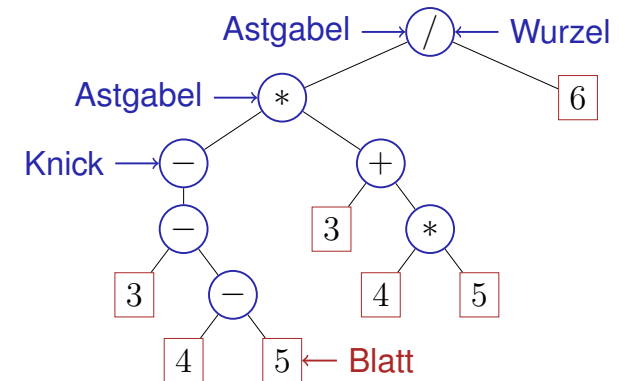


19. Vererbung und Polymorphie

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

(Ausdrucks-)Bäume

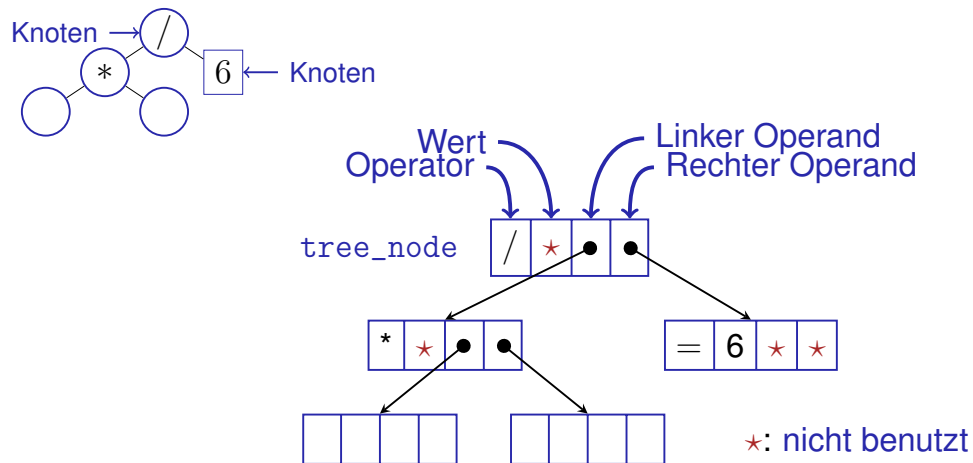
$$-(3-(4-5))*(3+4*5)/6$$



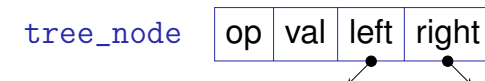
683

684

Astgabeln + Blätter + Knicke = Knoten



Knoten (struct tree_node)

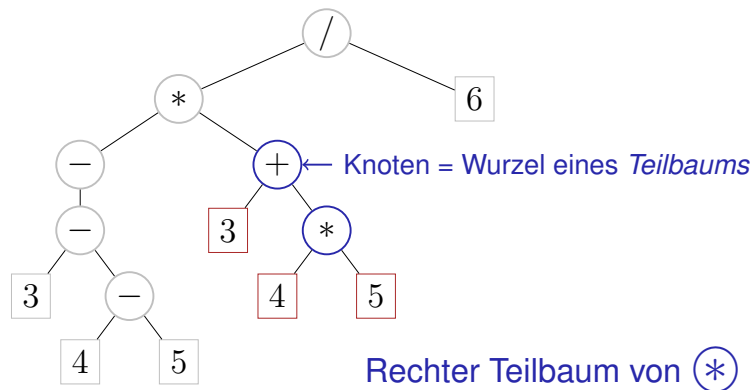


```
struct tree_node {
    char op;
    // leaf node (op: '=')
    double val;
    // internal node ( op: '+', '-', '*', '/')
    tree_node* left; // == 0 für unäres minus
    tree_node* right;
    // constructor
    tree_node (char o, double v, tree_node* l, tree_node* r)
        : op (o), val (v),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

685

686

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) // kurz für left != 0
            s += left->size();
        if (right)
            s += right->size();
        return s;
    }
};
```



687

688

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;
    }
};
```

689

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;
    }
};
```

690

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);
    }
};
```

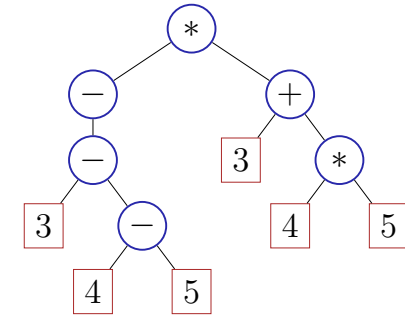


cond ? expr1 : expr2 hat Wert *expr1*, falls *cond* gilt, *expr2* sonst

691

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;
    }
};
```



692

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 ();
};
```

693

Bäume pflanzen

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

erzeugt Baum mit
einem Blatt

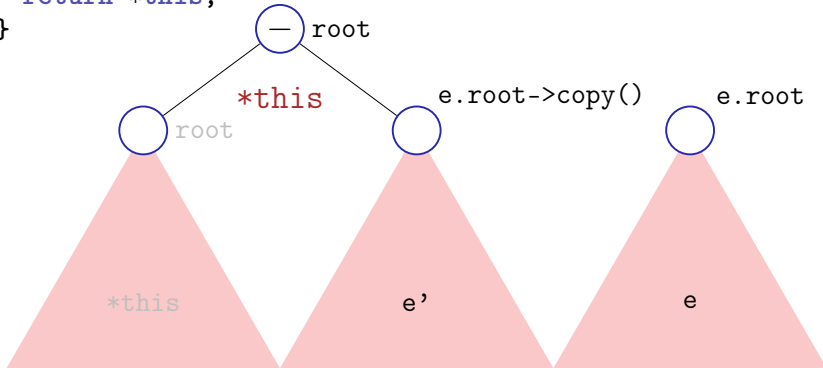
694

Bäume wachsen lassen

```

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

```



695

Bäume züchten

```

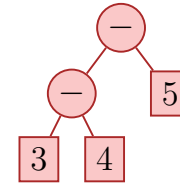
texpression operator- (const texpression& l,
                      const texpression& r)
{
    texpression result = l;
    return result -= r;
}

```

```

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

```



696

Bäume züchten

Es gibt für texpression auch noch

- Default-Konstruktor, Copy-Konstruktor, Assignment-Operator, Destruktor
- die arithematischen Zuweisungen +=, *=, /=
- die binären Operatoren +, *, /
- das unäre-

697

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;
    }
}
}

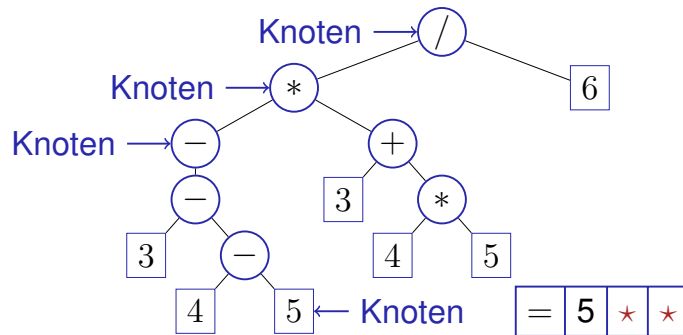
```

double_calculator.cpp
(Ausdruckswert)
→
texpression_calculator_1.cpp
(Ausdrucksbaum)

698

Motivation Vererbung:

Bisher

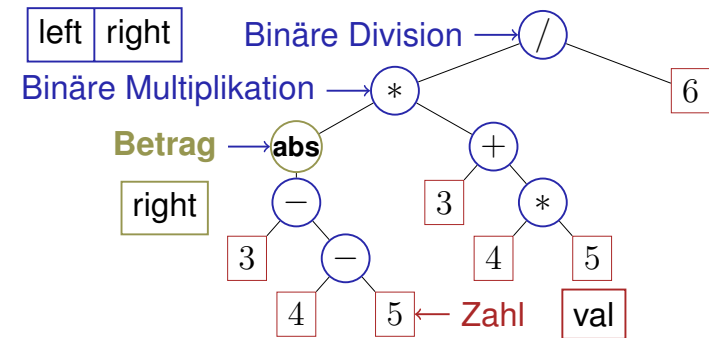


- Astgabeln + Blätter + Knicke = Knoten
- $\Rightarrow$ Unbenutzte Membervariablen *

699

Motivation Vererbung:

Die Idee



- Überall nur die benötigten Membervariablen!
- Zoo-Erweiterung mit neuen Arten!

700

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;
}
```

Nachteile:

- Veränderung des Originalcodes (unerwünscht)
- Noch mehr unbenutzte Membervariablen...

701

Vererbung – Der Hack, zum zweiten...

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

- Anpassung jeder einzelnen Memberfunktion member function

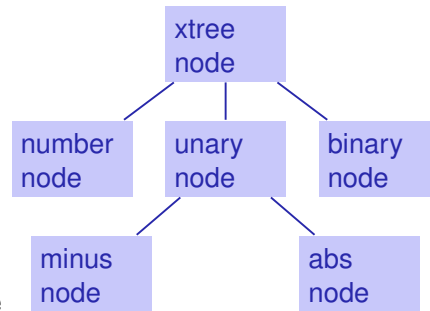
```
double eval () const
{
    ...
    else if (op == 'f')
        if (name == "abs")
            return std::abs(right->eval());
    ...
}
```

Nachteile:

- Verlust der Übersichtlichkeit
- Zusammenarbeit mehrerer Entwickler schwierig

702

Vererbung – die saubere Lösung



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

Vererbung

Klassen können Eigenschaften (*ver*)erben:

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

erbt von

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

Vererbung sichtbar

nur für `number_node`

Mitglieder von `xtree_node` werden überschrieben

703

704

Vererbung – Nomenklatur

```
class A {
    ...
}

class B: public A{
    ...
}

class C: public B{
    ...
}
```

Basisklasse
(Superklasse)

Abgeleitete Klasse
(Subklasse)

„B und C erben von A“
„C erbt von B“

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;
    }
};
```

705

706

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
```

707

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;
    }
    ...
}
```

statischer Typ

dynamischer Typ

708

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.

709

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!

710

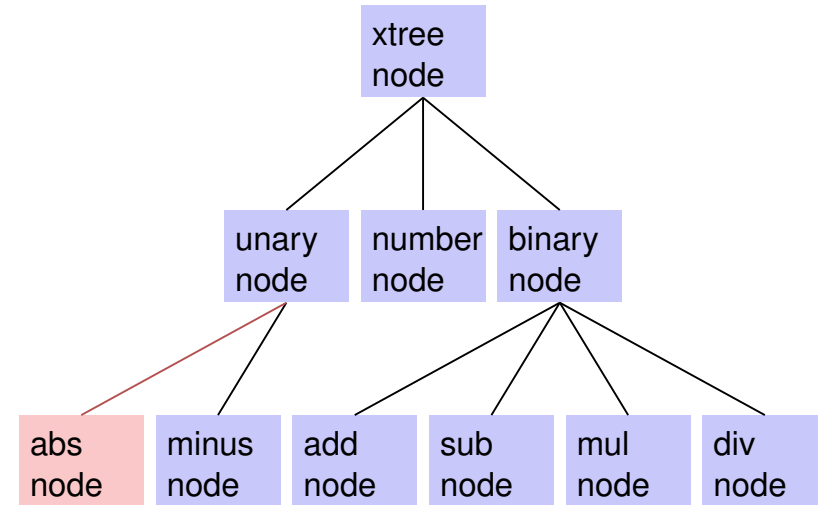
Aufgabenteilung: +, -, * ...

```
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



711

712

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());
    }
};
```

713

Da ist noch was...

Speicherbehandlung

```
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 () {};
};
```

714

Da ist noch was...

Speicherbehandlung

```
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
- 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 ();
};
```

Diagramm: Pfeile zeigen von `xtree_node::copy` zum Copy-Konstruktor und von `xtree_node::clear` zum Destruktor.

715

716

Mission: Monolithisch → 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 { ... }
};
```

Diagramm: Ein Pluszeichen (+) steht zwischen dem monolithischen Code und einer Box, die die modulare Struktur zeigt:

```
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();
    }
};
```

717

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

718

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.

719

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.

720