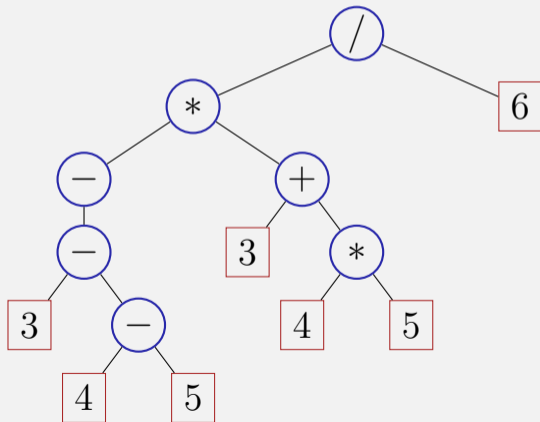


20. Vererbung und Polymorphie

Ausdrucksbäume, Vererbung, Code-Wiederverwendung, virtuelle Funktionen, Polymorphie, Konzepte des objektorientierten Programmierens

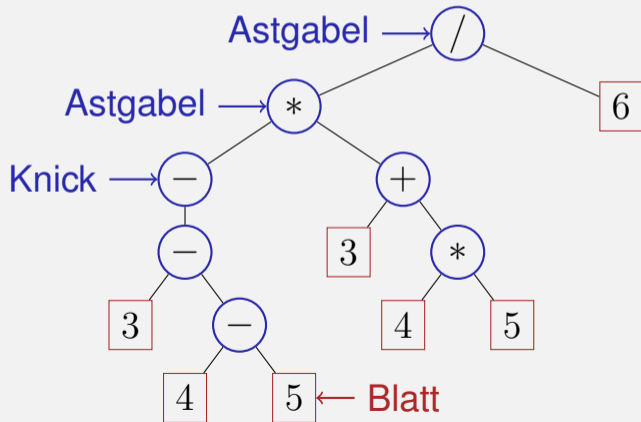
(Ausdrucks-)Bäume

$$-(3-(4-5))*(3+4*5)/6$$



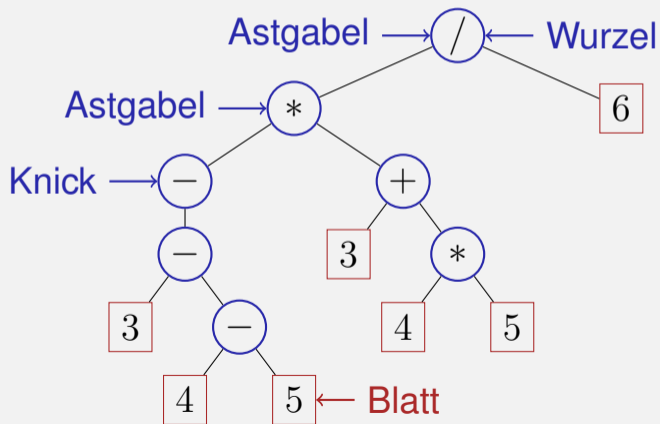
(Ausdrucks-)Bäume

$$-(3-(4-5))*(3+4*5)/6$$



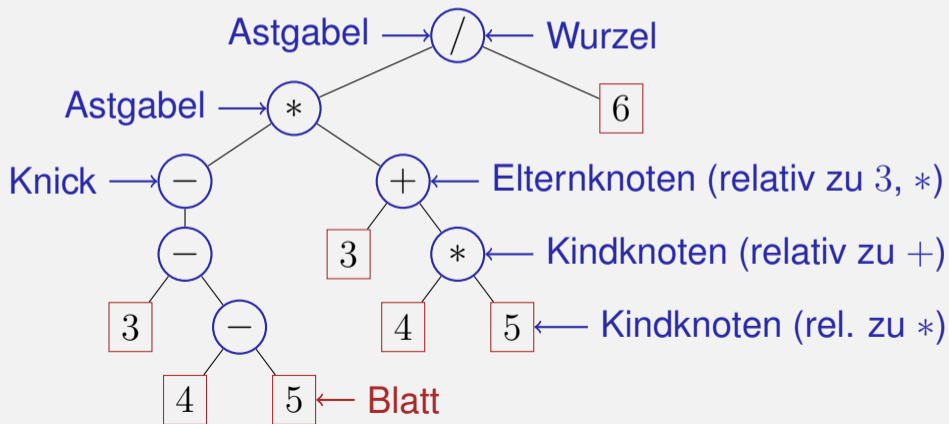
(Ausdrucks-)Bäume

$$-(3-(4-5))*(3+4*5)/6$$

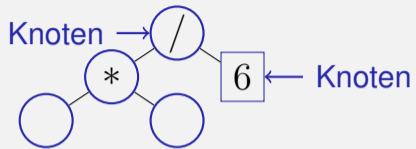


(Ausdrucks-)Bäume

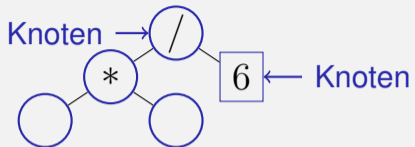
$$-(3-(4-5))*(3+4*5)/6$$



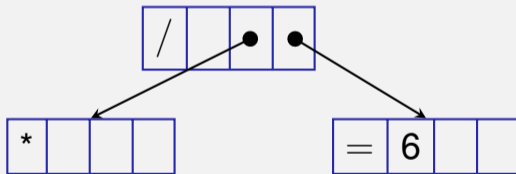
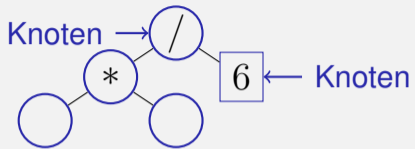
Astgabeln + Blätter + Knicke = Knoten



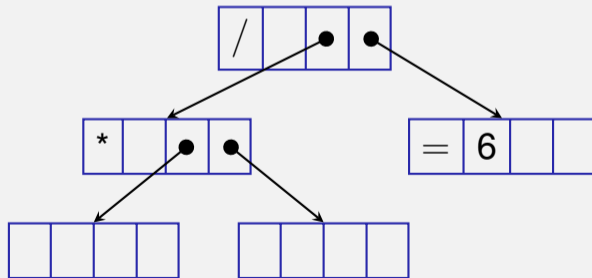
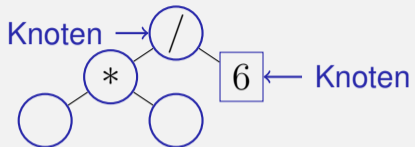
Astgabeln + Blätter + Knicke = Knoten



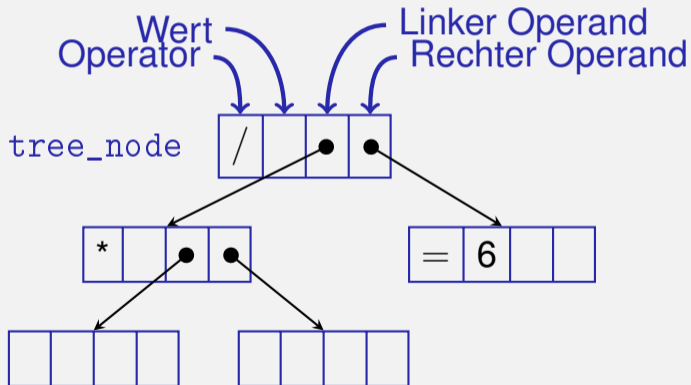
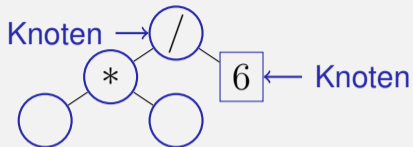
Astgabeln + Blätter + Knicke = Knoten



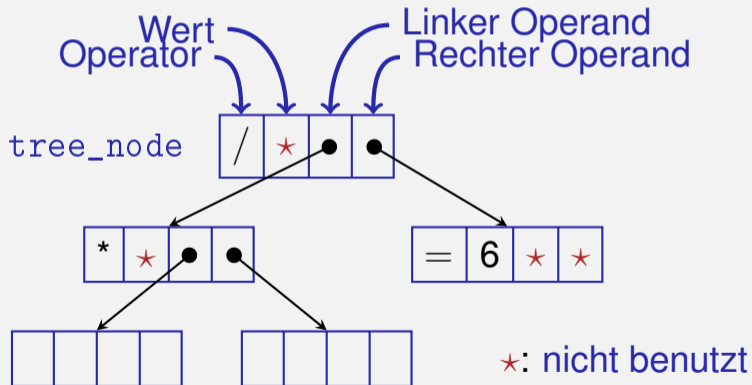
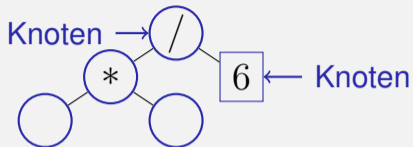
Astgabeln + Blätter + Knicke = Knoten



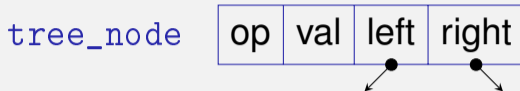
Astgabeln + Blätter + Knicke = Knoten



Astgabeln + Blätter + Knicke = Knoten



Knoten (struct tree_node)

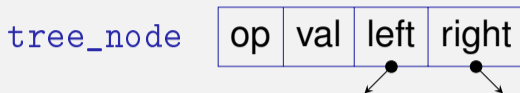


```
struct tree_node {
    char op; // leaf node: op is '='
             // internal node: op is '+', '-', '*', or '/'
    double val;
    tree_node* left;
    tree_node* right;

    tree_node(char o, double v, tree_node* l, tree_node* r)
        : op(o), val(v), left(l->copy()), right(r->copy()) {}

    ...
};
```

Knoten (struct tree_node)

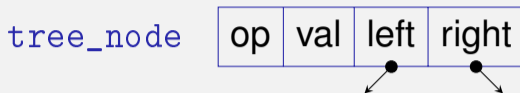


```
struct tree_node {
    char op; // leaf node: op is '='
             // internal node: op is '+', '-', '*', or '/'
    double val;
    tree_node* left; // == nullptr for unary minus
    tree_node* right;

    tree_node(char o, double v, tree_node* l, tree_node* r)
        : op(o), val(v), left(l->copy()), right(r->copy()) {}

    ...
};
```

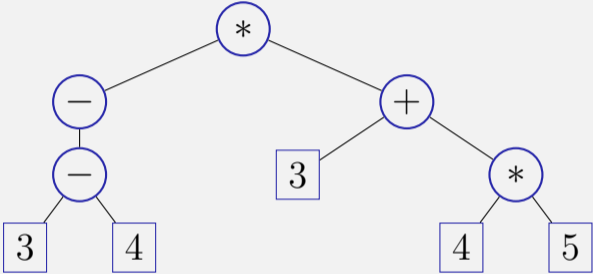
Knoten (struct tree_node)



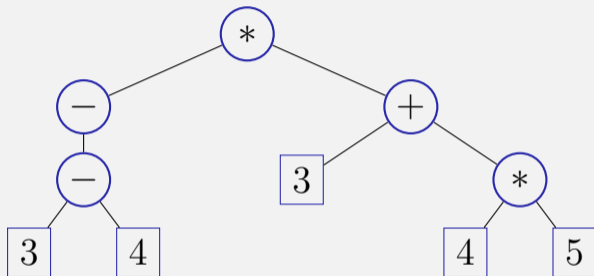
```
struct tree_node {
    char op; // leaf node: op is '='
             // internal node: op is '+', '-', '*', or '/'
    double val;
    tree_node* left; // == nullptr for unary minus
    tree_node* right;

    tree_node(char o, double v, tree_node* l, tree_node* r)
        : op(o), val(v), left(l->copy()), right(r->copy()) {}
    // Copy constructor: previous lecture; handout/Codeboard
    ...
};
```

Grösse = Knoten in Teilbäumen zählen

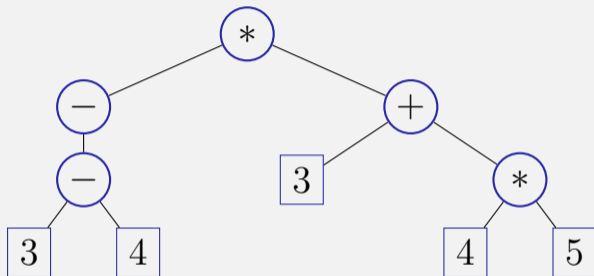


Grösse = Knoten in Teilbäumen zählen



- Grösse eines Blattes: 1
- Grösse anderer Knoten: 1 + Gesamtgrösse aller Kindknoten

Grösse = Knoten in Teilbäumen zählen



- Grösse eines Blattes: 1
- Grösse anderer Knoten: 1 + Gesamtgrösse aller Kindknoten
- Z.B. Grösse des „+“-Knoten ist 5

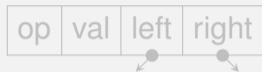
Knoten in Teilbäumen zählen

```
// POST: returns the size (number of nodes) of
//       the subtree with root *this
int tree_node::size () const {
    int s=1;
    if (left)
        s += left->size();
    if (right)
        s += right->size();
    return s;
}
```



Knoten in Teilbäumen zählen

```
// POST: returns the size (number of nodes) of
//       the subtree with root *this
int tree_node::size () const {
    int s=1;
    if (left) // shortcut for left != nullptr
        s += left->size();
    if (right)
        s += right->size();
    return s;
}
```



Teilbäume auswerten

```
// POST: evaluates the subtree with root *this
double tree_node::eval () const {
    if (op == '=') return val; ← Blatt...
    double l = 0;                ...oder Astgabel:
    if (left) l = left->eval(); ← op unär, oder linker Ast
    double r = right->eval(); ← rechter Ast
    if (op == '+') return l + r;
    if (op == '-') return l - r;
    if (op == '*') return l * r;
    if (op == '/') return l / r;
    return 0;
}
```



Ausdrucksbäume benutzen

```
// Construct a tree for  $1 - (-(3 + 7))$ 
tree_node* n1 = new tree_node('=', 3, nullptr, nullptr);
tree_node* n2 = new tree_node('=', 7, nullptr, nullptr);
tree_node* n3 = new tree_node('+', 0, n1, n2);
tree_node* n4 = new tree_node('-', 0, nullptr, n3);
tree_node* n5 = new tree_node('=', 1, nullptr, nullptr);
tree_node* root = new tree_node('-', 0, n5, n4);

// Evaluate the overall tree
std::cout << "1 - (-(3 + 7)) = " << root->eval() << '\n';

// Evaluate a subtree
std::cout << "3 + 7 = " << n3->eval() << '\n';
```

Beobachtungen

- Klasse `tree_node` ist die „Summe“ aller benötigten Knoten (Konstanten, Addition, ...)
- Gilt für die Datenrepräsentation ...

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

Beobachtungen I

- Klasse `tree_node` ist die „Summe“ aller benötigten Knoten
- ... und den Code

```
double tree_node::eval () const {  
    if (op == '=') ...  
    if (op == '+') ...  
    ...  
}
```

Beobachtungen I

- Klasse `tree_node` ist die „Summe“ aller benötigten Knoten
- ... und den Code

```
double tree_node::eval () const {  
    if (op == '=') ...  
    if (op == '+') ...  
    ...  
}
```

- **Aber:** Jedes *Objekt* (Instanz der Klasse) repräsentiert genau *einen* Knoten

Beobachtungen I

- Klasse `tree_node` ist die „Summe“ aller benötigten Knoten
- ... und den Code

```
double tree_node::eval () const {  
    if (op == '=') ...  
    if (op == '+') ...  
    ...  
}
```

- **Aber:** Jedes *Objekt* (Instanz der Klasse) repräsentiert genau *einen* Knoten
- Objekte „spezialisieren sich selbst“ mittels `if`-Anweisungen

Nachteile I

- Datenrepräsentation: Speicherverschwendung
 - Code: Implementierung der verschiedenen `evals` (und `sizes`, `copies`, ...) in einer Memberfunktion
- ⇒ Erhöhte Komplexität macht den Code umständlicher und somit fehleranfälliger

Beobachtungen II

Szenario: **Erweiterung** des Ausdrucksbaumes um mathematische Funktionen, z.B. `abs`, `sin`, `avg`:

Beobachtungen II

Szenario: **Erweiterung** des Ausdrucksbaumes um mathematische Funktionen, z.B. `abs`, `sin`, `avg`:

- **Erweiterung der Klasse `tree_node` um noch mehr Membervariablen**

```
struct tree_node {  
    char op; // new: op = 'f' -> function  
    ...  
    std::string name; // function name;  
    std::vector<tree_node*> children; // for avg  
    ...  
}
```

Beobachtungen II

Szenario: **Erweiterung** des Ausdrucksbaumes um mathematische Funktionen, z.B. `abs`, `sin`, `avg`:

Beobachtungen II

Szenario: **Erweiterung** des Ausdrucksbaumes um mathematische Funktionen, z.B. `abs`, `sin`, `avg`:

- **Anpassung jeder einzelnen Memberfunktion member function**

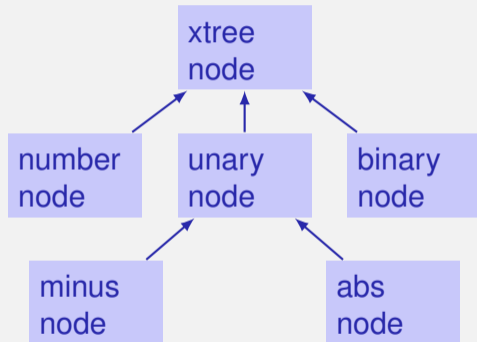
```
double eval () const
{
    ...
    else if (op == 'f')
        if (name == "abs")
            return std::abs(right->eval());
    ...
}
```

Nachteile II

- Neue Knotentypen erfordern Modifikation des bereits existierenden Codes
- Konsequenzen:
 - ⇒ Code wird noch komplexer und fehleranfälliger
 - ⇒ Quellcode muss Kunden/Nutzern vorliegen
(eigentlich: Verteilen einer Bibliothek als Headerdatei plus Objektdatei)

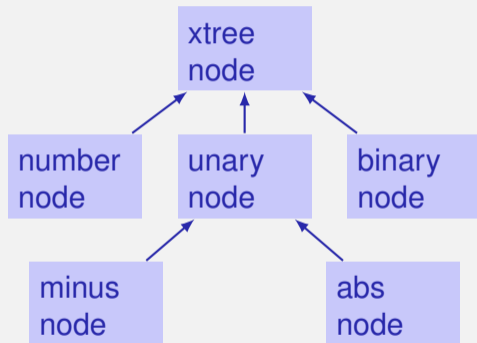
Vererbung: Die Idee

- Aufspaltung von `tree_node`



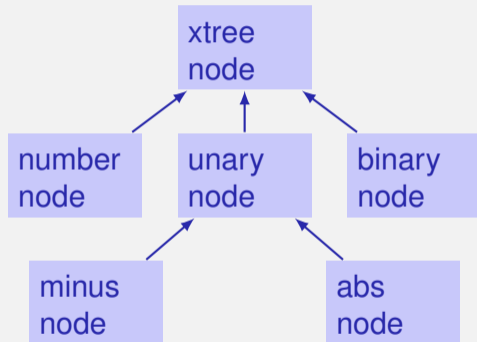
Vererbung: Die Idee

- Aufspaltung von `tree_node`
- „ist-ein“-Relation: `number_node` ist ein `xtree_node`



Vererbung: Die Idee

- Aufspaltung von `tree_node`
- „ist-ein“-Relation: `number_node` ist ein `xtree_node`
- Spezialisierung von oben nach unten: eine *abgeleitete Klasse*¹⁾ darf mehr Funktionalität anbieten als ihre *Basisklasse*²⁾

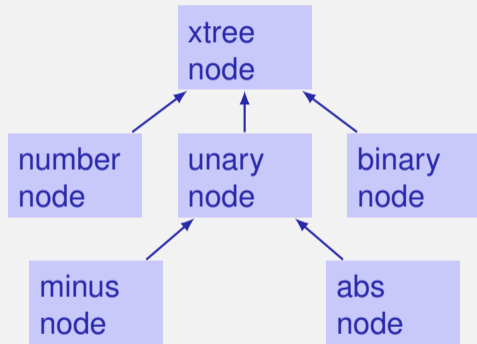


1) auch: Kind-/Unter-/Subklasse

2) auch: Eltern-/Ober-/Superklasse

Vererbung: Die Idee

- Aufspaltung von `tree_node`
- „ist-ein“-Relation: `number_node` ist ein `xtree_node`
- Spezialisierung von oben nach unten: eine *abgeleitete Klasse*¹⁾ darf mehr Funktionalität anbieten als ihre *Basisklasse*²⁾
- Gemeinsame Eigenschaften verbleiben in allgemeineren („höheren“) Klassen



1) auch: Kind-/Unter-/Subklasse

2) auch: Eltern-/Ober-/Superklasse

Vererbung: Der Code

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

Vererbung: Der Code

```
struct xtree_node {
```

```
    ...
```

```
    virtual int size() const = 0;
```

```
    virtual double eval() const = 0;
```

```
};
```

Erzwingt Impl. in Kindklassen



Vererbung: Der Code

```
struct xtree_node {
```

```
    ...
```

```
    virtual int size() const = 0;
```

```
    virtual double eval() const = 0;
```

Erzwingt Impl. in Kindklassen

```
};
```

A blue curved arrow originates from the text 'Erzwingt Impl. in Kindklassen' and points to the 'virtual double eval() const = 0;' line in the xtree_node struct. A second blue curved arrow originates from the same text and points to the 'virtual int size() const = 0;' line in the xtree_node struct. A third blue curved arrow originates from the 'virtual double eval() const = 0;' line in the xtree_node struct and points to the 'double eval () const;' line in the number_node struct, indicating that number_node inherits and implements this method.

```
struct number_node : public xtree_node {
```

```
    double val;
```

```
    int size () const;
```

```
    double eval () const;
```

```
};
```

Vererbung: Der Code

```
struct xtree_node {
```

```
    ...
```

```
    virtual int size() const = 0;
```

```
    virtual double eval() const = 0;
```

Erzwingt Impl. in Kindklassen

```
};
```

erbt von

Vererbung sichtbar

```
struct number_node : public xtree_node {
```

```
    double val; ← nur für number_node
```

```
    int size () const;
```

```
    double eval () const;
```

Mitglieder von xtree_node
werden von number_node
implementiert

```
};
```

Aufgabenteilung: Der Zahlknoten

```
struct number_node: public xtree_node {
    double val;

    number_node (double v) : val (v) {}

    double eval () const {
        return val;
    }

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


Ein Zahlknoten ist ein Baumknoten...

- Ein (Zeiger auf ein) erbendes Objekt kann überall dort verwendet werden, wo ein (Zeiger auf ein) Basisobjekt gefordert ist

```
number_node* num = new number_node (5);
```

Ein Zahlknoten ist ein Baumknoten...

- Ein (Zeiger auf ein) erbendes Objekt kann überall dort verwendet werden, wo ein (Zeiger auf ein) Basisobjekt gefordert ist

```
number_node* num = new number_node (5);  
  
xtree_node* tn = num; // ok, number_node is  
                       // just a special xtree_node
```

Ein Zahlknoten ist ein Baumknoten...

- Ein (Zeiger auf ein) erbendes Objekt kann überall dort verwendet werden, wo ein (Zeiger auf ein) Basisobjekt gefordert ist

```
number_node* num = new number_node (5);  
  
xtree_node* tn = num; // ok, number_node is  
                      // just a special xtree_node  
  
xtree_node* bn = new add_node (tn, num); // ok
```

Ein Zahlknoten ist ein Baumknoten...

- Ein (Zeiger auf ein) erbendes Objekt kann überall dort verwendet werden, wo ein (Zeiger auf ein) Basisobjekt gefordert ist, **aber nicht umgekehrt**.

```
number_node* num = new number_node (5);  
  
xtree_node* tn = num; // ok, number_node is  
                      // just a special xtree_node  
  
xtree_node* bn = new add_node (tn, num); // ok  
  
number_node* nn = tn; //error: invalid conversion
```

Polymorphie

```
struct xtree_node {  
    ...  
    virtual double eval() = 0;  
};  
...  
double number_node::eval()  
    {...}
```

```
xtree_node* n = new number_node(3);  
std::cout << n->eval(); // 3
```

Polymorphie

```
struct xtree_node {  
    ...  
    virtual double eval() = 0;  
};  
...  
double number_node::eval()  
    {...}
```

```
xtree_node* n = new number_node(3);  
std::cout << n->eval(); // 3
```

- **Virtuelle** Memberfunktionen: der *dynamische* Typ (Laufzeittyp) bestimmt die auszuführenden Memberfunktionen

Polymorphie

```
struct xtree_node {  
    ...  
    virtual double eval() = 0;  
};  
...  
double number_node::eval()  
    {...}
```

```
xtree_node* n = new number_node(3);  
std::cout << n->eval(); // 3
```

- **Virtuelle** Memberfunktionen: der *dynamische* Typ (Laufzeittyp) bestimmt die auszuführenden Memberfunktionen
- Ohne `virtual` wird der *statische Typ* zur Bestimmung der auszuführenden Funktion herangezogen

Polymorphie

```
struct xtree_node {  
    ...  
    virtual double eval() = 0;  
};  
...  
double number_node::eval()  
    {...}
```

```
xtree_node* n = new number_node(3);  
std::cout << n->eval(); // 3
```

- **Virtuelle** Memberfunktionen: der *dynamische* Typ (Laufzeittyp) bestimmt die auszuführenden Memberfunktionen
- Ohne `virtual` wird der *statische Typ* zur Bestimmung der auszuführenden Funktion herangezogen

Wir vertiefen das nicht weiter

Aufgabenteilung: Binäre Knoten

```
struct binary_node: public xtree_node {
    xtree_node* left;
    xtree_node* right;

    binary_node (xtree_node* l, xtree_node* r)
        : left (l->copy()), right (r->copy())
    {
        assert (left);
        assert (right);
    }

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

Aufgabenteilung: Binäre Knoten

```
struct binary_node: public xtree_node {
    xtree_node* left;
    xtree_node* right;

    binary_node (xtree_node* l, xtree_node* r)
        : left (l->copy()), right (r->copy())
    {
        assert (left);           size funktioniert für alle binären
        assert (right);         Knoten.   Abgeleitete Klassen
                                (add_node, sub_node...) erben
                                diese Funktion!
    }

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

Aufgabenteilung: +, -, * ...

```
struct add_node: public binary_node {  
    add_node (xtree_node* l, xtree_node* r)  
        : binary_node(l, r) {}
```

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

Aufgabenteilung: +, -, * ...

```
struct add_node: public binary_node {  
    add_node (xtree_node* l, xtree_node* r)  
        : binary_node(l, r) {}
```

Delegiert Initialisierung an
Superkonstruktor



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

eval spezifisch für +

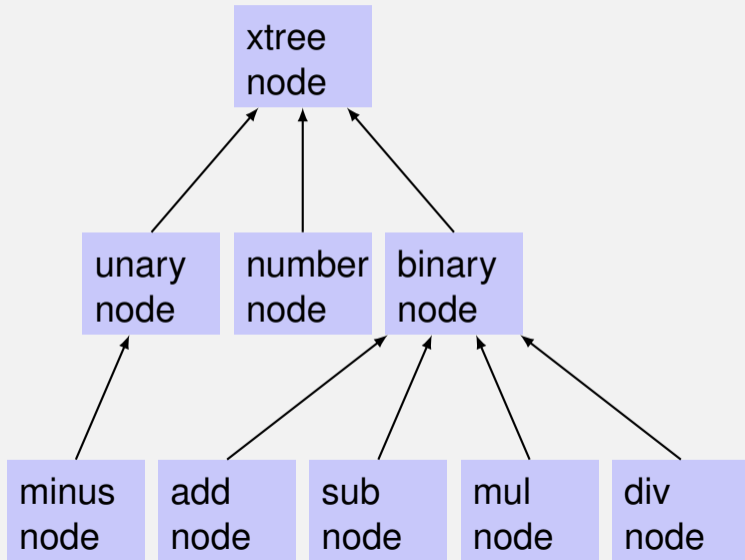


Aufgabenteilung: +, -, * ...

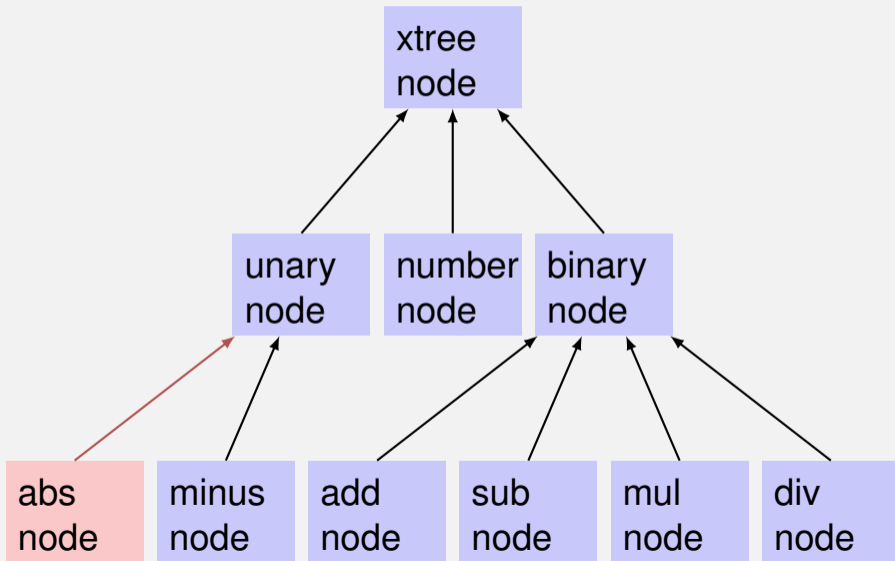
```
struct sub_node: public binary_node {
    sub_node (xtree_node* l, xtree_node* r)
        : binary_node (l, r) {}

    double eval () const {
        return left->eval() - right->eval();
    }
};
```

Erweiterung um abs Funktion



Erweiterung um abs Funktion



Erweiterung um abs Funktion

```
struct unary_node: public xtree_node {
    xtree_node* right;

    unary_node (xtree_node* r): right(r->copy()) {
        assert(right);
    }
    int size () const { return 1 + right->size(); }
};
```

```
struct abs_node: public unary_node {
    abs_node (xtree_node* arg): unary_node (arg) {}

    double eval () const {
        return std::abs (right->eval());
    }
};
```


Erweiterung um abs Funktion

```
struct unary_node: public xtree_node {
    xtree_node* right;

    unary_node (xtree_node* r): right(r->copy()) {
        assert(right);
    }
    int size () const { return 1 + right->size(); }
};
```

```
struct abs_node: public unary_node {
    abs_node (xtree_node* arg): unary_node (arg) {}

    double eval () const {
        return std::abs (right->eval());
    }
};
```

Mission: Monolithisch \rightarrow modular ✓

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

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

```
int size () const { ... }
```

```
void clear() { ... }
```

```
tree_node* copy () const { ... }
```

```
};
```

```
struct number_node: public xtree_node {  
    double val;  
    ...  
    double eval () const {  
        return val;  
    }  
};
```

```
struct minus_node: public unary_node {  
    ...  
    double eval () const {  
        return -right->eval();  
    }  
};
```

```
struct minus_node : public binary_node {  
    ...  
    double eval () const {  
        return left->eval() - right->eval();  
    }  
};
```

```
struct abs_node : public unary_node {  
    ...  
    double eval () const {  
        return left->eval() - right->eval();  
    }  
};
```



Zusammenfassung der Konzepte

.. der objektorientierten Programmierung

Kapselung (vorherigen beiden Vorlesungen)

- Verbergen der Implementierungsdetails von Typen (privater Bereich)
- Definition einer Schnittstelle zum Zugriff auf Werte und Funktionalität (öffentlicher Bereich)
- Ermöglicht das Sicherstellen von Invarianten und den Austausch der Implementierung

Zusammenfassung der Konzepte

.. der objektorientierten Programmierung

Vererbung

- Typen können Eigenschaften von Typen erben
- Abgeleitete Typen können neue Eigenschaften besitzen oder vorhandene überschreiben
- Macht Code- und Datenwiederverwendung möglich

Zusammenfassung der Konzepte

.. der objektorientierten Programmierung

Polymorphie

- Ein Zeiger (oder eine Referenz) kann abhängig von seiner Verwendung unterschiedliche zugrundeliegende Typen haben
- Die unterschiedlichen Typen können bei gleichem Zugriff auf ihre gemeinsame Schnittstelle verschieden reagieren
- Macht „nicht invasive“ Erweiterung von Bibliotheken möglich