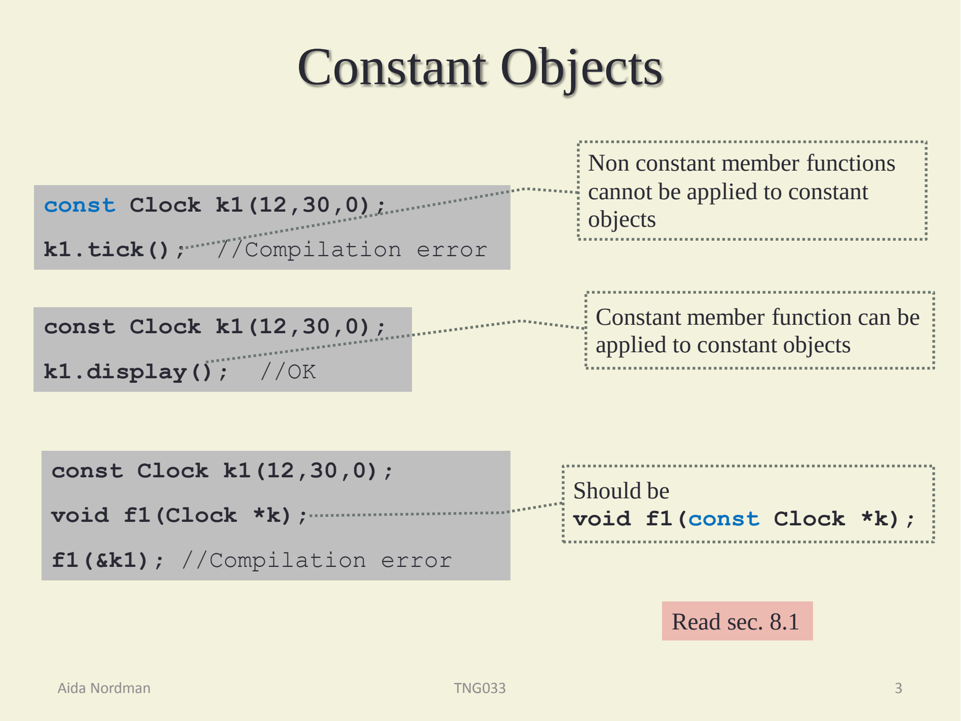
Lecture 8

- Constant members functions [8.1]
- Constant objects [8.1]
- Pointer **`this`** [8.2]
- Operator overloading (*operatorer*)
  - Assignment operator (*tilldelningsoperatorn*) [8.4.4]
  - **`operator<<`** (**`operator>>`**)
- Friend functions (*vänner*) [8.3]
- Exemples
  - Class **`Clock`**
  - Class **`Matrice`**

Class **`Clock`**

```
class Clock
{
  public:
    //constructors
    Clock() = default;
    Clock(int h, int m, int s);

    int get_hours()   const;
    int get_minutes() const;
    int get_seconds() const;
    void display(bool write_sec = true) const;

    void tick(); //add 1s more to the clock

    void reset(); //resets the clock to 00:00:00

    …

  private:
    //represent time as hh:mm:ss
     int hh {0};  //0-23
     int mm {0};  //0-59
     int ss {0};  //0-59
} ;
```

Constant member functions



Constant Objects

```
const Clock k1(12,30,0);
k1.tick();   //Compilation error
```

Non constant member functions cannot be applied to constant objects

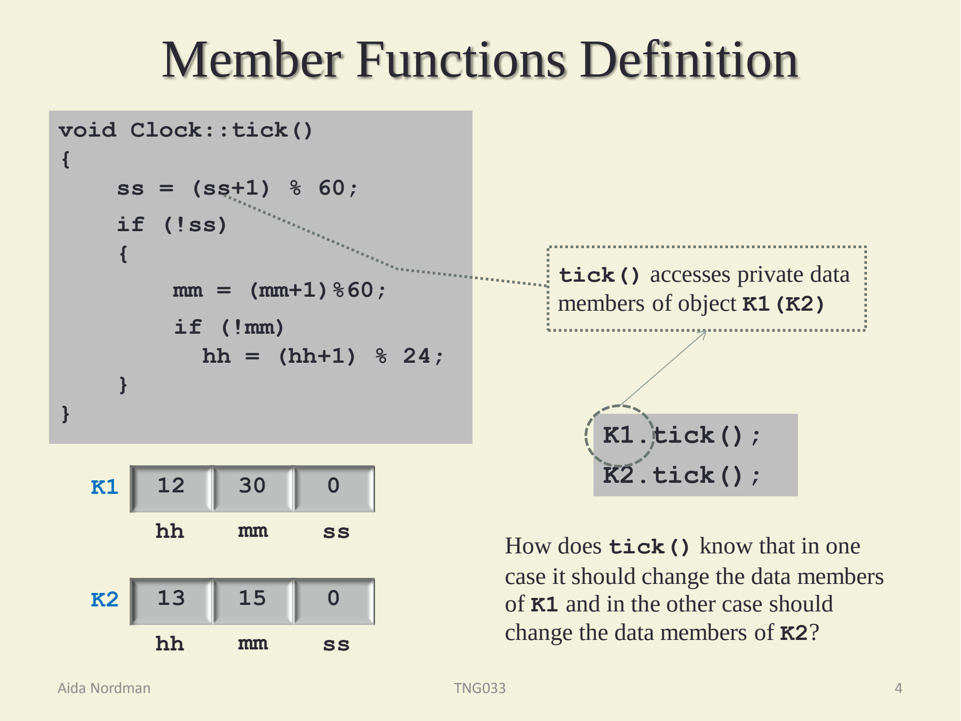
```
const Clock k1(12,30,0);
k1.display();   //OK
```

Constant member function can be applied to constant objects

```
const Clock k1(12,30,0);
void f1(Clock *k);
f1(&k1); //Compilation error
```

Should be
`void f1(const Clock *k);`

Read sec. 8.1



Member Functions Definition

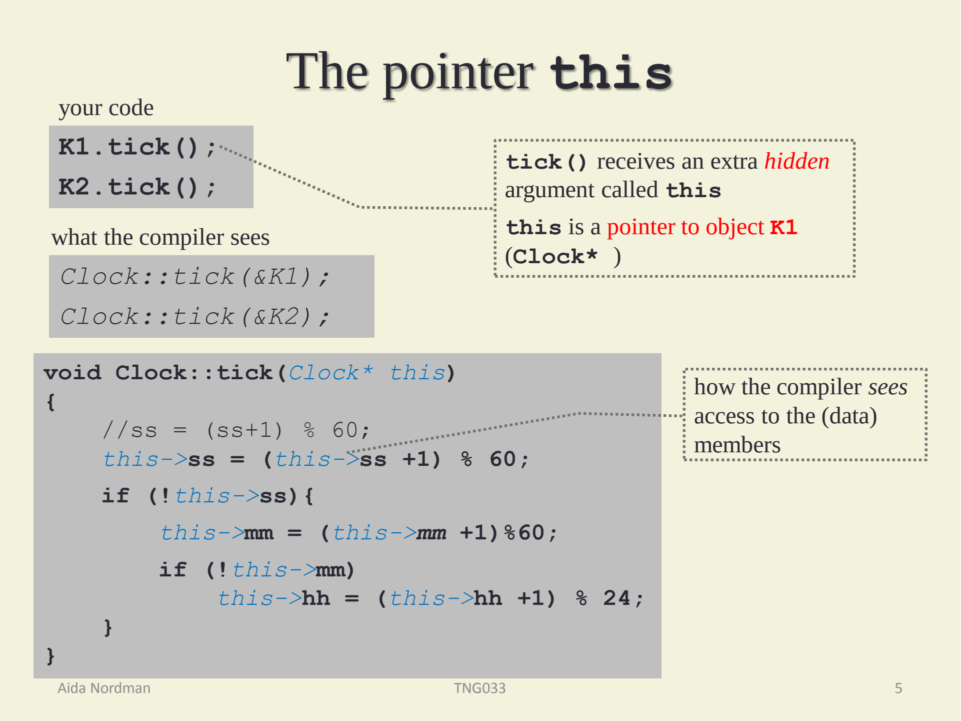
```
void Clock::tick()
{
    ss = (ss+1) % 60;
    if (!ss)
    {
        mm = (mm+1)%60;
        if (!mm)
          hh = (hh+1) % 24;
    }
}
```

`tick()` accesses private data members of object `K1(K2)`

```
K1.tick();
K2.tick();
```

K1	12	30	0
	hh	mm	ss
K2	13	15	0
	hh	mm	ss

How does `tick()` know that in one case it should change the data members of `K1` and in the other case should change the data members of `K2`?



The pointer `this`

your code

```
K1.tick();
K2.tick();
```

what the compiler sees

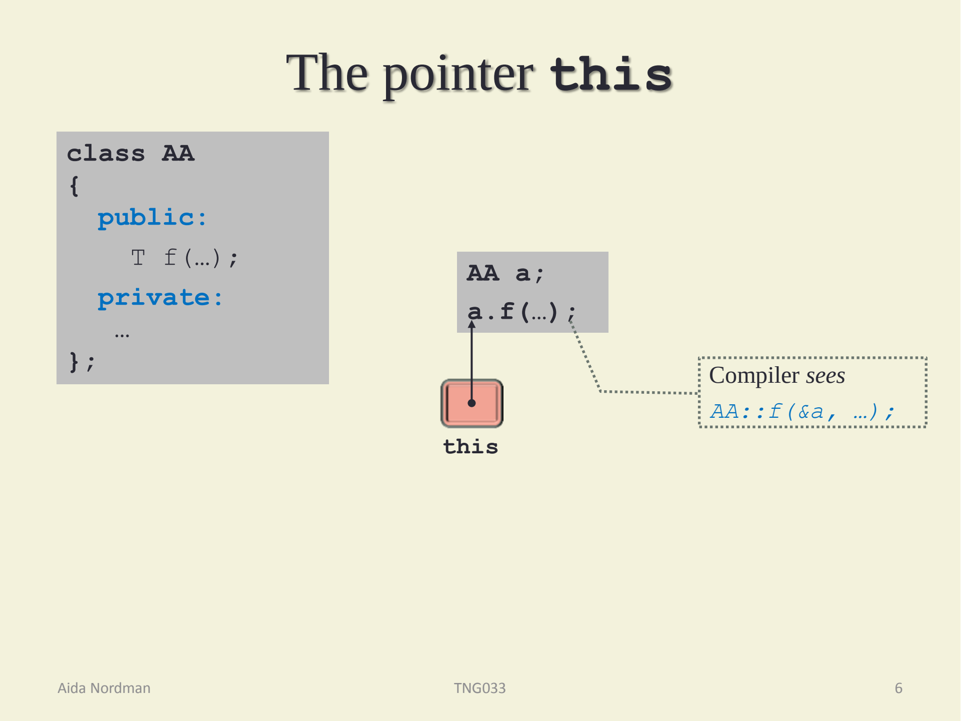
```
Clock::tick(&K1);
Clock::tick(&K2);
```

tick() receives an extra *hidden* argument called **this**

this is a pointer to object **K1** (**Clock***)

```
void Clock::tick(Clock* this)
{
    //ss = (ss+1) % 60;
    this->ss = (this->ss +1) % 60;
    if (!this->ss){
        this->mm = (this->mm +1)%60;
        if (!this->mm)
            this->hh = (this->hh +1) % 24;
    }
}
```

how the compiler *sees* access to the (data) members



The pointer `this`

```
class AA
{
  public:
    T f(…);
  private:
    …
};
```

```
AA a;
a.f(…);
```

Compiler *sees*

```
AA::f(&a, …);
```

When to use pointer `this` in the code?

- Yes! *e.g.* to call member functions in sequence (cascading)

```
//add 3 seconds to K1
K1.tick().tick().tick();
```

`void`

Does not work!!
Must return a (reference) to a `Clock` (`K1`)

```
void Clock::tick()
{ …; }
```

Every `tick()` should return (reference to) clock `K1` after the inc. of 1sec.

```
Clock& Clock::tick()
{
    … //as slide 5
    return *this;
}
```

It works ☺

The compiler automatically gets the address of object `*this` and returns its address

What if …?

```
Clock Clock::tick()
{
    … //as slide 4
    return *this;
}
```

```
Clock K1(10,20,0);
//add 3 seconds to K1
K1.tick().tick().tick();

display(K1);
```

A copy of the object `K1` is returned

Read sec. 8.2

What if …?

```
Clock* Clock::tick()
{
    … //as in slide 10
    return this;
}
```

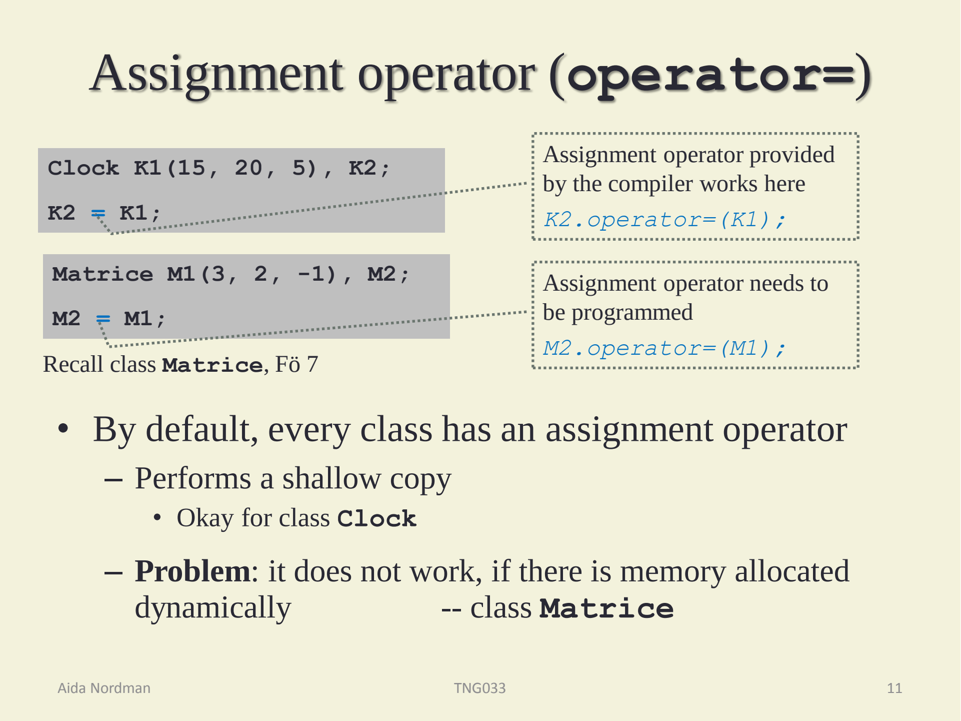
```
//add 3 seconds to K1
K1.tick().tick().tick();
```

Clock*

Compilation error!

```
//add 3 seconds to K1
((K1.tick())->tick())->tick();
```

- Constant objects/members functions [8.1]
- Pointer **this** [8.2]
- Operator overloading (*operatorer*)
  - Assignment operator (*tilldelningsoperatorn*) [8.4.4]
  - **operator<<** (**operator>>**)
- Friend functions (*vänner*) [8.3]
- Exemples
  - Class **Clock**
  - Class **Matrice**



Assignment operator (operator=)

```
Clock K1(15, 20, 5), K2;

K2 = K1;
```

Assignment operator provided by the compiler works here

K2.operator=(K1);

```
Matrice M1(3, 2, -1), M2;

M2 = M1;
```

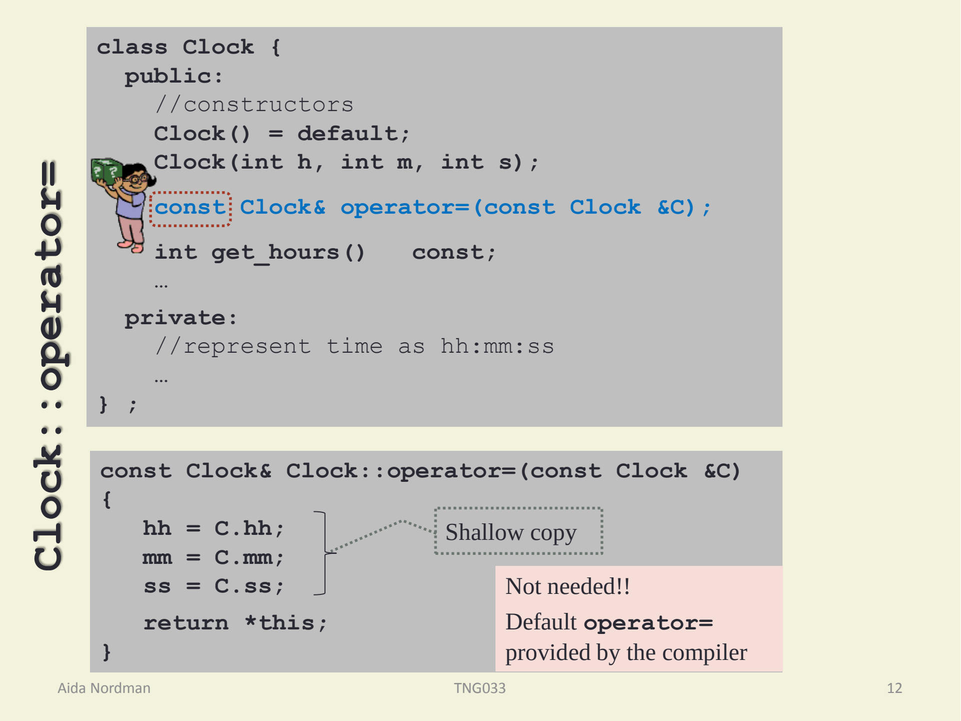
Assignment operator needs to be programmed

M2.operator=(M1);

Recall class **Matrice**, Fö 7

- By default, every class has an assignment operator
  - Performs a shallow copy
    - Okay for class **Clock**
  - **Problem**: it does not work, if there is memory allocated dynamically -- class **Matrice**

Aida Nordman TNG033 11



Clock::operator=

```
class Clock {
  public:
    //constructors
    Clock() = default;
    Clock(int h, int m, int s);

    const Clock& operator=(const Clock &C);

    int get_hours()   const;
    …
  private:
    //represent time as hh:mm:ss
    …
} ;
```

```
const Clock& Clock::operator=(const Clock &C)
{
   hh = C.hh;
   mm = C.mm;
   ss = C.ss;

   return *this;
}
```

Shallow copy

Not needed!!

Default **operator=** provided by the compiler

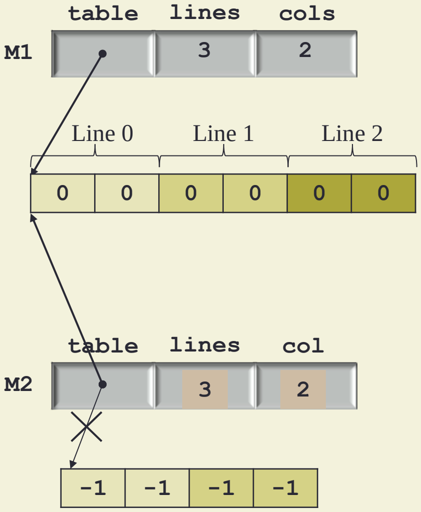
Aida Nordman TNG033 12

Example: Matrices *(revisited)*

```
Matrice M1(3,2,0)
Matrice M2(2,2,-1);

M2 = M1;
```

Shallow copy is not the solution!!

Memory leak!!



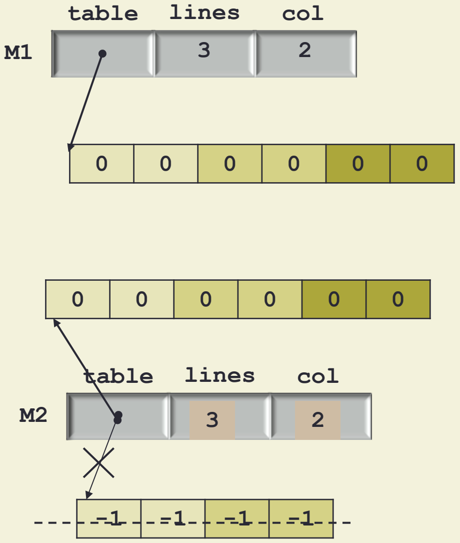
Deep copy

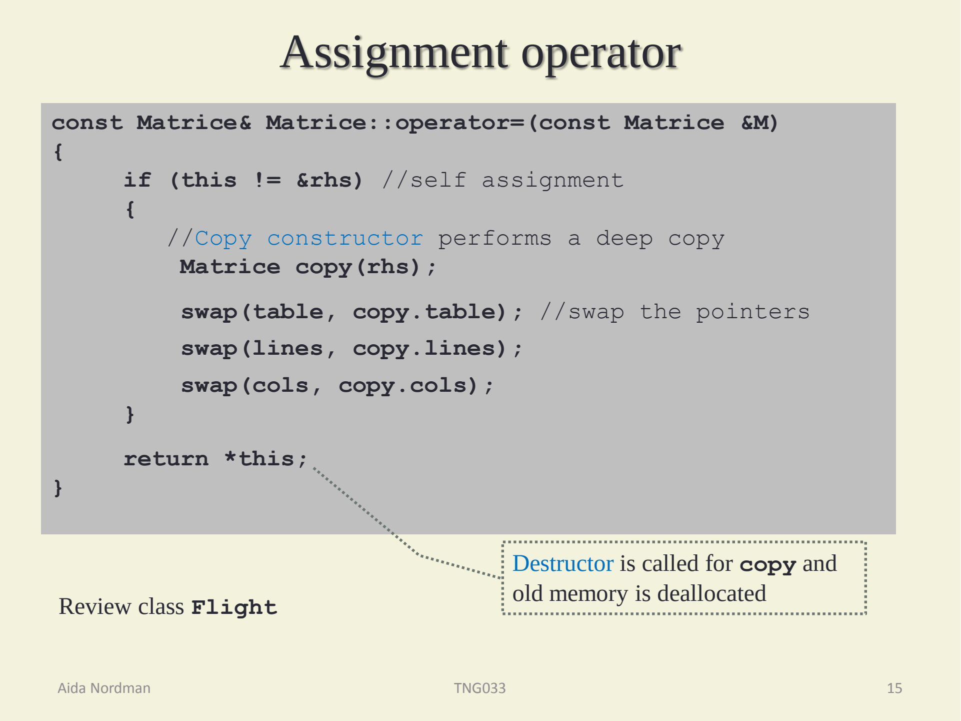
```
Matrice M1(3,2,0)
Matrice M2(2,2, -1);

M2 = M1;
```

Assignment operator should

1. Free old memory
2. Perform a **deep copy**

Important: If one programs a copy constructor then one needs also to program an assignment operator



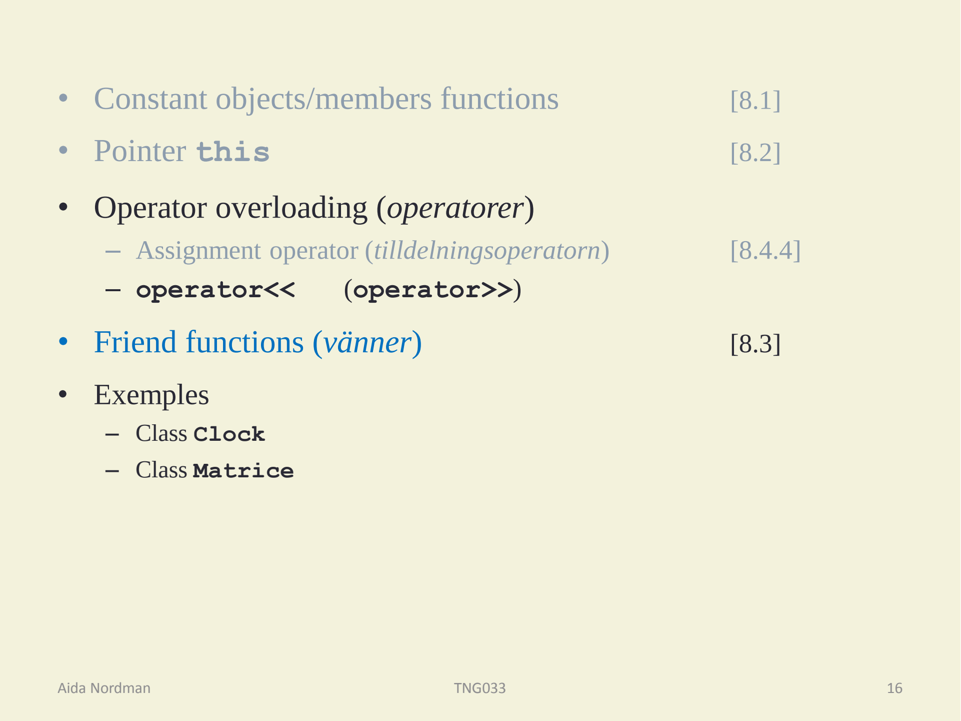


Assignment operator

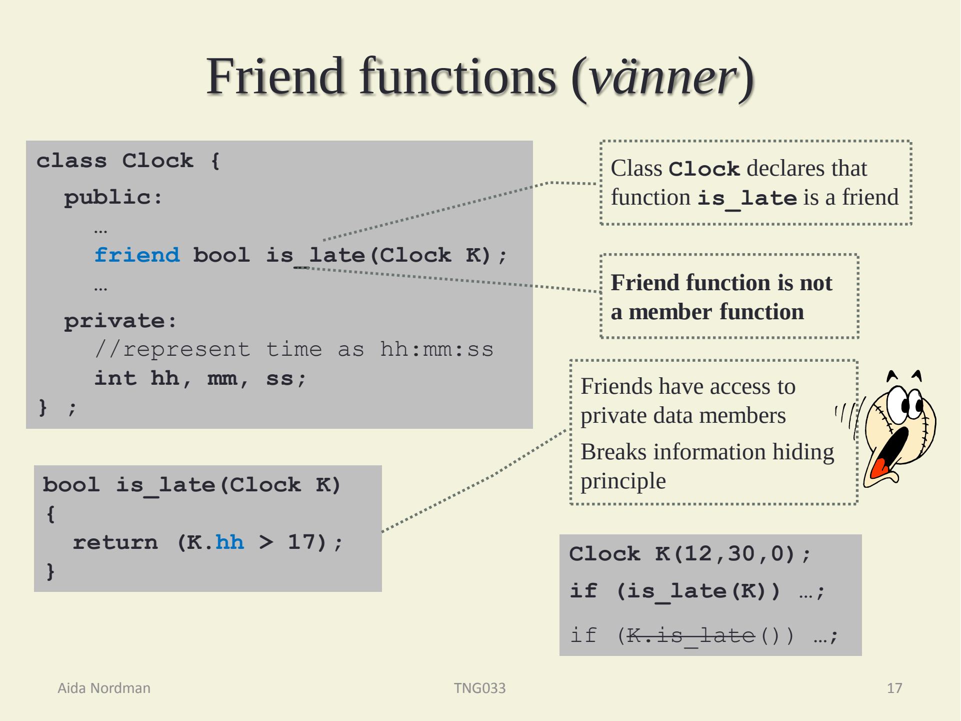
```
const Matrice& Matrice::operator=(const Matrice &M)
{
      if (this != &rhs) //self assignment
      {
         //Copy constructor performs a deep copy
          Matrice copy(rhs);

          swap(table, copy.table); //swap the pointers
          swap(lines, copy.lines);
          swap(cols, copy.cols);
      }

      return *this;
}
```

Destructor is called for **copy** and old memory is deallocated

Review class **Flight**



- Constant objects/members functions [8.1]
- Pointer **this** [8.2]
- Operator overloading (*operatorer*)
  - Assignment operator (*tilldelningsoperatorn*) [8.4.4]
  - **operator<<** (**operator>>**)
- Friend functions (*vänner*) [8.3]
- Exemples
  - Class **Clock**
  - Class **Matrice**



Friend functions (*vänner*)

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

Class **Clock** declares that function **is_late** is a friend

Friend function is not a member function

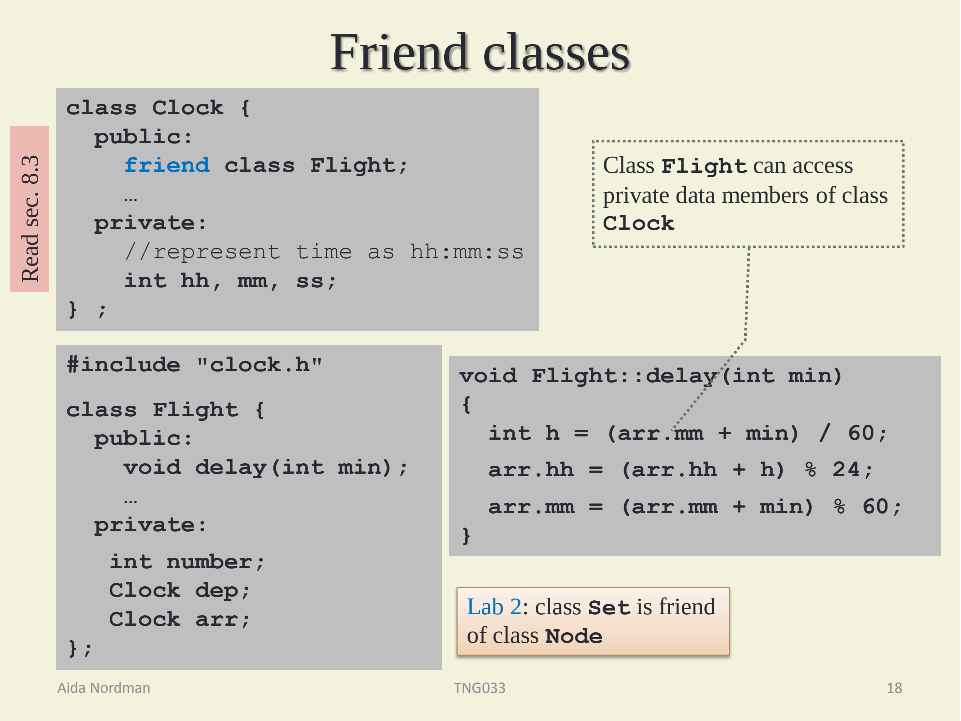
Friends have access to private data members

Breaks information hiding principle

```
bool is_late(Clock K)
{
  return (K.hh > 17);
}
```

```
Clock K(12,30,0);
if (is_late(K)) …;
```

if (~~K.is_late~~()) …;



Friend classes

Read sec. 8.3

```
class Clock {
  public:
    friend class Flight;
    …
  private:
    //represent time as hh:mm:ss
    int hh, mm, ss;
} ;
```

Class **Flight** can access private data members of class **Clock**

```
#include "clock.h"

class Flight {
  public:
    void delay(int min);
    …
  private:
     int number;
     Clock dep;
     Clock arr;
};
```

```
void Flight::delay(int min)
{
  int h = (arr.mm + min) / 60;
  arr.hh = (arr.hh + h) % 24;
  arr.mm = (arr.mm + min) % 60;
}
```

Lab 2: class **Set** is friend of class **Node**

operator<<

```
int i = 5, j = 6;
cout << i << endl;
cout << j << i;
```

```
Clock rilex(12,30,0);
Clock K3;
cout << rilex << endl;
cout << K3 << rilex;
```

We need to implement the function

```
ostream& operator<<(ostream& os, const Clock& K);
```

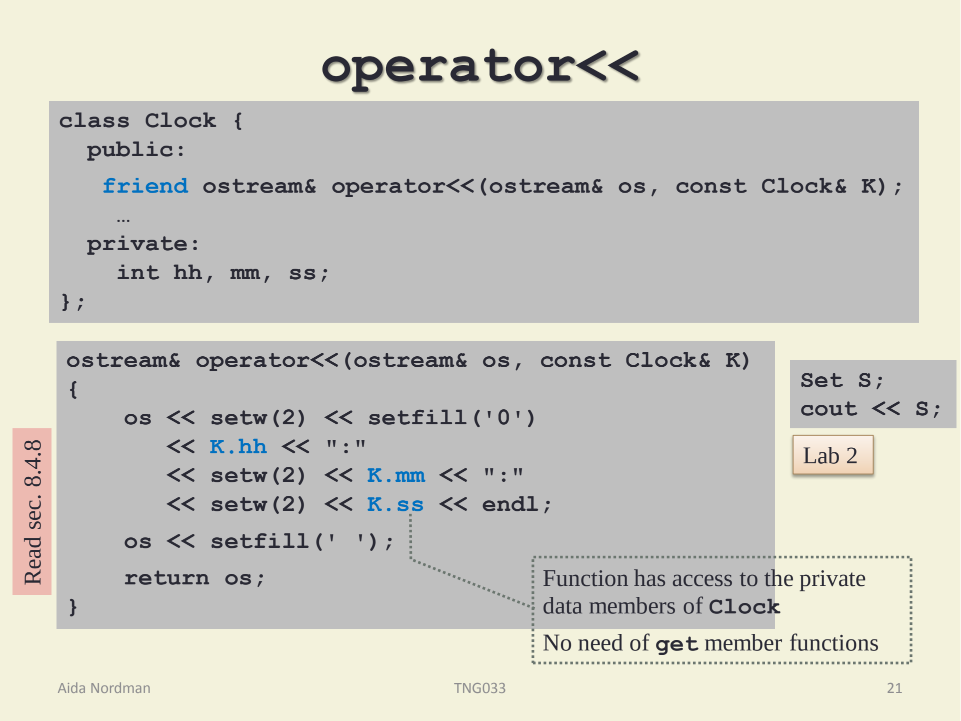
operator<<

```
ostream& operator<<(ostream& os, const Clock& K)
{
    os << setw(2) << setfill('0')
       << K.get_hours() << ":"
       << setw(2) << K.get_minutes() << ":"
       << setw(2) << K.get_seconds() << endl;
    os << setfill(' ');
    return os;
}
```

```
12:30:00
```

But, what if the class does not provide **`get`** member functions?

Solution: define a friend function
`operator<<(ostream& os, const Set& S);`



operator<<

```
class Clock {
  public:
   friend ostream& operator<<(ostream& os, const Clock& K);
    …
  private:
    int hh, mm, ss;
};
```

```
ostream& operator<<(ostream& os, const Clock& K)
{
    os << setw(2) << setfill('0')
       << K.hh << ":"
       << setw(2) << K.mm << ":"
       << setw(2) << K.ss << endl;
    os << setfill(' ');
    return os;
}
```

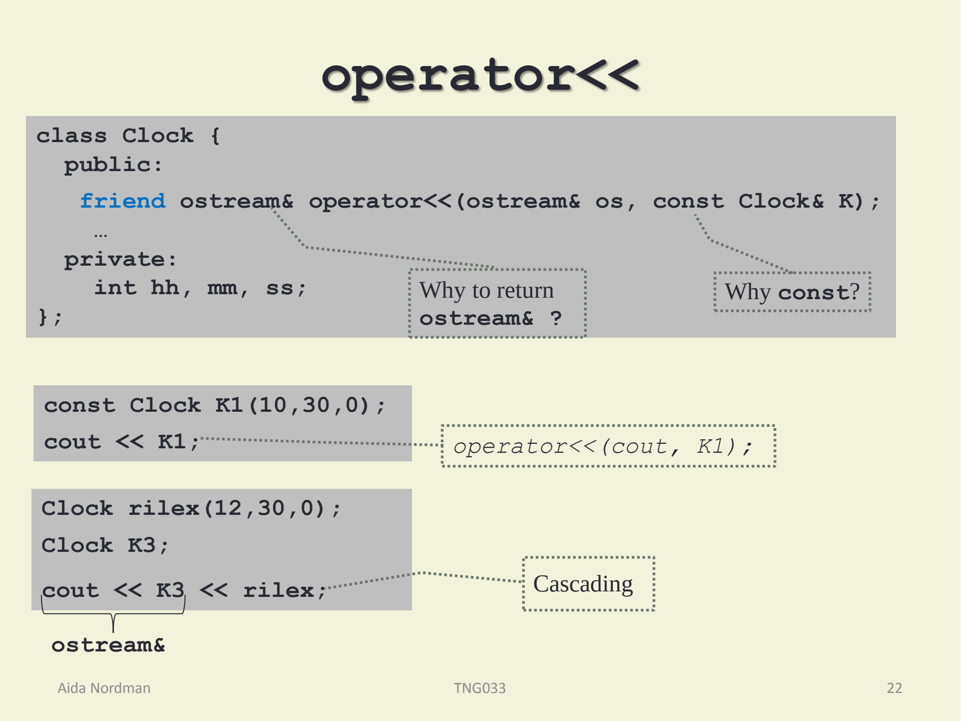
```
Set S;
cout << S;
```

Lab 2

Read sec. 8.4.8

Function has access to the private data members of **`Clock`**

No need of **`get`** member functions



operator<<

```
class Clock {
  public:
   friend ostream& operator<<(ostream& os, const Clock& K);
    …
  private:
    int hh, mm, ss;
};
```

Why to return **`ostream& ?`**

Why **`const`**?

```
const Clock K1(10,30,0);
cout << K1;
```

`operator<<(cout, K1);`

```
Clock rilex(12,30,0);
Clock K3;
cout << K3 << rilex;
```

`ostream&`

Cascading

Important

1. Read advised book sections
2. Study class **`Clock`** and **`Matrice`**
   - Class **`Matrice`** in Fö 8
     - **`operator=`**, **`operator<<`**, **`operator>>`**

Where do we want to go?

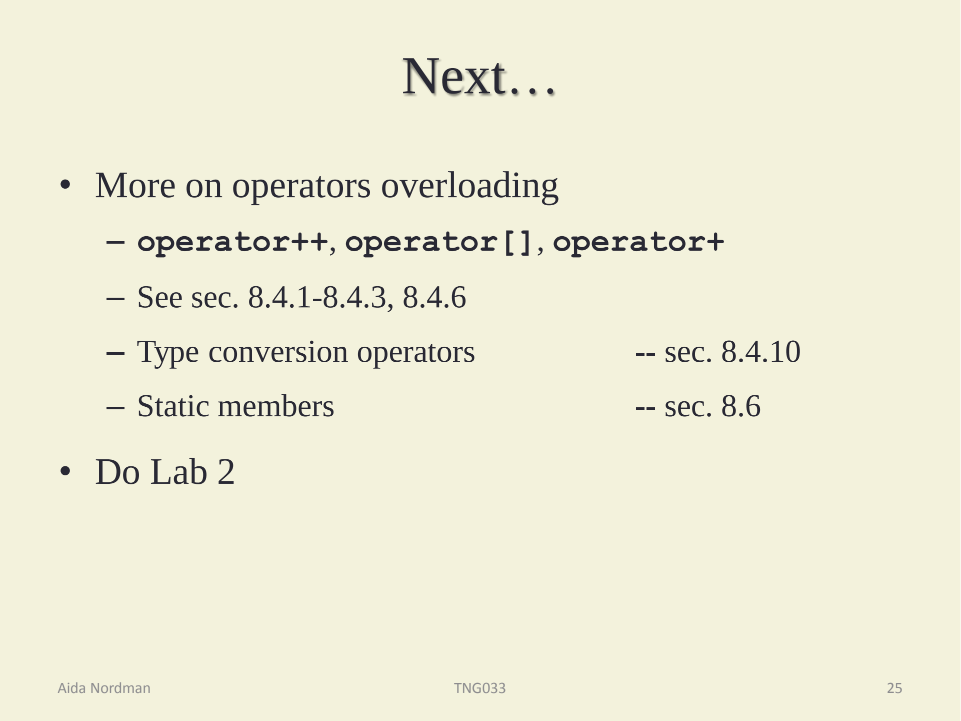
- **Aim**: to be able to define new classes (data types) with the same *look and feel* as the predefined types

```
int i = 2, j = 0, k;
j = ++i;
cout << ++(++i);
k = i+j;
cout << i << j << k << endl;
```

```
Clock K1(10,20,0), K2, K3;
K2 = ++K1;    //same as K1.tick();
cout << ++(++K1);
K3 = K1 + K2;
K3 = K1 + 2;
cout << K1 << K2 << K3 << endl;
```

Discussed in today’s lecture

`operator+`
`operator++`
`operator=`
`operator<<`
conversion operators
need to be defined for class **`Clock`**



Next…

- More on operators overloading
  - **`operator++`**, **`operator[]`**, **`operator+`**
  - See sec. 8.4.1-8.4.3, 8.4.6
  - Type conversion operators -- sec. 8.4.10
  - Static members -- sec. 8.6
- Do Lab 2