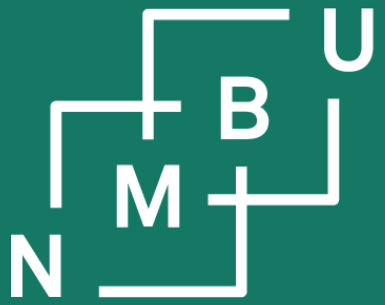


Norges miljø- og biovitenskapelige universitet

INF205
Resource-efficient programming

1 C++ basics

1.8 Structures
1.9 Argument passing
1.10 Memory (de)allocation
1.11 Working with pointers/arrays

Norwegian University of Life Sciences

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

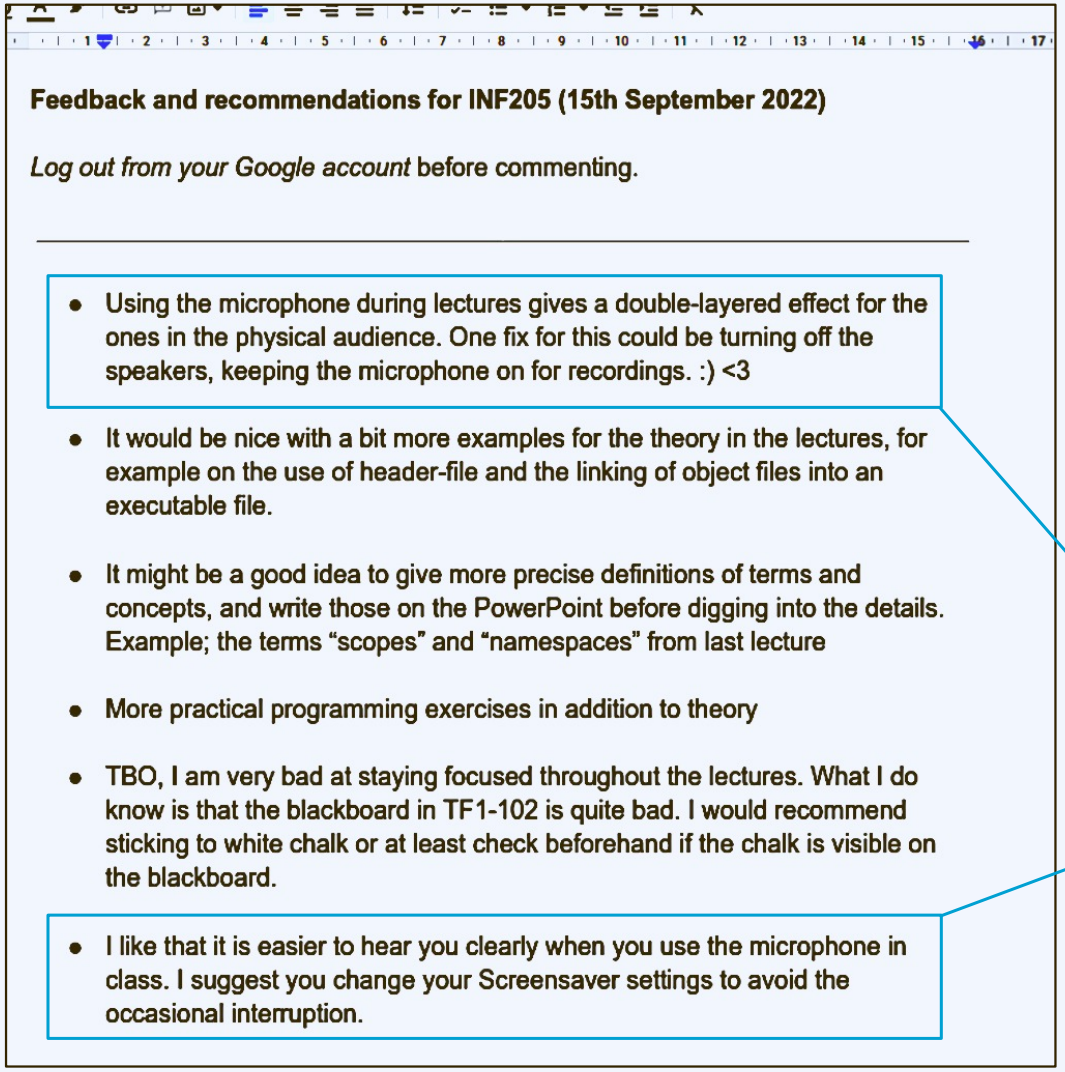
Why object orientation?

The job of variables is to store data. In object oriented programming (OOP) the focus is on *how data belong together* and how we can facilitate *safe and correct access to data*. How do data-centered tools (DBs, *etc.*) present data?

Example: "Largest cities by country" query on Wikidata.

city	cityLabel	population	country	countryLabel	loc
wd:Q172	Toronto	2731571	wd:Q16	Canada	Point(-79.386666666 43.670277777)
wd:Q1490	Tokyo	14047594	wd:Q17	Japan	Point(139.691722222 35.689555555)
wd:Q585	Oslo	693494	wd:Q20	Norway	Point(10.738888888 59.913333333)
wd:Q1761	Dublin	553165	wd:Q27	Republic of Ireland	Point(-6.260277777 53.349722222)
wd:Q1781	Budapest	1723836	wd:Q28	Hungary	Point(19.040833333 47.498333333)
wd:Q2807	Madrid	3305408	wd:Q29	Spain	Point(-3.7025 40.416666666)
wd:Q60	New York City	8804190	wd:Q30	United States of America	Point(-74.0 40.7)
wd:Q240	Brussels-Capital Region	1218255	wd:Q31	Belgium	Point(4.3525 50.846666666)
wd:Q1842	Luxembourg	128512	wd:Q32	Luxembourg	Point(6.132777777 49.610555555)
wd:Q1757	Helsinki	643272	wd:Q33	Finland	Point(24.93417 60.17556)
wd:Q1754	Stockholm	978770	wd:Q34	Sweden	Point(18.068611111 59.329444444)
wd:Q1748	Copenhagen	644431	wd:Q35	Denmark	Point(12.568888888 55.676111111)
wd:Q270	Warsaw	1790658	wd:Q36	Poland	Point(21.011111111 52.23)

Module feedback (& mike test)



Feedback and recommendations for INF205 (15th September 2022)

Log out from your Google account before commenting.

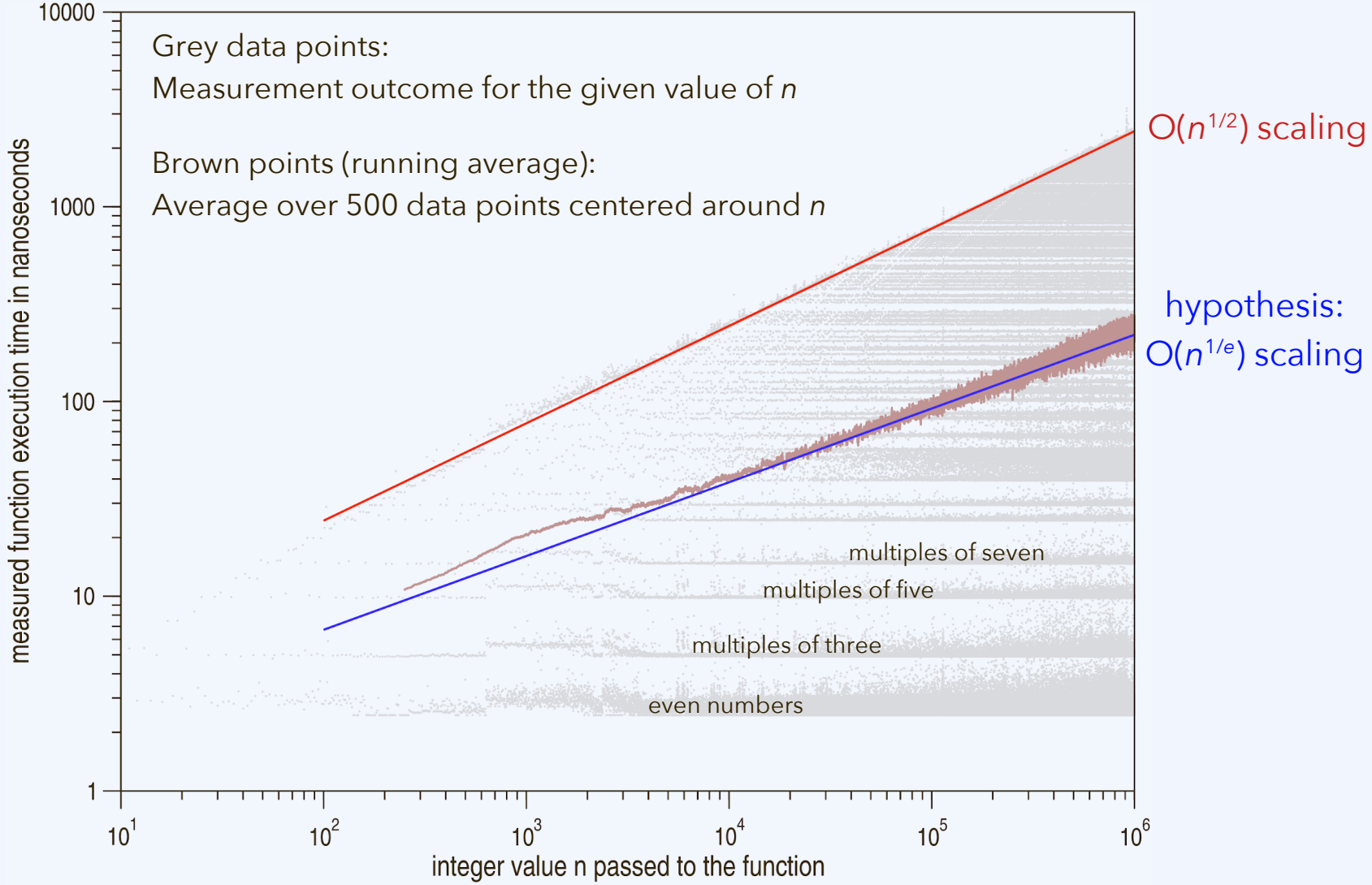
- Using the microphone during lectures gives a double-layered effect for the ones in the physical audience. One fix for this could be turning off the speakers, keeping the microphone on for recordings. :) <3
- It would be nice with a bit more examples for the theory in the lectures, for example on the use of header-file and the linking of object files into an executable file.
- It might be a good idea to give more precise definitions of terms and concepts, and write those on the PowerPoint before digging into the details. Example; the terms “scopes” and “namespaces” from last lecture
- More practical programming exercises in addition to theory
- TBO, I am very bad at staying focused throughout the lectures. What I do know is that the blackboard in TF1-102 is quite bad. I would recommend sticking to white chalk or at least check beforehand if the chalk is visible on the blackboard.
- I like that it is easier to hear you clearly when you use the microphone in class. I suggest you change your Screensaver settings to avoid the occasional interruption.

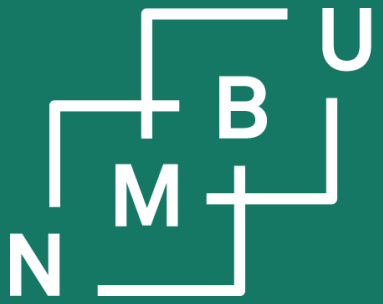
Thanks for providing the recommendations!

(Next in week 39.)

some acoustics test is needed …

Tutorial: Performance measurement





Noregs miljø- og biovitskaplege universitet

1 C++ basics

1.8 Structures

Structures: Object orientation from C

Norwegian University of Life Sciences

Structures (**struct**) were introduced as device for OOP-like programming in C. Most of the time, OOP in C++ uses classes, but C++ still has structures.

Structures group different *variables* (**properties**) together into one **composite data type**. Each instance of a structure is called an **object**; each object has its own instance of the properties. In C++, structures can have **methods**, *i.e.*, *functions* defined for the structure and carried out for an individual object.

Structures are what comes closest in C/C++ to object orientation from Python: Structure elements are accessible from outside the structure. They're "**public**".

Example:

- A book indexation structure, giving a position in a book in terms of chapter number, section number, and page number.
- Methods can be used for going to the next chapter, section, page.
- But we can also directly set chapter and section numbers from outside.

Example: struct as a collection of fields

```
struct BookIndex
{
  int chapter = 1;
  int section = 1;
  int page = 1;
};
```

Here we declare to BookIndex objects, i and j.

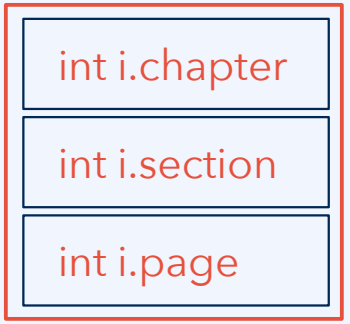
Are we initializing them also?

```
BookIndex i, j;

i.chapter = 1;
i.section = 8;
i.page = 8;
```

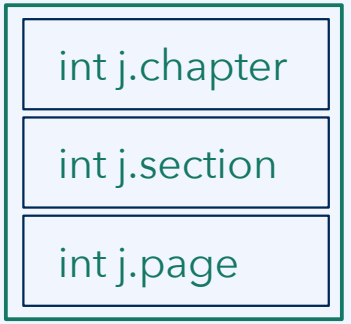
We can directly access the fields of the structure because they are public.

Each BookIndex object contains three int variables:

BookIndex i



BookIndex j



Toward OOP: Syntax for methods

```
struct BookIndex
{
  int chapter = 1;
  int section = 1;
  int page = 1;

  int next_chapter();
  int next_section();
  int next_page();

  void out();
};
```

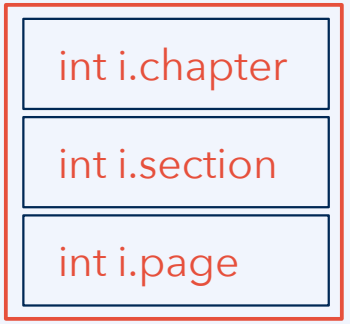
header file contains **method declarations**

code file contains the **method implementation**

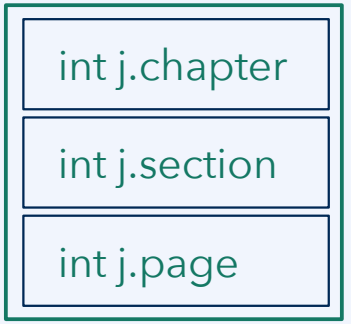
```
int BookIndex::next_chapter()
{
  chapter++;
  section = 1;
  page++;
  return chapter;
}

…

void BookIndex::out()
{
  std::cout
    << "Section " << chapter
    << "." << section
    << ", p. "<< page << "\n";
}
```

Each BookIndex object contains three int variables:

BookIndex i



BookIndex j



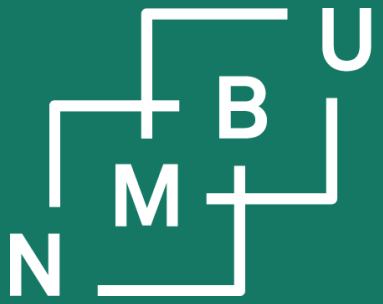
Python classes and C++ structures

```
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()
```

```
struct BookIndex
{
  int chapter = 1;
  int section = 1;
  int page = 1;

  int next_chapter();

  int next_section();

  int next_page();

  void out();
}

int main( ) {
  BookIndex idx;
  idx.chapter = 1;
  idx.section = 8;
  idx.page = 25;

  idx.out();
}
```

```
int BookIndex::next_chapter( ) {
  chapter++;
  section = 1;
  page++;
  return chapter;
}

int BookIndex::next_section( ) {
  section++;
  return section;
}

int BookIndex::next_page( ) {
  page++;
  return page;
}

void BookIndex::out( ) {
  cout << "Section " << chapter << "."
       << section << ", p. " << page << "\n";
}
```

How do Python and C++ deal with argument passing?



Noregs miljø- og biovitskaplege universitet

1 C++ basics

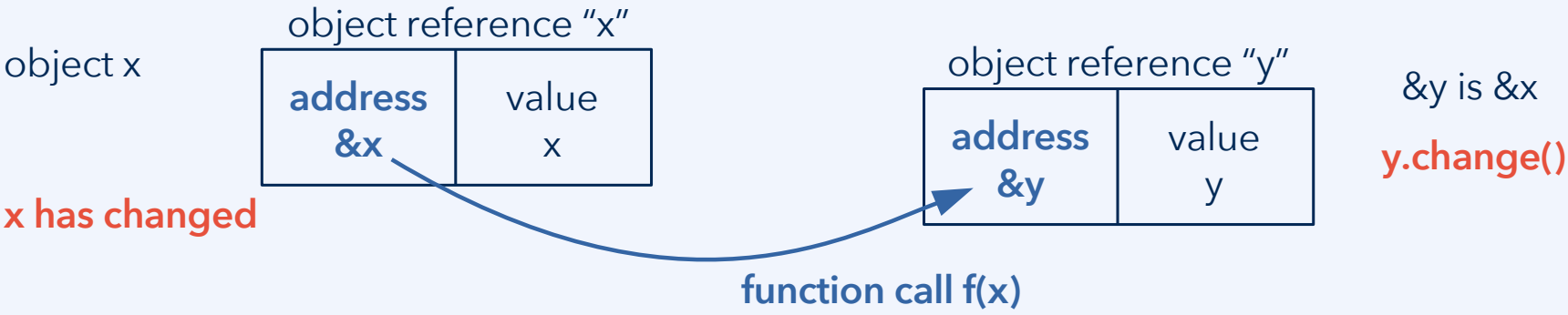
1.8 Structures

1.9 Argument passing

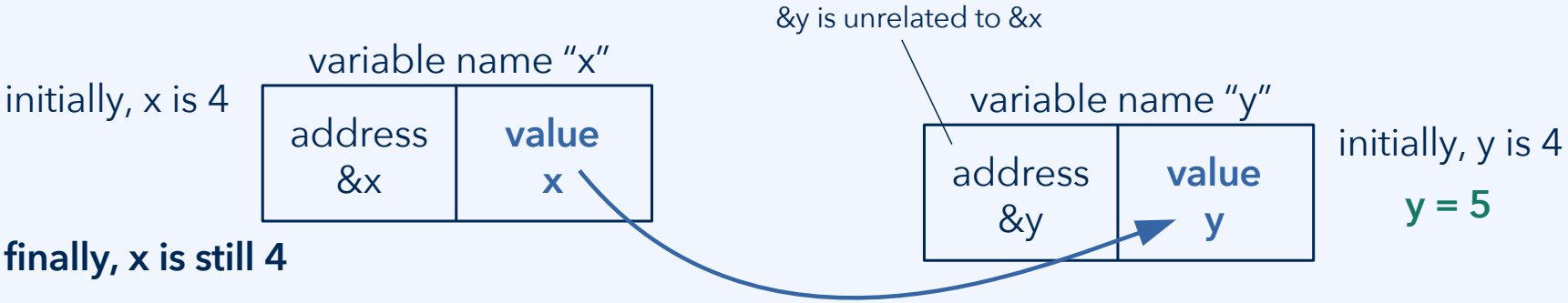
Argument passing

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



Example: struct passed by value

In **book-index-test.cpp** from the "struct BookIndex" example:

- In int main(), an object BookIndex idx is declared.
- idx is set to Section 1.8, p. 25.
- idx is then **passed by value** to a method that increments the chapter:

```
int main()
{
  … /* idx set to Section 1.8, p. 25 */
  using namespace pass_by_value;
  start_chapter(idx);
  idx.out();
  …
}
```

What output can we expect?

```
namespace pass_by_value
{
  void start_chapter(BookIndex b)
  {
    b.next_chapter();
    b.out();
  }
}
```

What output can we expect?

What is happening at the memory level?
How many BookIndex objects are there in memory, and what do they contain?

Example: struct passed by reference

Let us pass the parameter by reference, by adding "&" to the called function:

- In int main(), an object BookIndex idx is declared.
- idx is set to Section 1.8, p. 25.
- idx is then **passed by reference** to a method that increments the chapter:

```
int main()
{
  … /* idx set to Section 1.8, p. 25 */
  using namespace pass_by_reference;
  start_chapter(idx);
  idx.out();
  …
}
```

What output can we expect?

```
namespace pass_by_reference
{
  void start_chapter(BookIndex& b)
  {
    b.next_chapter();
    b.out();
  }
}
```

What output can we expect?

What is happening at the memory level?
How many BookIndex objects are there in memory, and what do they contain?

Pointers for memory address data

A **pointer** is a variable that has a **memory address** as its value.

- BookIndex* b is a pointer to the address of a BookIndex object.
- The address of an object is obtained by **referencing**, e.g., b = &idx;
- While b is the address, it can be **dereferenced** (*b) to access the content.
- For objects, there is the -> operator: Write b->prop instead of (*b).prop.

```
int main()
{
  … /* idx set to Section 1.8, p. 25 */
  using namespace pass_by_reference;
  start_chapter(&idx);
  idx.out();
  …
}
```

What output can we expect?

```
namespace pass_by_reference
{
  void start_chapter(BookIndex* b)
  {
    b->next_chapter();
    b->out();
  }
}
```

What output can we expect?

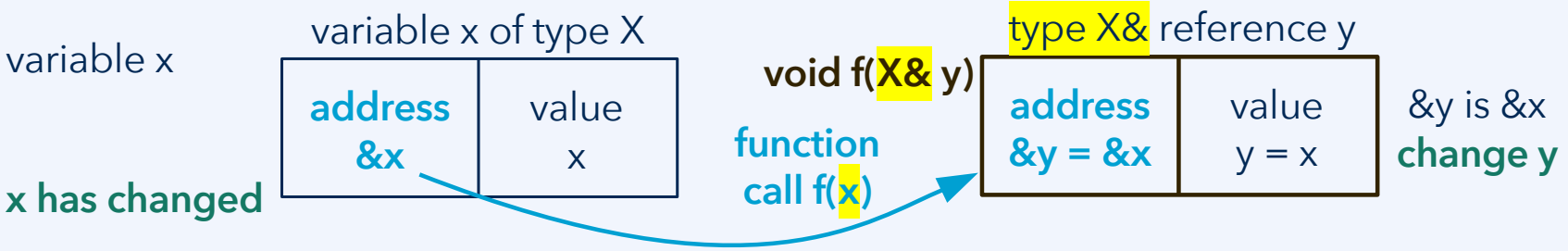
What is happening at the memory level?

How many BookIndex objects are there in memory, and what do they contain?

Two ways of "passing by reference"

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 reference** and passing a pointer.*

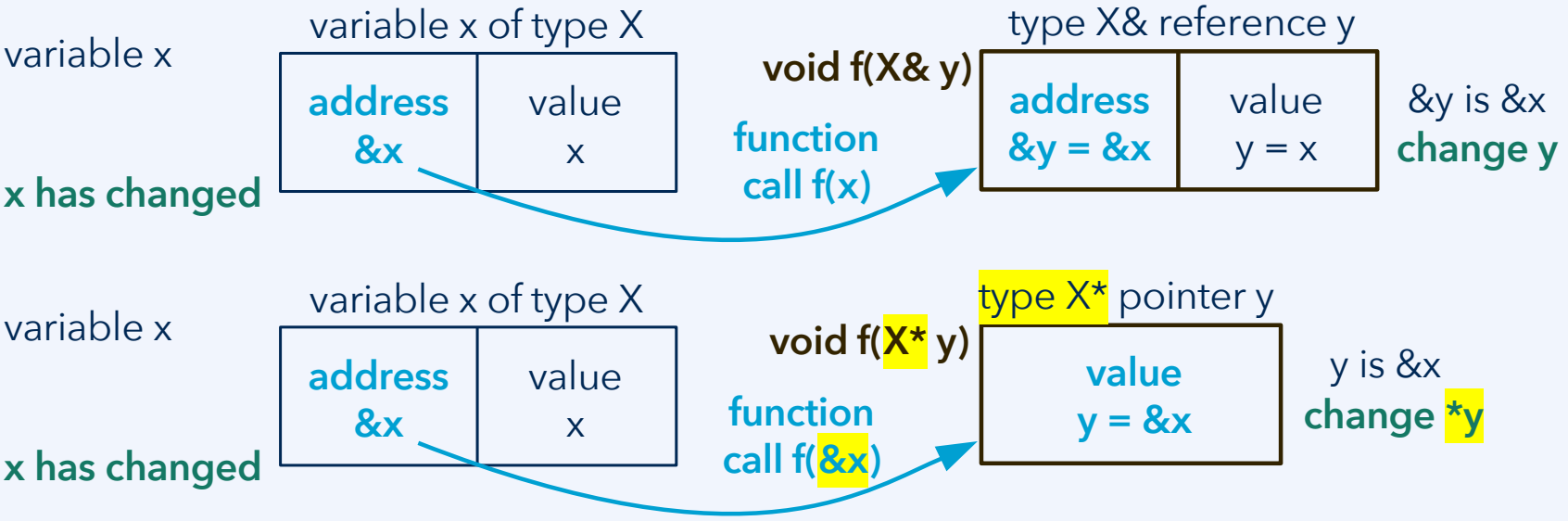


*Unfortunately there is some terminology confusion about this. We will call both "**pass by reference**."

Two ways of "passing by reference"

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 reference** and **passing a pointer**.*



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

Referencing/dereferencing operators

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;
- A second, independent use of this operator is "passing a reference" as a function argument, *e.g.*, as in void start_chapter(BookIndex& b);

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;

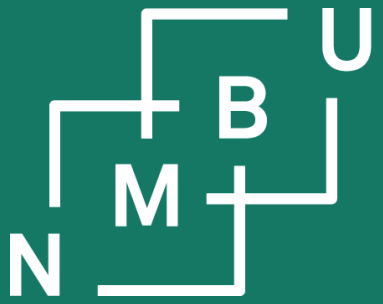
The combined **dereferencing and object property access** operator **->** is an abbreviation: y->prop is short for (*y).prop. It is applied to pointers to objects.

Lifetime of variables: What is wrong?

- What happens upon execution of the code below **at the memory level**?
- Why does it lead to an error? (**Segmentation fault.**)
- What **lifetime** do the **variables** have?

```
BookIndex* input_book_index()
{
  BookIndex x;

  std::cout << "input Chapter no.: ";
  std::cin >> x.chapter;
  // …

  std::cout << "BookIndex object: ";
  x.out();  // output: where are we in the book?

  // return pointer to x
  return &x;
}
```

```
int main()
{
  BookIndex* idx = input_book_index();

  // go to the next chapter
  //
  idx->next_chapter();
  std::cout << "Updated book index: ";
  idx->out();  // output: where are we now?
}
```



Noregs miljø- og biovitskaplege universitet

1 C++ basics

1.8 Structures

1.9 Argument passing

1.10 Memory (de)allocation

Norwegian University of Life Sciences

Memory allocation and deallocation

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

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**.

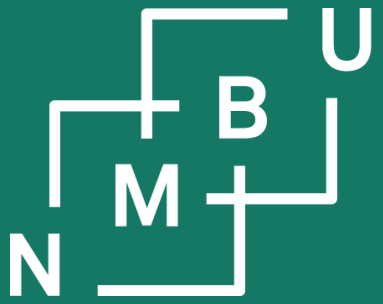
Memory on the heap is managed independent of the stack, at runtime, subject to **explicit allocation and deallocation** instructions that must come from the programmer. There is no garbage collection in C++!

- **Allocation** is done with **new**. Example: **int* i = new int(42);**
- **Deallocation** is done with **delete**. Example: **delete i;**

initialization to *i = 42

Discussion:

- How can it ever be justified to use the heap? When shouldn't we do it?
- What "new" and "delete" statements are needed in order to fix the bug from the "lifetime of variables" example? (Slide 19.)



Noregs miljø- og biovitskaplege universitet

1 C++ basics

1.8 Structures

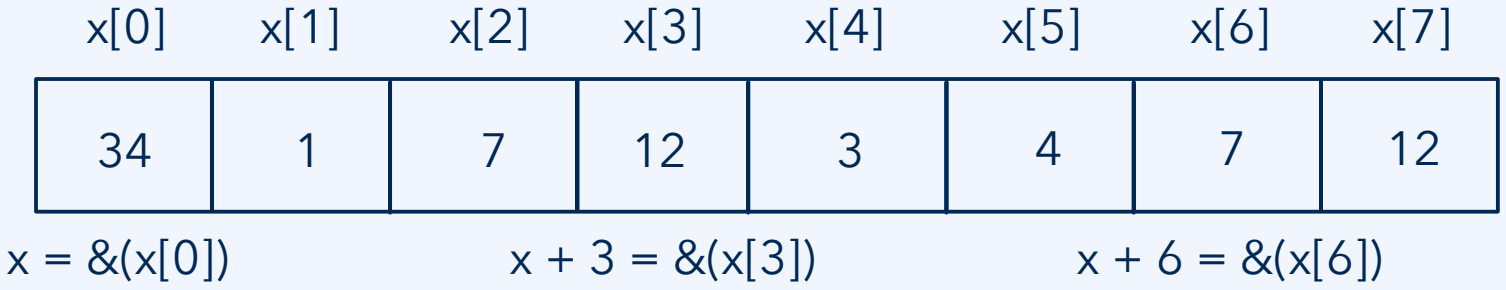
1.9 Argument passing

1.10 Referencing/dereferencing

1.11 Working with pointers and arrays

C/C++ arrays (static arrays)

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**.

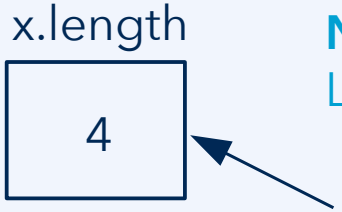
This is highly efficient since when x[i] is accessed, the compiler transforms this into accessing the memory address x + sizeof(int) * i.

Lists in Python (dynamic arrays)

Conventional arrays are **static data structures**. Their size in memory is constant, and memory needs to be allocated only once, *e.g.*, at declaration time. (Details depend on programming language, compiler, flags/optimization level, *etc.*).

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.

logical size is 4

Remarks on working with 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).

Remarks on working with 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**.

Pointers: Three most typical mistakes

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**.")

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

- **Discussion:** Why is this dangerous? Why is it hard to fix in debugging?

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**!

- In the "variable lifetime" bug example, there was a dangling pointer.

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. We will look at several of these techniques further ahead in the course of the module.

Memory leak: Example

The "**memory-leak.zip**" code seems to work well, at least for some time.
In fact it has a **memory leak**, leading to a crash or even hanging up the system.

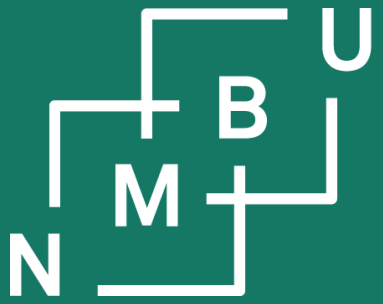
```
bool is_prime(int64_t* n) {
  if((*n%2 == 0) || (*n%3 == 0)) {
    delete n;  return false;
  }

  for(int64_t i = 5; *n >= i*i; i += 6) {
    if((*n % i == 0) || (*n % (i+2) == 0)) {
      delete n;  return false;
    }
  }

  return true;
}
```

```
double time_measurement(int64_t n) {
  auto t0 = high_resolution_clock::now();
  for(int i = 0; i < num_tests; i++) {
    is_prime(new int64_t{n});
  }
  auto t1 = high_resolution_clock::now();

  return duration_cast<nanoseconds>(t1-t0).count()
    / (double)num_tests;
}

int main() {
  for(int64_t x = xmin; xmax >= x; x += xstep)
    cout << x << "\t" << time_measurement(x) << "\n";
}
```

Discussion:

- What is strictly wrong about the code?
- In what ways is the code ill-designed?

… to be continued in the tutorial.

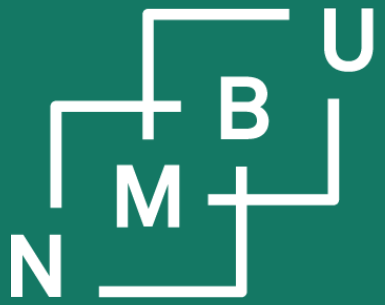


Noregs miljø- og biovitskaplege universitet

Conclusion

INF205

21st September 2022



Norges miljø- og biovitenskapelige universitet

INF205
Resource-efficient programming

1 C++ basics

1.8 Structures
1.9 Argument passing
1.10 Memory (de)allocation
1.11 Working with pointers/arrays