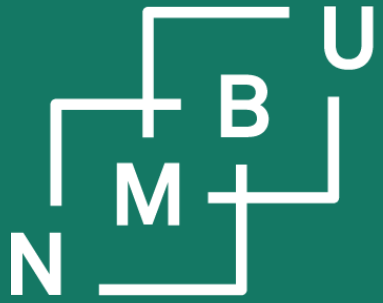
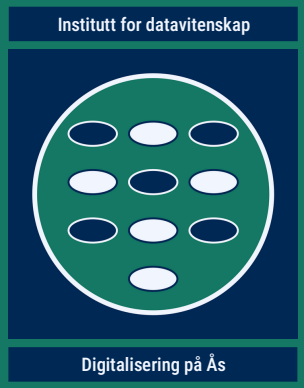


Norges miljø- og biovitenskapelige universitet



INF205
Resource-efficient programming

1 C++ basics

INF205 learning outcomes

Norwegian University of Life Sciences

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 concurrent processes and 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 of the course

1) C++ basics (week 6)

- Getting started – the lecture today
- Procedural programming in C++ (not much different from Python)

2) Memory and objects (week 7)

- Direct access to memory addresses, allocating and deallocating memory
- Object oriented programming in C++ (somewhat different from Python)

3) Data structures and libaries (weeks 8 and 9)

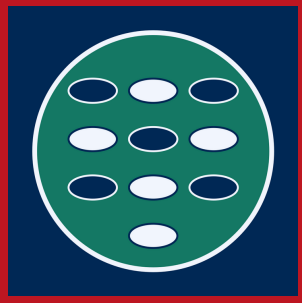
- Containers (incl. lists, graphs), standard template library, other libraries
- Memory management for container data structures

4) Concurrency (week 10 to 11)

- Concurrent process models and paradigms of parallel programming
- Using the ROS2 robot operating system from within C++

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap



Digitalisering på Ås

1 C++ basics

1.1 Why C++

1.2 From Python to C++

1.3 Static arrays

INF205

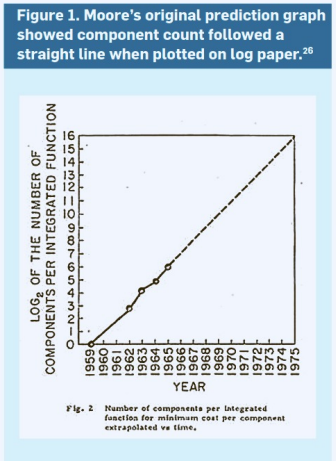
4th February 2025

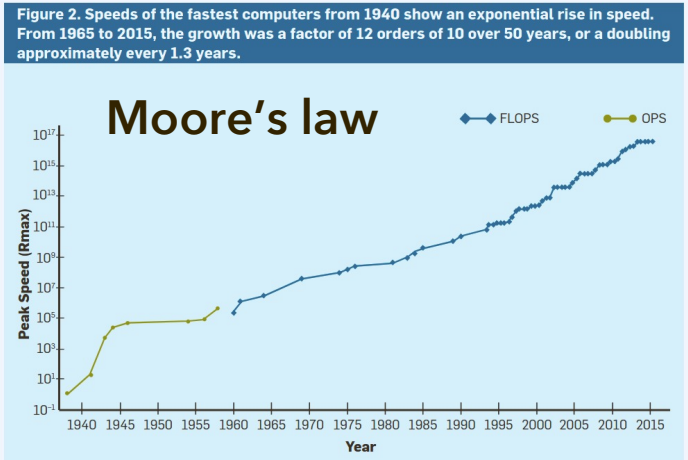
C++ vs. Python: Language features

C++	Python

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

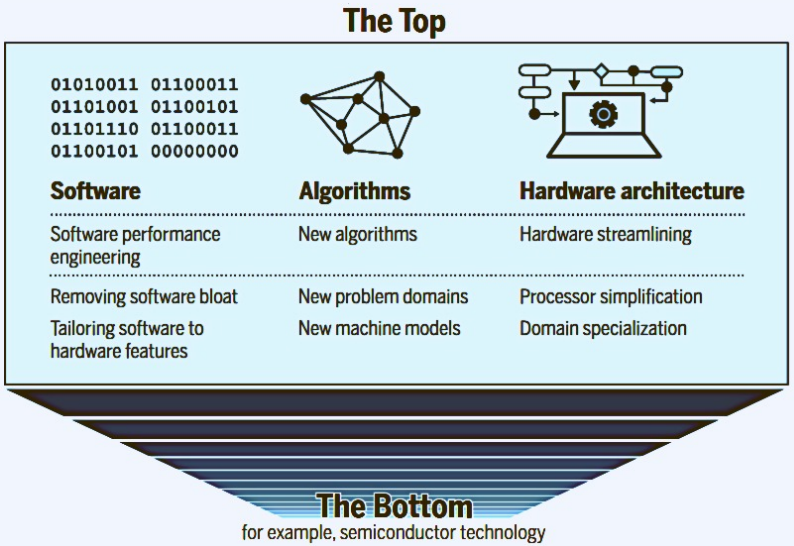
Resource efficient computing: Why?





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

"What comes after Moore's law?"



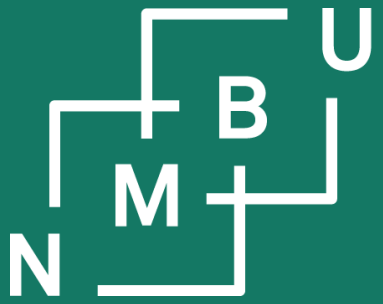
Performance gains after Moore's law ends. In the post-Moore era, improvements in computing power will increasingly come from technologies at the "Top" of the computing stack, not from those at the "Bottom", reversing the historical trend.

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

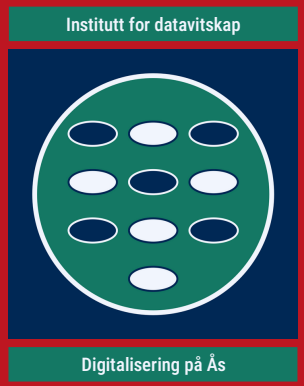
Embedded systems

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

therein, see Tab. 1



Noregs miljø- og biovitskaplege universitet



1 C++ basics

INF205

4th February 2025

Comparison between two simple codes

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

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

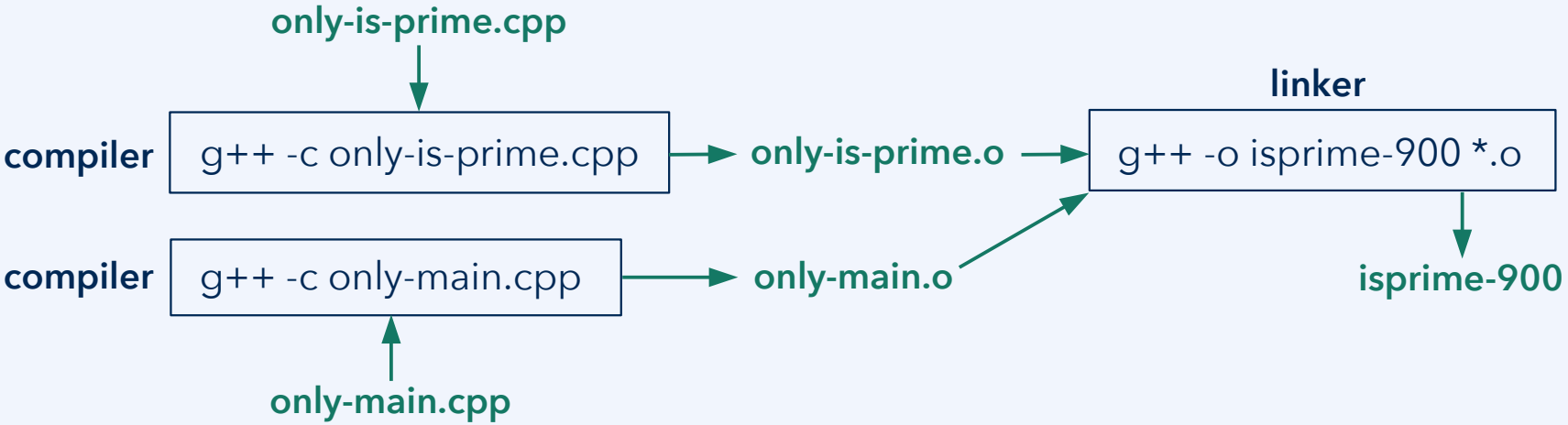
Example file: is-prime-900.cpp

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)
{
        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";
}
```

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

What differences can we see?

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

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:

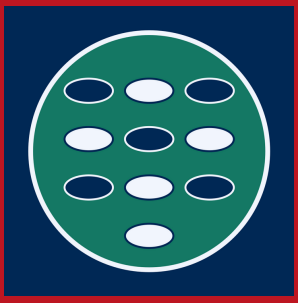
Multiple versions of a function (named equally) with different argument types:

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

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

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap



Digitalisering på Ås

1 C++ basics

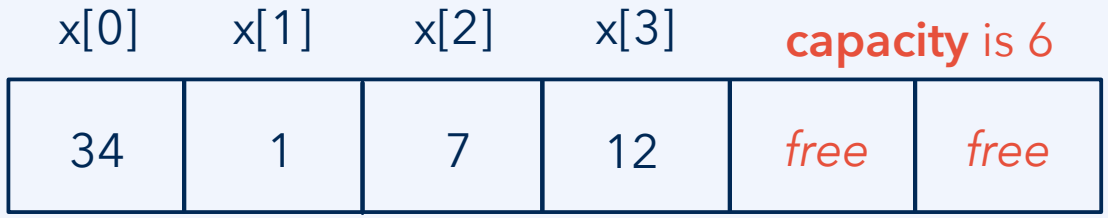
INF205

4th February 2025

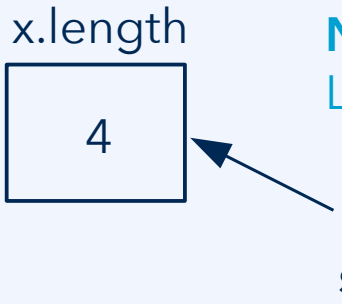
Dynamic arrays (such as lists in Python)

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.



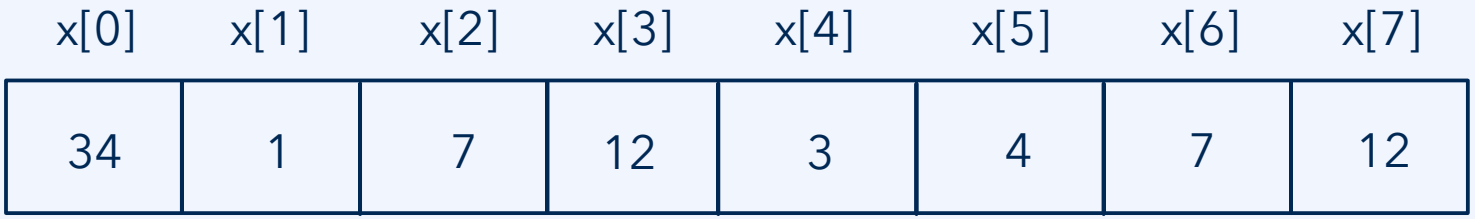
x = [34, 1, 7, 12]



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.



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];

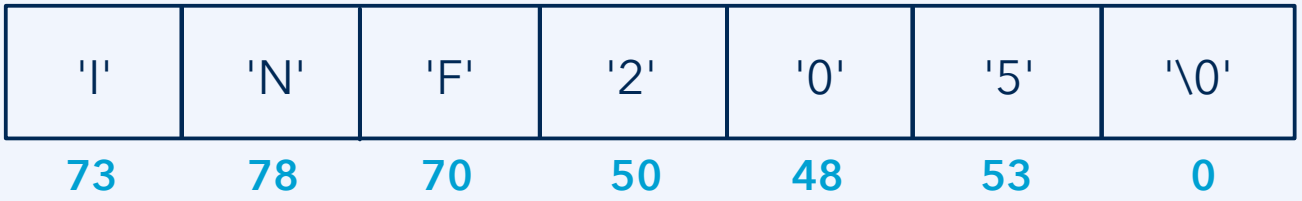
How do we initialize an array?

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

C strings: Character arrays

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:

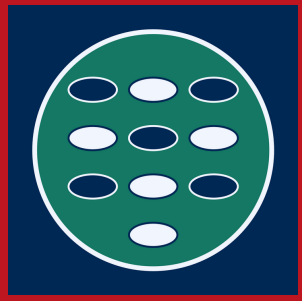


Note that while the string length above is six, one more is allocated in memory. The array has seven elements: It ends with the **null character '\0'**.

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



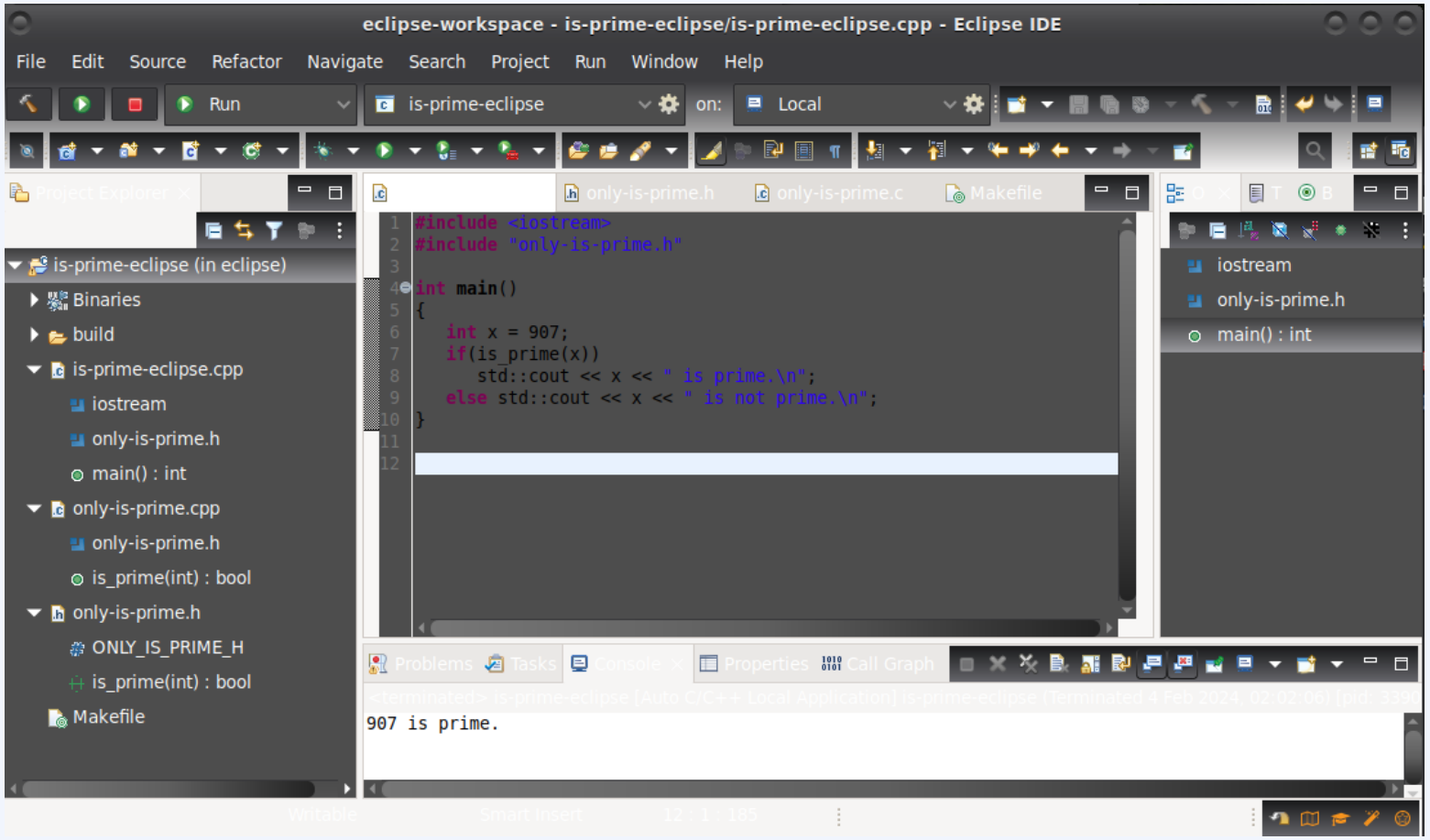
Digitalisering på Ås

Conclusion and discussion: How to get started? And how does the INF205 course work?

INF205

4th February 2025

IDE example: Eclipse



https://www.eclipse.org/downloads/packages/release/2024-12/r/eclipse-ide-cc-developers

GNU make

2.2 A Simple Makefile

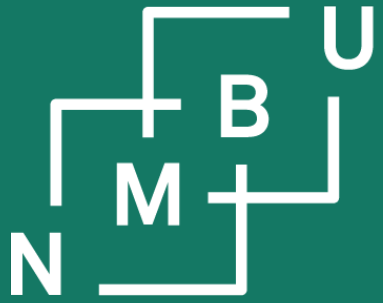
Here is a straightforward makefile that describes the way an executable file called `edit` depends on eight object files which, in turn, depend on eight C source and three header files.

In this example, all the C files include `defs.h`, but only those defining editing commands include `command.h`, and only low level files that change the editor buffer include `buffer.h`.

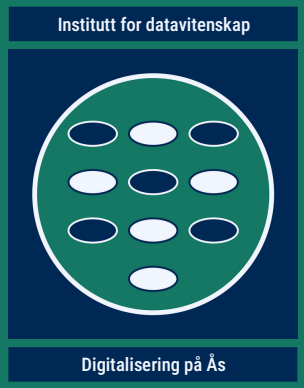
```
edit : main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o \
                   insert.o search.o files.o utils.o

main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit main.o kbd.o command.o display.o \
           insert.o search.o files.o utils.o
```

(from GNU make documentation)



Norges miljø- og biovitenskapelige universitet



INF205
Resource-efficient programming

1 C++ basics