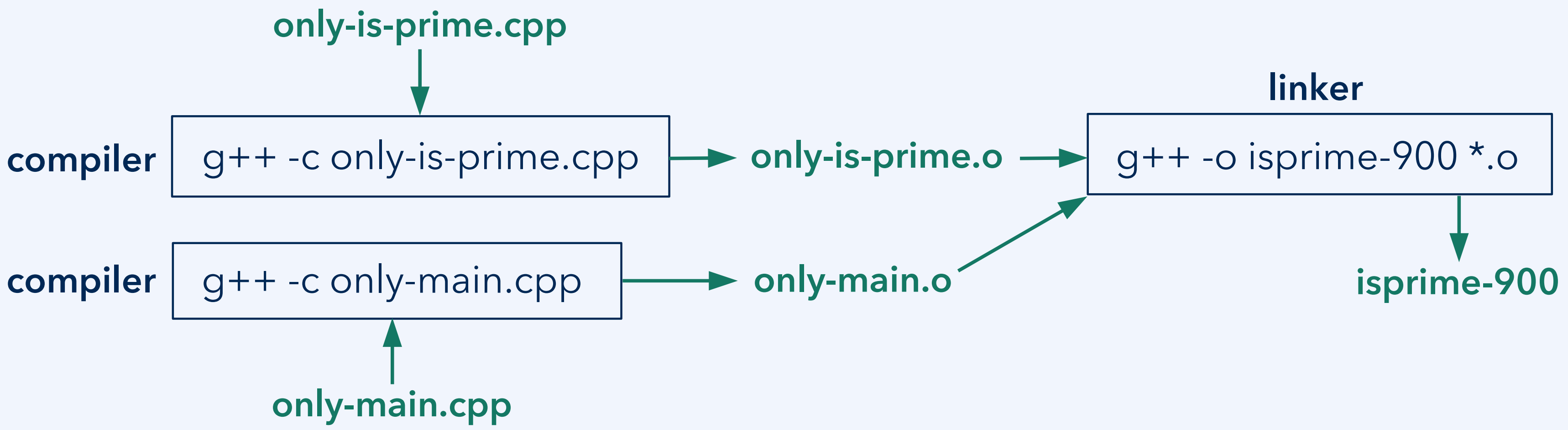
book Section 1.2

C/C++ as a compiled language

Compile the code from the previous example (file name: isprime-900.cpp), using the GNU C++ compiler: **g++ isprime-900.cpp -o isprime-900**

Alternatively, in a Linux environment, we have GNU make: **make isprime-900**

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**:



Example file: is-prime-separate-files.zip

Split into header files (*.h) and code files (*.cpp)

```
#include <iostream>

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))
    std::cout << x << " is prime.\n";
  else std::cout << x << " is not prime.\n";
}
```

Before, we split the code into two code files, one for each function.

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

bool is_prime(int n);

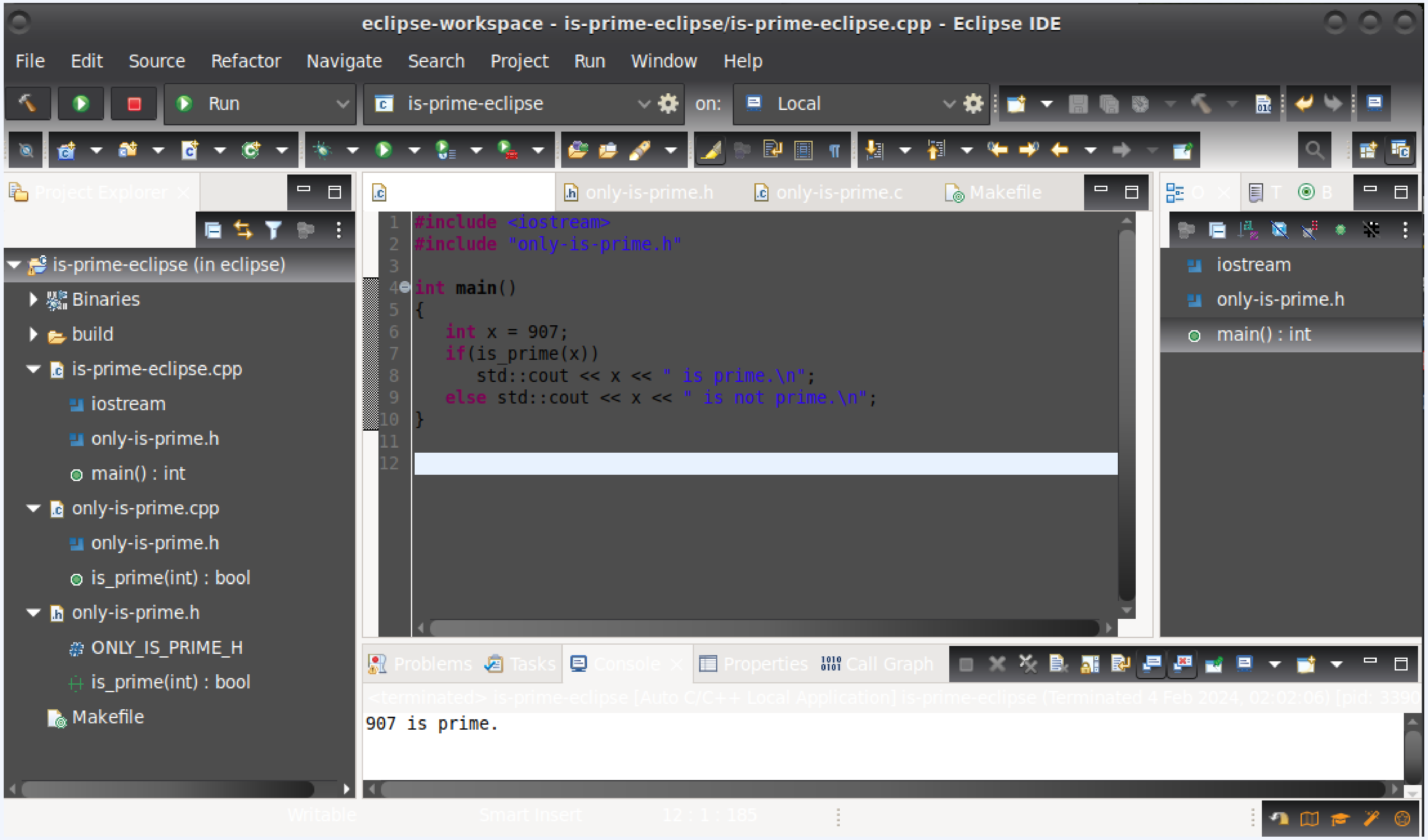
must be split from the **definition**:

bool is_prime(int n) { … }

Such declarations are normally stored in **header files** with the ending ".h". In this way, the header can be included by all external code that requires the same declarations.

Example file: is-prime-three-files.zip

IDE example: Eclipse



```
#include <iostream>
#include "only-is-prime.h"

int main()
{
    int x = 907;
    if(is_prime(x))
        std::cout << x << " is prime.\n";
    else std::cout << x << " is not prime.\n";
}
```

```
<terminated> is-prime-eclipse [Auto C/C++ Local Application] is-prime-eclipse (Terminated 4 Feb 2024, 02:02:06) [pid: 3390
907 is prime.
```

https://www.eclipse.org/downloads/packages/release/2024-03/m2/eclipse-ide-cc-developers

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap

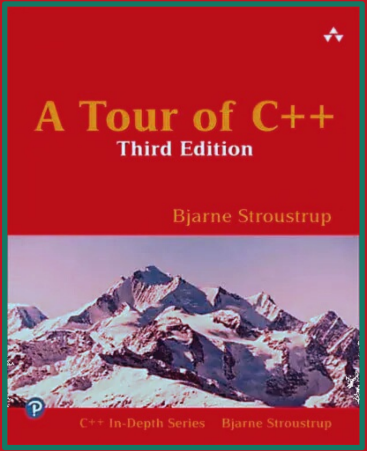
Digitalisering på Ås

1 Introduction

1.1 Why C++?

1.2 C/C++ compiler

1.3 From Python to C/C++



1.3, 1.4, 1.5, 1.8

What differences can we see?

C++ code

```
#include <iostream>

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))
    std::cout << x << " is prime.\n";
  else std::cout << x << " is not prime.\n";
}
```

Python code

```
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 **differences between the languages** can we recognize from the introductory example?

C/C++ is a statically typed language

Most compiled programming languages are **statically typed** languages: The **data type** of each variable must be known to the compiler, at compile time.

Therefore, the type of a variable must be given when the variable is declared.

float, double

- single-precision and double-precision floating-point numbers

int

- the default signed integer type

short (int), long (int), long long (int)

- less/more memory and smaller/larger range of values

unsigned, unsigned short (int), unsigned long (int), …

- holds natural number (or zero); modulo-arithmetic applies: $-n = 2^k - n$

bool

- integer-like; *meant to* hold the value **false** (0) or **true** (1, or any value ≠ 0)

char, wchar_t

- integer-like; *meant to* hold a ASCII (char) or Unicode (wchar_t) character

Functions require argument types and a return type

```
// 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
}
```

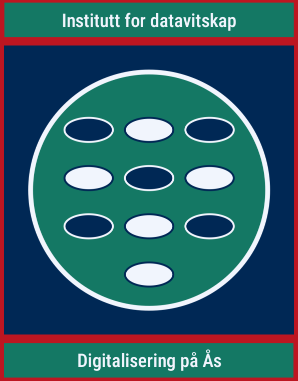
Function overloading:

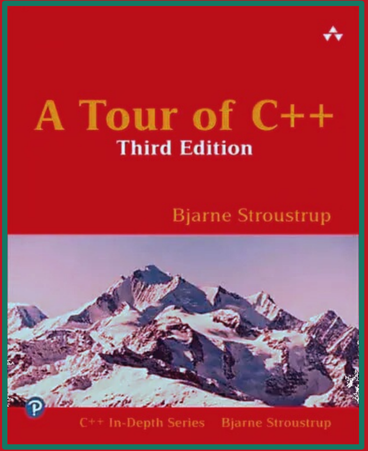
Multiple versions of a function (named equally) with different argument types:

```
// takes an integer argument
//
void do_something(int n) { … }
```

```
// takes a floating-point argument
//
void do_something(double x) { … }
```

Noregs miljø- og biovitskaplege universitet



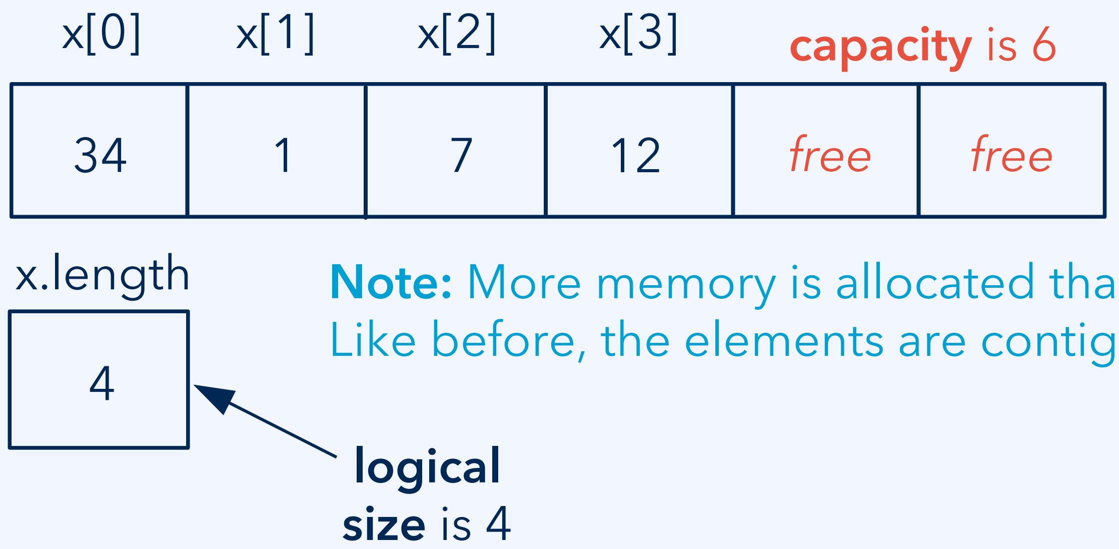


book Section 1.7

Dynamic arrays (such as lists in Python)

An array is a sequence of data items of the same type that is contiguous in memory. Python lists are contiguous in memory, *i.e.*, they are arrays. They can also grow or shrink in size over time: **Python lists are dynamic arrays**.

Dynamic data structures can change in size and/or structure at runtime. For an array, this can be implemented by **allocating reserve memory** for any elements that may be appended in the future. When the capacity of the dynamic array is exhausted, all of its contents need to be shifted to another position in memory.



x = [34, 1, 7, 12]

Note: More memory is allocated than strictly necessary.
Like before, the elements are contiguously arranged in memory.

Static arrays

Arrays in C/C++ are static: When declaring the array, the array size is specified and the exact amount of memory required for these data items is allocated. The array size does not change over time.

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
34	1	7	12	3	4	7	12

Accessing elements of an array is highly efficient: When x[i] is accessed, the compiler transforms this into accessing the memory address x + sizeof(int) * i.

How do we declare a array?

- Give the size as constant expression in square brackets; *e.g.*, `int values[6];`

How do we initialize an array?

- Explicitly give all the values: `int values[ ] = {4, 2, 3, -7, 2, 3};`
- Initialize to **all zeroes**, indicating the array size: `int values[6] = { };`

Character arrays: The way of defining a string in C

In C++, there is an explicit std:string datatype. But since C++ is backwards compatible to C, there is also the more traditional string type: The char array.

string s = "INF205"; or **char s[] = "INF205";** produce the following in memory:

'I'	'N'	'F'	'2'	'0'	'5'	'\0'
73	78	70	50	48	53	0

Note that while the string length above is six, one more is allocated in memory. The array has seven elements: It ends with the **null character '\0'**.

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';**

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap

Digitalisering på Ås

1 Introduction

Mandatory activities in INF205

The grade from INF205 will be determined from a programming project. (Work on this is to begin from calendar week 12.)

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.

Mandatory activities are required to pass, but don't contribute to the grade.

First tutorial worksheet

The first tutorial worksheet (deadline 13.2.) has seven problems:

1. Basic tools.
2. From C++ to Python.
3. From Python to C++.
4. Size of the primary data types, in bytes.
5. Using C/C++ arrays.
6. Program analysis – termination of a recursive function.
7. Program analysis – return value of a function.

It is published on Canvas and on the INF205 course website.

How do we proceed?

For the first worksheet, we will proceed as follows:

- Monday, 5th February: The worksheet was introduced at today's lecture.
- Wednesday, 7th February: Tutorial session for working on the problems.
- Monday, 12th February: Assigning presentation slots (during lecture time). If they cannot all be assigned at lecture time, ASAP thereafter.
- Tuesday, 13th February: Submission deadline.
- Wednesday, 14th February: Presentation of the solutions at the tutorial.

Hopefully, this process works well and can be followed for all five worksheets.

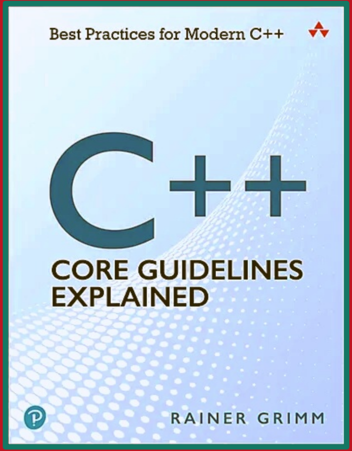
Note: There is only one tutorial group, Wednesdays, 14.15 – 16.00, TF1-105.
(Intially another, two hours earlier, had been announced. There is only one!)

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap

Digitalisering på Ås

1 Introduction



Appendix C

Program flow graphs

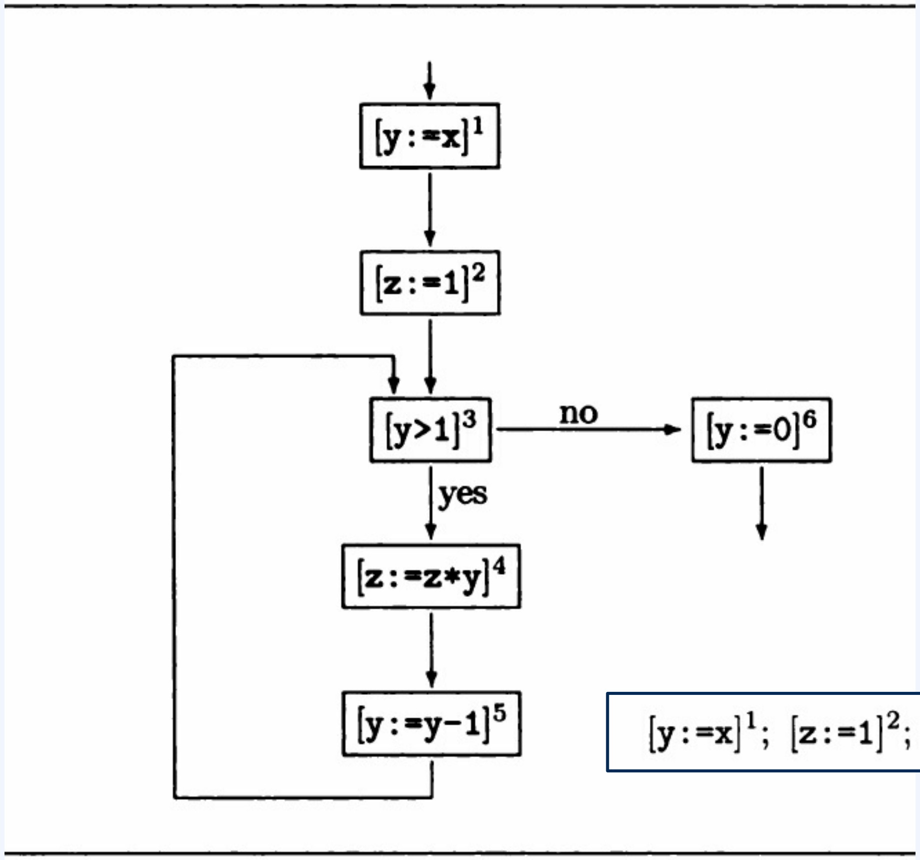


Figure 1.2: Flow graph for the factorial program.

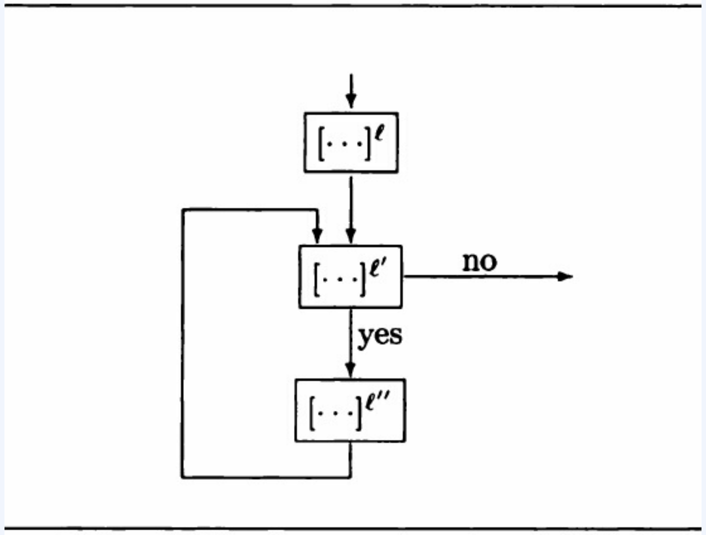


Figure 2.2: A schematic flow graph.

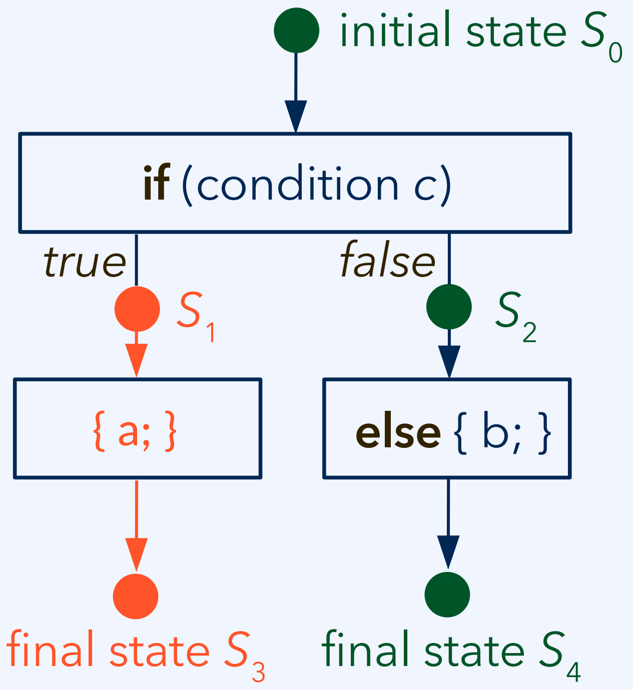
$[\mathtt{y:=x}]^1$; $[\mathtt{z:=1}]^2$; **while** $[\mathtt{y>1}]^3$ **do** ($[\mathtt{z:=z*y}]^4$; $[\mathtt{y:=y-1}]^5$); $[\mathtt{y:=0}]^6$

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

Preconditions and postconditions

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.



Note

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

- The execution state S_1 is the **precondition** of statement a.
- The execution state S_3 is the **postcondition** of statement a.

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

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)

"contracts" between specifier, designer, and programmer

Norges miljø- og biovitenskapelige universitet

Institutt for datavitenskap

Digitalisering på Ås

INF205
Resource-efficient programming

1 Introduction