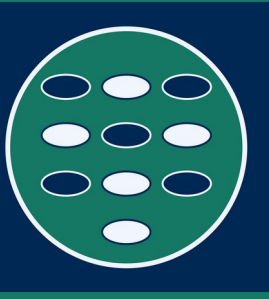
Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap



Digitalisering på Ås

IND205

Ressurseffektiv programmering

1 Moderne C++

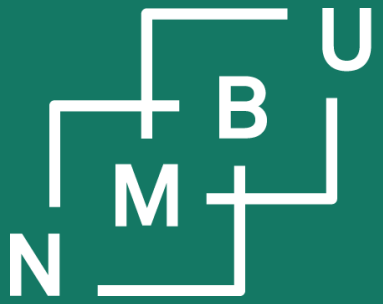
1.1 Frå Python til C++

1.2 Peikarar og datatabellar

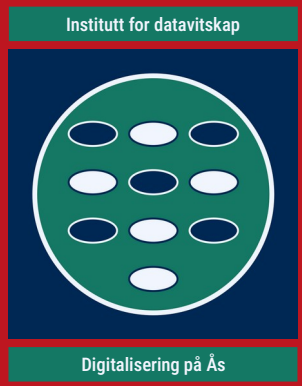
1.3 Programmeringsparadigme

1.4 God praksis i moderne C++

1.5 Moderne C++ og gamaldags C/C++



Noregs miljø- og biovitskaplege universitet



1 Moderne C++

INF205

12. februar 2026

Ordliste

Command-line argument	Kommandolineargument
Constant expression	Konstantutrykk
Global variable	Global variabel
Scope	Gyldigheitsområde
String	Streng
Template	Templat

I/O operator overloading

See example code **io-operator-overloading.zip** for the following.

Assume that for some **class C**, we have defined methods that write content to a stream, or that analogously read from a stream.

```
void C::out(ostream* target) const {
  *target << … ;
}
```

```
void C::in(istream* source) {
  *source >> … ;
}
```

You can convert this to overloaded I/O operator definitions:

```
ostream& operator<<(
  ostream& str, const C& x
) const {
  x.out(&str);
  return str;
}
```

```
istream& operator>>(istream& str, C& x)
{
  x.in(&str);
  return str;
}
```

Now you can use the operator **<<** and the operator **>>** on objects of type C just like for numbers, *etc.*

Advice: Input & output methods/operators should use the same serialization.

File input/output

We **must serialize the data** in order to store them in a file!

To transfer data through a communication channel as a message, the data items and their parts need to be serialized (ordered) in a well-defined way that is understood both by the sender and the receiver.

- As a contiguous chunk of memory, *if the exchange is memory-based*.
- As a file, *if file I/O is the mechanism* by which data are exchanged.

File stream objects can be used in order to read or write a file.

```
// open in-filestream
std::ifstream infile(argv[1]);      // file name given as command-line argument argv[1]
```

The keywords auto, const, constexpr

auto: Leave it to the compiler to determine the type

This requires an initialization.

Remark:

typeid(x).name() can be used to output the type assigned to x.

```
for (auto i = 0; i < 26; i++)
{
  auto c = 'a';
  c += i;
  cout << c;
}
```

should become **int**

should become **char**

const: Used to declare an immutable variable

constexpr: Immutable and, additionally, can be evaluated at compile time

```
constexpr int space_dimension = 3;  int n = 0;  cin >> n;  const int num_coords = n*space_dimension;
```

Con.1: By default, make objects immutable

"make objects non-const only when there is a need to change their value"

Con.4: Use **const** to define objects with values that do not change

Con.5: Use **constexpr** for values that can be computed at compile time

Example file: **explicit-implicit-typing.cpp**

“const” parameters of a function

const-array-broken.cpp – does not compile

If we pass an argument by reference but do not intend to modify it, the parameter should be declared as **const**. Such as:

```
void do_something(const int N);
void do_something(const int* const x);
void do_something(const int x[]);
```

Const variables may only be passed by reference if the parameter is also const.

1. What is the code supposed to do?
2. Why does it not compile?
3. What should be changed?
4. What more const/-expr can we add?

```
int second_of(int N, int* x) {
        int largest = x[0];
        int second_largest
                = std::numeric_limits<int>::min();

        for(int i = 1; i < N; i++)
                if(x[i] > largest) {
                        second_largest = largest;
                        largest = x[i];
                }
                else if(x[i] > second_largest)
                        second_largest = x[i];
        return second_largest;
}

int main() {
   int fixed_array_size = 5;
   const int x[fixed_array_size] = {4, 0, 6, 5, 2};
   int t = second_of(fixed_array_size, x);
}
```

Making proper use of const and constexpr

const-array-fixed.cpp – fixed as follows:

If x[] in main() is a const int array, or even "constexpr" (which is stronger than "const"), we must make the x parameter in second_of() const int*.

1. Variables are declared as const if we do not plan to modify them after initialization. If their value can be determined at compile time, we can even use "constexpr".

2. We declare pointers to const (of type T) as const T*. (References as const T&.)

3. Pointers that are constant (*i.e.*, have as value an address that cannot be changed) are of the type T* const. This can can be combined with the above: const T* const.

```
int second_of(const int N, const int* const x) {
        int largest = x[0];
        int second_largest
                = std::numeric_limits<int>::min();

        for(int i = 1; i < N; i++)
                if(x[i] > largest) {
                        second_largest = largest;
                        largest = x[i];
                }
                else if(x[i] > second_largest)
                        second_largest = x[i];
        return second_largest;
}

int main() {
  constexpr int fixed_array_size = 5;
  constexpr int x[fixed_array_size] = {4, 0, 6, 5, 2};
  const int t = second_of(fixed_array_size, x);
}
```

“const”, pass by reference, and const pointers

1) If you can pass by value, that is always to be preferred!
2) If you pass an argument by reference, the compiler assumes that the function will modify it. Write “const” whenever that’s not the case.

An array is a pointer. Therefore **it is impossible to pass an array by value**. If you don’t intend the function to write to the array, it should be a const parameter.

Pay attention to C++ syntax for combining pointers with “const”. Illustration:

```
int v = 3;
const int x[3] = {1, v, v*v};      // x is an array of constant integers
const int* y = &x[1];              // y is a pointer to a constant integer
int* const pv = &v;                // pv will forever point to address of v
const int* const z = &x[2];        // z will forever point to address of x[2]

(*pv)++;                           // this is legal, we may change *pv, just not pv
y++;                               // this is legal, we may change y, just not *y
```

C++ standard template library

The standard template library (STL) provides typical **container** data structures. They are **templates**: They can contain any type of fundamental data items or objects as their elements. The **element type** is specified in angular brackets.

```
// declare a list of int values
std::list<int> my_list();
```

```
// declare a list of std::string objects
std::list<std::string> my_list();
```

- **vector<T>** is a **dynamic array** for type **T** elements, similar to Python lists.
- **deque<T>** ("double ended queue"): **Dynamic array** with capacity both ends.

- **forward_list<T>** is a **singly linked list** data structure for type **T**.
- **list<T>** is a **doubly linked list** data structure for type **T**.

- **set<T>** is a container where each **key** (element) occurs only (at most) once.
- **map<T, V>** contains **key-value** pairs, which each key occurring at most once.
- **multimap<T, V>** contains **key-value** pairs; keys may occur multiple times.

- **array<T, n>** is a **static array** for type **T**, with array size **n**, similar to **T[]** arrays.

Gamaldags C/C++ som framleis blir brukt mykje

- Datatabellar slik dei er typiske for C/C++
  Det som er anbefalt i moderne C++ er std::array eller std::vector …

- C-stil streng, dvs. datatabell av char jf. **komline-argsett.cpp**
  Ved overføring av kommandolineargument er dette det som må brukast.

- *assert*

- kompilator-preprosessordirektiv med #define, #ifdef, #ifndef
  Kan lett føre til intransparent kode

jf. **book-index-c.zip**

- Struktur (**struct**) i staden for class
  Ikkje brukt veldig ofte, men somme synest det der god stil å bruke **struct** *dersom det ikkje er nokre private medlemer (alt er public)*

- globale variablar jf. **levetid.cpp**

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';**

Remark: Strings in C and C++

The C++ language only prescribes what functionalities a **std::string** should provide, not how it is realized at the memory level, which is up to the compiler.

Most implementations remain close to that from the C language, where character arrays terminated by the null character '\0' are employed. (If you want to enforce this, you can also still use all the C style constructs explicitly.)

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

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';**

Remark: Strings in C and C++

C++ strings may be the same as arrays at the memory level, but they are not arrays to the language. Therefore, **it is possible to pass C++ strings by value**.

C strings, however, **can never be passed by value** because they are arrays.

```
void increment_at(int p, char* str)
{
  str[p]++;
}

int main()
{
  char c_style_str[] = "INF205";
  increment_at(5, c_style_str);
  cout << c_style_str << "\n";
}
```

```
void increment_at(int p, std::string str)
{
  str[p]++;
}

int main()
{
  std::string cpp_style_str = "INF205";
  increment_at(5, cpp_style_str);
  cout << cpp_style_str << "\n";
}
```

Example file: **string-argument-passing.cpp**

Glossar

Global variabel

Definisjon: variabel som er tilgjengeleg gjennom eit namn med uavgrensa gyldigheitsområde; namn til slike variablar kan oppløysast overalt i kjeldekoden

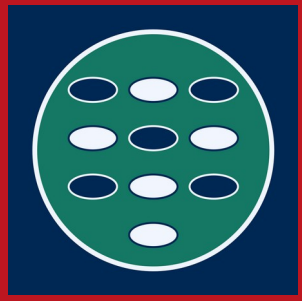
- Somme påstår det stort sett er dårleg stil å bruke globale variablar. Grunnen til det er m.a. at bruk av globale variablar vanskeleggjer feilsøking og formell verifisering.

- I skriptspråk (fortolka språk) er det veldig vanleg praksis med globale variablar.

Også å jamføre: **levetid.cpp**

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap

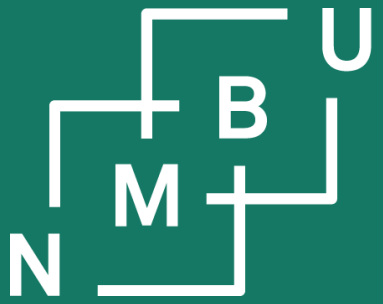


Digitalisering på Ås

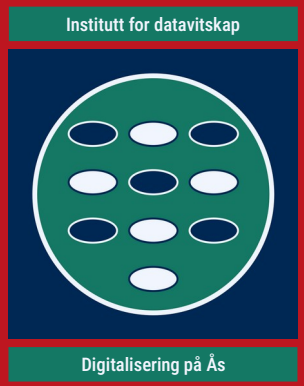
Samandrag / diskusjon

INF205

12. februar 2026



Noregs miljø- og biovitskaplege universitet



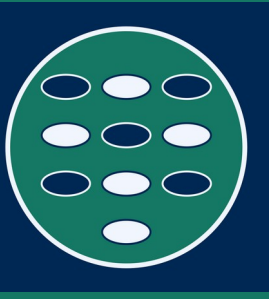
Eksempelproblem og oblig. oppgåve 1

INF205

12. februar 2026

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap



Digitalisering på Ås

IND205

Ressurseffektiv programmering

1 Moderne C++

1.1 Frå Python til C++

1.2 Peikarar og datatabellar

1.3 Programmeringsparadigme

1.4 God praksis i moderne C++

1.5 Moderne C++ og gamaldags C/C++