C++: An Introduction

Jörg Faschingbauer

Table of Contents

FASCHINGBAUER

1. Introduction
   - Introduction
2. Data Encapsulation
   - Classes and Objects
   - Constructors and Destructors
   - Object Copy
3. Functions and Methods
   - Overloading
   - Methods
   - `const`
   - `this`
   - References
   - `static`
   - Operatoren
4. Exceptions
   - Basics
   - `try` — `catch`
5. Templates
   - Function Templates
   - Class Templates
6. Standard Template Library
   - Basics
   - Sorting
   - Searching
   - Sequential Containers
   - Associative Containers
7. Dynamic Memory Allocation
8. Inheritance
9. Object Oriented Design

Overview

FASCHINGBAUER

1. Introduction
   - Introduction
2. Data Encapsulation
   - Classes and Objects
   - Constructors and Destructors
   - Object Copy
3. Functions and Methods
   - Overloading
   - Methods
   - `const`
   - `this`
   - References
   - `static`
   - Operatoren
4. Exceptions
   - Basics
   - `try` — `catch`
5. Templates
   - Function Templates
   - Class Templates
6. Standard Template Library
   - Basics
   - Sorting
   - Searching
   - Sequential Containers
   - Associative Containers
7. Dynamic Memory Allocation
8. Inheritance
9. Object Oriented Design

Overview

FASCHINGBAUER

1. Introduction
   - Introduction
2. Data Encapsulation
   - Classes and Objects
   - Constructors and Destructors
   - Object Copy
3. Functions and Methods
   - Overloading
   - Methods
   - `const`
   - `this`
   - References
   - `static`
   - Operatoren
4. Exceptions
   - Basics
   - `try` — `catch`
5. Templates
   - Function Templates
   - Class Templates
6. Standard Template Library
   - Basics
   - Sorting
   - Searching
   - Sequential Containers
   - Associative Containers
7. Dynamic Memory Allocation
8. Inheritance
9. Object Oriented Design

Background

FASCHINGBAUER

C is great:

- One has control over everything
- Dominant language for years
- Real Men don't do Object Oriented Programming

But ...

- Same bugs over and over
  - Memory leaks and other cleanup
  - Dangling pointers
  - ...
- The need for inheritance was obvious even without language support
  - Implementation idioms were similar
  - ... but never the same
  - ... and always had to have too much code

History

- 1980: Bjarne Stroustrup — “C with Classes”
- 1983: “C++”
- 1985: “The C++ Reference Manual” (Stroustrup)
- 1990: “The Annotated C++ Reference Manual” (Stroustrup)
- 1998: C++ ISO/IEC 14882:1998 — “C++98”
  - Standard Template Library
- 2003: Standard “Revision” (Fix) — “C++03”
- 2005 ...: “Technical Reports” (Fixes)
- 2011: “C++11”: most recent standard
  - Library extension
  - New syntax (“Unification” with C, ...)

Content

FASCHINGBAUER

- Data encapsulation — "Object"
  - Constructor, destructor
  - Access control: `public`, `protected`, `private`
  - Methods and operators (*Copy!*)
- Inheritance
  - "Classic" OO Design → science on its own
- Templates
- Standard Library and Standard Template Library (STL)
  - String
  - IO streams
  - Container classes
  - Threading

Overview

1. Introduction
   - Introduction
2. Data Encapsulation
   - Classes and Objects
   - Constructors and Destructors
   - Object Copy
3. Functions and Methods
   - Overloading
   - Methods
   - `const`
   - `this`
   - References
   - `static`
   - Operatoren
4. Exceptions
   - Basics
   - `try` — `catch`
5. Templates
   - Function Templates
   - Class Templates
6. Standard Template Library
   - Basics
   - Sorting
   - Searching
   - Sequential Containers
   - Associative Containers
7. Dynamic Memory Allocation
8. Inheritance
9. Object Oriented Design

Overview

FASCHINGBAUER

1. Introduction
   - Introduction
2. Data Encapsulation
   - Classes and Objects
   - Constructors and Destructors
   - Object Copy
3. Functions and Methods
   - Overloading
   - Methods
   - `const`
   - `this`
   - References
   - `static`
   - Operatoren
4. Exceptions
   - Basics
   - `try` — `catch`
5. Templates
   - Function Templates
   - Class Templates
6. Standard Template Library
   - Basics
   - Sorting
   - Searching
   - Sequential Containers
   - Associative Containers
7. Dynamic Memory Allocation
8. Inheritance
9. Object Oriented Design

"Objects" in C — `struct`

FASCHINGBAUER

Objects in C — `struct`

- Self defined "Composite" types
- *Copy* is supported by the compiler — but nothing else
  - Explicit assignment
  - Parameter passing
  - Function return value

```
struct point
{
    int x;
    int y;
};
struct point add_points(
    struct point p1,
    struct point p2)
{
     struct point ret;
     ret.x = p1.x + p2.x;
     ret.y = p1.y + p2.y;
     return ret;
}
```

Example: struct point

FASCHINGBAUER

Definition

```
struct point
{
    int x;
    int y;
};
struct point add_points(
    struct point rhs,
    struct point lhs);
void add_to_point(
    struct point *rhs,
    struct point lhs);
```

Usage

```
struct point A = {1,2},
             B = {2,4};
struct point C;

C = add_points(A, B);
add_to_point(&A, B);
```

`struct point` — Criticism

FASCHINGBAUER

Is `struct` good enough?

- Members are public
  - → Bugs are only a matter of time
  - Counter argument: "Real programmers don't write bugs"
- Function just hang around
- Clean initialization
  - *Contructor* (and *Destructor*)
  - Error checking
- Self defined operators — e.g. addition of `struct point`, using operator "+"
- *Methods* on *Objects*

Example: `class point`

FASCHINGBAUER

Definition

```
class point
{
public:
    point(int x, int y);
    int x() const;
    int y() const;
    point& operator+=(
        point addend);
private:
    int _x;
    int _y;
};
point operator+(
    point lhs,
```

Usage

```
point A(1,2), B(2,4);

point C = A + B;
A += B;
```

class point, analyzed (1)

FASCHINGBAUER

Access Specifier

Definition

```
class point
{
private:
    int _x;
};
```

Usage

```
int x = A._x;
```

- **Compiler error**: "Access to private member ..."
- **Access Specifier**: specifies, who can call a method or access a member
  - `public`: access allowed from everywhere
  - `private`: access only from withni methods of the same class
  - `protected`: access only from within methods of same or derived class (→ Inheritance)

`class point`, analyzed (2)

FASCHINGBAUER

Access Specifier and **Access Methods**

Definition

```
class point
{
public:
    int x() const { return _x; }
};
```

Usage

```
int x = A.x();
```

- Public Access $\Longrightarrow$ compiler does not complain
- Access Specifier: matter of taste ("Design")
  - Public Member Access: eerybody could modify everything → C
  - Access Methods: read-only member access → *inline*
- `const`: `x()` does not modify the object → excellent type system

`class point`, analyzed (3)

Constuctors

Declaration

```
class point
{
public:
    point(int x, int y);
};
```

Usage

```
point A(1,2);
```

- **Constuctor**: initializes the object
- Here: initialization of the members `x` and `y`
- Multiple constuctors possible

class point, analyzed (4)

FASCHINGBAUER

Operators

Declaration

```
class point
{
public:
    point& operator+=(point addend);
};
```

Usage

```
A += B;
C = A += B;
```

- **Operator Overloading**
- `A += B` has the value of `A` *after assignment*

`class point`, analyzed (5)

FASCHINGBAUER

Operators

Declaration

```
class point
{
    // ...
};
point operator+(point lhs, point rhs);
```

Usage

```
C = A + B;
```

- Operator "+=" modifies an objekt (left hand side) $\Longrightarrow$ member
- Operator "+" creates a new object $\Longrightarrow$ global

Terminology

FASCHINGBAUER

One says:

```
class point
{
    // ...
};
```

- `point` is a type ...
- ... a *Class* of objects

```
point A(1,2), B(3,4);
```

- `A` and `B` are *instanzen* of *Class* `point`
- *Instance* $\Longleftrightarrow$ *Object*

Overview

1. Introduction
   - Introduction
2. Data Encapsulation
   - Classes and Objects
   - Constructors and Destructors
   - Object Copy
3. Functions and Methods
   - Overloading
   - Methods
   - `const`
   - `this`
   - References
   - `static`
   - Operatoren
4. Exceptions
   - Basics
   - `try` — `catch`
5. Templates
   - Function Templates
   - Class Templates
6. Standard Template Library
   - Basics
   - Sorting
   - Searching
   - Sequential Containers
   - Associative Containers
7. Dynamic Memory Allocation
8. Inheritance
9. Object Oriented Design

Constructors: why? (1)

FASCHINGBAUER

Initialization in C

- left to the programmer
- → sheer number of bugs!

```
struct point A;
```

- A remains uninitialized → "random" values

```
struct point A = {1,2};
```

- A initialized with `x = 1` and `y = 2`

```
struct point A;
...
A.x = 1;
A.y = 2;
```

- "Initialization" at same later point

Constructors: why? (1)

FASCHINGBAUER

Initialization in C++

- Programmierer has no choice
- Whenever you think about a `point` object, you *have* to think about its value
- → Initialization error excluded from the beginning

```
point A;
```

- Compiler errors: "`void constructor for point` not defined"

```
point A(1,2);
```

- Only possibility to create a `point`

Construktors: Implementation — *Inline*

FASCHINGBAUER

“Short” methods are best defined in the class definition itself → *inline*

point.h: “Inline” definition

```
class point
{
public:
    point(int x, int y)
    {
        _x = x;
        _y = y;
    }
```

Constructors: Implementation — *Out-of-Line*

FASCHINGBAUER

"Long" methods are best defined in the implementation file

point.h: declaration

```
class point
{
public:
    point(int x, int y);
```

point.cc: definition

```
point::point(int x, int y)
{
    _x = x;
    _y = y;
}
```

Constructors: *Initializer List* (1)

FASCHINGBAUER

What about `const` members?

```
class point
{
public:
    point(int x, int y)
    {
        _x = x;
        _y = y;
    }
private:
    const int _x;
    const int _y;
};
```

- Compiler error
  - "`const` members x und y not initialized"
  - "Assignment to const member"
- Constructor body is *normal* Code
- `const` pollution?
- → *No!*

Constructors: *Initializer List* (2)

FASCHINGBAUER

Initializer List: different form of assignment — *Initialization*

```
class point
{
public:
    point(int x, int y) : _x(x), _y(y) {}
private:
    const int _x;
    const int _y;
};
```

Default Constructor (1)

Constructor without parameter — *Default Constructor*

```
class point
{
public:
    point() : _x(0), _y(0) {}
    point(int x, int y) : _x(x), _y(y) {}
};

...

point p; // -> (0, 0)
```

Default Constructor (2)

```
class rectangle
{
    point nw;
    point se;
};
```

- Compiler *generates* default constructor
- ... but only when none is defined explicitly

- Always ask whether a default constructor makes sense
- Here: rectangle ((0,0),(0,0)) $\rightarrow$ nonsense
- If one wants a real ctor *and* a default ctor $\rightarrow$ define one explicitly

Object Lifecycle — Destructor

FASCHINGBAUER

Like in C. Well almost. The end of an object is ...

- Scope: end of block
- `return` from function → End for *local* objects
- Explicit lifetime (dynamic memory): `delete`
- Static (global) lifetime: program termination

In any case: as soon as life is over → *Destructor*

- Implicitly defined (compiler generated)
- → memberwise destruction
- Explicitly defined

Destructors (1)

FASCHINGBAUER

What happens when life is over?

```
class String
{
public:
    String(const char *from)
      : _c_str(new char[strlen(from)+1])
    {
        strcpy(_c_str, from);
    }
private:
    char *_c_str;
};
```

Destructors (2)

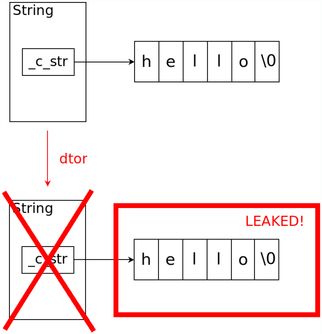
FASCHINGBAUER

Implementation detail of `String`:

- *Heap-allocated* memory
- `String` is only as big as all of its members
- → `sizeof(char *)` (4 or 8 bytes)
- *Data are on the heap*
- → variable length

Destructors (3)

```
void f()
{
    String s("hello");
    ...
    // LEAKED 6 bytes!
}
```



Destructors (4)

FASCHINGBAUER

Solution: program a destructor

```
class String
{
public:
    ~String()
    {
        delete[] _c_str;
    }
};
```

- Not only with dynamically allocated memory
- ... but with all kinds of explicit resource allocation (e.g. file descriptors)
- More details for `new` and `delete` → later

Overview

FASCHINGBAUER

1 Introduction
- Introduction

2 Data Encapsulation
- Classes and Objects
- Constructors and Destructors
- Object Copy

3 Functions and Methods
- Overloading
- Methods
- `const`
- `this`
- References
- `static`
- Operatoren

4 Exceptions
- Basics
- `try` — `catch`

5 Templates
- Function Templates
- Class Templates

6 Standard Template Library
- Basics
- Sorting
- Searching
- Sequential Containers
- Associative Containers

7 Dynamic Memory Allocation

8 Inheritance

9 Object Oriented Design

Copy in C

FASCHINGBAUER

Copy of "Objects" in C: `struct`

```
struct point
{
    int x;
    int y;
};
```

```
struct point p1 = {2,7};
struct point p2;

p2 = p1;
```

- `struct point` *memberwise* copy
- Simple: transfer of memory image

Copy Constructor

FASCHINGBAUER

Copying objects in C++: similar to C++

```
class point
{
    // ...
};
...
point p1;
point p2(p1);
```

- Compiler *generates copy constructor*
- → member by member
- → simple data types just as in C

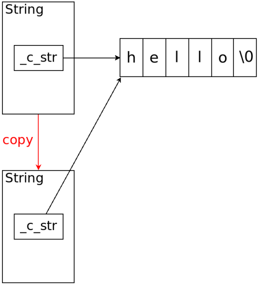
But ...

Copy Constructor and Pointer Members (1)

Caution, Trap: pointer members

```
class String
{
public:
    String(const char *c_str);
private:
    char *_c_str;
};
```

```
String s1("hello");
String s2 = s1; // ctor!
```

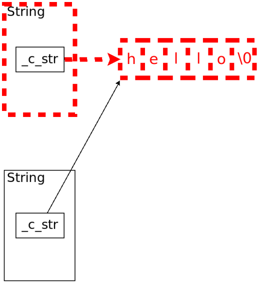


Copy Constructor and Pointer Members (2)

FASCHINGBAUER

Segmentation Fault in the best of all cases ...

- Pointer member is to compiler simply *a pointer*
- Pointers are copied
- But not what they point to
- *How should the compiler know!*



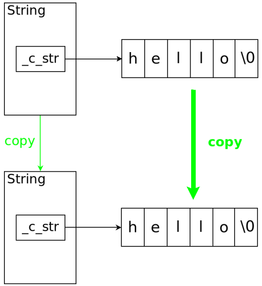
Copy Constructor and Pointer Members (3)

FASCHINGBAUER

Solution: explicit copy constructor

Copy the pointed-to memory!

```
String::String(const String& s)
{
    _c_str = new char[
          strlen(s._c_str)+1];
    strcpy(_c_str, s._c_str);
}
```



Copy Constructor: Rekursivs/Memberwise

FASCHINGBAUER

```
struct TwoStrings
{
    String s1;
    String s2;
};
struct TwoTwoStrings
{
    TwoStrings s21;
    TwoStrings s22;
};
```

- `String` has copy constructor (correct because handwritten)
- $\Longrightarrow$ `TwoStrings` is correct
- $\Longrightarrow$ `TwoTwoStrings` is correct
- $\Longrightarrow$ ...

Assignment Operator

FASCHINGBAUER

Second way of copying objects: overwrite an existing object

```
class point
{
    // ...
};
```

```
point p1, p2;
// ...
p2 = p1; // assignment!
```

- Like *Copy Constructor* generated by compiler
- → Member by member
- → simple data types just as in C

But ... as with the copy constructor → pointer members!

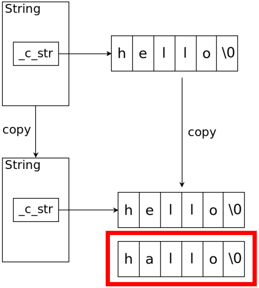
- Assignment operator is best self defined

Assignment Operator and Pointer Members (1)

FASCHINGBAUER

Caution, naively buggy!

```
String& String::operator=(
    const String& s)
{
    _c_str = new char[
        strlen(s._c_str)+1];
    strcpy(_c_str, s._c_str);
    return *this;
}
```




```
String s1("hello");
String s2("hallo");
s2 = s1; // LEAK!
```

Assignment Operator and Pointer Members (2)

FASCHINGBAUER

Straightforward fix — caution, still naively buggy!

```
String& String::operator=(
    const String& s)
{
    delete[] _c_str;
    _c_str = new char[
        strlen(s._c_str)+1];
    strcpy(_c_str, s._c_str);
    return *this;
}
```

```
String s("hello");
s = s; // SEGFAULT!
```

- *"Self Assignment"*
- Rare but true!
- User expects that this is not an error

Correct nonsense

```
int i = 42;
i = i;
```

Assignment Operator: *Self Assignment*

FASCHINGBAUER

Ultimate Fix: *Self Assignment Check*

```
String& String::operator=(
    const String& s)
{
    if (this != &s) {
        delete[] _c_str;
        _c_str = new char[
            strlen(s._c_str)+1];
        strcpy(_c_str, s._c_str);
    }
    return *this;
}
```

Overview

1. Introduction
   - Introduction
2. Data Encapsulation
   - Classes and Objects
   - Constructors and Destructors
   - Object Copy
3. Functions and Methods
   - Overloading
   - Methods
   - const
   - this
   - References
   - static
   - Operatoren
4. Exceptions
   - Basics
   - try — catch
5. Templates
   - Function Templates
   - Class Templates
6. Standard Template Library
   - Basics
   - Sorting
   - Searching
   - Sequential Containers
   - Associative Containers
7. Dynamic Memory Allocation
8. Inheritance
9. Object Oriented Design

Overview

1. Introduction
   - Introduction
2. Data Encapsulation
   - Classes and Objects
   - Constructors and Destructors
   - Object Copy
3. Functions and Methods
   - Overloading
   - Methods
   - `const`
   - `this`
   - References
   - `static`
   - Operatoren
4. Exceptions
   - Basics
   - `try` — `catch`
5. Templates
   - Function Templates
   - Class Templates
6. Standard Template Library
   - Basics
   - Sorting
   - Searching
   - Sequential Containers
   - Associative Containers
7. Dynamic Memory Allocation
8. Inheritance
9. Object Oriented Design

Functions in C

FASCHINGBAUER

In C everything is simple

Declaration of x

```
int x(int i);
```

Ok

```
int ret = x(42);
```

2x Error

```
char *ret = x("huh?");
```

Error: x declared twice

```
char *x(char* str);
```

Functions in C++ — Overloading

FASCHINGBAUER

Two declarations of `x`

```
int x(int i);
char *x(const char *str);
```

Ok

```
int ret = x(42);
```

Ok

```
char *ret = x("huh?");
```

Error: no appropriate `x` found

```
char *ret = x(42);
```

Overview

FASCHINGBAUER

1. Introduction
   - Introduction
2. Data Encapsulation
   - Classes and Objects
   - Constructors and Destructors
   - Object Copy
3. Functions and Methods
   - Overloading
   - Methods
   - const
   - this
   - References
   - static
   - Operatoren
4. Exceptions
   - Basics
   - try — catch
5. Templates
   - Function Templates
   - Class Templates
6. Standard Template Library
   - Basics
   - Sorting
   - Searching
   - Sequential Containers
   - Associative Containers
7. Dynamic Memory Allocation
8. Inheritance
9. Object Oriented Design

Objects — Data and Methods

FASCHINGBAUER

C

- Object $\Longleftrightarrow$ `struct`
- Operations on "Objects": free functions
- $\rightarrow$ can be defined anywhere

C++

- Classes: data and "methods"
- Methods: functions *bound* to objects

Method — Example point (1)

FASCHINGBAUER

- What is a point? → x and y
- What is the responsibility of a point?
  - move itself
  - compute its distance to origin
  - … or from another point …

```
class point
{
public:
    void move(int x, int y);
    float distance_origin() const;
    float distance(const point&) const;
};
```

Method — Example `point` (2)

- `point` offers functionality
- `point` should be used *as simply and clearly as possible!*

```
point p(2, 0);
p.move(1, 0);
if (fabs(p.distance_origin() - 3.0) > 0.0001)
    std::cerr << "FPU bogus?" << std::endl;
```

Methods and Design

FASCHINGBAUER

Question: what should a point be able to? Difficult to answer …

- Should it offer its coordinates?
  - I think so → small `inline` access methods
- Should it offer two dimensional arithmetic methods?
  - Why not? This is what a point is there for.
- Should it be able to print a plot of itself?
  - Why not? As long as users of a point are willing to link 28 more libraries.
  - → Coupling

Methods: Wrap-Up

FASCHINGBAUER

Many but simple (?) Nuances ...

- `const`: type system
- References: performance
- `static`, with yet another meaning of the keyword

Overview

1 Introduction
- Introduction

2 Data Encapsulation
- Classes and Objects
- Constructors and Destructors
- Object Copy

3 Functions and Methods
- Overloading
- Methods
- `const`
- `this`
- References
- `static`
- Operatoren

4 Exceptions
- Basics
- `try` — `catch`

5 Templates
- Function Templates
- Class Templates

6 Standard Template Library
- Basics
- Sorting
- Searching
- Sequential Containers
- Associative Containers

7 Dynamic Memory Allocation

8 Inheritance

9 Object Oriented Design

const: Immutable Variable

FASCHINGBAUER

Already possible in C: immutable variables

```
const point *p;
p->x = 7; /*ERROR!*/
```

```
void f(const struct point *p)
{
    p->x = 7; /*ERROR!*/
}
```

- Variables → Modification impossible
- Parameter → Modification impossible

const: Methods (1)

FASCHINGBAUER

- const methods *promise to the compiler* not to modify the object
- No promise → compiler *has to assume* that the method modifies the object

```
class point
{
public:
    int x() { return _x; }
    int y() { return _y; }
private:
    int _x;
    int _y;
};
```

```
void f(const point *p)
{
    // ERROR!
    cout << p->x();
}
```

const: Methods (2)

FASCHINGBAUER

```
class point
{
public:
    int x() const { return _x; }
    int y() const { return _y; }
private:
    int _x;
    int _y;
};
```

- "const pollution" $\Longleftrightarrow$ "being correct is very cumbersome"
- "const correctness": best possible state
- Nice goodie offered by the language

Overview

1. Introduction
   - Introduction
2. Data Encapsulation
   - Classes and Objects
   - Constructors and Destructors
   - Object Copy
3. Functions and Methods
   - Overloading
   - Methods
   - const
   - this
   - References
   - static
   - Operatoren
4. Exceptions
   - Basics
   - try — catch
5. Templates
   - Function Templates
   - Class Templates
6. Standard Template Library
   - Basics
   - Sorting
   - Searching
   - Sequential Containers
   - Associative Containers
7. Dynamic Memory Allocation
8. Inheritance
9. Object Oriented Design

Hidden Pointer: this (1)

FASCHINGBAUER

```
class point
{
public:
    void move(int x, int y)
    {
        _x += x;
        _y += y;
    }
};
```

- Where's the object?
- What's the object?
- Where's the *member* x?

Hidden Pointer: this (2)

FASCHINGBAUER

Explanation: how would this be done in C?

C++

```
point p(5, 6);
p.move(2, 3);
```

- First parameter of each method: this
- Method name is: move in class point

C

```
struct point p = {5, 6};
point_move(&p /*this*/, 2, 3)
```

C++: writing this explicitly

```
void move(int x, int y)
{
    this->_x += x;
    this->_y += y;
}
```

Overview

1. Introduction
   - Introduction
2. Data Encapsulation
   - Classes and Objects
   - Constructors and Destructors
   - Object Copy
3. Functions and Methods
   - Overloading
   - Methods
   - const
   - this
   - References
   - static
   - Operatoren
4. Exceptions
   - Basics
   - try — catch
5. Templates
   - Function Templates
   - Class Templates
6. Standard Template Library
   - Basics
   - Sorting
   - Searching
   - Sequential Containers
   - Associative Containers
7. Dynamic Memory Allocation
8. Inheritance
9. Object Oriented Design

Pointers, Seen Differently: Referenzen (1)

FASCHINGBAUER

Problem: parameter passing (of large objects)

```
class point
{
public:
    float distance(point p) const
    {
        int dx = abs(_x-p._x);
        int dy = abs(_y-p._y);
        return sqrt(dx*dx+dy*dy);
    }
};
```

- **Problem**
  - Parameter is a *copy*
- **Solution**
  - Pass by pointer
  - Even better: `const pointer`

Pointers, Seen Differently: Referenzen (2)

FASCHINGBAUER

```
class point
{
public:
    float distance(const point *p) const
    {
        int dx = abs(_x-p->_x);
        int dy = abs(_y-p->_y);
        return sqrt(dx*dx+dy*dy);
    }
};
```

- **Problem**
  - User has to take the address
  - `p1.distance(&p2`
  - Pointers can be `NULL`
- **Solution**
  - References

Pointers, Seen Differently: Referenzen (3)

FASCHINGBAUER

```
class point
{
public:
    float distance(const point &p) const
    {
        int dx = abs(_x-p._x);
        int dy = abs(_y-p._y);
        return sqrt(dx*dx+dy*dy);
    }
};
```

Overview

FASCHINGBAUER

1. Introduction
   - Introduction
2. Data Encapsulation
   - Classes and Objects
   - Constructors and Destructors
   - Object Copy
3. Functions and Methods
   - Overloading
   - Methods
   - const
   - this
   - References
   - static
   - Operatoren
4. Exceptions
   - Basics
   - try — catch
5. Templates
   - Function Templates
   - Class Templates
6. Standard Template Library
   - Basics
   - Sorting
   - Searching
   - Sequential Containers
   - Associative Containers
7. Dynamic Memory Allocation
8. Inheritance
9. Object Oriented Design

Methods without Object — `static` (1)

FASCHINGBAUER

What we know now:

- Methods are great
- Name and variable → Method (`p.move(1,2)`)
- → clear writing

But: *global* functions? Methods without an object?

- Not bound to objects
- Same scheme ("method of the class")?

In C ...

```
point point_add(const point &l, const point &l);
```

Methods without Object — static (2)

FASCHINGBAUER

Declaration/definition

```
class point
{
public:
    static point add(const point &l, const point &r)
    {
        return point(l.x()+r.x(), l.y()+r.y());
    }
};
```

Usage

```
point p1, p2, p3;
...
p3 = point::add(p1, p2);
```

Overview

1. Introduction
   - Introduction
2. Data Encapsulation
   - Classes and Objects
   - Constructors and Destructors
   - Object Copy
3. Functions and Methods
   - Overloading
   - Methods
   - `const`
   - `this`
   - References
   - `static`
   - Operatoren
4. Exceptions
   - Basics
   - `try` — `catch`
5. Templates
   - Function Templates
   - Class Templates
6. Standard Template Library
   - Basics
   - Sorting
   - Searching
   - Sequential Containers
   - Associative Containers
7. Dynamic Memory Allocation
8. Inheritance
9. Object Oriented Design

Motivation

FASCHINGBAUER

Operators (`+`, `+=`, `->`) etc. in C

- Only available for simple data types (`int`, `float`, pointers, ...)
- → *defined by the language*

Problem: we want more ...

- Arithmetic operators for `class point`?
- Intelligent pointers which have a different definition of `->`?
- ... unbounded fantasy here ...

Operators, Functions, and Methods

FASCHINGBAUER

Why shouldn't this be possible? Operators, after all, are functions that are implemented by the compiler.

- `i += 42`. Method "+=" on object of type `int`, with parameter `int`
- `i = j + 42`. *Static* method "+". Two parameters (type `int`), return type `int`
- `p += point(1,2)`. Define as you like!
- `str += "hallo!"`. Someone else did this already ...
  - `std::string`
  - *C++ Standard Library*

Example: Operator += *on the Objekt* (1)

FASCHINGBAUER

Without further ado ...

```
class point
{
public:
    point& operator+=(const point &addend)
    {
        _x += addend._x;
        _y += addend._y;
        return *this;
    }
private:
    int _x;
    int _y;
};
```

Example: Operator += *on the Objekt* (2)

FASCHINGBAUER

```
operator+=(const point &addend)
```

- `this`: left hand side of `p1 += p2`
- `addend`: right hand side of `p1 += p2`

```
point& operator+=(...)
```

```
p3 = p2 += p1;
```

```
return *this;
```

- Value of the expression `p1 += p2` is `p1`
- → use `p1` onwards

Example: Operator + *nicht* on the Object (1)

FASCHINGBAUER

Without further ado ...

```
class point
{
public:
    int x() const { return _x; }
    int y() const { return _y; }
};

point operator+(const point &l, const point &r)
{
    return point(l.x()+r.x(), l.y()+r.y());
}
```

Example: Operator + *not* on the Object (2)

FASCHINGBAUER

```
operator+(const point &l,
          const point &r)
```

- No object → no `this`
- Two real parameters

```
point operator+(...)
```

- "+" creates *new* object
- → return by *copy*

```
l.x()+r.x() ...
```

- Global function → `private` not visible
- `friend` — not a solution

Example: Function Objects — *Functors* (1)

FASCHINGBAUER

Function Call Operator "()": for example ...

- Class without comparison operator

```
class Item
{
public:
    Item(int dies, int das)
    : _dies(dies), _das(das) {}

    int dies() const { return _dies; }
    int das() const { return _das; }

private:
    int _dies, _das;
};
```

Example: Function Objects — *Functors* (2)

Problem: one wants to sort → comparison operator needed

```
bool operator<(const Item &lhs, const Item &rhs)
{
    if (lhs.dies() < rhs.dies())
        return true;
    if (lhs.dies() > rhs.das())
        return false;
    return lhs.das() < rhs.das();
}
```

Problem: he's *global*

- → Ambiguity!
- Not everybody agrees

Example: Function Objects — *Functors* (3)

FASCHINGBAUER

Solution:

- Functors that everybody can write
- *Function Call Operator*

```
class LessOp
{
public:
    bool operator()(const Item &lhs, const Item &rhs) const
    {
        // same as operator<(lhs, rhs)
    }
};
```

Example: Function Objects — *Functors* (4)

FASCHINGBAUER

Usage ...

```
LessOp less;
if (less(item1, item2))
    ...
```

- Container classes
- Algorithms

Overview

1. Introduction
   - Introduction
2. Data Encapsulation
   - Classes and Objects
   - Constructors and Destructors
   - Object Copy
3. Functions and Methods
   - Overloading
   - Methods
   - `const`
   - `this`
   - References
   - `static`
   - Operatoren
4. Exceptions
   - Basics
   - `try` — `catch`
5. Templates
   - Function Templates
   - Class Templates
6. Standard Template Library
   - Basics
   - Sorting
   - Searching
   - Sequential Containers
   - Associative Containers
7. Dynamic Memory Allocation
8. Inheritance
9. Object Oriented Design

Overview

FASCHINGBAUER

1. Introduction
   - Introduction
2. Data Encapsulation
   - Classes and Objects
   - Constructors and Destructors
   - Object Copy
3. Functions and Methods
   - Overloading
   - Methods
   - const
   - this
   - References
   - static
   - Operatoren
4. Exceptions
   - Basics
   - try — catch
5. Templates
   - Function Templates
   - Class Templates
6. Standard Template Library
   - Basics
   - Sorting
   - Searching
   - Sequential Containers
   - Associative Containers
7. Dynamic Memory Allocation
8. Inheritance
9. Object Oriented Design

Error Handling: `if — else if — else`

FASCHINGBAUER

Tradional Errorhandling

```
if (dothat())
    if (dothis())
        if (dothose())
            finally();
        else
            dammit();
    else
        dammit();
else
    dammit();
```

My Wish ...

```
dothat();
dothis();
dothose();
finally();
// only if anything happens:
dammit();
```

Overview

1 Introduction
- Introduction

2 Data Encapsulation
- Classes and Objects
- Constructors and Destructors
- Object Copy

3 Functions and Methods
- Overloading
- Methods
- const
- this
- References
- static
- Operatoren

4 Exceptions
- Basics
- try — catch

5 Templates
- Function Templates
- Class Templates

6 Standard Template Library
- Basics
- Sorting
- Searching
- Sequential Containers
- Associative Containers

7 Dynamic Memory Allocation

8 Inheritance

9 Object Oriented Design

try - Block

FASCHINGBAUER

Try do do something:

```
try {
    dothat();
    dothis();
    dothose();
    finally();
}
...
```

- Linear execution
- Error handling not after every step
- ... but rather in a separate block

catch - Block

```
try {
    ...
}
catch (const ThisException &e) {
    std::cerr << e.what() << std::endl;
    // ... react ...
}
catch (const ThatException &e) {
    std::cerr << e.what() << std::endl;
    // ... react ...
}
catch (const std::exception &e) {
    std::cerr << e.what() << std::endl;
    // ... give up ...
}
```

Exceptions

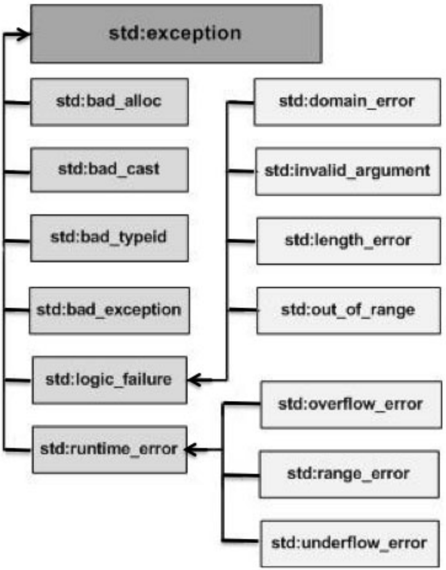
No restrictions: everything can be thrown and caught

```
try {
    ...
}
catch (int i) {
    ...
}
```

- → One has to think if it makes sense!
- Some structure is recommended

Standard Library: Exception-Hierarchy

FASCHINGBAUER



Custom Exceptions (1)

FASCHINGBAUER

Recommendation:

- Don't throw numbers ...
- Don't throw strings ...
- ... fit yourself into the exception hierarchy
- → minimal inheritance

```
namespace std {
  class exception
  {
  public:
    virtual const char* what() const throw() = 0;
  };
}
```

Custom Exceptions (2)

FASCHINGBAUER

```
class MyException : public std::exception
{
public:
    virtual const char* what() const throw()
    {
        return "dammit!";
    }
};
```

- Here: `void` constructor
- Can be arbitrary
- … as far as interface is ok

Throwing Exceptions — throw

FASCHINGBAUER

```
void dothis()
{
    // ...
    if (error_detected)
        throw MyException();
    // ...
}
```

Last Words

- `return` if ok, `throw` if error
- → alternative return path
- Destructors of local objects are called
- Important design decision
  - How many custom exception do I define?
  - → Error handling at which granularity?

Overview

1. Introduction
   - Introduction
2. Data Encapsulation
   - Classes and Objects
   - Constructors and Destructors
   - Object Copy
3. Functions and Methods
   - Overloading
   - Methods
   - `const`
   - `this`
   - References
   - `static`
   - Operatoren
4. Exceptions
   - Basics
   - `try` — `catch`
5. Templates
   - Function Templates
   - Class Templates
6. Standard Template Library
   - Basics
   - Sorting
   - Searching
   - Sequential Containers
   - Associative Containers
7. Dynamic Memory Allocation
8. Inheritance
9. Object Oriented Design

Overview

FASCHINGBAUER

1. Introduction
   - Introduction
2. Data Encapsulation
   - Classes and Objects
   - Constructors and Destructors
   - Object Copy
3. Functions and Methods
   - Overloading
   - Methods
   - `const`
   - `this`
   - References
   - `static`
   - Operatoren
4. Exceptions
   - Basics
   - `try` — `catch`
5. Templates
   - Function Templates
   - Class Templates
6. Standard Template Library
   - Basics
   - Sorting
   - Searching
   - Sequential Containers
   - Associative Containers
7. Dynamic Memory Allocation
8. Inheritance
9. Object Oriented Design

Origin: Duplicated Code

FASCHINGBAUER

Overloading: function `max` with different implementations ...

```
int max(int a, int b)
{
    return (a<b)?b:a;
}
float max(float a, float b)
{
    return (a<b)?b:a;
}
```

→ Duplicated Code

A simple Function-Template

FASCHINGBAUER

Solution: "code generator" → Templates

```
template <typename T> T max(T a, T b)
{
    return (a<b)?b:a;
}
```

- Generation recipe
- T ... "Template Parameter"
- Requirement: `operator<()` must be valid

More Templates from the STL

FASCHINGBAUER

Better: look into the STL. For example ...

```
#include <algorithm>

float f = max(1.2, 1.3);
int i = max(1, 2);
std::string s = max("abc", "abd");
```

Overview

1. Introduction
   - Introduction
2. Data Encapsulation
   - Classes and Objects
   - Constructors and Destructors
   - Object Copy
3. Functions and Methods
   - Overloading
   - Methods
   - const
   - this
   - References
   - static
   - Operatoren
4. Exceptions
   - Basics
   - try — catch
5. Templates
   - Function Templates
   - Class Templates
6. Standard Template Library
   - Basics
   - Sorting
   - Searching
   - Sequential Containers
   - Associative Containers
7. Dynamic Memory Allocation
8. Inheritance
9. Object Oriented Design

Does This Work With Classes?

FASCHINGBAUER

```
class point
{
public:
    point(int x, int y);
    int x() const;
    int y() const;
private:
    int _x;
    int _y;
};
```

What about other data types?

- point with int
- point with float
- ...

Example: point as a Class Template (1)

```
template <typename T>
class point
{
public:
    point(T x, T y) : _x(x), _y(y) {}
    T x() const;
    T y() const;
private:
    T _x;
    T _y;
};
```

Example: point as a Class Template (2)

FASCHINGBAUER

```
template <typename T>  // method template parameter
T                      // method return type
point<T>::x() const    // class template parameter
{
    return _x;
}
```

Pooh ...

Last Words

FASCHINGBAUER

- Template classes *must* be defined in the header file
- *Compiler* instantiates code
- *Linker* recognizes duplicates → unifies
- Rules are very complicated
- → "Language in a language"
- Compiler error message often very confusing

Overview

FASCHINGBAUER

1 Introduction
- Introduction

2 Data Encapsulation
- Classes and Objects
- Constructors and Destructors
- Object Copy

3 Functions and Methods
- Overloading
- Methods
- `const`
- `this`
- References
- `static`
- Operatoren

4 Exceptions
- Basics
- `try` — `catch`

5 Templates
- Function Templates
- Class Templates

6 Standard Template Library
- Basics
- Sorting
- Searching
- Sequential Containers
- Associative Containers

7 Dynamic Memory Allocation

8 Inheritance

9 Object Oriented Design

Overview

1 Introduction
- Introduction

2 Data Encapsulation
- Classes and Objects
- Constructors and Destructors
- Object Copy

3 Functions and Methods
- Overloading
- Methods
- const
- this
- References
- static
- Operatoren

4 Exceptions
- Basics
- try — catch

5 Templates
- Function Templates
- Class Templates

6 Standard Template Library
- Basics
- Sorting
- Searching
- Sequential Containers
- Associative Containers

7 Dynamic Memory Allocation

8 Inheritance

9 Object Oriented Design

Containers, Iterators, Algorithms

Genius Combination of ...

- Operator overloading (->, *, +, +=, ++)
- Abstract containers
- Abstract "Algorithms"
- ... based upon *pointer arithmetic*!

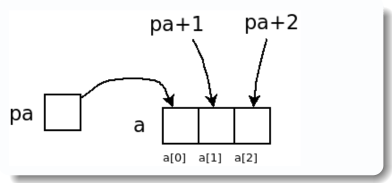
→ *Pointer arithmetic*, revisited ...

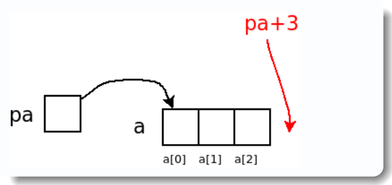
Pointer Arithmetic (1)

FASCHINGBAUER

Pointer and arrary index

- *Pointer + Integer = Pointer*
- Exactly the same as subscript ("index") operator
- *No range check*
- → Error prone
- But: performance!



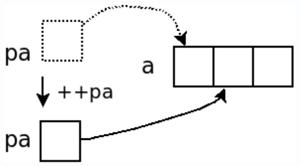


Pointer Arithmetic (2)

FASCHINGBAUER

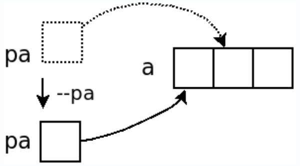
Pointer Increment

```
int *pa = a;
++pa;
```



Pointer Decrement

```
int *pa = &a[1];
--pa;
```

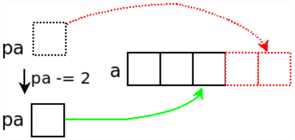


Pointer Arithmetic (3)

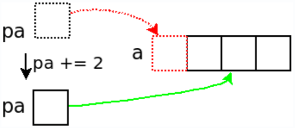
FASCHINGBAUER

Pointer don't necessarily point to valid memory locations ...

```
*pa = a + 4;
pa -= 2;
i = *pa; /* ok */
```




```
*pa = a - 1;
pa += 2;
i = *pa; /* ok */
```

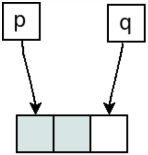


Pointer Arithmetic: Difference

FASCHINGBAUER

How many array elements are there between two pointers?

```
p = &a[0];
q = &a[2];
num = q - p; /* 2 */
```



General practice ("The Spirit of C"):

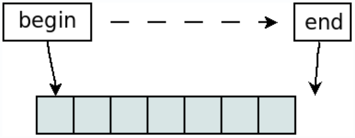
- *Beginning* of an array (a *set* of elements) is a *pointer to the first element*
- *End* is *pointer past the last element*

Pointer Arithmetic: Array Algorithms

FASCHINGBAUER

Iteration over all elements of an array ...

```
int sum(const int *begin, const int *end)
{
    int sum = 0;

    while (begin < end)
        sum += *begin++; /* precedence? what? */
    return sum;
}
```



Pretty, isn't it?

Pointer Arithmetic: Step Width? (1)

FASCHINGBAUER

So far: pointer to int `int` — how about different datatypes?
→ same!

- *pointer* + *n*: points to the *n*-th array element from *pointer*
- Type system knows about sizes
- Pointer knows the type of the data it points to
- Careful with `void` and `void*`

Pointer Arithmetic: Step Width? (2)

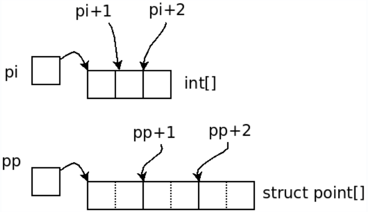
```
struct point
{
    int x, y;
};

struct point points[3], *begin, *end;

begin = points;
end = points + sizeof(points)/sizeof(struct point);

while (begin < end) {
    ...
    ++begin;
}
```

Pointer Arithmetic: Arbitrary Data Types?

FASCHINGBAUER

- *sizeof*: size (in bytes) of a type or variable

```
sizeof(int)
sizeof(struct point)
sizeof(i)
sizeof(pi)
sizeof(pp)
```



Container

FASCHINGBAUER

Container

- Extremely practical collection of template classes
- Sequential container $\rightarrow$ array, list
- Associative containers

Dynamically growing array: std::vector

```
#include <vector>

std::vector<int> int_array;
int_array.push_back(42);
int_array.push_back(7);
int_array.push_back(666);

for (int i=0; i<int_array.size(); ++i)
    std::cout << int_array[i] << ' ';
```

Pointer Arithmetic

```
std::vector<int>::const_iterator begin = int_array.begin();
std::vector<int>::const_iterator end = int_array.end();
while (begin < end) {
    std::cout << *begin << ' ';
    ++begin;
}
```

Algorithms: std::copy (1)

Copy array by hand

```
std::vector<int> int_array;
int_array.push_back(42);
int_array.push_back(7);
int_array.push_back(666);

int int_array_c[3];
std::vector<int>::const_iterator src_begin = int_array.begin();
std::vector<int>::const_iterator src_end = int_array.end();
int *dst_begin = int_array_c;

while (src_begin < src_end)
    *dst_begin++ = *src_begin++;
```

Algorithms: std::copy (2)

FASCHINGBAUER

Copy using STL

```
#include <algorithm>

std::vector<int> int_array;
// ...
int int_array_c[3];

std::copy(int_array.begin(), int_array.end(), int_array_c);
```

Adapting Iterators: `std::ostream_iterator`

Copy: array to `std::ostream`, which looks like another array

```
#include <iterator>

int int_array_c[] = { 34, 45, 1, 3, 2, 666 };
std::copy(int_array_c, int_array_c+6,
          std::ostream_iterator<int>(std::cout, " "));

std::vector<int> int_array;
// ...
std::copy(int_array.begin(), int_array.end(),
          std::ostream_iterator<int>(std::cout, " "));
```

Adapting Iterators: std::back_insert_iterator

Problem

- std::copy() requires *existing/allocated memory* → *performance!*
- → copying onto empty std::vector impossible

Segmentation Fault

```
int int_array_c[] = { 34, 45, 1, 3, 2, 666 };
std::vector<int> int_array; // empty!

std::copy(int_array_c, int_array_c+6, int_array.begin());
```

Adapting Iterators: std::back_insert_iterator

Solution: std::back_insert_iterator

```
int int_array_c[] = { 34, 45, 1, 3, 2, 666 };
std::vector<int> int_array;

std::copy(
    int_array_c, int_array_c+6,
    std::back_insert_iterator<std::vector<int> >(int_array));
```

Overview

1. Introduction
   - Introduction
2. Data Encapsulation
   - Classes and Objects
   - Constructors and Destructors
   - Object Copy
3. Functions and Methods
   - Overloading
   - Methods
   - `const`
   - `this`
   - References
   - `static`
   - Operatoren
4. Exceptions
   - Basics
   - `try` — `catch`
5. Templates
   - Function Templates
   - Class Templates
6. Standard Template Library
   - Basics
   - Sorting
   - Searching
   - Sequential Containers
   - Associative Containers
7. Dynamic Memory Allocation
8. Inheritance
9. Object Oriented Design

Algorithms: std::sort

Now for something simple ...

C

```
int int_array[] = { 34, 45, 1, 3, 2, 666 };
std::sort(int_array, int_array + 6);
```

C++

```
std::vector<int> int_array;
int_array.push_back(42);
int_array.push_back(7);
int_array.push_back(666);

std::sort(int_array.begin(), int_array.end());
```

Algorithms: std::sort, custom comparison

```
bool less_reverse(int l, int r)
{
    return l > r;
}

int int_array[] = { 34, 45, 1, 3, 2, 666 };
std::sort(int_array, int_array + 6, less_reverse);
```

Overview

1 Introduction
- Introduction

2 Data Encapsulation
- Classes and Objects
- Constructors and Destructors
- Object Copy

3 Functions and Methods
- Overloading
- Methods
- `const`
- `this`
- References
- `static`
- Operatoren

4 Exceptions
- Basics
- `try` — `catch`

5 Templates
- Function Templates
- Class Templates

6 Standard Template Library
- Basics
- Sorting
- Searching
- Sequential Containers
- Associative Containers

7 Dynamic Memory Allocation

8 Inheritance

9 Object Oriented Design

Algorithms: std::find

FASCHINGBAUER

Search at its simplest: linearly for *equality*

```
int int_array_c[] = { 34, 45, 1, 3, 2, 666 };

const int *found = std::find(int_array_c, int_array_c+6, 3);
if (found == int_array_c+6)
    std::cout << "not found" << std::endl;
else
    std::cout << *found << std::endl;
```

Attention: “not found” $\Longleftrightarrow$ pointer one past the last element

Algorithms: std::find — end()

Important concept: "not found" $\iff$ pointer past the last element

```
std::vector<int> int_array;
// ...

std::vector<int>::const_iterator found =
    std::find(int_array.begin(), int_array.end(), 7);
if (found == int_array.end())
    std::cout << "not found" << std::endl;
else
    std::cout << *found << std::endl;
```

More Intelligent Search: `std::binary_search`

Sorted `std::vector` → more efficient search → *binary* search

std::binary_search

```
int int_array[] = { 34, 45, 1, 3, 2, 666 };
std::sort(int_array, int_array+6);
bool found = std::binary_search(int_array, int_array+6, 3);
```

Problem

- One can only decide whether the element is contained
- Searching for data?

More Intelligent Search: `std::lower_bound`

Result: Pointer/iterator to element found *or past* → very flexible

```
std::vector<int> int_array;
int_array.push_back(7);
int_array.push_back(42);
int_array.push_back(42);
int_array.push_back(666);

std::vector<int>::const_iterator lower =
    std::lower_bound(int_array.begin(), int_array.end(), 42);
while (lower != int_array.end() && *lower == 42) {
    std::cout << *lower << std::endl;
    ++lower;
}
```

Overview

FASCHINGBAUER

1. Introduction
   - Introduction
2. Data Encapsulation
   - Classes and Objects
   - Constructors and Destructors
   - Object Copy
3. Functions and Methods
   - Overloading
   - Methods
   - `const`
   - `this`
   - References
   - `static`
   - Operatoren
4. Exceptions
   - Basics
   - `try` — `catch`
5. Templates
   - Function Templates
   - Class Templates
6. Standard Template Library
   - Basics
   - Sorting
   - Searching
   - Sequential Containers
   - Associative Containers
7. Dynamic Memory Allocation
8. Inheritance
9. Object Oriented Design

Characteristics of `std::vector<>`

`std::vector<>` is an efficient sequential container because ...

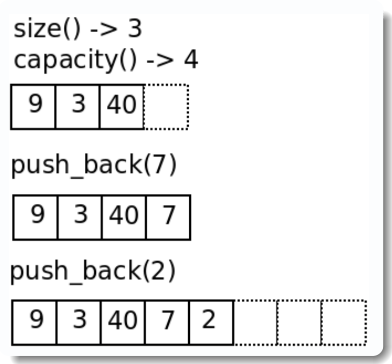
- Organization: contiguous memory → perfect utilization of processor caches
- Appending is performs liek with strings (logarithmic time)

But ...

- Removal at arbitrary position is slow
- Insertion at arbitrary position is slow
- → Unwanted copies

std::vector<>: Modification at the Back

- Appending at the back
  - There is room → immediate
  - No room → allocate (double the space), copy over, and append
- Removing from the back
  - Immediate
  - `capacity()` remains same
  - `size()` decremented by one

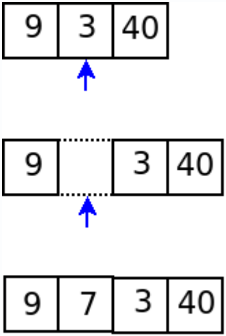


`std::vector<>`: Insertion

FASCHINGBAUER

Performance is miserable!

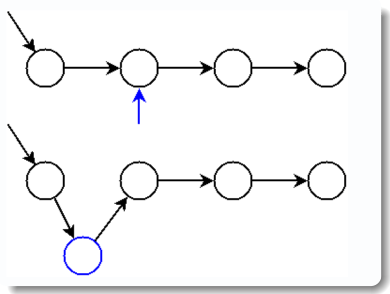
- Insert at arbitrary position
  - All elements from there on have to be copied toward the end by one position
  - Reallocation is also possible
- Removal at arbitrary position
  - All elements from there have to be copied one down



std::list<>: Insertion and Deletion

FASCHINGBAUER

- Insertion at arbitrary position
  - Pointer rearrangement → constant time
- Deletion at arbitrary position
  - Pointer rearrangement → constant time



Overview

1 Introduction
- Introduction

2 Data Encapsulation
- Classes and Objects
- Constructors and Destructors
- Object Copy

3 Functions and Methods
- Overloading
- Methods
- `const`
- `this`
- References
- `static`
- Operatoren

4 Exceptions
- Basics
- `try` — `catch`

5 Templates
- Function Templates
- Class Templates

6 Standard Template Library
- Basics
- Sorting
- Searching
- Sequential Containers
- Associative Containers

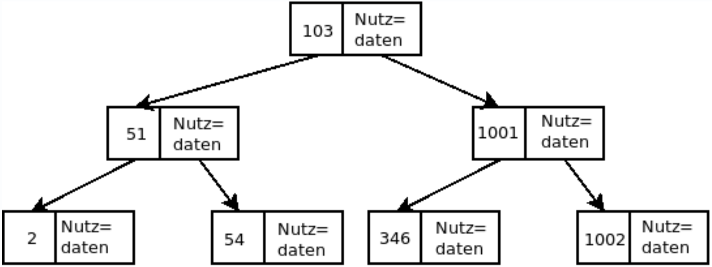
7 Dynamic Memory Allocation

8 Inheritance

9 Object Oriented Design

Associative Containers

- Sorted by nature
- Many kinds of associative containers → degree in Computer Science



Overview

1. Introduction
   - Introduction
2. Data Encapsulation
   - Classes and Objects
   - Constructors and Destructors
   - Object Copy
3. Functions and Methods
   - Overloading
   - Methods
   - `const`
   - `this`
   - References
   - `static`
   - Operatoren
4. Exceptions
   - Basics
   - `try` — `catch`
5. Templates
   - Function Templates
   - Class Templates
6. Standard Template Library
   - Basics
   - Sorting
   - Searching
   - Sequential Containers
   - Associative Containers
7. Dynamic Memory Allocation
8. Inheritance
9. Object Oriented Design

Overview

1. Introduction
   - Introduction
2. Data Encapsulation
   - Classes and Objects
   - Constructors and Destructors
   - Object Copy
3. Functions and Methods
   - Overloading
   - Methods
   - const
   - this
   - References
   - static
   - Operatoren
4. Exceptions
   - Basics
   - try — catch
5. Templates
   - Function Templates
   - Class Templates
6. Standard Template Library
   - Basics
   - Sorting
   - Searching
   - Sequential Containers
   - Associative Containers
7. Dynamic Memory Allocation
8. **Inheritance**
9. Object Oriented Design

Overview

1. Introduction
   - Introduction
2. Data Encapsulation
   - Classes and Objects
   - Constructors and Destructors
   - Object Copy
3. Functions and Methods
   - Overloading
   - Methods
   - `const`
   - `this`
   - References
   - `static`
   - Operatoren
4. Exceptions
   - Basics
   - `try` — `catch`
5. Templates
   - Function Templates
   - Class Templates
6. Standard Template Library
   - Basics
   - Sorting
   - Searching
   - Sequential Containers
   - Associative Containers
7. Dynamic Memory Allocation
8. Inheritance
9. Object Oriented Design

Dummy