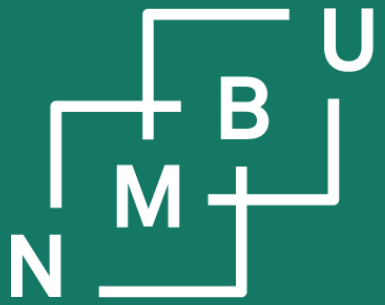


Norges miljø- og biovitenskapelige universitet

INF205
Resource-efficient programming

1 C++ basics

1.12 On pass-by-reference
1.13 Classes
1.14 Inheritance
1.15 Streams

Working with pointers (recapitulation)

Norwegian University of Life Sciences

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).

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**.

Tutorial problem: Structure

Low-level oriented arrangement of data: (**sphere-collisions-low-level.zip**)

```
int count_collisions(
  int N, float size[], float coordx[],
  float coordy[], float coordz[]
);

int main() {
  …
  cin >> N;
  float* size = new float[N]();
  float* coordx = new float[N]();
  float* coordy = new float[N]();
  float* coordz = new float[N]();
  …
  int result = count_collisions(
    N, size, coordx, coordy, coordz
  );
  …
}
```

Q: Can we also use "float size[N];" *etc.*?

The task was to arrange the data in a more object-oriented way, using a structure.

In this way, all data on the same sphere should be stored in the same object.

Tutorial problem: Structure

Low-level oriented arrangement of data: (**sphere-collisions-low-level.zip**)

```
int count_collisions(
  int N, float size[], float coordx[],
  float coordy[], float coordz[]
);

int main() {
  …
  cin >> N;
  float* size = new float[N]();
  float* coordx = new float[N]();
  float* coordy = new float[N]();
  float* coordz = new float[N]();
  …
  int result = count_collisions(
    N, size, coordx, coordy, coordz
  );
  …
}
```

Object-oriented arrangement of data: (**sphere-collisions-struct.zip**)

```
struct Sphere {
  float size = 0.0;
  float coords[3] = {0.0, 0.0, 0.0};
};

int count_collisions(int N, Sphere spheres[]);

int main() {
  …
  cin >> N;
  Sphere* spheres = new Sphere[N]();
  …
  int result = count_collisions(N, spheres);
  …
}
```

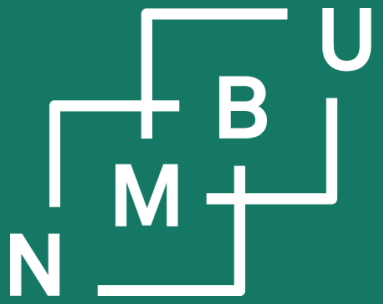
Q: Can we also use "Sphere spheres[N];"?

Tutorial problem: Pass by reference

Let us collect opinions from the groups' discussions on the "memory leak" code: In what ways was it poorly designed – what would have been better?

```
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";
}
```



Noregs miljø- og biovitskaplege universitet

1 C++ basics

1.12 On pass-by-reference

Pass by value vs. pass by reference

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. Even Python uses it when dealing with objects. **Discussion:** What is the advantage?

Pass a reference vs. pass a pointer

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.

“const” parameters of a function

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 Sphere& s);
void do_something(const Sphere* s);
void do_something(const Sphere s[]);
```

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

Examples:

1) The “**const-array**” code does not compile. How do we fix it?
2) Let us look at the “**sphere-collisions-struct**” code. It is advisable to say that something is constant wherever possible. Where can we do it?

const-array.cpp: What is the mistake?

```
int second_largest_of(int N, int x[]) {
  int largest = numeric_limits<int>::min();
  int second_largest = numeric_limits<int>::min();

  for(int i = 0; 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;
  const int x[fixed_array_size] = {4, 0, 6, 5, 2};
  cout << second_largest_of(fixed_array_size, x);
}
```

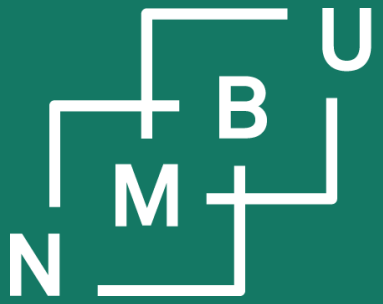
Pass by reference and “const”

1) If you can pass by value, that is always to be preferred!
2) But pass large objects by reference; otherwise all data must be copied.
3) 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
```



Noregs miljø- og biovitskaplege universitet

1 C++ basics

1.12 On pass-by-reference

1.13 Classes

C++ as an object-oriented language

C++ can be used in many ways, including old-fashioned procedural programming like in C. But C++ is also a true **object-oriented programming language**. Its functionality in that respect goes beyond that of Python.

By convention, if we use only the features that we have seen before, we will use the **struct** keyword, but we tend to define a **class** when using any of these:

- Members that are **private**, *i.e.*, they can only be accessed from inside the class. Both properties (variables of an object) and methods (functions of an object) can be either **public** or **private**.
- Class hierarchies with **inheritance**, where we define a more generic superclass (*e.g.*, Shape) and one or several more specific subclasses that are derived from it (*e.g.*, Sphere and Cuboid).
- Specialized functionality for **constructing**, **copying**, or **deleting** objects.

Methods: Recapitulation

```
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 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";
}
```

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**.

The “this” pointer and “const” methods

Norwegian University of Life Sciences

```
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() 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 and public members of a class

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.

Class definition: Common techniques

Class definitions are done in header files. Method definitions are mostly done in the code file. In many cases, each class has its own header and code file (*e.g.*, example-class**.h**, example-class**.cpp**), but see also Core Guidelines **NR.4**.

```
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();
};
```

Very simple methods can be in the header file. That saves you lines of code.

But it also helps the compiler: They could become "inline" methods. The compiler then replaces the method call by the method definition.

For propertyA, we are using an object or data item.
For propertyB, we are using a pointer.
What could be the reasons behind such choices?

Example: Making properties private

Task: Develop the "struct BookIndex" example into a class where the properties are private.

1) Use "class" instead of "struct".
2) Split class definition into a public part and a private part.
3) Introduce "get" and "set" methods to access the properties.
4) Adjust the remaining code so that direct access to private properties is replaced with calling the "get" and "set" methods.

```
class BookIndex
{
public:
  int get_chapter() const;
  …
  void set_chapter(int c);
  …

private:
  int chapter = 1;
  …
};
```

```
int main()
{
  …
  BookIndex idx;
  idx.set_chapter(1);
  idx.set_section(8);
  idx.set_page(8);
  …
}
```

Constructors and destructors

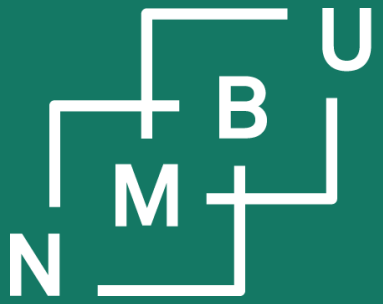
Constructor: A method that is called when an object is **allocated**.
Destructor: A method that is (implicitly) called when an object is **deallocated**.

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". (Not to be discussed right now.)
- 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";
}
```



Noregs miljø- og biovitskaplege universitet

1 C++ basics

1.12 On pass-by-reference

1.13 Classes

1.14 Inheritance

Inheritance and virtual methods

Classes can stand in a hierarchical relationship: A more general superclass and its more specific subclass (also, “derived class” or “child”).

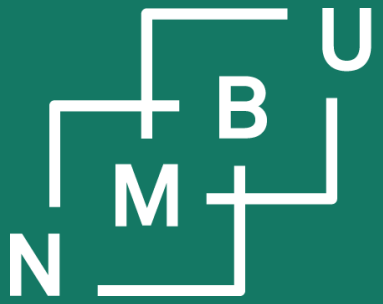
An object of the subclass then (automatically) is **also an object of the superclass**; it has all the members defined in its class definition, but also **inherits the members defined for the superclass**, to which it also belongs.


```
class LiteratureIndex {
public:
  virtual int next_page();
  …
private:
  int year = 0;
  …
};
```

```
class JournalArticleIndex: public LiteratureIndex {
public:
  int next_page();
  …
private:
  int volume = 0;
  …
};
```

JournalArticleIndex can override the **next_page** method definition from its superclass, because it is **virtual**.

It has the property **volume**, but it also **inherits the property year**.



Noregs miljø- og biovitskaplege universitet

1 C++ basics

1.12 On pass-by-reference

1.13 Classes

1.14 Inheritance

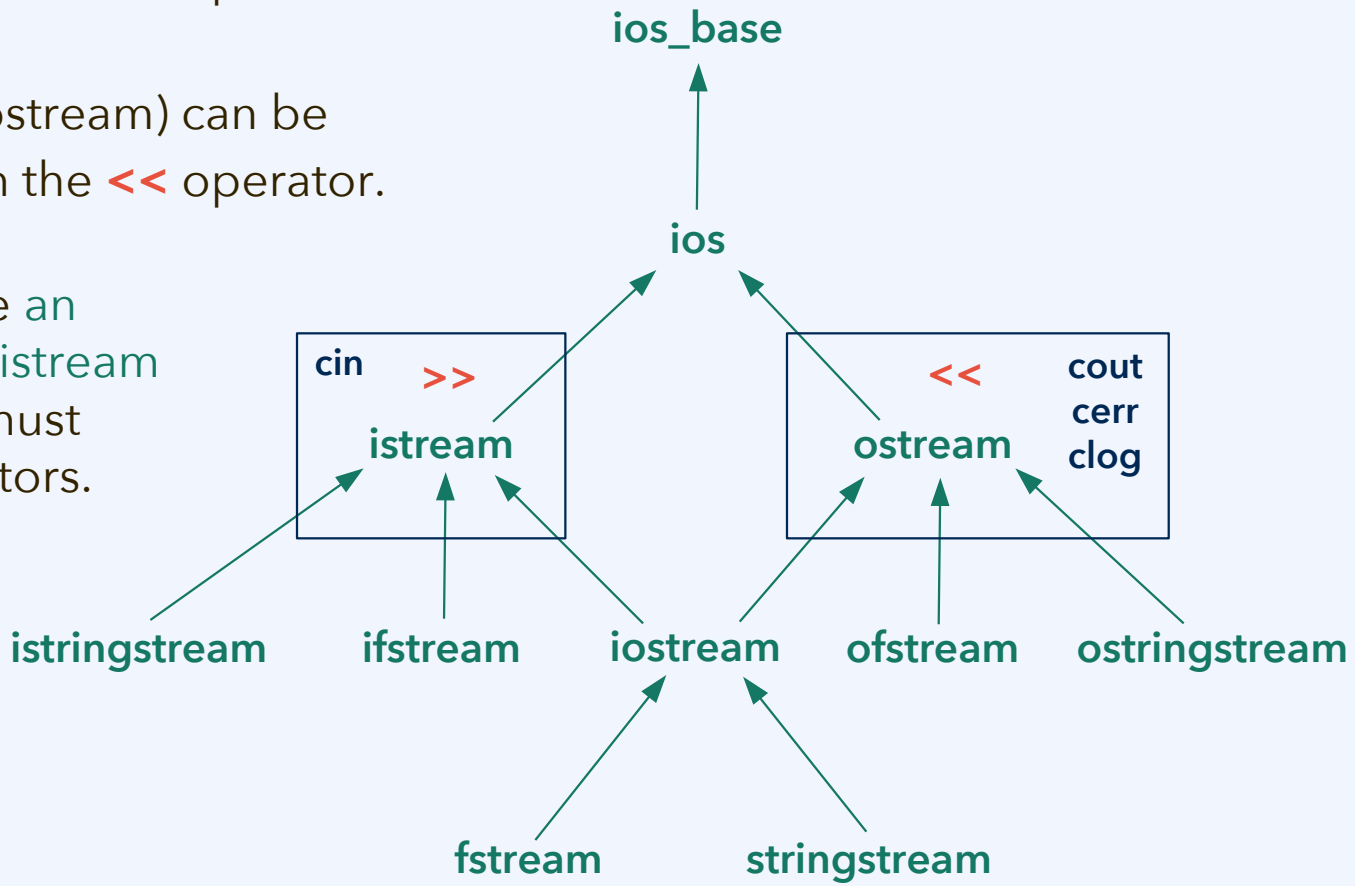
1.15 Streams

Streams: An example of inheritance

An input stream (istream) can be used for output with the >> operator.

An output stream (ostream) can be used for output with the << operator.

Consequently, since an iostream is both an istream and an ostream, it must provide both operators.



I/O with the stream as a parameter

We often develop code where we expect to write to cout, or read from cin.

However, the code is more reusable if we include a stream object among the function parameters, and then pass cin/cout or something else (such as a stream that accesses a file) as appropriate. If used for reading, the parameter type should be istream (provides **>>**); for writing, it should be ostream (**<<**).

Code from the "inheritance" example:

```
// write to *target
void BookIndex::out(std::ostream* target) const
{
   *target  << "Section " << this->chapter
            << "." << this->section
             << ", p. "<< this->page << "\n";
}
```

```
int main() {
   …
   std::cout << "\nBookIndex example:\n";
   litindex::BookIndex idx(1, 11, 24);
   idx.out(&std::cout);  // print status
   …
}
```

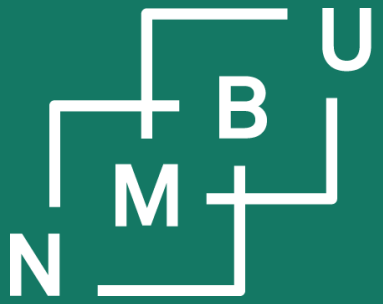
File input/output

To read from a file, open an **ifstream**, to write to a file, open an **ofstream**.

Example code "**read-from-file**" takes the name of a data file as a command line argument, passed to **int main(int argc, char** argv)** as argv[1].

```
int main(int argc, char** argv)
{
  ...
  char* file_name = argv[1];
  ...
  ifstream read_from_file(file_name);
  if(!read_from_file)
  {
    cerr << "Error! " << file_name << " cannot be read.\n";
    return EXIT_FAILURE;
  }

  int N = 0;          // N will be the number of spheres
  read_from_file >> N;  // input number of spheres

  // array containing the spheres; allocated on the heap
  Sphere* spheres = read_sphere_data(N, &read_from_file);

  ...
  delete[] spheres;
}
```

```
Sphere* read_sphere_data(int N, istream* source)
{
  /*
   * array containing all the spheres
   */
  Sphere* spheres = new Sphere[N]();

  /*
   * read all the data from standard input
   */
  for(int i = 0; i < N; i++)
  {
    *source >> spheres[i].size;
    for(int d = 0; d < 3; d++)
      *source >> spheres[i].coords[d];
  }
  return spheres;
}
```



Noregs miljø- og biovitskaplege universitet

Conclusion

INF205

28th September 2022

Glossary building and group formation

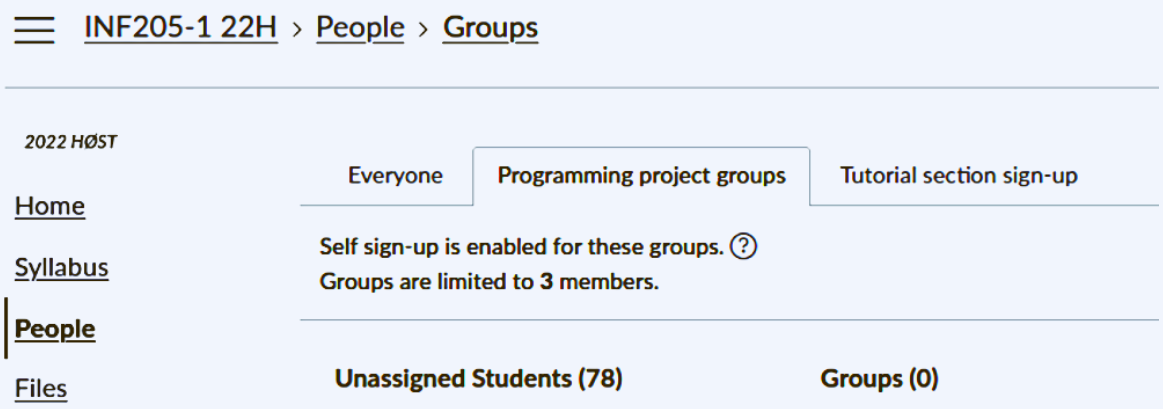
Norwegian University of Life Sciences

Glossary building:

- The glossary is up and continuously growing. We will continue to collect terms, including at the end of lectures if there is time.
- (If we have time: Let us reflect on the key concepts from this lecture.)

Formation of programming project groups:

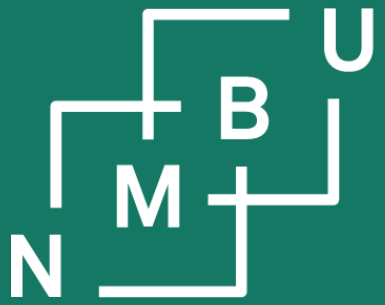
- Groups should ideally have three members; two are also acceptable.
- If you do not have a project group, you cannot pass INF205!



https://nmbu.instructure.com/courses/8427/groups#tab-5215

Deadline:

Self sign-up by 12th October (lecture time)



Norges miljø- og biovitenskapelige universitet

INF205
Resource-efficient programming

1 C++ basics

1.12 On pass-by-reference
1.13 Classes
1.14 Inheritance
1.15 Streams