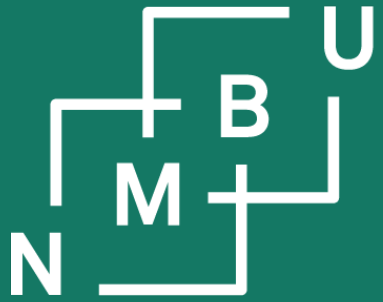
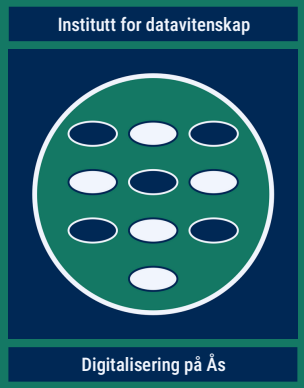


Norges miljø- og biovitenskapelige universitet



INF205
Resource-efficient programming

2 Memory and objects

2.1 Pass by value/reference
2.2 Memory allocation
2.3 C++ object orientation
2.4 Immutability
2.5 Streams and file I/O
2.6 Class hierarchies

Referencing (&) and dereferencing (*)

Norwegian University of Life Sciences

Referencing operator &:

- Used to obtain the address of a variable: &x is the address of x.
- If x has type X, the address has the type X*, *i.e.*, "pointer to X."

```
int x = 5;  int* y = &x;
```

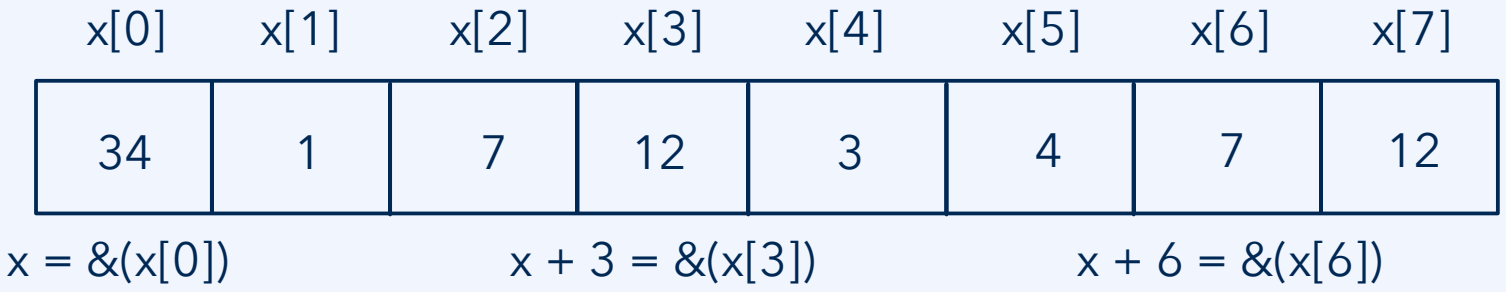
Dereferencing operator *:

- If y is a pointer of type X* (pointer to X), the value of y is an address.
- To access the value stored at the address y, we dereference it as *y.
- The value stored at y, and accessed by *y, is then of type X.
- & and * are inverse operators, therefore, *(&x) is the same as x:

```
int x = 5;  int* y = &x;  cout << x << " is the same as " << *y;
```

C/C++ arrays are pointers

An array contains a sequence of elements of the same type, arranged **contiguously in memory**. This supports fast access using **pointer arithmetics**. Once created, the size of a C/C++ array is fixed; we cannot append elements.



In C/C++, the type of an array such as **int[] is the same as the corresponding pointer type int***, *i.e.*, **the array actually is a pointer**. Its value is an address at which an integer is stored, namely, the memory **address of the first element**.

When x[i] is accessed, the compiler transforms this into x + sizeof(int) * i.

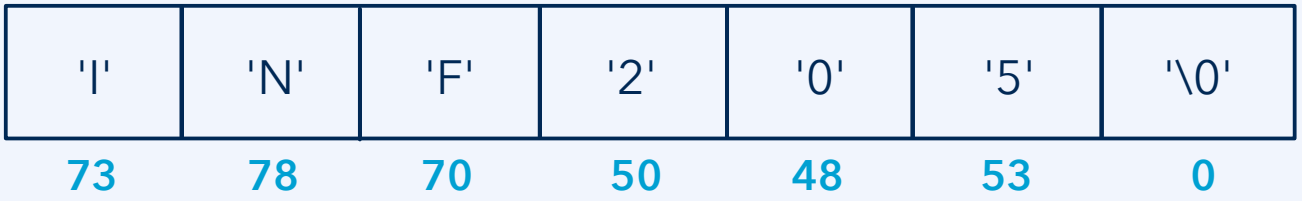
- **Allocation** is done with **new**. Example: **int* i = new int[8]();**
- **Deallocation** is done with **delete[]**. Example: **delete[] i;**

Remark: Strings in C and C++

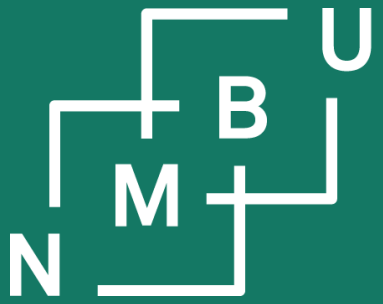
The C++ language only prescribes what functionalities a **std::string** should provide, not how it is realized at the memory level, which is up to the compiler.

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.)

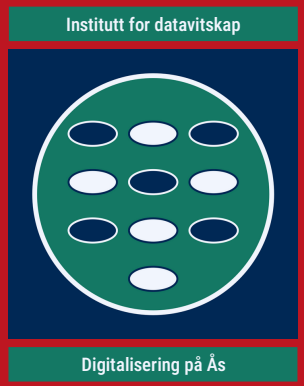
string s = "INF205"; or **char s[] = "INF205";** produce the following in memory:



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';**



Noregs miljø- og biovitskaplege universitet



2 Memory and objects

2.1 Pass by value or reference

2.2 Memory allocation

2.3 OOP in C++

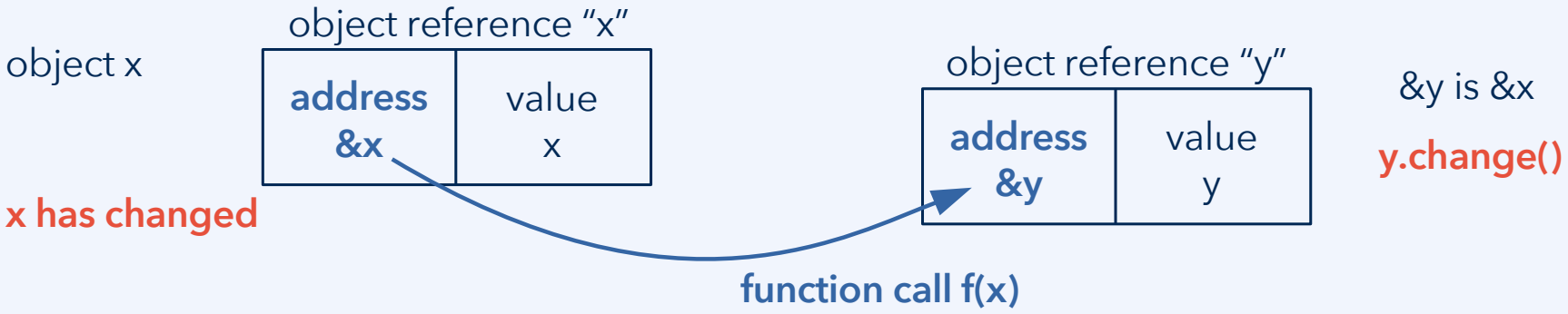
INF205

11th February 2025

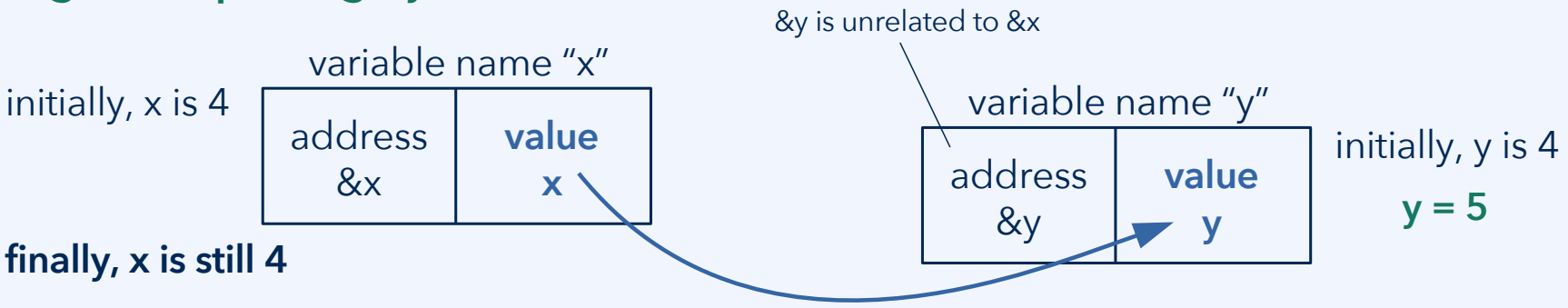
Pass by value in C++ (compared to Python)

In Python, object references are passed by value (*i.e.*, "pass by object reference"):

Argument passing by object reference in Python (similarly, in Java)



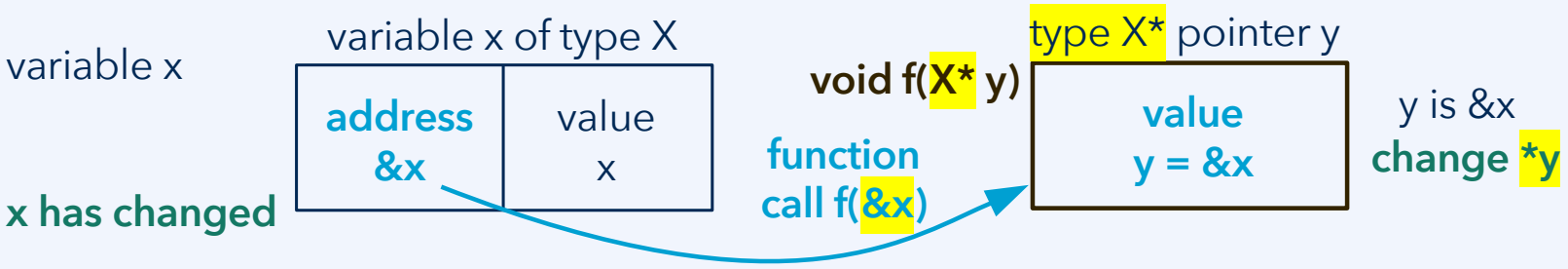
Argument passing by value



Pass by reference in C++ (compared to pass by value)

Pass by value: A new copy of the argument value(s) is created in memory. The function works with the copy. The function cannot access the original variable.

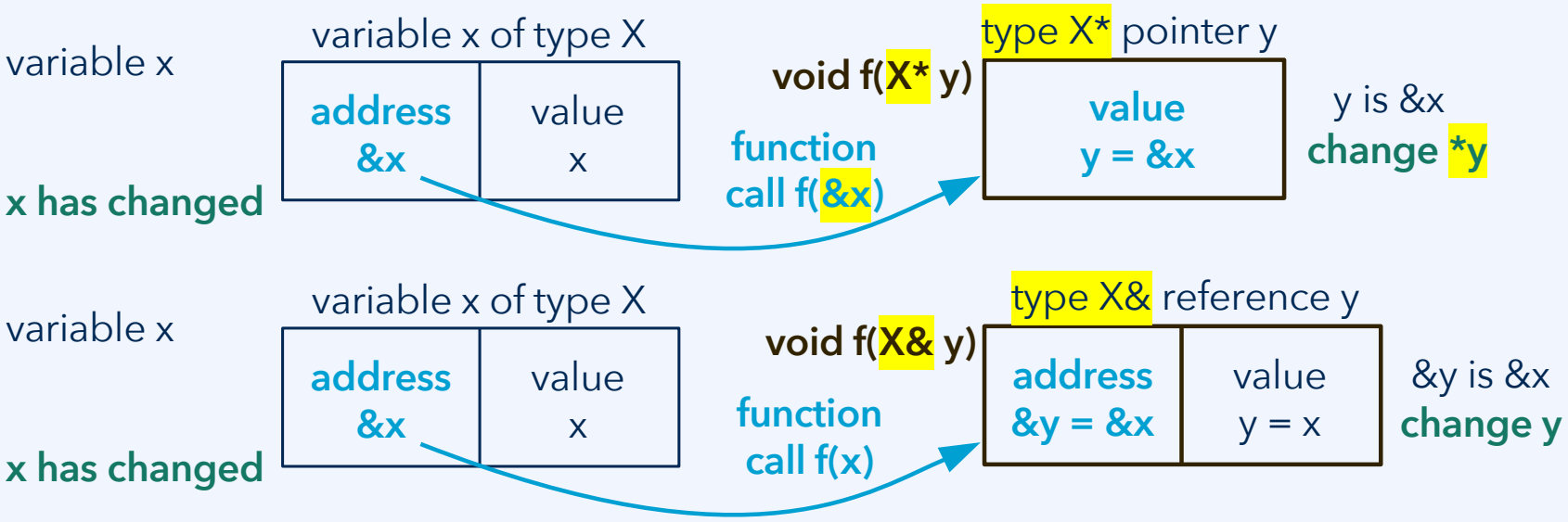
Pass by reference: The function is enabled to access the original variable at its address in memory. No copy is created. Changes affect the original variable.



Pass by reference in C++ (two ways of doing it)

Pass by value: A new copy of the argument value(s) is created in memory. The function works with the copy. The function cannot access the original variable.

Pass by reference: The function is enabled to access the original variable at its address in memory. No copy is created. Changes affect the original variable. C++ has two mechanisms for this: **Passing a pointer** and **passing a reference**.*



*Unfortunately there is some terminology confusion about this. We will call both "**pass by reference**."

Pass by reference vs. pass by value

Advantages of **passing** a function argument **by value**:

- **Memory management** is done at the stack level, **by the compiler**. The programmer can relax and does not need to deal with this aspect.
- The stack can be optimized at compile time, and it is **faster to access** memory on the stack because there is no need to look up an address.
- **Variable lifetime** coincides with the runtime of functions that use them.
- The value of **the variable in the calling function is protected** from any intransparent changes by the called function.
- This makes the code **more modular**. It is easier to understand and even verify the function. (The point of using local instead of global variables.)

Advantages of **passing** a function argument **by reference**:

There must be a reason there is a second mechanism, pass-by-reference.
Why? What is the advantage?

Pass by reference using a pointer vs. a reference

Pointers and references are two equivalent notations for the same techniques.

```
void some_function(int& parameter) {
  …
  // convert the reference to a pointer
  int* y = &parameter;
  // now we can work with pointer y
  …
}
```

```
void some_function(int* parameter) {
  …
  // convert the pointer to a reference
  int& x = *parameter;
  // now we can work with reference x
  …
}
```

Advantages of pass-by-reference **using a reference**:

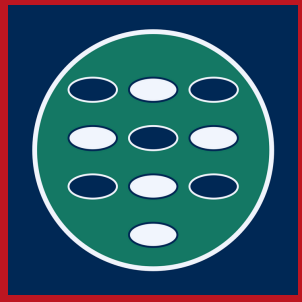
- Some memory-related errors become less likely if we only work with references; *e.g.*, errors from applying incorrect pointer arithmetics.
- Looks more like Java, Python, and other modern high-level languages.

Advantages of pass-by-reference **using a pointer**:

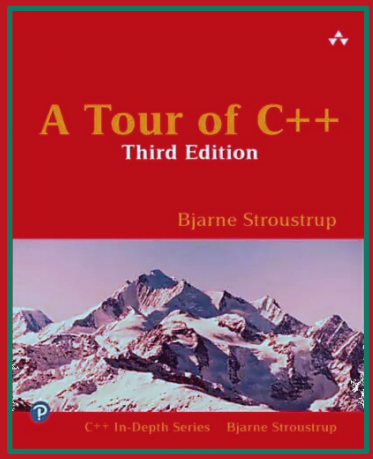
- It is visible to the programmer at all times that we deal with memory.
- Looks more like C, and it is closer to the object-code representation.

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap



Digitalisering på Ås



book Section 1.5

2 Memory and objects

2.1 Pass by value or reference

2.2 Memory allocation

2.3 OOP in C++

INF205

11th February 2025

Allocate: new. Deallocate: delete.

Allocation: Reserve memory to store data.
Deallocation: Release the memory.

On the stack

The stack is already handled completely and safely by the compiler. **Memory on the stack** (local variables of functions) is **allocated** as part of a **stack frame** **when the function is called**. It is **deallocated** again **when the function returns**.

On the heap

Memory on the heap is managed independent of the stack, at runtime, subject to **explicit allocation and deallocation** instructions that must come from the programmer. There is no garbage collection in C++!

- **Allocation** is done with **new**. Example: **int* i = new int;**
- **Deallocation** is done with **delete**. Example: **delete i;**

Summary: Allocation and deallocation by pointers

How do we declare a pointer?

- Like any other variable. Its type is a pointer type; *e.g.*, int* my_int_pointer;

How do we initialize a pointer?

- Initialize to **nullptr** (pointer version of 0): int* my_int_pointer = **nullptr**;
- Initialize to **another variable's address**: int* my_int_pointer = &my_index;
- **Allocate memory** on the heap: int* my_int_pointer = **new** int(0);

How do we deallocate a variable if it is stored on the heap?

- Delete the pointer to it. Example: b = **new** BookIndex; …; **delete** b;

How to release the memory if it is a local variable that is stored on the stack?

- Don't do that! You can only call "delete" on memory allocated with "new".

What if we call **new**, but there is not enough free memory left on the system?

- **new** VeryBigObject may throw an exception (a high-level construct).
- **new(std::nothrow)** VeryBigObject may return **nullptr** (low-level construct).

Summary: Allocation and deallocation of arrays

How do we declare a array?

- Give the size as constant expression in square brackets; *e.g.*, int values[6];
- Also possible: Just declare a pointer; *e.g.*, int* values;

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] = { };
- **Allocate memory** with **default initialization**: int* values = **new** int[6]**()**;

How do we deallocate an array if it is stored on the heap?

- Use **delete[]**. Example: b = **new** BookIndex[100]**()**; …; **delete[]** b;
- **Pitfall:** If you use **delete** instead of **delete[]**, only b[0] will be deallocated!

What if we call **new**, but there is not enough free memory left on the system?

- **new** BigObject[100000]**()** may throw an exception.
- **new(std::nothrow)** BigObject[100000]**()** may return **nullptr**.

The three most typical memory bugs

Norwegian University of Life Sciences

1) Access a pointer that was **not initialized**, or that has the value **nullptr**, or that for any other reason points to an **invalid address** in memory. ("**Wild pointer**.")

- **Question:** Why is this dangerous?

2) Memory is **allocated** using new, **but not deallocated again** using delete. This is called a **memory leak**.

- **Question:** Why is this dangerous?

3) Memory **has been deallocated**: Either it was on the stack in a stack frame that has been removed, or there has been a delete statement. But the address information was stored in **a pointer that still exists**: A **dangling pointer**!

- **Question:** Why is this dangerous?

Code that can produce wild pointers

```
#include <iostream>

void crop(int num_strings,  char** strings,  int characters_cropped);

int main(int argc,  char** argv) {
        int jump_to_index = 2;
        crop(argc-1,  argv+1,  jump_to_index);

        for(int i = 1;  i < argc;  i++)
                std::cout  <<  "Argument no. "  <<  i  <<  " was cropped to \""  <<  argv[i]  <<  "\".\n";
}

void crop(int num_strings,  char** strings,  int characters_cropped)  {

        for(int i = 0; i < num_strings; i++)
                strings[i] += characters_cropped;
}
```

1. Where is argv in memory? What does it consist of?

Plan: Remove the first 2 characters from each command-line argument.

Implementation: Simply jump ahead by <characters_cropped> char's, that is, here, by 2 characters.

Example file: wildptr.cpp

Code that can produce wild pointers

```
#include <iostream>

void crop(int num_strings,  char** strings,  int characters_cropped);

int main(int argc,  char** argv) {
        int jump_to_index = 2;
        crop(argc-1,  argv+1,  jump_to_index);

        for(int i = 1;  i < argc;  i++)
                std::cout  <<  "Argument no. "  <<  i  <<  " was cropped to \""  <<  argv[i]  <<  "\".\n";
}

void crop(int num_strings,  char** strings,  int characters_cropped) {

        for(int i = 0; i < num_strings; i++)
                strings[i] += characters_cropped;

}
```

2. What sort of a statement is this? Why do we need it?

3. Why this "-1" and "+1" on argc and argv?

4. How was this supposed to work? When will it inadvertently generate a wild pointer?

Example file: wildptr.cpp

used for assert(condition), which checks that condition is true

used for strlen(char str), which returns the length of a C string*

Note that this is one less than the size of str as a char array:
For strlen, the terminal \0 character does not count.

```
#include <cassert>
#include <cstring>
#include <iostream>

void crop(int num_strings,  char** strings,  int characters_cropped);

int main(int argc,  char** argv) {
        int jump_to_index = 2;
        crop(argc-1,  argv+1,  jump_to_index);

        for(int i = 1;  i < argc;  i++)
                std::cout  <<  "Argument no. "  <<  i  <<  " was cropped to \""  <<  argv[i]  <<  "\".\n";
}

void crop(int num_strings,  char** strings,  int characters_cropped) {
        assert(characters_cropped >= 0);
        for(int i = 0; i < num_strings; i++)
                if(strlen(strings[i]) >= characters_cropped)
                        strings[i] += characters_cropped;
                else strings[i] += strlen(strings[i]);
}
```

Example file: wildptr-fixed.cpp

How can memory leaks be fixed?

High-level languages operate with *automated garbage collection:* Memory is deallocated when there are no more variables referring to it.

C/C++ memory management on the heap must be done by hand. This causes two possible bugs:

Memory leak: Memory should have been deallocated, but was not.

Dangling pointers: Memory has been deallocated, but it should not.

Discussion:

- What approaches can we try in general, when we have detected a memory leak?
- Brainstorm a list of ideas.
-

How can memory leaks be avoided?

Discussion:

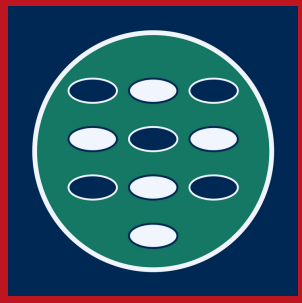
- What approaches can we try in general, if we want to decrease the risk of a bug in the form of a memory leak?

There are a few techniques in C++ that help us write safer code with explicit memory management, still done on the heap but less prone to pitfalls.

The key concept for safe manual memory management is **ownership** of a data item, *i.e.*, deciding what entity/part of the code has *responsibility for managing its allocation and deallocation* safely. The entity holding ownership is typically an object – so we will first need to discuss how OOP is done in C++.

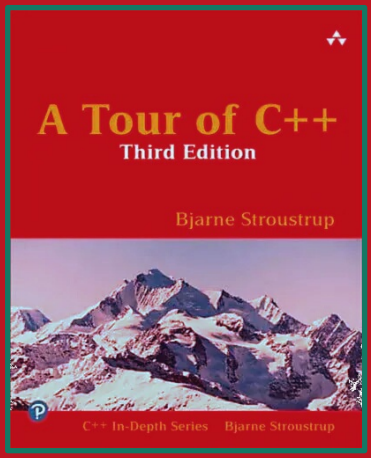
Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap



Digitalisering på Ås

2 Memory and objects



Sections 5.1, 5.2

Core Guidelines:

C.3 – C.11
C.43 – C.51

(and more in "C")

2.1 Pass by value or reference

2.2 Memory allocation

2.3 OOP in C++

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.»

Example file: book-index-python.ipynb

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.

Constructors and destructors

Constructor: A method that is called when an object is **allocated**.
Destructor: A method that is (implicitly) called when an object is **deallocated**.

They are not mandatory (as we have seen); use them if you need to specify some functionality for this purpose. Most typically:

- Provide a **constructor** if you want to give the user control over how the private properties of an object are initialized.
- There are also special "copy constructors" and "move constructors". (We will discuss them in detail at a later stage in the course.)
- Provide a **destructor** if your memory management strategy requires it; there might be properties stored as pointers that need to be deleted.

```
class BookIndex {
public:
  BookIndex(int c, int s, int p);
  ~BookIndex();
  …
};
```

```
BookIndex::BookIndex(int c, int s, int p) {
  this->chapter = c; this->section = s; this->page = p;
}
BookIndex::~BookIndex() {
  cout << "Deleting a BookIndex object.\n";
}
```

Constructors and destructors

General rule: For every "new" there must be a matching "delete".

The destructor **T::~T()** is called when an object of type T is deallocated.

This is the case both for objects on the stack and on the heap:

```
void function_name(…)
{
  // constructor is called
  T tobject;
  …
  // destructor is called
  return;
}
```

```
{
  …
  // constructor is called
  T* tpointer = new T;
  …
  // destructor is called
  delete T;
}
```

```
class T
{
public:
  …
  ~T() { delete this->p; }
  …
private:
  S* p …
}
```

If T has ownership over p, this must be done! Without it, there would be a **memory leak**!

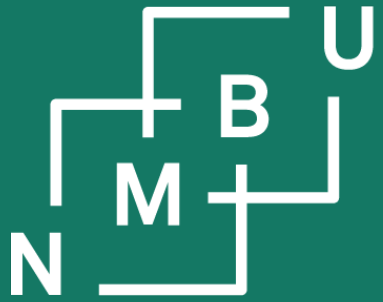
There might by other properties that do *not* need to be deallocated manually. (*Why?*)

Constructors and destructors

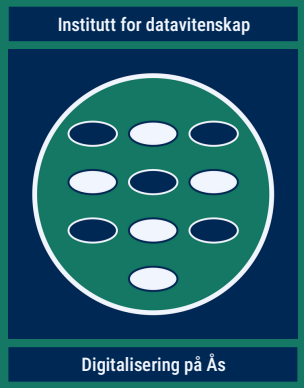
Core Guidelines:

- C.20: "If you can avoid defining default operations, do."
  - "This is known as "the **rule of zero**." Define zero constructors or destructors if it can be done without creating an inconsistent state.
  - A simple and good reason for defining a constructor is to force the user to provide some information that is required.
  - Define a constructor in cases where it does not make sense to initialize the object's properties to some specified default value.

- **C.30**: "Define a destructor if a class needs an explicit action at object destruction." And related, **C.31**: "All resources acquired by a class must be released by the class's destructor."
  - If a data structure needs to be built up (memory allocated, *etc.*), this normally requires both a constructor and a destructor.
  - For such cases, we will learn the *rule of three* and the *rule of five*.



Norges miljø- og biovitenskapelige universitet



INF205
Resource-efficient programming

2 Memory and objects

2.1 Pass by value/reference
2.2 Memory allocation
2.3 C++ object orientation
2.4 Immutability
2.5 Streams and file I/O
2.6 Class hierarchies