

22. Subtyping, Polymorphie und Vererbung

Ausdrückbäume, Aufgabenteilung und Modularisierung,
Typhierarchien, virtuelle Funktionen, dynamische Bindung,
Code-Wiederverwendung, Konzepte der objektorientierten
Programmierung

Letzte Woche: Ausdrucksbäume

- Ziel: Arithmetische Ausdrücke repräsentieren, z.B.

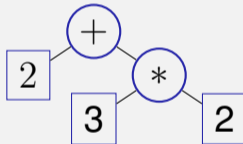
$$2 + 3 * 2$$

Letzte Woche: Ausdrucksbäume

- Ziel: Arithmetische Ausdrücke repräsentieren, z.B.

$$2 + 3 * 2$$

- Arithmetische Ausdrücke bilden eine *Baumstruktur*

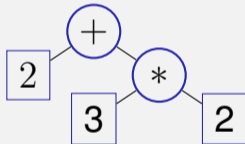


Letzte Woche: Ausdrucksbäume

- Ziel: Arithmetische Ausdrücke repräsentieren, z.B.

$$2 + 3 * 2$$

- Arithmetische Ausdrücke bilden eine *Baumstruktur*



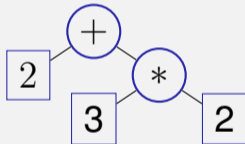
- Ausdrucksbäume bestehen aus *unterschiedlichen* Knoten:

Letzte Woche: Ausdrucksbäume

- Ziel: Arithmetische Ausdrücke repräsentieren, z.B.

$$2 + 3 * 2$$

- Arithmetische Ausdrücke bilden eine *Baumstruktur*

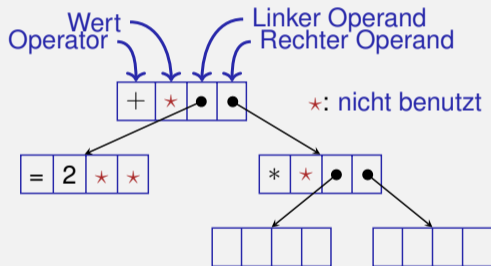


- Ausdrucksbäume bestehen aus *unterschiedlichen* Knoten: Literale (z.B. 2), binäre Operatoren (z.B. +), unäre Operatoren (z.B. $\sqrt{\quad}$), Funktionsanwendungen (z.B. \cos), etc.

Nachteile

Implementiert mittels *eines einzigen* Knotentyps:

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



Beobachtung: `tnode` ist die „Summe“ aller benötigten Knoten (Konstanten, Addition, ...) \Rightarrow Speicherverschwendung, unelegant

Nachteile

Beobachtung: `tnode` ist die „Summe“ aller benötigten Knoten –

Nachteile

Beobachtung: `tnode` ist die „Summe“ aller benötigten Knoten – und jede Funktion muss diese „Summe“ wieder „auseinander nehmen“, z.B.:

```
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; // ...
        ...
    }
}
```


Nachteile

Beobachtung: `tnode` ist die „Summe“ aller benötigten Knoten – und jede Funktion muss diese „Summe“ wieder „auseinander nehmen“, z.B.:

```
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; // ...
        ...
    }
}
```

⇒ Umständlich und somit fehleranfällig

Nachteile

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

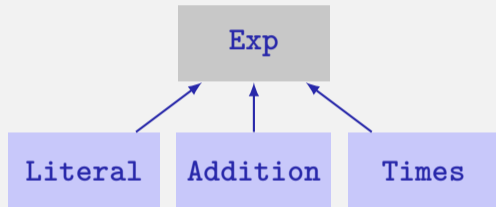
```
double eval(const tnode* n) {
    if (n->op == '=') return n->val;
    double l = 0;
    if (n->left) l = eval(n->left);
    double r = eval(n->right);
    switch(n->op) {
        case '+': return l+r;
        case '*': return l*r;
        ...
    }
```

Dieser Code ist nicht *modular* – das ändern wir heute!

Neue Konzepte heute

1. Subtyping

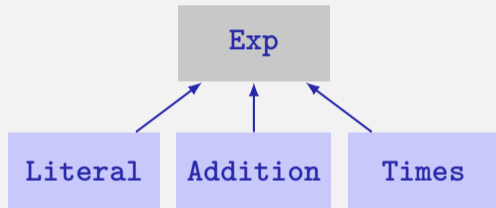
- Typhierarchie: `Exp` repräsentiert allgemeine Ausdrücke, `Literal` etc. sind konkrete Ausdrücke



Neue Konzepte heute

1. Subtyping

- Typhierarchie: `Exp` repräsentiert allgemeine Ausdrücke, `Literal` etc. sind konkrete Ausdrücke
- Jedes `Literal` etc. ist auch ein `Exp` (Subtyp-Beziehung)

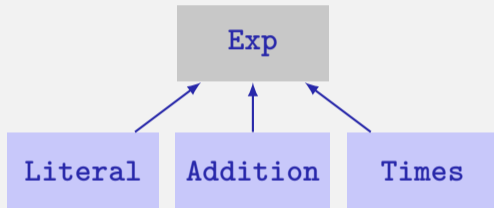


Neue Konzepte heute

1. Subtyping

- Typhierarchie: `Exp` repräsentiert allgemeine Ausdrücke, `Literal` etc. sind konkrete Ausdrücke
- Jedes `Literal` etc. ist auch ein `Exp` (Subtyp-Beziehung)
- Deswegen kann ein `Literal` etc. überall dort genutzt werden, wo ein `Exp` erwartet wird:

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



2. Polymorphie und dynamische Bindung

- Eine Variable vom *statischen* Typ `Exp` kann Ausdrücke mit unterschiedlichen *dynamischen* Typen „beherbergen“:

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

Neue Konzepte heute

2. Polymorphie und dynamische Bindung

- Eine Variable vom *statischen* Typ `Exp` kann Ausdrücke mit unterschiedlichen *dynamischen* Typen „beherbergen“:

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

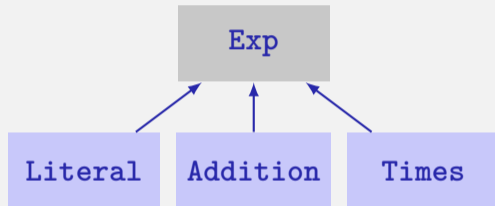
- Ausgeführt werden die Memberfunktionen des *dynamischen* Typs:

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

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

3. Vererbung

- Manche Funktionalität ist für mehrere Mitglieder der Typhierarchie gleich

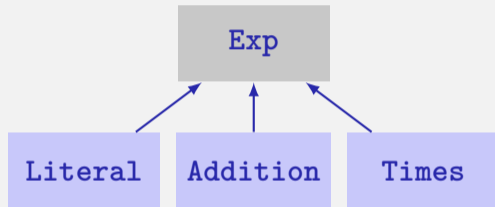


Neue Konzepte heute

3. Vererbung

- Manche Funktionalität ist für mehrere Mitglieder der Typhierarchie gleich
- Z.B. die Berechnung der Grösse (Verschachtelungstiefe) binärer Ausdrücke (**Addition**, **Times**):

$$1 + \textit{size}(\textit{left operand}) + \textit{size}(\textit{right operand})$$



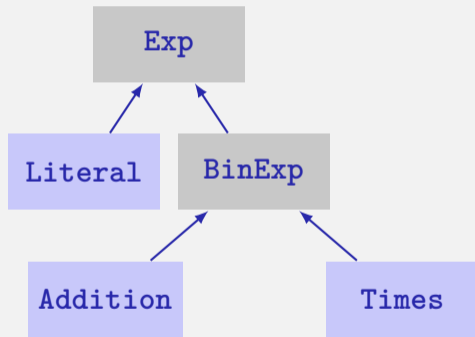
Neue Konzepte heute

3. Vererbung

- Manche Funktionalität ist für mehrere Mitglieder der Typhierarchie gleich
- Z.B. die Berechnung der Grösse (Verschachtelungstiefe) binärer Ausdrücke (**Addition**, **Times**):

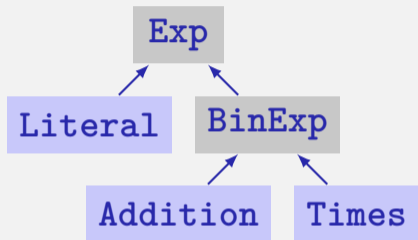
$$1 + \text{size}(\text{left operand}) + \text{size}(\text{right operand})$$

⇒ Funktionalität einmal implementieren und dann an Subtypen *vererben*



Vorteile

- Subtyping, Polymorphie und dynamische Bindung ermöglichen *Modularisierung durch Spezialisierung*

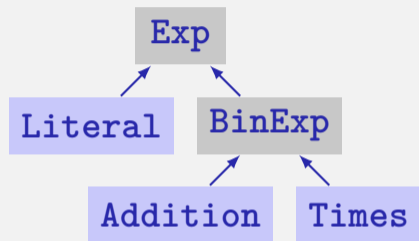


```
Exp* e = new Literal(2);  
std::cout << e->eval();
```

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

Vorteile

- Subtyping, Polymorphie und dynamische Bindung ermöglichen *Modularisierung durch Spezialisierung*
- Vererbung erlaubt gemeinsamen Code trotz Modularisierung
⇒ *Codeduplikation vermeiden*



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

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

Syntax und Terminologie

```
struct Exp {  
    ...  
}
```

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

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



Syntax und Terminologie

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



Anmerkung: Wir konzentrieren uns heute auf die neuen Konzepte (Subtyping, ...) und ignorieren den davon unabhängigen Aspekt der Kapselung (`class`, `private` vs. `public` Membervariablen)

Syntax und Terminologie

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



- BinExp ist eine von Exp *abgeleitete Klasse*¹

¹Subklasse, Kindklasse

²Superklasse, Elternklasse

Syntax und Terminologie

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



- BinExp ist eine von Exp *abgeleitete Klasse*¹
- Exp ist die *Basisklasse*² von BinExp

¹Subklasse, Kindklasse

²Superklasse, Elternklasse

Syntax und Terminologie

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



- BinExp ist eine von Exp *abgeleitete Klasse*¹
- Exp ist die *Basisklasse*² von BinExp
- BinExp *erbt* von Exp

¹Subklasse, Kindklasse

²Superklasse, Elternklasse

Syntax und Terminologie

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



- BinExp ist eine von Exp *abgeleitete Klasse*¹
- Exp ist die *Basisklasse*² von BinExp
- BinExp *erbt* von Exp
- Die Vererbung von Exp zu BinExp ist *öffentlich* (public), daher ist BinExp ein *Subtyp* von Exp

¹Subklasse, Kindklasse

²Superklasse, Elternklasse

Syntax und Terminologie

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



- BinExp ist eine von Exp *abgeleitete Klasse*¹
- Exp ist die *Basisklasse*² von BinExp
- BinExp *erbt* von Exp
- Die Vererbung von Exp zu BinExp ist *öffentlich* (public), daher ist BinExp ein *Subtyp* von Exp
- Analog: Times und BinExp

¹Subklasse, Kindklasse

²Superklasse, Elternklasse

Syntax und Terminologie

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



- BinExp ist eine von Exp *abgeleitete Klasse*¹
- Exp ist die *Basisklasse*² von BinExp
- BinExp *erbt* von Exp
- Die Vererbung von Exp zu BinExp ist *öffentlich* (public), daher ist BinExp ein *Subtyp* von Exp
- Analog: Times und BinExp
- Subtyprelation ist transitiv: Times ist ebenfalls ein Subtyp von Exp

¹Subklasse, Kindklasse ²Superklasse, Elternklasse

Abstrakte Klasse Exp und konkrete Klasse Literal

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

Abstrakte Klasse Exp und konkrete Klasse Literal

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

Aktiviert dynamische Bindung



Abstrakte Klasse Exp und konkrete Klasse Literal

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

Erzwingt Implementierung durch
abgeleitete Klassen ...

Abstrakte Klasse Exp und konkrete Klasse Literal

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

← ... das macht Exp zu einer *abstrakten* Klasse

← Erzwingt Implementierung durch abgeleitete Klassen ...

Abstrakte Klasse Exp und konkrete Klasse 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;  
};
```

Abstrakte Klasse Exp und konkrete Klasse Literal

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

```
struct Literal : public Exp { ← Literal erbt von Exp ...  
    double val;  
  
    Literal(double v);  
    int size() const;  
    double eval() const;  
};
```

Abstrakte Klasse Exp und konkrete Klasse Literal

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

```
struct Literal : public Exp { ← Literal erbt von Exp ...  
    double val;
```

```
    Literal(double v);  
    int size() const; ← ... ist aber ansonsten eine ganz normale Klasse  
    double eval() const;  
};
```

Literal: Implementierung

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

Literal: Implementierung

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

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

Literal: Implementierung

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

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

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

Subtyping: Ein Literal ist ein Ausdruck ...

Ein Zeiger auf einen Subtyp kann überall dort verwendet werden, wo ein Zeiger auf einen Supertyp gefordert ist:

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

Subtyping: Ein Literal ist ein Ausdruck ...

Ein Zeiger auf einen Subtyp kann überall dort verwendet werden, wo ein Zeiger auf einen Supertyp gefordert ist:

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


Subtyping: Ein Literal ist ein Ausdruck ...

Ein Zeiger auf einen Subtyp kann überall dort verwendet werden, wo ein Zeiger auf einen Supertyp gefordert ist:

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

Aber nicht umgekehrt:

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

Polymorphie: ... ein Literal verhält sich wie ein Literal

```
struct Exp {  
    ...  
    virtual double eval();  
};  
  
double Literal::eval() {  
    return this->val;  
}
```

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

Polymorphie: ... ein Literal verhält sich wie ein Literal

```
struct Exp {  
    ...  
    virtual double eval();  
};  
  
double Literal::eval() {  
    return this->val;  
}
```

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

- *Virtuelle* Memberfunktionen: der *dynamische* Typ (hier: `Literal`) bestimmt die auszuführenden Memberfunktionen
⇒ *dynamische Bindung*

Polymorphie: ... ein Literal verhält sich wie ein Literal

```
struct Exp {  
    ...  
    virtual double eval();  
};  
  
double Literal::eval() {  
    return this->val;  
}
```

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

- *Virtuelle* Memberfunktionen: der *dynamische* Typ (hier: `Literal`) bestimmt die auszuführenden Memberfunktionen
⇒ *dynamische Bindung*
- Ohne `virtual` bestimmt der *statische Typ* (hier: `Exp`) die auszuführende Funktion

Polymorphie: ... ein Literal verhält sich wie ein Literal

```
struct Exp {  
    ...  
    virtual double eval();  
};  
  
double Literal::eval() {  
    return this->val;  
}
```

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

- *Virtuelle* Memberfunktionen: der *dynamische* Typ (hier: `Literal`) bestimmt die auszuführenden Memberfunktionen
⇒ *dynamische Bindung*
- Ohne `virtual` bestimmt der *statische Typ* (hier: `Exp`) die auszuführende Funktion
- Wir vertiefen das nicht weiter

Weitere Ausdrücke: Addition und Times

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

Weitere Ausdrücke: Addition und 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  
    ...  
};
```

Weitere Ausdrücke: Addition und 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  
    ...  
};
```


Weitere Ausdrücke: Addition und 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();  
}
```

Weitere Ausdrücke: Addition und 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();  
}
```

😊 Aufgabenteilung

Weitere Ausdrücke: Addition und 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();  
}
```



Aufgabenteilung



Codeduplizierung

Gemeinsamkeiten auslagern ...: 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) {}
```

Gemeinsamkeiten auslagern ...: 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();  
}
```

Bemerkung: BinExp implementiert eval nicht und ist daher, genau wie Exp, eine abstrakte Klasse

... Gemeinsamkeiten erben: Addition

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

... Gemeinsamkeiten erben: Addition

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

← Addition erbt Membervariablen
(left, right) und Funktionen
(size) von BinExp

... Gemeinsamkeiten erben: Addition

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

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

Aufruf des *Superkonstruktors*
(Konstruktor von BinExp) zwecks
Initialisierung der Membervari-
ablen `left` und `right`

... Gemeinsamkeiten erben: 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();  
}
```

... Gemeinsamkeiten erben: 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();  
}
```

Beobachtung: `Additon::eval()` und `Times::eval()` sind sich sehr ähnlich und könnten ebenfalls zusammengelegt werden. Das dafür notwendige Konzept der *funktionalen Programmierung* geht jedoch über diesen Kurs hinaus.

Weitere Ausdrücke und Operationen

- Weitere Ausdrücke, als von `Exp` abgeleitete Klassen, sind möglich, z.B. $-$, $/$, $\sqrt{\quad}$, \cos , \log

Weitere Ausdrücke und Operationen

- Weitere Ausdrücke, als von `Exp` abgeleitete Klassen, sind möglich, z.B. $-$, $/$, $\sqrt{\quad}$, \cos , \log
- Eine ehemalige Bonusaufgabe (Teil der heutigen Vorlesungsbeispiele auf Code Expert) veranschaulicht, was alles möglich ist: Variablen, trigonometrische Funktionen, Parsing, Pretty-Printing, numerische Vereinfachungen, symbolische Ableitungen, ...

Mission: Monolithisch \rightarrow 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());  
    }  
};
```



Es gibt noch so viel mehr ...

Nicht gezeigt/besprochen:

- Private Vererbung (`class B : public A`)
- Subtyping und Polymorphie ohne Zeiger
- Nicht-virtuelle Memberfunktionen und statische Bindung (~~`virtual`~~ `double eval()`)
- Überschreiben geerbter Memberfunktionen und Aufrufen der überschriebenen Implementierung
- Mehrfachvererbung (multiple inheritance)
- ...

Objektorientierte Programmierung

Im letzten Kursdrittel wurden einige Konzepte der *objektorientierten Programmierung* vorgestellt, die auf den kommenden Folien noch einmal kurz zusammengefasst werden.

Kapselung (Wochen 10-13):

- Verbergen der Implementierungsdetails von Typen (privater Bereich) vor Benutzern
- Definition einer Schnittstelle (öffentlicher Bereich) zum kontrollierten Zugriff auf Werte und Funktionalität
- Ermöglicht das Sicherstellen von Invarianten, sowie den Austausch von Implementierungen ohne Anpassungen von Benutzercode

Objektorientierte Programmierung

Subtyping (Woche 14):

- Typhierarchien mit Super- und Subtypen können angelegt werden um Verwandtschaftbeziehungen sowie Abstraktionen und Spezialisierungen zu modellieren
- Ein Subtyp unterstützen mindestens die Funktionalität, die auch der Supertyp unterstützt – i.d.R. aber mehr, d.h. Subtypen erweitern die Schnittstelle (den öffentlichen Bereich) ihrer Supertypen
- Daher können Subtypen überall dort eingesetzt werden, wo Supertypen verlangt sind ...
- ... und Funktionen, die auf abstrakteren Typen (Supertypen) operieren können, können auch auf spezialisierteren Typen (Subtypen) operieren
- Die in Woche 7 vorgestellten Streams bilden eine solche Typhierarchie: `ostream` ist der abstrakte Supertyp, `ofstream` etc. sind spezialisierte Subtypen

Objektorientierte Programmierung

Polymorphie und *dynamische Bindung* (Woche 14):

- Ein Zeiger vom statischen Typ T_1 kann zur Laufzeit auf Objekte vom (dynamischen) Typ T_2 zeigen, falls T_2 ein Subtyp von T_1 ist
- Wird eine virtuelle Memberfunktion von einem solchen Zeiger aus aufgerufen, so entscheidet der dynamische Typ darüber, welche Funktion ausgeführt wird
- D.h.: Trotz gleichem statischen Typ kann beim Zugriff auf eine gemeinsame Schnittstelle (Memberfunktionen) eines solchen Zeigers ein anderes Verhalten auftreten
- Zusammen mit Subtyping ermöglicht es dies, neue konkrete Typen (Streams, Ausdrücke, ...) zu einem bestehenden System hinzuzufügen, ohne dieses abändern zu müssen

Objektorientierte Programmierung

Vererbung (Woche 14):

- Abgeleitete Klassen erben die Funktionalität, d.h. die Implementierungen von Memberfunktionen, ihrer Elternklassen
- Dies ermöglicht es, gemeinsam genutzten Code wiederverwenden zu können und vermeidet so Codeduplikation
- Geerbte Implementierungen können auch überschrieben werden, um zu erreichen, dass eine abgeleitete Klasse sich anders verhält als ihre Elternklasse (im Kurs nicht gezeigt)

— Ende der Vorlesung —