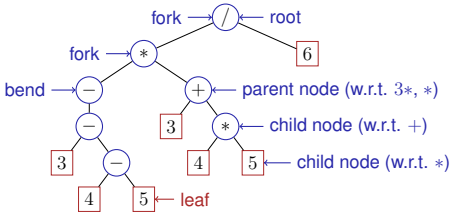
20. Inheritance and Polymorphism

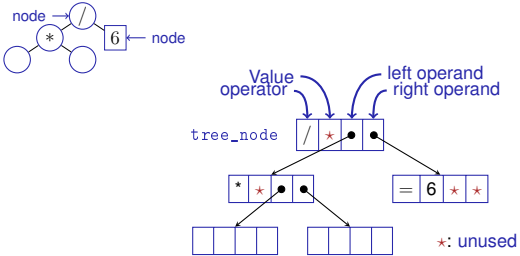
Expression Trees, Inheritance, Code-Reuse, Virtual Functions, Polymorphism, Concepts of Object Oriented Programming

(Expression) Trees

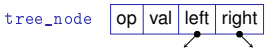
`-(3-(4-5))*(3+4*5)/6`



Nodes: Forks, Bends or Leaves



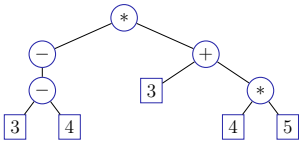
Nodes (`struct tree_node`)




```
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
  ...
};
```

Size = Count Nodes in Subtrees



- Size of a leave: 1
- Size of other nodes: 1 + sum of child nodes' size
- E.g. size of the "+"-node is 5

Count Nodes in Subtrees

```
// 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;
}
```

op | val | left | right

Evaluate Subtrees

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

op | val | left | right

leaf...

...or fork:

op unary, or left branch

right branch

Cloning Subtrees

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

op | val | left | right

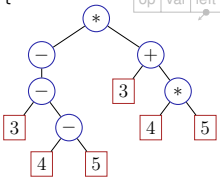
Cloning Subtrees – more Compact Notation

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

cond ? expr1 : expr 2 has value *expr1*, if *cond* holds, *expr2* otherwise

Felling Subtrees

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



Using Expression Trees

```
// 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';
```


Observations

- Class `tree_node` is the "sum" of all required nodes
- Holds for the data representation ...

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

Observations I

- Class `tree_node` is the "sum" of all required nodes
- ... and the code

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

- But: every *object* (instance of the class) represents exactly *one* node
- Objects "specialiase themselves" via `if` statements

Disadvantages I

- Data representation: Waste of memory
- Code: Implementation of different `eval`s (and `size`s, `copy`s, ...) in one member function

⇒ Increased complexity results in convoluted and thus more error-prone code

Observations II

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

- extension of the class **tree_node** by even more member variables

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


Observations II

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

- Adaption of every single member function

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

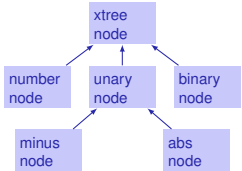
Disadvantages II

- New node types entail modifications of already existing code
- Consequences:
  - ⇒ Code complexity and error-proneness increases further
  - ⇒ Customers/users must have access to source code (originally: distributing libraries as header files plus object files)

Inheritance: the Idea

- Split-up of `tree_node`
- "is-a" relation: `number_node` is a `xtree_node`
- Specialisation happens top-down: a *derived class** may provide more functionality than its *base class***
- Common properties stay in more general ("higher") classes

1) also: child class/sub class
2) also: parent class, superclass



Inheritance: the Code

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

struct number_node : public xtree_node {
  double val;

  int size () const;
  double eval () const;
};
```

Erzwingt Impl. in Kindklassen

erbt von

inheritance visible

only for `number_node`

members of `xtree_node` are implemented by `number_node`

Inheritance: Notation

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

Base/Super Class

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

Subclass

```
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
```


Polymorphism

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

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

- 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;
  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();
  }
};
```

`size` works for *all* binary nodes. Derived classes (`add_node`, `sub_node`...) *inherit* this function!

Separation of Concerns: +, −, * ...

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

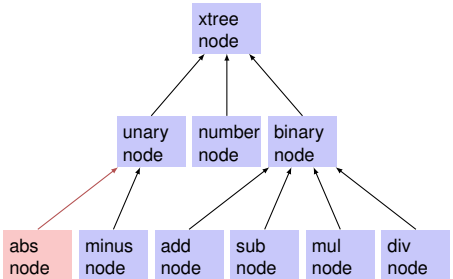
eval specific for +

Separation of Concerns: +, −, * ...

```
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



Extension by `abs` Function

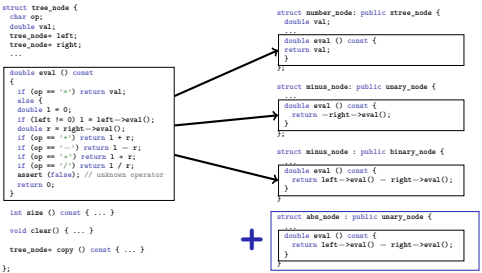
```
struct unary_node: public xtree_node {
  xtree_node* right;

  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: Monolithic → Modular ✓



Summary of the Concepts

.. of Object Oriented Programming

Encapsulation (previous two lectures)

- 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

Summary of Concepts

.. of Object Oriented Programming

Polymorphism

- A pointer (or reference) 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”