24. Subtyping, Inheritance and Polymorphism

Expression Trees, Separation of Concerns and Modularisation, Type Hierarchies, Virtual Functions, Dynamic Binding, Code Reuse, Concepts of Object-Oriented Programming

Last Week: Expression Trees

- Goal: Represent arithmetic expressions, e.g.

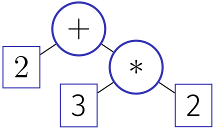
```
2 + 3 * 2
```

Last Week: Expression Trees

- Goal: Represent arithmetic expressions, e.g.

```
2 + 3 * 2
```

- Arithmetic expressions form a *tree structure*

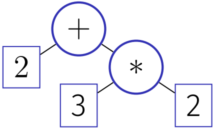


Last Week: Expression Trees

- Goal: Represent arithmetic expressions, e.g.

```
2 + 3 * 2
```

- Arithmetic expressions form a *tree structure*



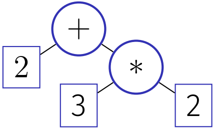
- Expression trees comprise *different* nodes:

Last Week: Expression Trees

- Goal: Represent arithmetic expressions, e.g.

```
2 + 3 * 2
```

- Arithmetic expressions form a *tree structure*

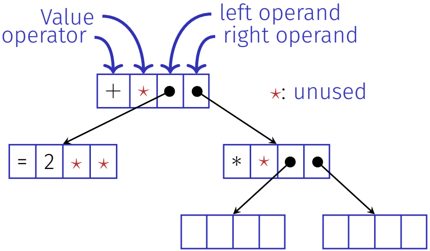


- Expression trees comprise *different* nodes: literals (e.g. 2), binary operators (e.g. $+$), unary operators (e.g. $\sqrt{\ }$), function applications (e.g. $\cos$), etc.

Disadvantages

Implemented via *a single* node type:

```
struct tnode {
  char op; // Operator ('=' for literals)
  double val; // Literal's value
  tnode* left; // Left child (or nullptr)
  tnode* right; // ...
  ...
};
```



Observation: **tnode** is the "sum" of all required nodes (constants, addition, ...) ⇒ memory wastage, inelegant

Disadvantages

Observation: `tnode` is the "sum" of all required nodes –

Disadvantages

Observation: `tnode` is the "sum" of all required nodes – and every function must "dissect" this "sum", e.g.:

```
double eval(const tnode* n) {
  if (n->op == '=') return n->val; // n is a constant
  double l = 0;
  if (n->left) l = eval(n->left); // n is not a unary operator
  double r = eval(n->right);
  switch(n->op) {
    case '+': return l+r; // n is an addition node
    case '*': return l*r; // ...
    ...
```

Disadvantages

Observation: **`tnode`** is the “sum” of all required nodes – and every function must “dissect” this “sum”, e.g.:

```
double eval(const tnode* n) {
  if (n->op == '=') return n->val; // n is a constant
  double l = 0;
  if (n->left) l = eval(n->left); // n is not a unary operator
  double r = eval(n->right);
  switch(n->op) {
    case '+': return l+r; // n is an addition node
    case '*': return l*r; // ...
    ...
```

⇒ Complex, and therefore error-prone

Disadvantages

```
struct tnode {
  char op;
  double val;
  tnode* left;
  tnode* right;
  ...
};
```

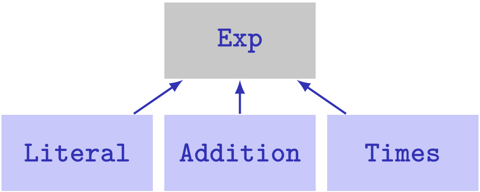
```
double eval(const tnode* n) {
  if (n->op == '=') return n->val;
  double l = 0;
  if (n->left) l = eval(n->left);
  double r = eval(n->right);
  switch(n->op) {
    case '+': return l+r;
    case '*': return l*r;
    ...
```

This code isn't *modular* – we'll change that today!

New Concepts Today

1. Subtyping

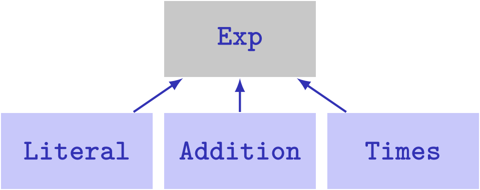
- Type hierarchy: **`Exp`** represents general expressions, **`Literal`** etc. are concrete expression



New Concepts Today

1. Subtyping

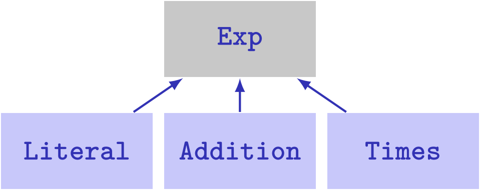
- Type hierarchy: **`Exp`** represents general expressions, **`Literal`** etc. are concrete expression
- Every **`Literal`** etc. also is an **`Exp`** (subtype relation)



New Concepts Today

1. Subtyping

- Type hierarchy: **Exp** represents general expressions, **Literal** etc. are concrete expression
- Every **Literal** etc. also is an **Exp** (subtype relation)



- That's why a **Literal** etc. can be used everywhere, where an **Exp** is expected:

```
Exp* e = new Literal(132);
```

New Concepts Today

2. Polymorphism and Dynamic Dispatch

- A variable of *static* type **Exp** can “host” expressions of different *dynamic* types:

```
Exp* e = new Literal(2); // e is the literal 2
e = new Addition(e, e); // e is the addition 2 + 2
```

New Concepts Today

2. Polymorphism and Dynamic Dispatch

- A variable of *static* type **`Exp`** can "host" expressions of different *dynamic* types:

```
Exp* e = new Literal(2); // e is the literal 2
e = new Addition(e, e); // e is the addition 2 + 2
```

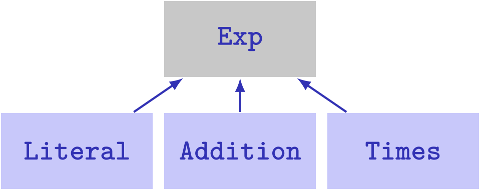
- Executed are the member functions of the *dynamic* type:

```
Exp* e = new Literal(2);
std::cout << e->eval(); // 2

e = new Addition(e, e);
std::cout << e->eval(); // 4
```

New Concepts Today

3. Inheritance

- Certain functionality is shared among type hierarchy members

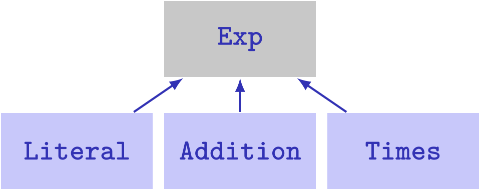


New Concepts Today

3. Inheritance

- Certain functionality is shared among type hierarchy members
- E.g. computing the size (nesting depth) of binary expressions (**`Addition`**, **`Times`**):

$$1 + size(left\ operand) + size(right\ operand)$$



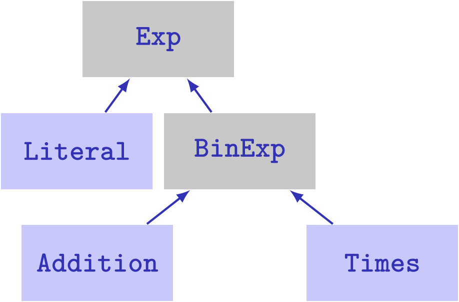
New Concepts Today

3. Inheritance

- Certain functionality is shared among type hierarchy members
- E.g. computing the size (nesting depth) of binary expressions (**`Addition`**, **`Times`**):

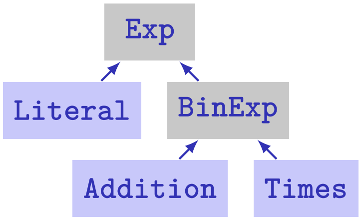
$$1 + size(left\ operand) + size(right\ operand)$$

⇒ Implement functionality once, and let subtypes *inherit* it



Advantages

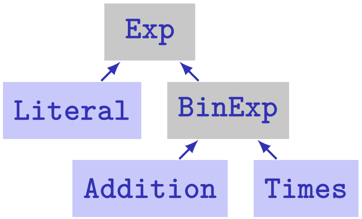
- Subtyping, inheritance and dynamic binding enable *modularisation through spezialisation*




```
Exp* e = new Literal(2);
std::cout << e->eval();

e = new Addition(e, e);
std::cout << e->eval();
```

Advantages

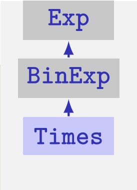
- Subtyping, inheritance and dynamic binding enable *modularisation through spezialisation*
- Inheritance enables sharing common code across modules
  $\Rightarrow$ *avoid code duplication*




```
Exp* e = new Literal(2);
std::cout << e->eval();

e = new Addition(e, e);
std::cout << e->eval();
```

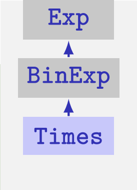
Syntax and Terminology

```
struct Exp {
  ...
}

struct BinExp : public Exp {
  ...
}

struct Times : public BinExp {
  ...
}
```



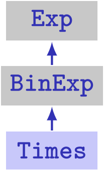
Syntax and Terminology

```
struct Exp {
  ...
}

struct BinExp : public Exp {
  ...
}

struct Times : public BinExp {
  ...
}
```



Note: Today, we focus on the new concepts (subtyping, ...) and ignore the orthogonal aspect of encapsulation (**class**, **private** vs. **public** member variables)

Syntax and Terminology

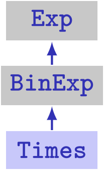



```
struct Exp {
  ...
}

struct BinExp : public Exp {
  ...
}

struct Times : public BinExp {
  ...
}
```

- **`BinExp`** is a *subclass*[1] of **`Exp`**

[1] derived class, child class   [2] base class, parent class

Syntax and Terminology



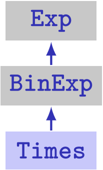

```
struct Exp {
  ...
}

struct BinExp : public Exp {
  ...
}

struct Times : public BinExp {
  ...
}
```

- **`BinExp`** is a *subclass*[1] of **`Exp`**
- **`Exp`** is the *superclass*[2] of **`BinExp`**

[1] derived class, child class   [2] base class, parent class

Syntax and Terminology

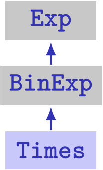
```
struct Exp {
  ...
}

struct BinExp : public Exp {
  ...
}

struct Times : public BinExp {
  ...
}
```



- **BinExp** is a *subclass*[1] of **Exp**
- **Exp** is the *superclass*[2] of **BinExp**
- **BinExp** *inherits* from **Exp**

[1] derived class, child class   [2] base class, parent class

Syntax and Terminology



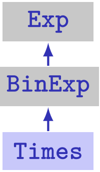

```
struct Exp {
  ...
}

struct BinExp : public Exp {
  ...
}

struct Times : public BinExp {
  ...
}
```

- **`BinExp`** is a *subclass*[1] of **`Exp`**
- **`Exp`** is the *superclass*[2] of **`BinExp`**
- **`BinExp`** *inherits* from **`Exp`**
- **`BinExp`** *publicly* inherits from **`Exp`** (**`public`**), that's why **`BinExp`** is a *subtype* of **`Exp`**

[1] derived class, child class [2] base class, parent class

Syntax and Terminology

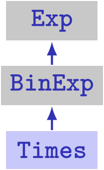
```
struct Exp {
  ...
}

struct BinExp : public Exp {
  ...
}

struct Times : public BinExp {
  ...
}
```



- **BinExp** is a *subclass*[1] of **Exp**
- **Exp** is the *superclass*[2] of **BinExp**
- **BinExp** *inherits* from **Exp**
- **BinExp** *publicly* inherits from **Exp** (**public**), that's why **BinExp** is a *subtype* of **Exp**
- Analogously: **Times** and **BinExp**

[1] derived class, child class   [2] base class, parent class

Syntax and Terminology




```
struct Exp {
  ...
}

struct BinExp : public Exp {
  ...
}

struct Times : public BinExp {
  ...
}
```

- **`BinExp`** is a *subclass*[1] of **`Exp`**
- **`Exp`** is the *superclass*[2] of **`BinExp`**
- **`BinExp`** *inherits* from **`Exp`**
- **`BinExp`** *publicly* inherits from **`Exp`** (**`public`**), that's why **`BinExp`** is a *subtype* of **`Exp`**
- Analogously: **`Times`** and **`BinExp`**
- Subtype relation is transitive: **`Times`** is also a subtype of **`Exp`**

[1] derived class, child class [2] base class, parent class

Abstract Class `Exp` and Concrete Class `Literal`

```
struct Exp {
  virtual int size() const = 0;
  virtual double eval() const = 0;
};
```

Abstract Class `Exp` and Concrete Class `Literal`

```
struct Exp {
  virtual int size() const = 0;
  virtual double eval() const = 0;
};
```

Activates dynamic dispatch

Abstract Class `Exp` and Concrete Class `Literal`

```
struct Exp {
  virtual int size() const = 0;
  virtual double eval() const = 0;
};
```

Enforces implementation by derived classes ...

Abstract Class `Exp` and Concrete Class `Literal`

```
struct Exp {
  virtual int size() const = 0;
  virtual double eval() const = 0;
};
```

… that makes **`Exp`** an *abstract* class

Enforces implementation by derived classes …

Abstract Class `Exp` and Concrete Class `Literal`

```
struct Exp {
  virtual int size() const = 0;
  virtual double eval() const = 0;
};
```

```
struct Literal : public Exp {
  double val;

  Literal(double v);
  int size() const;
  double eval() const;
};
```

Abstract Class Exp and Concrete Class Literal

```
struct Exp {
  virtual int size() const = 0;
  virtual double eval() const = 0;
};
```

```
struct Literal : public Exp {
  double val;

  Literal(double v);
  int size() const;
  double eval() const;
};
```

Literal inherits from Exp ...

Abstract Class `Exp` and Concrete Class `Literal`

```
struct Exp {
  virtual int size() const = 0;
  virtual double eval() const = 0;
};
```

```
struct Literal : public Exp {
  double val;

  Literal(double v);
  int size() const;
  double eval() const;
};
```

`Literal` inherits from **`Exp`** . . .

. . . but is otherwise just a regular class

`Literal`: Implementation

```
Literal::Literal(double v): val(v) {}
```

`Literal`: Implementation

```
Literal::Literal(double v): val(v) {}
```

```
int Literal::size() const {
  return 1;
}
```

Literal: Implementation

```
Literal::Literal(double v): val(v) {}
```

```
int Literal::size() const {
  return 1;
}
```

```
double Literal::eval() const {
  return this->val;
}
```

Subtyping: A Literal is an Expression

A pointer to a subtype can be used everywhere, where a pointer to a supertype is required:

```
Literal* lit = new Literal(5);
```

Subtyping: A Literal is an Expression

A pointer to a subtype can be used everywhere, where a pointer to a supertype is required:

```
Literal* lit = new Literal(5);
Exp* e = lit; // OK: Literal is a subtype of Exp
```

Subtyping: A Literal is an Expression

A pointer to a subtype can be used everywhere, where a pointer to a supertype is required:

```
Literal* lit = new Literal(5);
Exp* e = lit; // OK: Literal is a subtype of Exp
```

But not vice versa:

```
Exp* e = ...
Literal* lit = e; // ERROR: Exp is not a subtype of Literal
```

Polymorphie: a Literal Behaves Like a Literal

```
struct Exp {
  ...
  virtual double eval();
};

double Literal::eval() {
  return this->val;
}
```

```
Exp* e = new Literal(3);
std::cout << e->eval(); // 3
```

Polymorphie: a Literal Behaves Like a Literal

```
struct Exp {
  ...
  virtual double eval();
};

double Literal::eval() {
  return this->val;
}
```

```
Exp* e = new Literal(3);
std::cout << e->eval(); // 3
```

- *virtual* member function: the *dynamic* (here: **`Literal`**) type determines the member function to be executed
  $\Rightarrow$ *dynamic binding*

Polymorphie: a Literal Behaves Like a Literal

```
struct Exp {
  ...
  virtual double eval();
};

double Literal::eval() {
  return this->val;
}
```

```
Exp* e = new Literal(3);
std::cout << e->eval(); // 3
```

- *virtual* member function: the *dynamic* (here: **Literal**) type determines the member function to be executed
  $\Rightarrow$ *dynamic binding*
- Without **Virtual** the *static type* (hier: **Exp**) determines which function is executed

Polymorphie: a Literal Behaves Like a Literal

```
struct Exp {
  ...
  virtual double eval();
};

double Literal::eval() {
  return this->val;
}
```

```
Exp* e = new Literal(3);
std::cout << e->eval(); // 3
```

- *virtual* member function: the *dynamic* (here: **`Literal`**) type determines the member function to be executed
  ⇒ *dynamic binding*
- Without **`Virtual`** the *static type* (hier: **`Exp`**) determines which function is executed
- We won't go into further details

Further Expressions: Addition and Times

```
struct Addition : public Exp {
  Exp* left; // left operand
  Exp* right; // right operand
  ...
};
```

Further Expressions: `Addition` and `Times`

```
struct Addition : public Exp {
  Exp* left; // left operand
  Exp* right; // right operand
  ...
};
```

```
struct Times : public Exp {
  Exp* left; // left operand
  Exp* right; // right operand
  ...
};
```

Further Expressions: `Addition` and `Times`

```
struct Addition : public Exp {
  Exp* left; // left operand
  Exp* right; // right operand
  ...
};
```

```
int Addition::size() const {
  return 1 + left->size()
           + right->size();
}
```

```
struct Times : public Exp {
  Exp* left; // left operand
  Exp* right; // right operand
  ...
};
```

Further Expressions: Addition and Times

```
struct Addition : public Exp {
  Exp* left; // left operand
  Exp* right; // right operand
  ...
};
```

```
int Addition::size() const {
  return 1 + left->size()
           + right->size();
}
```

```
struct Times : public Exp {
  Exp* left; // left operand
  Exp* right; // right operand
  ...
};
```

```
int Times::size() const {
  return 1 + left->size()
           + right->size();
}
```

Further Expressions: Addition and Times

```
struct Addition : public Exp {
  Exp* left; // left operand
  Exp* right; // right operand
  ...
};
```

```
struct Times : public Exp {
  Exp* left; // left operand
  Exp* right; // right operand
  ...
};
```

```
int Addition::size() const {
  return 1 + left->size()
           + right->size();
}
```

```
int Times::size() const {
  return 1 + left->size()
           + right->size();
}
```

😃 Separation of concerns

Further Expressions: Addition and Times

```
struct Addition : public Exp {
  Exp* left; // left operand
  Exp* right; // right operand
  ...
};
```

```
struct Times : public Exp {
  Exp* left; // left operand
  Exp* right; // right operand
  ...
};
```

```
int Addition::size() const {
  return 1 + left->size()
           + right->size();
}
```

```
int Times::size() const {
  return 1 + left->size()
           + right->size();
}
```

😃 Separation of concerns

🤬 Code duplication

Extracting Commonalities ...: BinExp

```
struct BinExp : public Exp {
  Exp* left;
  Exp* right;

  BinExp(Exp* l, Exp* r);
  int size() const;
};
```

```
BinExp::BinExp(Exp* l, Exp* r): left(l), right(r) {}
```

Extracting Commonalities ...: BinExp

```
struct BinExp : public Exp {
  Exp* left;
  Exp* right;

  BinExp(Exp* l, Exp* r);
  int size() const;
};
```

```
BinExp::BinExp(Exp* l, Exp* r): left(l), right(r) {}
```

```
int BinExp::size() const {
  return 1 + this->left->size() + this->right->size();
}
```

Note: **`BinExp`** does not implement **`eval`** and is therefore also an abstract class, just like **`Exp`**

...Inheriting Commonalities: Addition

```
struct Addition : public BinExp {
  Addition(Exp* l, Exp* r);
  double eval() const;
};
```

...Inheriting Commonalities: `Addition`

```
struct Addition : public BinExp {
  Addition(Exp* l, Exp* r);
  double eval() const;
};
```

`Addition` inherits member variables (**`left`**, **`right`**) and functions (**`size`**) from **`BinExp`**

...Inheriting Commonalities: `Addition`

```
struct Addition : public BinExp {
  Addition(Exp* l, Exp* r);
  double eval() const;
};
```

```
Addition::Addition(Exp* l, Exp* r): BinExp(l, r) {}
```

Calling the *super constructor* (constructor of **`BinExp`**) initialises the member variables **`left`** and **`right`**

...Inheriting Commonalities: `Addition`

```
struct Addition : public BinExp {
  Addition(Exp* l, Exp* r);
  double eval() const;
};
```

```
Addition::Addition(Exp* l, Exp* r): BinExp(l, r) {}
```

```
double Addition::eval() const {
  return
    this->left->eval() +
    this->right->eval();
}
```

...Inheriting Commonalities: `Times`

```
struct Times : public BinExp {
  Times(Exp* l, Exp* r);
  double eval() const;
};
```

```
Times::Times(Exp* l, Exp* r): BinExp(l, r) {}
```

```
double Times::eval() const {
  return
    this->left->eval() *
    this->right->eval();
}
```

Observation: **`Additon::eval()`** and **`Times::eval()`** are very similar and could also be unified. However, this would require the concept of *functional programming*, which is outside the scope of this course.

Further Expressions and Operations

- Further expressions, as classes derived from **`Exp`**, are possible, e.g. $-$, $/$, $\sqrt{}$, $\cos$, $\log$

Further Expressions and Operations

- Further expressions, as classes derived from **`Exp`**, are possible, e.g. $-$, $/$, $\sqrt{}$, $\cos$, $\log$
- A former bonus exercise (included in today's lecture examples on Code Expert) illustrates possibilities: variables, trigonometric functions, parsing, pretty-printing, numeric simplifications, symbolic derivations, ...

Mission: Monolithic → Modular ✓

```
struct tnode {
  char op;
  double val;
  tnode* left;
  tnode* right;
  ...
}
```

```
double eval(const tnode* n) {
  if (n->op == '=') return n->val;
  double l = 0;
  if (n->left != 0) l = eval(n->left);
  double r = eval(n->right);
  switch(n->op) {
    case '+': return l + r;
    case '*': return l - r;
    case '-': return l - r;
    case '/': return l / r;
    default:
      // unknown operator
      assert (false);
  }
}
```

```
int size (const tnode* n) const { ... }

...
```

```
struct Literal : public Exp {
  double val;
  ...
  double eval() const {
    return val;
  }
};
```

```
struct Addition : public Exp {
  ...
  double eval() const {
    return left->eval() + right->eval();
  }
};
```

```
struct Times : public Exp {
  ...
  double eval() const {
    return left->eval() * right->eval();
  }
}
```

\+

```
struct Cos : public Exp {
  ...
  double eval() const {
    return std::cos(argument->eval());
  }
}
```

And there is so much more …

Not shown/discussed:

- Private inheritance (`class B :` ~~`public`~~ `A`)
- Subtyping and polymorphism without pointers
- Non-virtuell member functions and static dispatch (~~`virtual`~~ `double eval()`)
- Overriding inherited member functions and invoking overridden implementations
- Multiple inheritance
- …

Object-Oriented Programming

In the last 3rd of the course, several concepts of *object-oriented programming* were introduced, that are briefly summarised on the upcoming slides.

Encapsulation (weeks 10-13):

- Hide the implementation details of types (private section) from users
- Definition of an interface (public area) for accessing values and functionality in a controlled way
- Enables ensuring invariants, and the modification of implementations without affecting user code

Object-Oriented Programming

Subtyping (week 14):

- Type hierarchies, with super- and subtypes, can be created to model relationships between more abstract and more specialised entities
- A subtype supports at least the functionality that its supertype supports – typically more, though, i.e. a subtype extends the interface (public section) of its supertype
- That's why supertypes can be used anywhere, where subtypes are required ...
- ...and functions that can operate on more abstract type (supertypes) can also operate on more specialised types (subtypes)
- The streams introduced in week 7 form such a type hierarchy: **`ostream`** is the abstract supertyp, **`ofstream`** etc. are specialised subtypes

Object-Oriented Programming

Polymorphism and *dynamic binding* (week 14):

- A pointer of static typ T_1 can, at runtime, point to objects of (dynamic) type T_2, if T_2 is a subtype of T_1
- When a virtual member function is invoked from such a pointer, the dynamic type determines which function is invoked
- I.e.: despite having the same static type, a different behaviour can be observed when accessing the common interface (member functions) of such pointers
- In combination with subtyping, this enables adding further concrete types (streams, expressions, $\ldots$) to an existing system, without having to modify the latter

Object-Oriented Programming

Inheritance (week 14):

- Derived classes inherit the functionality, i.e. the implementation of member functions, of their parent classes
- This enables sharing common code and thereby avoids code duplication
- An inherited implementation can be overridden, which allows derived classes to behave differently than their parent classes (not shown in this course)

25. Conclusion

Purpose and Format

Name the most important key words to each chapter. Checklist: “does every notion make some sense for me?”

- Ⓜ motivating example for each chapter
- Ⓒ concepts that do not depend from the implementation (language)
- Ⓛ language (C++): all that depends on the chosen language
- Ⓔ examples from the lectures

Kapitelüberblick

- 1. Introduction
- 2. Integers
- 3. Booleans
- 4. Defensive Programming
- 5./6. Control Statements
- 7./8. Floating Point Numbers
- 9./10. Functions
- 11. Reference Types
- 12./13. Vectors and Strings
- 14./15. Recursion
- 16. Structs and Overloading
- 17. Classes
- 18./19. Dynamic Datastructures
- 20. Containers, Iterators and Algorithms
- 21. Dynamic Datatypes and Memory Management
- 22. Subtyping, Polymorphism and Inheritance

1. Introduction

- Ⓜ Euclidean algorithm
- Ⓒ algorithm, Turing machine, programming languages, compilation, syntax and semantics
- values and effects, fundamental types, literals, variables
- Ⓛ include directive `#include <iostream>`
- main function `int main(){...}`
- comments, layout `// Kommentar`
- types, variables, L-value `a` , R-value `a+b`
- expression statement `b=b*b;` , declaration statement `int a;`, return statement `return 0;`

2. Integers

Ⓜ
- Celsius to Fahrenheit

Ⓒ
- associativity and precedence, arity
- expression trees, evaluation order
- arithmetic operators
- binary representation, hexadecimal numbers
- signed numbers, twos complement

Ⓛ
- arithmetic operators `9 * celsius / 5 + 32`
- increment / decrement `expr++`
- arithmetic assignment `expr1 += expr2`
- conversion `int` ↔ `unsigned int`

Ⓔ
- Celsius to Fahrenheit, equivalent resistance

3. Booleans

Ⓒ

- Boolean functions, completeness
- DeMorgan rules

Ⓛ

- the type `bool`
- logical operators `a && !b`
- relational operators `x < y`
- precedences `7 + x < y && y != 3 * z`
- short circuit evaluation `x != 0 && z / x > y`
- the **`assert`**-statement, **`#include <cassert>`**

Ⓔ

- Div-Mod identity.

4. Definsive Programming

Ⓒ

- Assertions and Constants

Ⓛ

- The **`assert`**-statement, **`#include <cassert>`**
- **`const int speed_of_light=2999792458`**

Ⓔ

- Assertions for the GCD

5./6. Control Statements

Ⓜ
- linear control flow vs. interesting programs

Ⓒ
- selection statements, iteration statements
- (avoiding) endless loops, halting problem
- Visibility and scopes, automatic memory
- equivalence of iteration statement

Ⓛ
- if statements `if (a % 2 == 0) {..}`
- for statements `for (unsigned int i = 1; i <= n; ++i) ...`
- while and do-statements `while (n > 1) {...}`
- blocks and branches `if (a < 0) continue;`
- Switch statement `switch(grade) {case 6: }`

Ⓔ
- sum computation (Gauss), prime number tests, Collatz sequence, Fibonacci numbers, calculator, output grades

7./8. Floating Point Numbers

Ⓜ
- correct computation: Celsius / Fahrenheit

Ⓒ
- fixpoint vs. floating point
- holes in the value range
- compute using floating point numbers
- floating point number systems, normalisation, IEEE standard 754
- *guidelines for computing with floating point numbers*

Ⓛ
- types `float`, `double`
- floating point literals `1.23e-7f`

Ⓔ
- Celsius/Fahrenheit, Euler, Harmonic Numbers

9./10. Functions

Ⓜ
- Computation of Powers

Ⓒ
- Encapsulation of Functionality
- functions, formal arguments, arguments
- scope, forward declarations
- procedural programming, modularization, separate compilation
- *Stepwise Refinement*

Ⓛ
- declaration and definition of functions

  `double pow(double b, int e){ ... }`
- function call `pow (2.0, -2)`
- the type `void`

Ⓔ
- powers, perfect numbers, minimum, calendar

11. Reference Types

Ⓜ
- Swap

Ⓒ
- value- / reference- semantics, pass by value, pass by reference, return by reference
- lifetime of objects / temporary objects
- constants

Ⓛ
- reference type `int& a`
- call by reference, return by reference `int& increment (int& i)`
- const guideline, const references, reference guideline

Ⓔ
- swap, increment

12./13. Vectors and Strings

Ⓜ
- Iterate over data: sieve of erathosthenes

Ⓒ
- vectors, memory layout, random access
- (missing) bound checks
- vectors
- characters: ASCII, UTF8, texts, strings

Ⓛ
- vector types `std::vector<int> a {4,3,5,2,1};`
- characters and texts, the type char `char c = 'a';`, Konversion nach **int**
- vectors of vectors
- Streams `std::istream, std::ostream`

Ⓔ
- sieve of Erathosthenes, Caesar-code, shortest paths

14./15. Recursion

Ⓜ

- recursive math. functions, the n-Queen problem, Lindenmayer systems, a command line calculator

Ⓒ

- recursion
- call stack, memory of recursion
- correctness, termination,
- recursion vs. iteration
- Backtracking, EBNF, formal grammars, parsing

Ⓔ

- factorial, GCD, sudoku-solver, command line calcoulator

16. Structs and Overloading

Ⓜ

- build your own rational number

Ⓒ

- heterogeneous data types
- function and operator overloading
- encapsulation of data

Ⓛ

- struct definition `struct rational {int n; int d;};`
- member access `result.n = a.n * b.d + a.d * b.n;`
- initialization and assignment,
- function overloading `pow(2)` vs. `pow(3,3);`, operator overloading

Ⓔ

- rational numbers, complex numbers

17. Classes

Ⓜ
- rational numbers with encapsulation

Ⓒ
- Encapsulation, Construction, Member Functions

Ⓛ
- classes `class rational { ... };`
- access control `public:` / `private:`
- member functions `int rational::denominator () const`
- The implicit argument of the member functions

Ⓔ
- finite rings, complex numbers

18./19. Dynamic Datastructures

Ⓜ
- Our own vector

Ⓒ
- linked list, allocation, deallocation, dynamic data type

Ⓛ
- The **`new`** statement
- pointer `int* x;`, Null-pointer `nullptr`.
- address and derference operator `int *ip = &i; int j = *ip;`
- pointer and const `const int *a;`

Ⓔ
- linked list, stack

20. Containers, Iterators and Algorithms

M

- vectors are containers

C

- iteration with pointers
- containers and iterators
- algorithms

L

- Iterators `std::vector<int>::iterator`
- Algorithms of the standard library `std::fill (a, a+5, 1);`
- implement an iterator
- iterators and const

E

- output a vector, a set

21. Dynamic Datatypes and Memory Management

Ⓜ

- Stack
- Expression Tree

Ⓒ

- Guideline ”dynamic memory“
- Pointer sharing
- Dynamic Datatype
- Tree-Structure

Ⓛ

- **`new`** and **`delete`**
- Destructor **`stack::~stack()`**
- Copy-Constructor **`stack::stack(const stack& s)`**
- Assignment operator **`stack& stack::operator=(const stack& s)`**
- Rule of Three

Ⓔ

- Binary Search Tree

22. Subtyping, Polymorphism and Inheritance

Ⓜ
- extend and generalize expression trees

Ⓒ
- Subtyping
- polymorphism and dynamic binding
- Inheritance

Ⓛ
- base class `struct Exp{}`
- derived class `struct BinExp: public Exp{}`
- abstract class `struct Exp{virtual int size() const = 0...}`
- polymorphie `virtual double eval()`

Ⓔ
- expression node and extensions

The End

End of the Course