# Lecture 8

## Classes, operator overload

TDDD86: DALP

Utskriftsversion av Lecture in *Data Structures, Algorithms and Programming Paradigms*
September 30th, 2024

IDA, Linköping University

8.1

## Content

## Contents

**1 Classes and objects** **1**
1.1 `class` . . . 2
1.2 `struct` . . . 3

**2 Operator overload** **4**
2.1 Print out objects . . . 5
2.2 `this` . . . 5
2.3 `const` . . . 5
2.4 Increment and decrement . . . 6

8.2

## 1 Classes and objects

### Classes and objects

- *Class*: A unit in a program that represents:
  - a "plan" or "blueprint" for a type of object
- *object*: a unit that combines *state* and *behavior*:
  - *Object-oriented programming (OOP)*: Program consisting of interactions between objects
  - *abstraction*: Separation between concepts and implementations. Objects give us abstraction in programming.

8.3

### Interface vs code

- When writting a class in C++ there is a seperation between:
  - **interface**: Declarations of functions, classes, members, etc...
  - **implementation**: Definition of how these are implemented
- C++ implements this separation between two type of source files:
  - `.h`: A "header"-file containing only the interface
  - `.cpp`: A "source"-fil containing the definition
- The content of `.h`-file is included in `.cpp`-files with `#include`
  - Lets the source file know about declarations implemented somewhere else
  - All compiled definitions are *linked* into an executable

8.4

## Preprocessor in C++

- Part of the compilation process in C++; recognize special statements #-statements, modify the source

code before compilation

| function | description |
|---|---|
| `#include <filename>` | insert a library file's contents into this file |
| `#include "filename"` | insert a user file's contents into this file |
| `#define name [value]` | create a preprocessor symbol ("variable") |
| `#if test` | `if` statement |
| `#else` | `else` statement |
| `#elif test` | `else if` statement |
| `#endif` | terminates an `if` or `if/else` statement |
| `#ifdef name` | `if` statement; `true` if ***name*** is defined |
| `#ifndef name` | `if` statement; `true` if ***name*** is *not* defined |
| `#undef name` | deletes the given symbol name |

8.5

## Structure in a .h-file

```
// foobar.h
#ifndef _foobar_h
#define _foobar_h

declarations;

#endif
```

8.6

# 1.1 class

## A "header" for a class

```
class ClassName {                  // i ClassName.h
public:
    ClassName(parameters);         // konstruktor
    returnType name(parameters); // medlemsfunktioner
    returnType name(parameters); // (beteende hos
    returnType name(parameters); // varje objekt)
private:
    type name;                     // medlemsvariabler
    type name;                     // (data hos varje objekt)
};
```

- Do not forget the semi-column at the end of the class definition!

8.7

## Exempel: Date.h

```
#ifndef _date_h
#define _date_h
class Date {
public:
    Date(int y, int m, int d);
    int daysInMonth();
    int getYear();
    int getMonth();
    int getDay();
    bool isLeapYear();
    void nextDay();
    string toString();

private:
    int year;
    int month;
    int day;
};
#endif
```

8.8

### Definition of class member

- In `ClassName.cpp` we write the body (definition) for the member functions and constructors declared in .h-file

```
// ClassName.cpp
#include "ClassName.h"

// medlemsfunktion
returnType ClassName::methodName(parameters) {
    statements;
}

// konstruktor
ClassName::ClassName(parameters) {
    statements;
}
```

  - Member functions/constructors can use object member

8.9

### Example: Date.cpp

```
#include <string>
#include "Date.h"

Date::Date(int y, int m, int d) { // construsctor
    year = y;
    month = m;
    day = d;
}

int Date::getYear() {             // member function
    return year;
}

bool Date::isLeapYear() {
    return year % 400 == 0 || (year % 100 != 0 && year % 4 == 0);
}

string Date::toString() {
    return std::to_string(year) + "-" +
    std::to_string(month) + "-" +
    std::to_string(day);
}
```

8.10

## 1.2 `struct`

### `struct`

- C++ also has a syntactic unit called **`struct`**
- Very much like a class, except that by default all members of a **`struct`** are **`public`** while members of a **`class`** are **`private`**

```
struct Point {
    int x;
    int y;
};
...
point p;
p.x = 15;
...
```

8.11

"A struct simply feels like an open pile of bits with very little in the way of encapsulation or functionality. A class feels like a living and responsible member of society with intelligent services, a strong encapsulation barrier, and a well defined interface"

-Bjarne Stroustrup

8.12

## 2 Operator overload

### Operator overload

- C++ allows to *overload* or redefine the behavior of most operators of the language:

```
+      -      *      /      %      ^      &      |      ~      !      <<     >>
=      +=     -=     *=     /=     %=     ^=     &=     |=     <<=    >>=
<      >      <=     >=     ==     !=     &&     ||
++     --     ->     ->*    ,      [ ]    ( )
new    new[]         delete       delete[]
```

- Be sensible when overloading it can lead to confusing code otherwise

8.13

### Syntax for operator overload

- Declare the operator in a .h-file, implementation goes in .cpp-fil

```
returnType operator op(parameters); // .h

returnType operator op(parameters) { // .cpp
    statements;
}
```

  - where `op` is an operator such as +, ==, <<, etc.
  - parameters are the operands of the operator; for example for `a + b` it would be **operator**+`(Foo a, Foo b)`

8.14

### Example

```
// Date.h
class Date {
    ...
};

bool operator ==(Date& d1, Date& d2);
bool operator !=(Date& d1, Date& d2);

// Date.cpp
bool operator ==(Date& d1, Date& d2) {
    return d1.getYear() == d2.getYear() &&
```

```
        d1.getMonth() == d2.getMonth() &&
        d1.getDay() == d2.getDay();
}

bool operator !=(Date& d1, Date& d2) {
    return !(d1 == d2); // calls operator ==
}
```

8.15

## 2.1 Print out objects

### Print out objects

- Overload the operator << with `ostream` and the type

```
ostream& operator <<(ostream& out, Type& name) {
    statements;
    return out;
}
```

  - operator return a reference to the stream so that it can be chained
    - `cout << a << b << c` is equivalent to `((cout << a) << b) << c`
    - technically `cout` is returned for each <<-operation

8.16

### Example

```
// Date.h
class Date {
   ...
};

ostream& operator <<(ostream& out, Date& date);

// Date.cpp
ostream& operator <<(ostream& out, Date& d) {
    out << d.getYear() << "-"
        << d.getMonth() << "-"
        << d.getDay();              // eller out << d.toString()
    return out;
}
```

8.17

## 2.2 this

### The reserved word **this**

- Just as in Java, the keyword **this** in C++ can be used to reference a pointer to the object

```
Date::Date(int year, int month, int day) {
    this->year = year;
    this->month = month;
    this->day = day;
}
```

- **this** uses -> not . because it is a pointer

8.18

## 2.3 const

### The reserved keyword **const**

- The keyword **const** indicates that a value cannot be changed

```
const int x = 4; // x will always be 4
```

- A reference paramter to **const** cannot be changed by the function:

```
void foo(const Date& d) { ... // foo cannot change d
```

  - Any attempt to change `d` within `foo` gives a compilation error

- A **const**-declared function cannot change the object state

```
class Date { ...
    int getYear() const; // doesn't change object state
```

  - A **const** `Date`-reference can only access **const**-members

8.19

### Date.h with **const**

```
#ifndef _date_h
#define _date_h
class Date {
public:
    Date(int y, int m, int d);
    int daysInMonth() const;
    int getYear() const;
    int getMonth() const;
    int getDay() const;
    bool isLeapYear() const;
    void nextDay();           // not const since it modifies the date
    string toString() const;

private:
    int year;
    int month;
    int day;
};

bool operator ==(const Date& d1, const Date& d2);
ostream& operator <<(ostream& out, const Date& date);
#endif
```

8.20

### Date.cpp with **const**

```
#include "Date.h"
...
bool Date::isLeapYear() const { // member function
    return year % 400 == 0 || (year % 100 != 0 && year % 4 == 0);
}

bool operator ==(const Date& d1, const Date& d2) {
    return d1.getYear() == d2.getYear() &&
         d1.getMonth() == d2.getMonth() &&
         d1.getDay() == d2.getDay();
}

ostream& operator <<(ostream& out, const Date& d) {
    out << d.getYear() << "-"
        << d.getMonth() << "-"
        << d.getDay();
    return out; // eller << d.toString()
}
```

- Correct use of **const** allows to check for **const**-correctness

8.21

## 2.4 Increment and decrement

### Operator ++ and --

```
int x = 0
cout << x++ << endl; // write: 0
cout << x << endl;   // write: 1

x = 0;
cout << ++x << endl; // write: 1
cout << x << endl;   // write: 1
```

- C++ use a hack to distinguish between prefix- and postfix- overload

```
class MyClass {
public:
...
    MyClass& operator ++(); // Prefix
    MyClass operator ++(int dummy); // Postfix
...
};
```

  - compilers give `dummy` the value of 0

8.22

## Example

- implementation:

```
MyClass& MyClass::operator ++() {
    *this += 1;
    return *this;
}

MyClass MyClass::operator ++(int dummy) {
    MyClass oldValue = *this; // Store the object's current value
    ++*this;
    return oldValue;
}
```

8.23