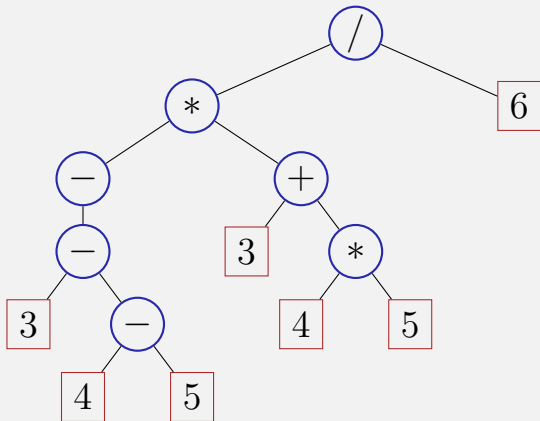


18. Vererbung und Polymorphie

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

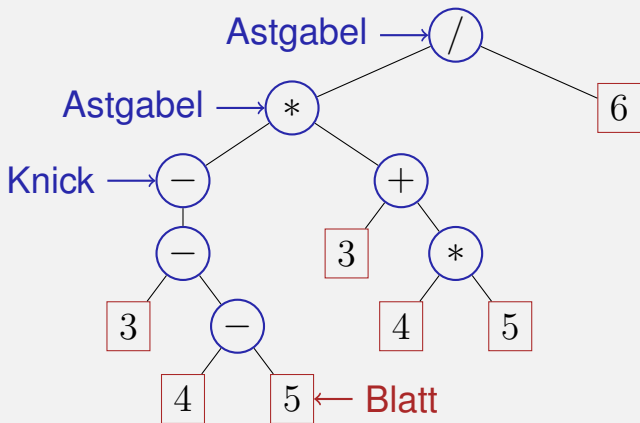
(Ausdrucks-)Bäume

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



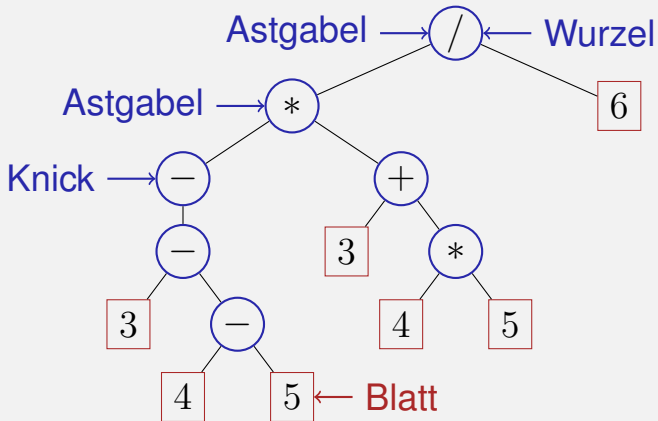
(Ausdrucks-)Bäume

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

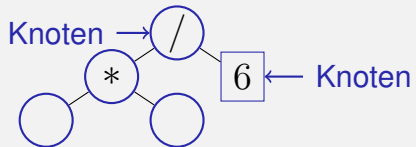


(Ausdrucks-)Bäume

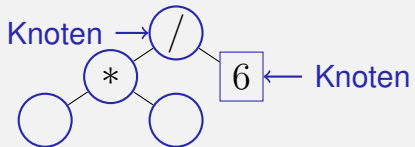
$$-(3-(4-5))*(3+4*5)/6$$



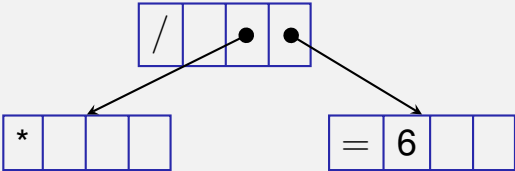
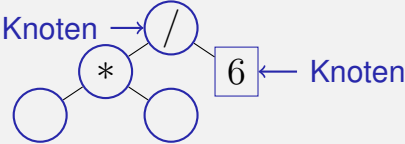
Astgabeln + Blätter + Knicke = Knoten



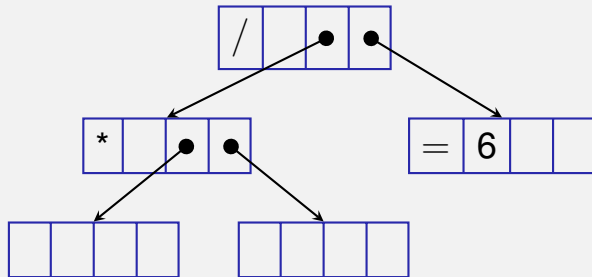
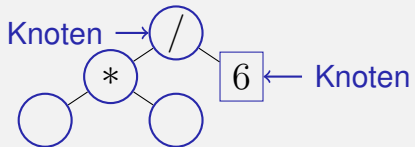
Astgabeln + Blätter + Knicke = Knoten



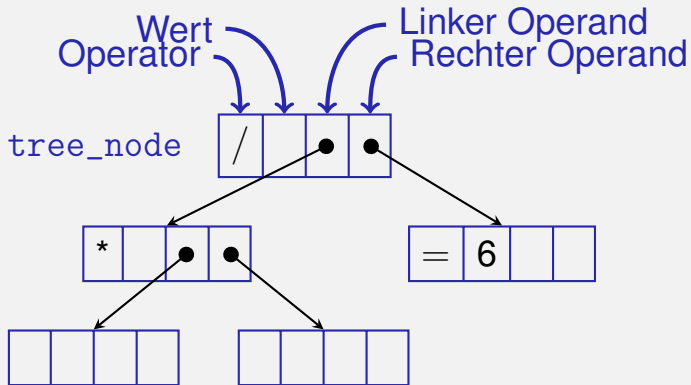
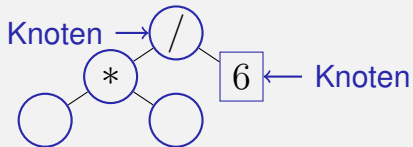
Astgabeln + Blätter + Knicke = Knoten



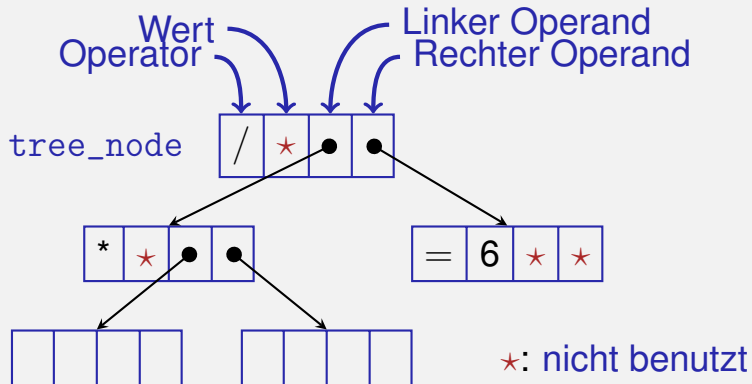
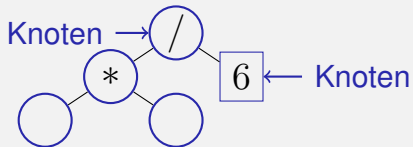
Astgabeln + Blätter + Knicke = Knoten



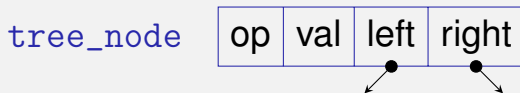
Astgabeln + Blätter + Knicke = Knoten



Astgabeln + Blätter + Knicke = Knoten

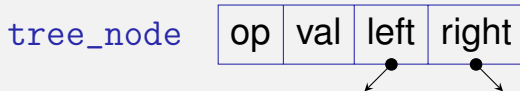


Knoten (struct tree_node)



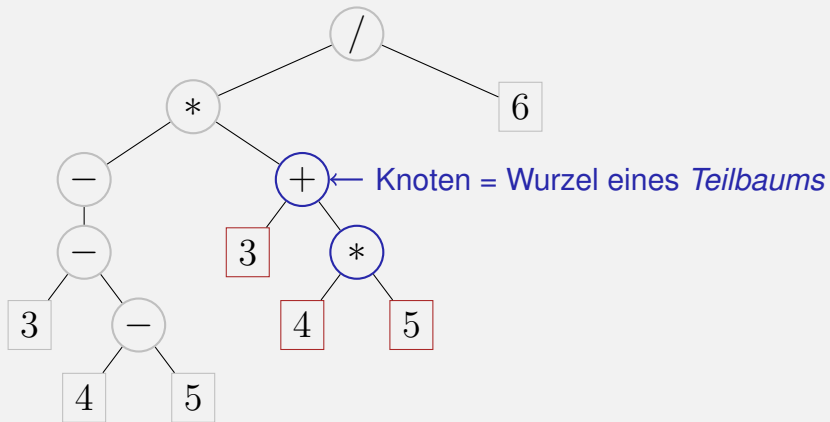
```
struct tree_node {
    char op;
    // leaf node (op: '=')
    double val;
    // internal node ( op: '+', '-', '*', '/')
    tree_node* left;
    tree_node* right;
    // constructor
    tree_node (char o, double v, tree_node* l, tree_node* r)
        : op (o), val (v), left (l), right (r)
    {}
};
```

Knoten (struct tree_node)

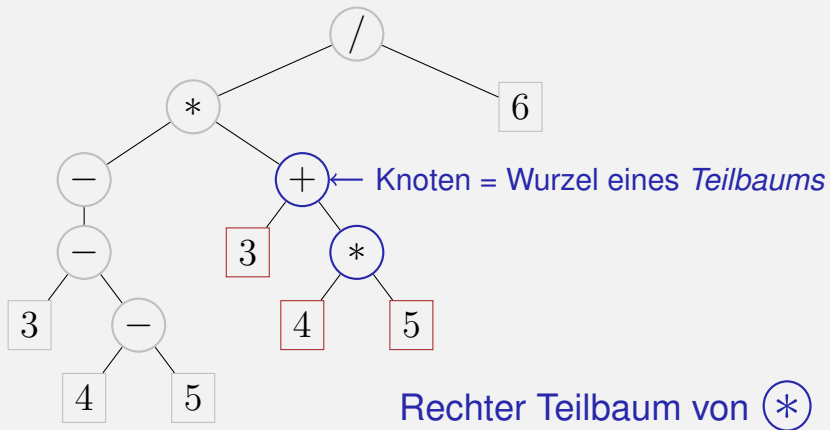


```
struct tree_node {
    char op;
    // leaf node (op: '=')
    double val;
    // internal node ( op: '+', '-', '*', '/')
    tree_node* left; // == 0 für unäres minus
    tree_node* right;
    // constructor
    tree_node (char o, double v, tree_node* l, tree_node* r)
        : op (o), val (v), left (l), right (r)
    {}
};
```

Knoten und Teilbäume



Knoten und Teilbäume



Knoten in Teilbäumen zählen

```
struct tree_node {
```



```
...
```

```
// POST: returns the size (number of nodes) of  
//       the subtree with root *this
```

```
int size () const
```

```
{
```

```
    int s=1;
```

```
    if (left)
```

```
        s += left->size();
```

```
    if (right)
```

```
        s += right->size();
```

```
    return s;
```

```
}
```

```
};
```



Knoten in Teilbäumen zählen

```
struct tree_node {
```



```
...
```

```
// POST: returns the size (number of nodes) of  
// the subtree with root *this
```

```
int size () const
```

```
{
```

```
    int s=1;
```

```
    if (left) // kurz für left != 0
```

```
        s += left->size();
```

```
    if (right)
```

```
        s += right->size();
```

```
    return s;
```

```
}
```

```
};
```




Teilbäume auswerten

```
struct tree_node {  
    ...  
    // POST: evaluates the subtree with root *this  
    double 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;  
    }  
};
```



Teilbäume klonen

```
struct tree_node {  
    ...  
    // POST: a copy of the subtree with root *this is  
    //         made, and a pointer to its root node is  
    //         returned  
    tree_node* copy () const {  
        tree_node* to = new tree_node (op, val, 0, 0);  
        if (left)  
            to->left = left->copy();  
        if (right)  
            to->right = right->copy();  
        return to;  
    }  
};
```



The diagram shows a horizontal rectangle divided into four equal-width cells. From left to right, the cells are labeled 'op', 'val', 'left', and 'right'. Below the 'left' and 'right' cells, there is a small grey circle. From each of these circles, a thin grey arrow points downwards and slightly outwards, representing pointers to the left and right child nodes.


Teilbäume klonen - Kompaktere Schreibweise

```
struct tree_node {  
    ...  
    // POST: a copy of the subtree with root *this is  
    //         made, and a pointer to its root node is  
    //         returned  
    tree_node* copy () const {  
        return new tree_node (op, val,  
                               left ? left->copy() : 0,  
                               right ? right->copy() : 0);  
    }  
};
```



Teilbäume klonen - Kompaktere Schreibweise

```
struct tree_node {  
    ...  
    // POST: a copy of the subtree with root *this is  
    //         made, and a pointer to its root node is  
    //         returned  
    tree_node* copy () const {  
        return new tree_node (op, val,  
                                left ? left->copy() : 0,  
                                right ? right->copy() : 0);  
    }  
};
```

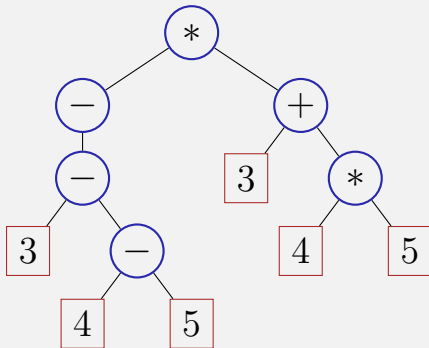


The diagram shows a horizontal box divided into four sections labeled 'op', 'val', 'left', and 'right'. Below the 'left' and 'right' sections, there are small grey circles with arrows pointing downwards and outwards, representing pointers to child nodes.

cond ? expr1 : expr2 hat Wert *expr1*, falls *cond* gilt, *expr2* sonst

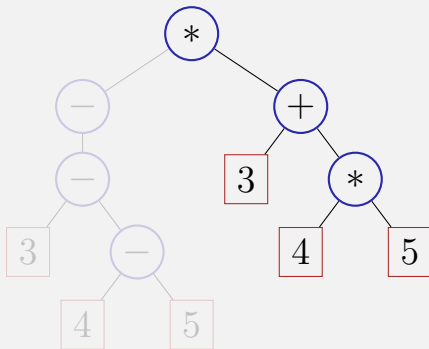
Teilbäume fällen

```
struct tree_node {  
    ...  
    // POST: all nodes in the subtree with root  
    // *this are deleted  
    void clear() {  
        if (left) {  
            left->clear();  
        }  
        if (right) {  
            right->clear();  
        }  
        delete this;  
    }  
};
```



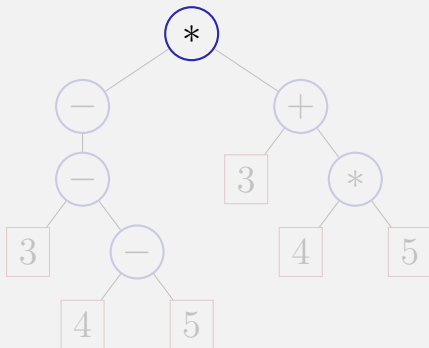
Teilbäume fällen

```
struct tree_node {  
    ...  
    // POST: all nodes in the subtree with root  
    // *this are deleted  
    void clear() {  
        if (left) {  
            left->clear();  
        }  
        if (right) {  
            right->clear();  
        }  
        delete this;  
    }  
};
```



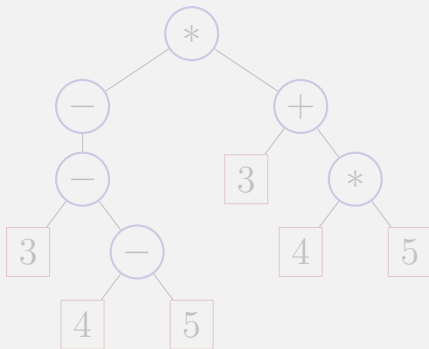
Teilbäume fällen

```
struct tree_node {  
    ...  
    // POST: all nodes in the subtree with root  
    // *this are deleted  
    void clear() {  
        if (left) {  
            left->clear();  
        }  
        if (right) {  
            right->clear();  
        }  
        delete this;  
    }  
};
```



Teilbäume fällen

```
struct tree_node {  
    ...  
    // POST: all nodes in the subtree with root  
    // *this are deleted  
    void clear() {  
        if (left) {  
            left->clear();  
        }  
        if (right) {  
            right->clear();  
        }  
        delete this;  
    }  
};
```



Bäumige Teilbäume

```
struct tree_node {  
    ...  
    // constructor  
    tree_node (char o, tree_node* l,  
               tree_node* r, double v)  
  
    // functionality  
    double eval () const;  
    void print (std::ostream& o) const;  
    int size () const;  
    tree_node* copy () const;  
    void clear ();  
};
```

Bäume pflanzen

```
class texpression {  
private:  
    tree_node* root;  
public:  
    ...  
    texpression (double d) ← erzeugt Baum mit  
        : root (new tree_node ('=', d, 0, 0)) {}  
    ...  
};
```

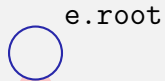
erzeugt Baum mit
einem Blatt

Bäume wachsen lassen

```
texpression& operator-= (const texpression& e)
{
    assert (e.root);
    root = new tree_node ('-', 0, root, e.root->copy());
    return *this;
}
```



*this



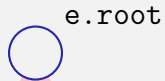
e

Bäume wachsen lassen

```
texpression& operator-= (const texpression& e)
{
    assert (e.root);
    root = new tree_node ('-', 0, root, e.root->copy());
    return *this;
}
```



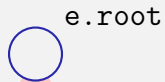
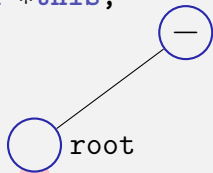
*this



e

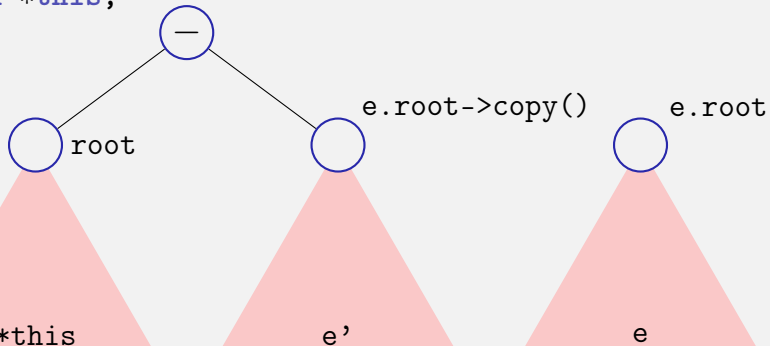
Bäume wachsen lassen

```
texpression& operator-= (const texpression& e)
{
    assert (e.root);
    root = new tree_node ('-', 0, root, e.root->copy());
    return *this;
}
```



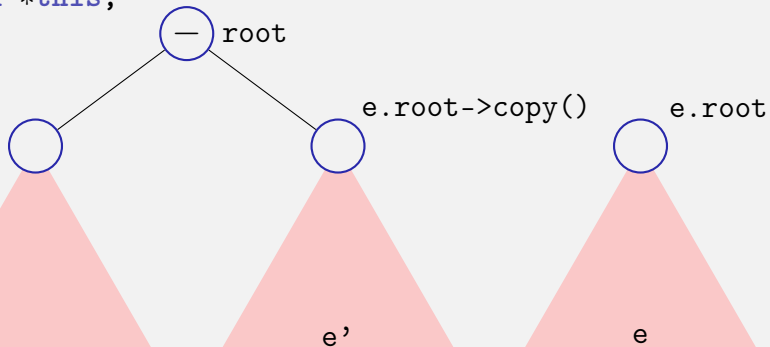
Bäume wachsen lassen

```
texpression& operator-= (const texpression& e)
{
    assert (e.root);
    root = new tree_node ('-', 0, root, e.root->copy());
    return *this;
}
```



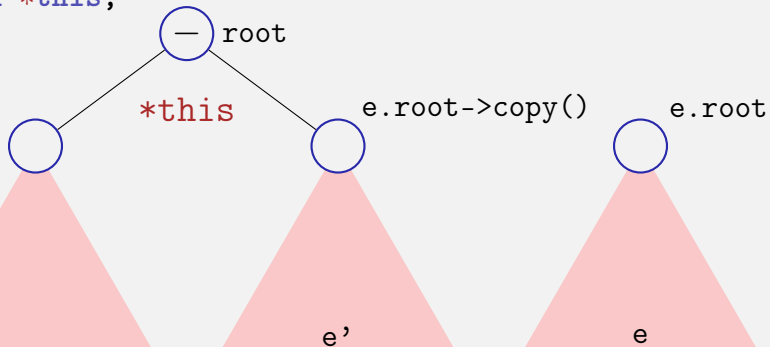
Bäume wachsen lassen

```
texpression& operator-= (const texpression& e)
{
    assert (e.root);
    root = new tree_node ('-', 0, root, e.root->copy());
    return *this;
}
```



Bäume wachsen lassen

```
texpression& operator-= (const texpression& e)
{
    assert (e.root);
    root = new tree_node ('-', 0, root, e.root->copy());
    return *this;
}
```



Bäume züchten

```
texpression operator- (const texpression& l,  
                      const texpression& r)  
{  
    texpression result = l;  
    return result -= r;  
}
```

```
texpression a = 3;  
texpression b = 4;  
texpression c = 5;  
texpression d = a-b-c;
```

Bäume züchten

```
texpression operator- (const texpression& l,  
                      const texpression& r)  
{  
    texpression result = l;  
    return result -= r;  
}
```

```
texpression a = 3;  
texpression b = 4;  
texpression c = 5;  
texpression d = a-b-c;
```

3

Bäume züchten

```
texpression operator- (const texpression& l,  
                      const texpression& r)  
{  
    texpression result = l;  
    return result -= r;  
}
```

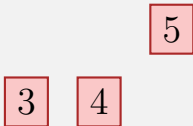
```
texpression a = 3;  
texpression b = 4;  
texpression c = 5;  
texpression d = a-b-c;
```

3 4

Bäume züchten

```
texpression operator- (const texpression& l,  
                      const texpression& r)  
{  
    texpression result = l;  
    return result -= r;  
}
```

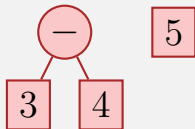
```
texpression a = 3;  
texpression b = 4;  
texpression c = 5;  
texpression d = a-b-c;
```



Bäume züchten

```
texpression operator- (const texpression& l,  
                      const texpression& r)  
{  
    texpression result = l;  
    return result -= r;  
}
```

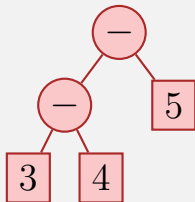
```
texpression a = 3;  
texpression b = 4;  
texpression c = 5;  
texpression d = a-b-c;
```



Bäume züchten

```
texpression operator- (const texpression& l,  
                      const texpression& r)  
{  
    texpression result = l;  
    return result -= r;  
}
```

```
texpression a = 3;  
texpression b = 4;  
texpression c = 5;  
texpression d = a-b-c;
```



Von Werten zu Bäumen!

```
// term = factor { "*" factor | "/" factor }
double term (std::istream& is){
{
    double value = factor (is);
    while (true) {
        if (consume (is, '*'))
            value *= factor (is);
        else if (consume (is, '/'))
            value /= factor (is);
        else
            return value;
    }
}
```

calculator.cpp
(Ausdruckswert)

Von Werten zu Bäumen!

```
typedef double result_type; // Typ-Alias
```

```
// term = factor { "*" factor | "/" factor }
```

```
result_type term (std::istream& is){  
{  
    result_type value = factor (is);  
    while (true) {  
        if (consume (is, '*'))  
            value *= factor (is);  
        else if (consume (is, '/'))  
            value /= factor (is);  
        else  
            return value;  
    }  
}
```

double_calculator.cpp
(Ausdruckswert)

Von Werten zu Bäumen!

```
typedef texpression result_type; // Typ-Alias
```

```
// term = factor { "*" factor | "/" factor }
```

```
result_type term (std::istream& is){  
{  
    result_type value = factor (is);  
    while (true) {  
        if (consume (is, '*'))  
            value *= factor (is);  
        else if (consume (is, '/'))  
            value /= factor (is);  
        else  
            return value;  
    }  
}
```

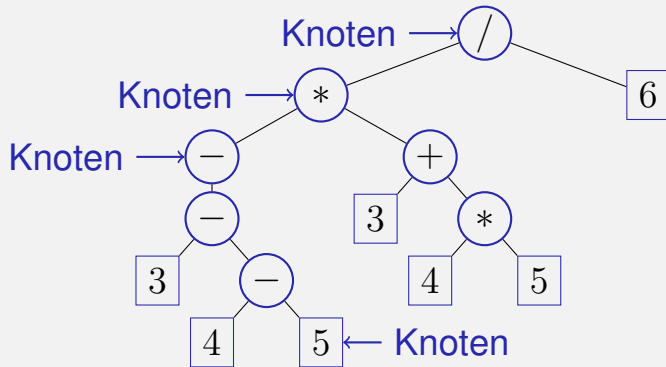
double_calculator.cpp

(Ausdruckswert)

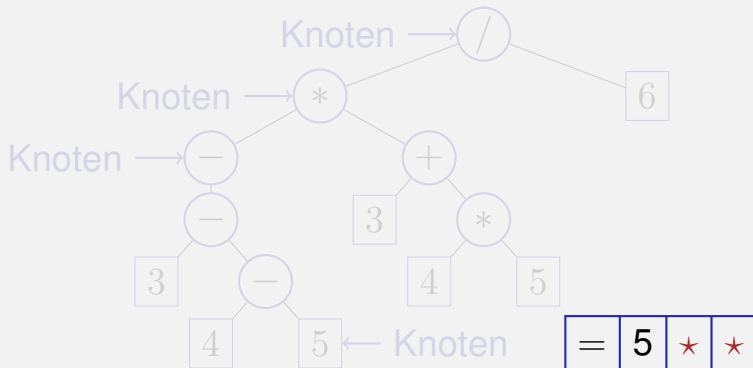
→

texpression_calculator_1.cpp

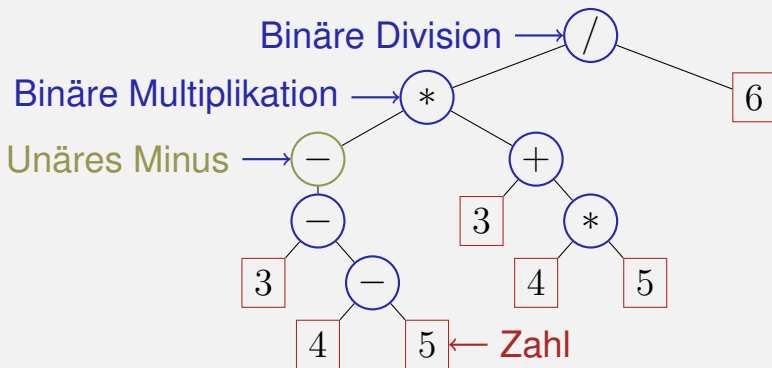
(Ausdrucksbaum)



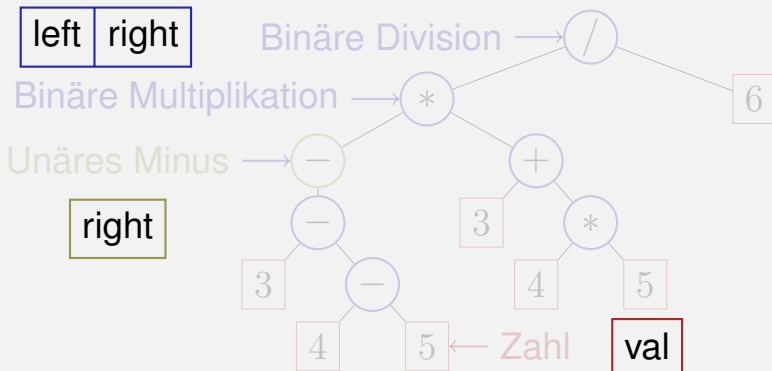
- Astgabeln + Blätter + Knicke = Knoten



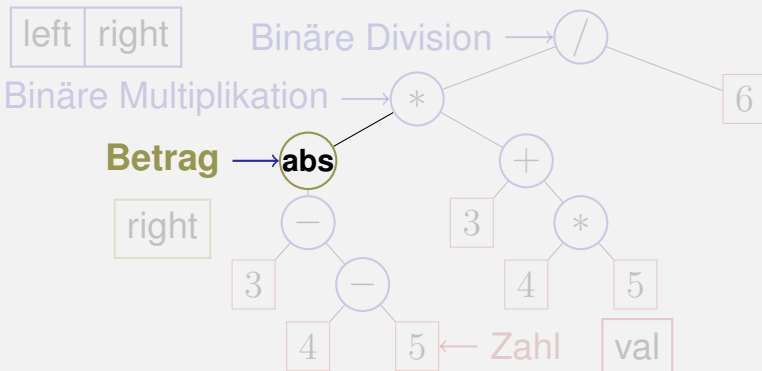
- Astgabeln + Blätter + Knicke = Knoten
- \Rightarrow Unbenutzte Membervariablen *



- “Zoo” von Astgabeln, Blättern und Knicken



- “Zoo” von Astgabeln, Blättern und Knicken
- Überall nur die benötigten Membervariablen!



- “Zoo” von Astgabeln, Blättern und Knicken
- Zoo-Erweiterung mit neuen Arten!

Vererbung – Der Hack, zum ersten...

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

Vererbung – Der Hack, zum ersten...

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

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

```
struct tree_node{
    char op; // neu: op = 'f' -> Funktion
    ...
    std::string name; // function name;
}
```


Vererbung – Der Hack, zum zweiten...

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

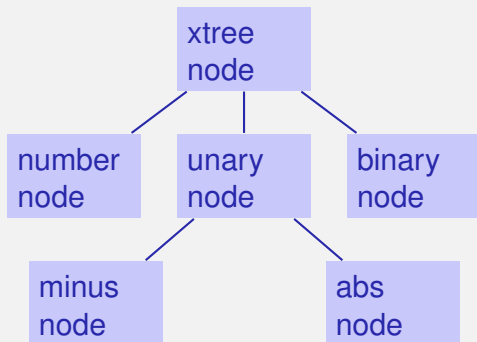
Vererbung – Der Hack, zum zweiten...

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

- **Anpassung jeder einzelnen Memberfunktion member function**

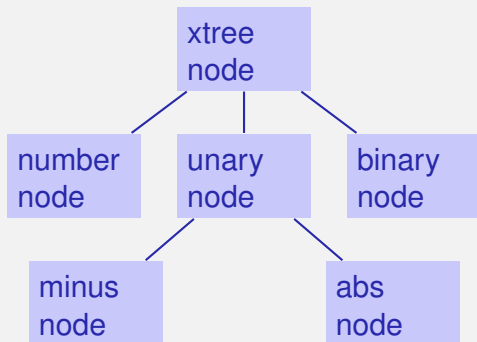
```
double eval () const
{
    ...
    else if (op == 'f')
        if (name == "abs")
            return std::abs(right->eval());
    ...
}
```

Vererbung – die saubere Lösung



- „Aufspaltung” von `tree_node`

Vererbung – die saubere Lösung



- „Aufspaltung“ von `tree_node`
- Gemeinsame Eigenschaften verbleiben in der *Basisklasse* `xtree_node` (Erklärung folgt)

Vererbung

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

Vererbung

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

```
struct number_node : public xtree_node {  
    double val;  
  
    int size () const;  
    double eval () const;  
};
```

Vererbung

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

erbt von

Vererbung sichtbar

```
struct number_node : public xtree_node {  
    double val; ← nur für number_node
```

```
    int size () const; ← Mitglieder von xtree_node  
    double eval () const; ← werden überschrieben
```

```
};
```

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
```

Anwendung

```
class xexpression {
private:
    xtree_node* root;
public:
    xexpression (double d)
        : root (new number_node (d)) {}

    xexpression& operator-= (const xexpression& t)
    {
        assert (t.root);
        root = new sub_node (root, t.root->copy());
        return *this;
    }
    ...
}
```

Anwendung

```
class xexpression {  
private:                                     statischer Typ  
    xtree_node* root; ←  
public:                                     dynamischer Typ  
    xexpression (double d) ←  
        : root (new number_node (d)) {}  
  
    xexpression& operator-= (const xexpression& t)  
    {  
        assert (t.root);  
        root = new sub_node (root, t.root->copy());  
        return *this;  
    }  
    ...  
}
```

Polymorphie

- **Virtuelle** Mitgliedsfunktion: der *dynamische* Typ bestimmt bei Zeigern auf erbende Objekte die auszuführenden Memberfunktionen

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

- Ohne `virtual` wird der *statische Typ* zur Bestimmung der auszuführenden Funktion herangezogen.

Wir vertiefen das nicht weiter.

Polymorphie

- **Virtuelle** Mitgliedsfunktion: der *dynamische* Typ bestimmt bei Zeigern auf erbende Objekte die auszuführenden Memberfunktionen

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

- Ohne `virtual` wird der *statische Typ* zur Bestimmung der auszuführenden Funktion herangezogen.

Wir vertiefen das nicht weiter.

Polymorphie

- **Virtuelle** Mitgliedsfunktion: der *dynamische* Typ bestimmt bei Zeigern auf erbende Objekte die auszuführenden Memberfunktionen

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

- Ohne `virtual` wird der *statische Typ* zur Bestimmung der auszuführenden Funktion herangezogen.

Wir vertiefen das nicht weiter.

Polymorphie

- **Virtuelle** Mitgliedsfunktion: der *dynamische* Typ bestimmt bei Zeigern auf erbende Objekte die auszuführenden Memberfunktionen

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

- 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; // INV != 0
    xtree_node* right; // INV != 0

    binary_node (xtree_node* l, xtree_node* r) :
        left (l), right (r)
    {
        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; // INV != 0
    xtree_node* right; // INV != 0

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

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

size funktioniert für alle binären Knoten. Abgeleitete Klassen (add_node, sub_node...) erben diese Funktion!



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

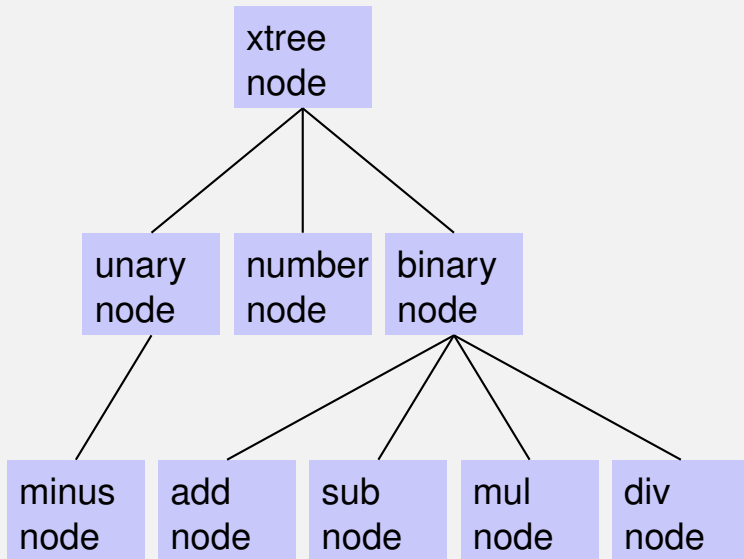
    double eval () const {
        return left->eval() - right->eval();
    }
};
```

Aufgabenteilung: +, -, * ...

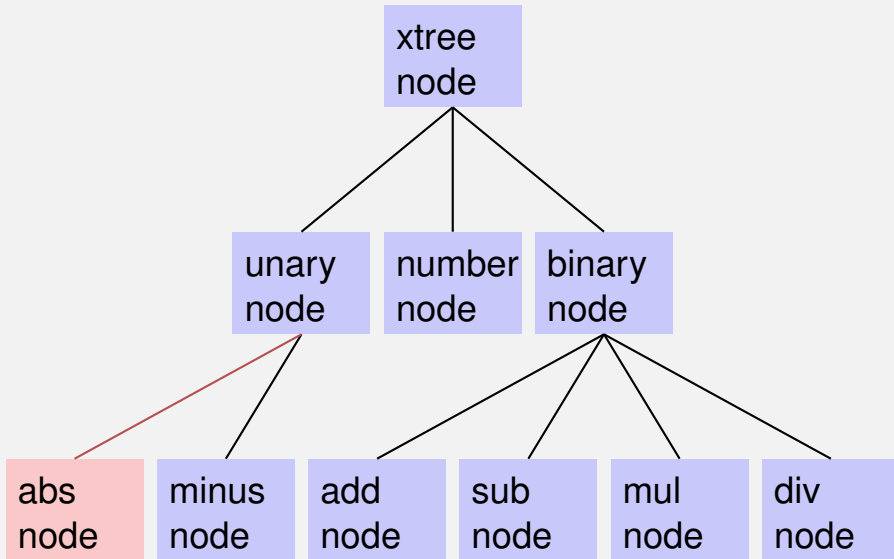
```
struct sub_node : public binary_node {  
    sub_node (xtree_node* l, xtree_node* r)  
        : binary_node (l, r) {}  
  
    double eval () const {  
        return left->eval() - right->eval();  
    }  
};
```

←
eval spezifisch
für +, -, *, /

Erweiterung um abs Funktion



Erweiterung um abs Funktion



Erweiterung um abs Funktion

```
struct unary_node: public xtree_node
{
    xtree_node* right; // INV != 0
    unary_node (xtree_node* r);
    int size () const;
};
```

```
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; // INV != 0
    unary_node (xtree_node* r);
    int size () const;
};

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

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

```
struct xtree_node {
    ...
    // POST: a copy of the subtree with root
    //        *this is made, and a pointer to
    //        its root node is returned
    virtual xtree_node* copy () const;

    // POST: all nodes in the subtree with
    //        root *this are deleted
    virtual void clear () {};
};
```

```
struct unary_node: public xtree_node {
    ...
    virtual void clear () {
        right->clear();
        delete this;
    }
};

struct minus_node: public unary_node {
    ...
    xtree_node* copy () const
    {
        return new minus_node (right->copy());
    }
};
```

`xtree_node` ist kein dynamischer Datentyp ??

- Wir haben keine Variablen vom Typ `xtree_node` mit automatischer Speicherdauer

`xtree_node` ist kein dynamischer Datentyp ??

- Wir haben keine Variablen vom Typ `xtree_node` mit automatischer Speicherdauer
- Copy-Konstruktor, Zuweisungsoperator und Destruktor sind überflüssig

xtree_node ist kein dynamischer Datentyp ??

- Wir haben keine Variablen vom Typ `xtree_node` mit automatischer Speicherdauer
- Copy-Konstruktor, Zuweisungsoperator und Destruktor sind überflüssig
- Speicherverwaltung in der *Container-Klasse*

```
class xexpression {  
    // Copy-Konstruktor  
    xexpression (const xexpression& v);  
    // Zuweisungsoperator  
    xexpression& operator=(const xexpression& v);  
    // Destruktor  
    ~xexpression ();  
};
```


xtree_node ist kein dynamischer Datentyp ??

- Wir haben keine Variablen vom Typ `xtree_node` mit automatischer Speicherdauer
- Copy-Konstruktor, Zuweisungsoperator und Destruktor sind überflüssig
- Speicherverwaltung in der *Container-Klasse*

```
class xexpression {  
    // Copy-Konstruktor  
    xexpression (const xexpression& v);  
    // Zuweisungsoperator  
    xexpression& operator=(const xexpression& v);  
    // Destruktor  
    ~xexpression ();  
};
```

`xtree_node::copy`

`xtree_node::clear`

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

- 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 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.