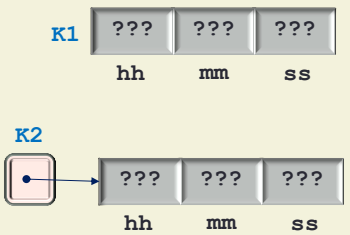
Lecture 7

- Classes *[6-7]*
  - Constructors (*konstruktorer*) *[7.3.1-7.3.3]*
  - Copy constructor (*kopieringskonstruktorer*) *[7.3.4]*
  - Destructor (*destruktorer*) *[7.4]*
- Examples
  - **`Clock`** and **`Flight`** classes
  - **`Matrice`** as a class -- (*revisited*)

Constructors (*konstruktorer*)

- When an object is declared, its data members are not initialized
  - Unless, constructor(s) are defined for the object's class

```
Clock K1;
Clock *K2 = new Clock;
```



A constructor is called **automatically** when an object of the class is declared or allocated dynamically (**`new`**)

```
class Clock {
  public:
    //constructors
    Clock();
    …
  private:
    //represent time as hh:mm:ss
    int hh, mm, ss;
} ;
```

Constructors

```
class Clock {
  public:
    //constructors
    Clock();
    …
  private:
    //represent time as hh:mm:ss
    int hh, mm, ss;
} ;
```

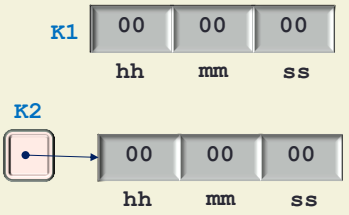
Default constructor
(no arguments)
- Same name as class name
- No return type

If no constructors are programmed then the compiler generates a default constructor

Default constructor is called

```
Clock K1;
Clock *K2 = new Clock;
```

```
Clock::Clock()
{
   hh = mm = ss = 0;
}
```



Constructors, C++11

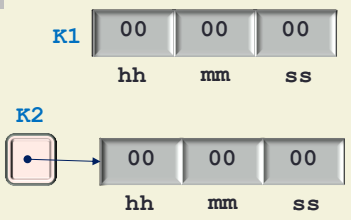
```
class Clock {
  public:
    //constructors
    Clock() = default;
    …
  private:
    //represent time as hh:mm:ss
    int hh {0};
    int mm {0};
    int ss {0};
} ;
```

Compiler generates default constructor

Member (brace) initializer can be used in the class definition

Default constructor is called

```
Clock K1;
Clock *K2 = new Clock;
```



Constructors

```
class Clock {
  public:
    //constructors
    Clock(int h, int m, int s);
    …
  private:
    //represent time as hh:mm:ss
    int hh, mm, ss;
} ;
```

No default constructor is generated by the compiler, if another constructor is provided

Constructor is called

```
Clock K3(12,30,5);
```

```
Clock K1; //compilation error
```


Constructors

```
Clock::Clock(int h, int m, int s)
{
   hh = h;
   mm = m;
   ss = s;
}
```

same as

```
Clock::Clock(int h, int m, int s)
 : hh(h), mm(m), ss(s)
{  }
```

Initialization list

```
Clock K3(12,30,5);
```

Initialization of constant data members

```
class AA
{
  public:
    //constructors
    AA(int i, Clock c) : SIZE(i), K_ref(c)
    { };
    …
  private:
    const int SIZE;
    Clock &K_ref;
} ;
```

```
Clock K1(10,30,0);
AA a(12, K1);
```

Constants and references are initialized through the initialization list of a constructor

Constructors

```
class Clock {
  public:
    //constructors
    Clock() = default;
    Clock(int h, int m, int s = 0);
    …
  private:
    //represent time as hh:mm:ss
    int hh {0};
    int mm {0};
    int ss {0};
} ;
```

Constructors can be overloaded

Constructor is called

```
Clock K3(12,30,5);
Clock K1;
```

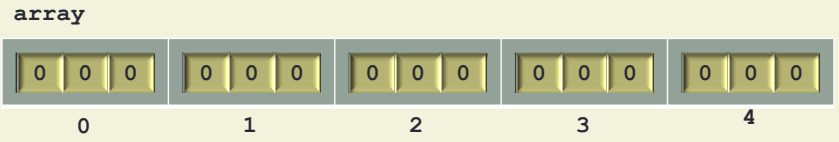
See `clock.h`
`clock.cpp`
`test_clock.cpp`

Constructors

```
const int SIZE = 5;
Clock array[SIZE];

for(int i = 0; i < 4; i++)
      array[i].display();
```

Default constructor
Clock::Clock() called for each object



Association: objects containing other objects

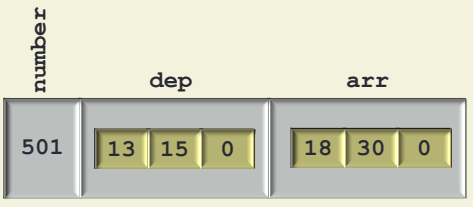
- A **Flight** object has two **Clock**s, to represent departure (**dep**) and arrival (**arr**) times

```
#include "clock.h"

class Flight
{
  public:
    ...
  private:
   int number;
   Clock dep;
   Clock arr;
};
```

Sec. 6 and 7.2: **Flight** example
See also Fö 7 code

UML: Composition (*has-relation*)




```
Flight TAP501;
```

Constructors

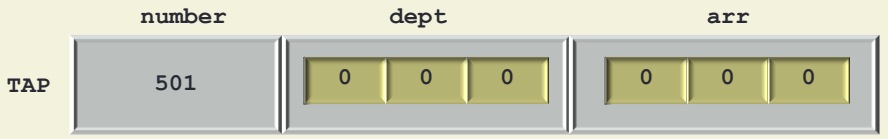
The default constructor for each **Clock** is automatically called

```
#include "clock.h"

class Flight {
  public:
    Flight(int n) : number(n) { };
    Flight(int n, int dh, int dm, int ah, int am);
    …
  private:
   int number;
   Clock dep;
   Clock arr;
};
```

arrival *departure*

```
Fight TAP(501);
```



Constructor

```
Flight::Flight(int n, int dh, int dm, int ah, int am)
 : number(n),
   dep {dh, dm, 0}, //dep(Clock(dh,dm,0)),
   arr {ah, am, 0}  //arr(Clock(ah,am,0))
{ };
```

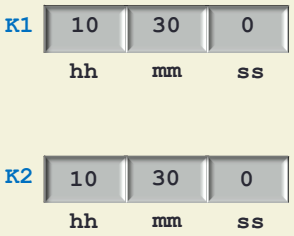
Copy Constructor (*kopieringskonstruktorer*)

```
Clock K1(10,30,0);
Clock K2(K1);
//Clock K2 = K1;
```

Copy constructor is called

There is a **default copy constructor**

- A **shallow copy** is performed



Copy Constructor

```
class Clock {
  public:
    //constructors
    Clock();
    Clock(int h, int m, int s);
    Clock(const &K);
    …
  private:
    //represent time as hh:mm:ss
    int hh {0};
    int mm {0};
    int ss {0};
} ;
```

Compiler generates a **default copy constructor**

- A **shallow copy** is performed

```
Clock::Clock(const &K)
{
  hh = K.hh;
  mm = K.mm;
  ss = K.ss;
}
```

When is the copy constructor called?

```
Clock f(Clock K1)
{
   Clock K2;
   …;
   return K2;
}
```

Call by value: copy constructor is called

Return an object: copy constructor is called

```
Clock my_rolex;
f(my_rolex);
```

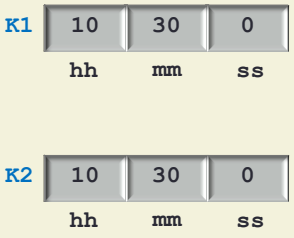
Copy constructor is called **automatically** when

- Call-by-value
- Return an object of the class

Copy Constructor

```
Clock K1(10,30,0);
//Clock K2(K1);
Clock K2 = K1;
```

Copy constructor is called



Default constructor called for **K2**

```
Clock K2;
K2 = K1;
```

Copy constructor **NOT** called
Assignment operator called

When is it needed to define a copy constructor?

- **Shallow copy** is not always enough
  - Dynamic memory allocation
- Let us recall the matrices example of Fö 4
  - Define class **`Matrice`**

Download the code from course web page, Fö 7

$$A = \begin{bmatrix} 1 & 2 & 3 & -1 \\ 4 & 5 & 0 & 0 \\ 6 & -1 & 6 & 8 \end{bmatrix}$$

4 columns

3 lines

Example: Matrices

- Let us create matrices where the number of lines and columns is set by the user

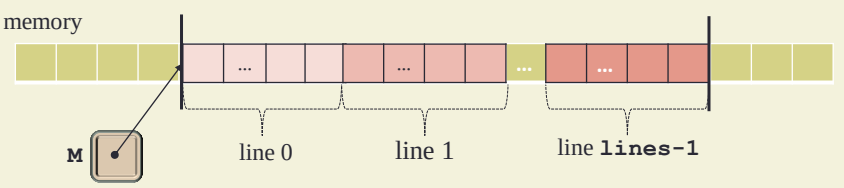
```
int lines, cols;
cout << "Number of lines and columns: ";
cin >> lines >> cols;

//create a lines×cols matrix M
double* M = new double [lines*cols];
```

Allocate space for an array with **`lines×cols`** slots

This is now the job of the constructor

M[i,j] is **`M[i*cols+j]`**



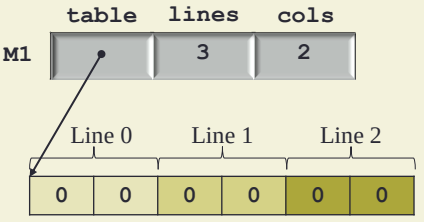
Example: Matrices

See matrices.h

```
class Matrice
{
  public:
     Matrice(int l, int c, int init);
    …
  private:
     int lines;
     int cols;
     double *table;
};
```

Constructor

```
Matrice M1(3,2,0);
```



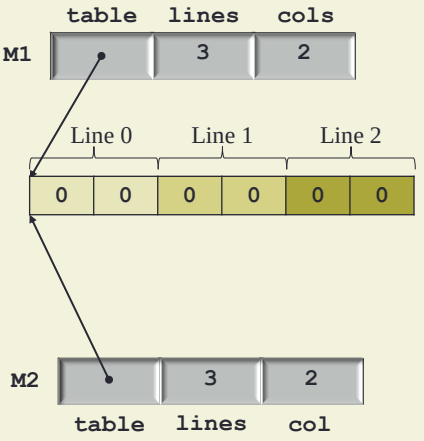
Example: Matrices

```
Matrice M1(3,2,0);
Matrice M2(M1);
```

Copy constructor is called

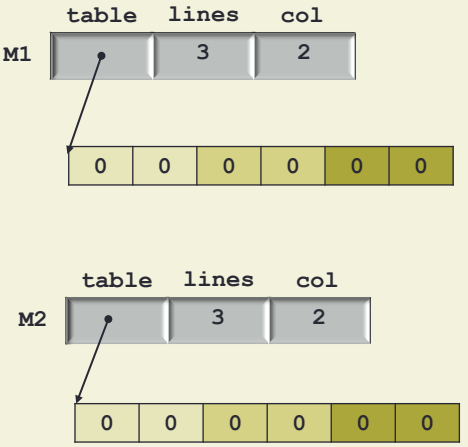
Answer: shallow copy!!

What is the problem?



Example: Matrices

```
class Matrice {
  public:
   Matrice(…);
   Matrice(const Matrice &M);
    …
  private:
    …
};
```




```
Matrice M1(3,2,0);
Matrice M2(M1);
```

A copy constructor needs to be defined
- To perform **deep copy**

Copy Constructor: deep copy

```
Matrice::Matrice(const Matrice &M)
{
  lines = M.lines;
  cols = M.cols;
  //Allocate memory
  table = new double [lines*cols];
  //Initialize matrix with values from M
  for(int i = 0; i < lines*cols; i++)
        table[i] = M.table[i];
}
//Assumption: M.lines != 0 and  M.cols != 0
```

Why should **M** be passed by reference? *(see slide 15)*

The destructors

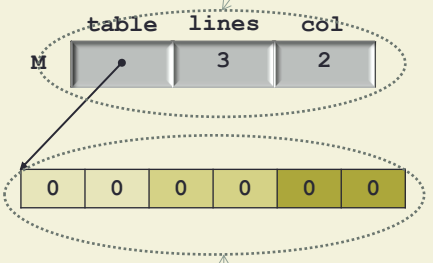
Constructor is called

```
void f(){
  Matrice M(3,2);
  …;
  return;
}
```

Object **M** is about to cease existing

Who deallocates the memory?

Memory allocated **automatically** is also automatically deallocated



Memory allocated **explicitly** with **new** must be explicitly deallocated with **delete**

```
delete [] M.table;
```


Read sec. 7.4

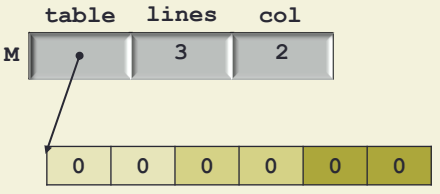
Destructor *(destruktorer)*

```
Matrice::~Matrice()
{
   //Deallocate memory
    delete [] table;
}
```

- Used to "*clean up*" when an object ceases to exist
- No arguments
- Cannot be overloaded
- Called automatically

```
void f(){
  Matrice M(3,2);
  …;
  return;
}
```

Destructor for object object **M** is called



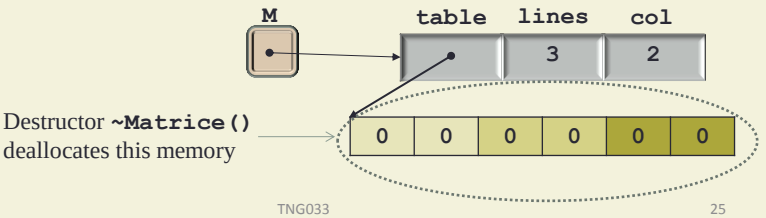
Destructor

- Destructor is also called when
  - **delete** of an object is executed

```
int main(){
  Matrice *M = new Matrice(3,2);
  …;
  delete M;
  …;
}
```

Constructor is called

Destructor is called for **M**



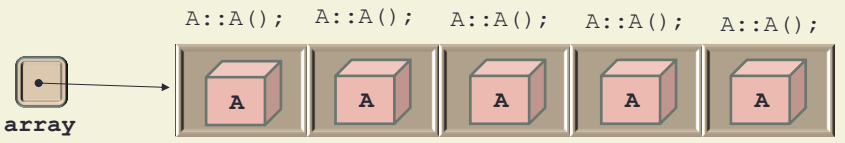
Destructor **~Matrice()** deallocates this memory

Destructor

```
class A
{
   public:
     A();
     …
     ~A();
    private:
     //data members
}
```

```
A *array = new A[5];
```

Constructor is called for each object in the array




```
delete [] array;
```

Destructor is called for each object in the array

The three essential

- If your class **allocates dynamically memory** then the class should have
  - Copy constructor
  - Destructor
  - Assignment operator -- coming Fö

If no destructor is provided for your class then the compiler generates one

```
Clock::~Clock()
{
  //Do nothing
}
```



```
//Return a matrix with same values as M but diagonal set to v
//M is a square matrix
Matrice diagonal_set(Matrice M, int n, double v)
{
   Matrice temp(M);
   for(int i = 0; i < n; i++)
       temp.set(i, i, v);
   return temp;
}

int main()
{
   Matrice M1(3,3,5);
   cout << diagonal_set(M1, 3, -1) << endl;
   Matrice* M2 = new Matrice(4,4,0);
   cout << M2 << endl;
   delete M2;
   return 0;
}
```

Copy constructor called

Copy constructor called

Destructor called twice: **M** and **temp**

Constructor called

Copy constructor called

Constructor called

Destructor called for **M2**

Destructor called for **M1**

Homework

- Study the class **`Matrice`** (download code)
  - **`matrice.h`**
  - **`matrice.cpp`**
  - **`test_matrices.cpp`**
- Read sections 7.1, 7.2, [7.3.1-7.3.3], 7.4 of course book

Next …

- **Read** about constant objects and constant member functions – [sec. 8.1]
- Classes (*cont.*) [sec. 8]
  - Pointer **`this`** [sec. 8.2]
  - Assignmnet operator [sec. 8.4.4]
  - **`friend`** (*vänner*) functions [sec. 8.3]
- Start looking into Lab 2
  - Lists are used to implement sets
  - No repeated values in the sets (lists)
  - Lists are sorted
  - Dummy node used in the lists implementation