19. Inheritance and Polymorphism

Expression Trees, Inheritance, Code-Reuse, Virtual Functions, Polymorphism, Concepts of Object Oriented Programming

Nodes: Forks, Bends or Leaves

(Expression) Trees

\[-(3-(4-5))*(3+4*5)/6\]

Nodes (struct tree_node)

```
struct tree_node {
    char op;
    // leaf node (op: '=')
    double val;
    // internal node (op: '+', '-', '*', '/')
    tree_node* left; // == 0 for unary 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)
    {}
};
```
### Nodes and Subtrees

Node = root of a subtree

Right subtree of *

### Count Nodes in Subtrees

```cpp
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;  
    }  
};
```

### Evaluate Subtrees

```cpp
struct tree_node {  
    // POST: evaluates the subtree with root *this
    double eval () const  
    {  
        if (op == '=') return val;  
        double l = 0;  
        if (left) 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;  
        return 0;  
    }  
};
```

### Cloning Subtrees

```cpp
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;  
    }  
};
```
Cloning Subtrees – more Compact Notation

```cpp
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 : expr 2 has value expr1, if cond holds, expr2 otherwise
```

Felling Subtrees

```cpp
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;
    }
};
```

```
```

Powerful Subtrees!

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

Planting Trees

```cpp
class texpression {
    private:
        tree_node∗ root;
    public:
        //... 
        texpression (double d)
            : root (new tree_node (′=′, d, 0, 0)) {}
        //... 
};
```

```
creates a tree with one leaf
```
Letting Trees Grow

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

Raising Trees

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;

For texpression we also provide
■ default constructor, copy constructor, assignment operator, destructor
■ arithmetic assignments +=, *=, /=
■ binary operators +, *, /
■ the unary−

From Values to Trees!
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 (expression value)
→
texpression_calculator_l.cpp (expression tree)
Motivation Inheritance: Previously

Nodes: Forks, Leafs and Bends

⇒ unused member variables

Motivation Inheritance: The Idea

Everywhere only the necessary member variables

Extension of “operator zoo” with new species!

Inheritance – The Hack, First...

Scenario: extension of the expression tree by mathematical functions abs, sin, cos:

- extension of the class tree_node by even more member variables

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

Disadvantages:
- Modification of the original code (undesirable)
- even more member variables...

Inheritance – The Hack, Second...

Scenario: extension of the expression tree by mathematical functions abs, sin, cos:

Adaption of every single member function

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

Disadvantages:
- Loss of clarity
- hard to work in a team of developers
Inheritance – the Clean Solution

“Split-up” of tree_node

- Common properties stay in the base class xtree_node (will be explained)

Inheritance

Classes can inherit properties

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

```
erbt von inheritance visible
only for number_node
members of xtree_node
are overwritten
```

```
struct number_node : public xtree_node {
    double val;
    int size() const {
        return 1;
    }
    double eval() const {
        return val;
    }
};
```

Inheritance – Notation

```
class A {
    ...
}
```

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

```
class C: public B {
    “B and C inherit from A”
    “C inherits from B”
    ...
}
```

Separation of Concerns: The Number Node

```
struct number_node: public xtree_node{
    double val;
    number_node(double v) : val(v) {}
    double eval() const {
        return val;
    }
    int size() const {
        return 1;
    }
};
```

A Number Node is a Tree Node...

- A (pointer to) an inheriting object can be used where (a pointer to) a base object is required, but not vice versa.

```
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
```

Application

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

Polymorphism

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

Without Virtual the static type determines which function is executed

We do not go into further details.
```

Separation of Concerns: Binary Nodes

```
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 {
        size works for all binary nodes. Derived classes (add_node, sub_node...) inherit this function!
        return 1 + left->size() + right->size();
    }
};
```
Separation of Concerns: +, -, * ...

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

Extension by `abs` Function

```cpp
struct unary_node: public xtree_node
{
    xtree_node* right; // INV != 0
    unary_node (xtree_node* arg)
        : unary_node (arg) {}
    double eval () const {
        return std::abs (right->eval());
    }
};
```

Do not forget... Memory Management

```cpp
struct xtree_node {
...
    virtual xtree_node* copy () const;
    // POST: all nodes in the subtree with root *this are deleted
    virtual void clear () {};
};
```
Do not forget... Memory Management

```
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 is no dynamic data type ??

- We do not have any variables of type xtree_node with automatic memory lifetime
- copy constructor, assignment operator and destructor are unnecessary
- memory management in the container class

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

Summary of the Concepts
.. of Object Oriented Programming

Encapsulation
- hide the implementation details of types
- definition of an interface for access to values and functionality (public area)
- make possible to ensure invariants and the modification of the implementation
Summary of Concepts

.. of Object Oriented Programming

Inheritance
- types can inherit properties of types
- inheriting types can provide new properties and overwrite existing ones
- allows to reuse code and data

Polymorphism
- A pointer may, depending on its use, have different underlying types
- the different underlying types can react differently on the same access to their common interface
- makes it possible to extend libraries “non invasively”