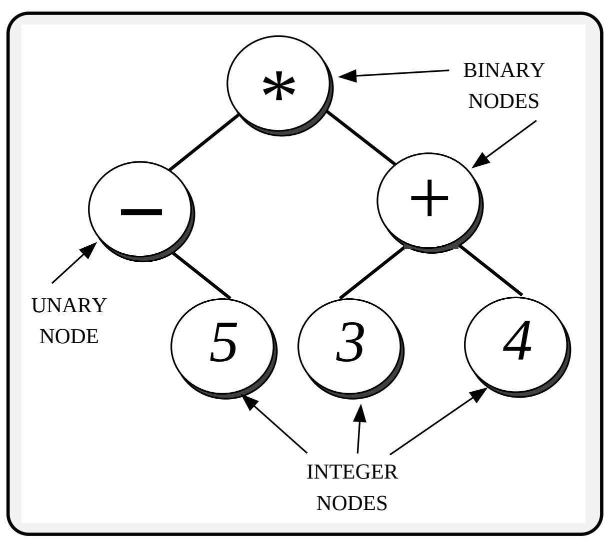
Object-Oriented Design and Programming

C++ Advanced Examples with Inheritance and Dynamic Binding

Introduction

- The following inheritance and dynamic binding example constructs *expression trees*
  - Expression trees consist of nodes containing operators and operands
    - Operators have different *precedence levels* and different *arities*, *e.g.*,
      - Multiplication takes precedence over addition
      - The multiplication operator has two arguments, whereas unary minus operator has only one
    - Operands are integers, doubles, variables, etc.
      - We'll just handle integers in the example...

Expression Tree Diagram



Expression Tree Behavior

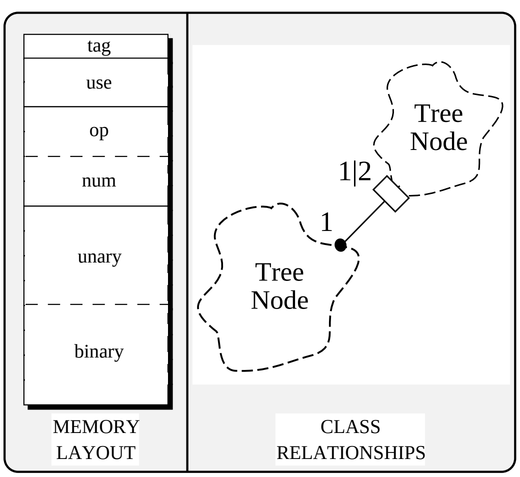
- *Expression trees*
  - These trees may be "evaluated" via different traversals
    - ▷ *e.g.*, in-order, post-order, pre-order, level-order
  - The evaluation step may perform various operations. . . , *e.g.*,
    - ▷ Traverse and print the expression tree
    - ▷ Return the "value" of the expression tree
    - ▷ Generate code
    - ▷ Perform semantic analysis

C Version

- A typical functional method for implementing expression trees in C or Ada involves using a **struct/union** to represent data structure, *e.g.*,

```
typedef struct Tree_Node Tree_Node;
struct Tree_Node {
      enum {
            NUM, UNARY, BINARY
      } tag;
      short use; /* reference count */
      union {
            int num;
            char op[2];
      } o;
#define num o.num
#define op o.op
      union {
            Tree_Node *unary;
            struct { Tree_Node *l, *r; } binary;
      } c;
#define unary c.unary
#define binary c.binary
};
```


Memory Layout of C Version



- Here's what the memory layout of a **struct** Tree_Node object looks like

Print_Tree Function

- Typical C or Ada implementation (cont'd)
  - Use a **switch** statement and a recursive function to build and evaluate a tree, *e.g.*,

```
void print_tree (Tree_Node *root) {
      switch (root->tag) {
      case NUM: cout << root->num; break;
      case UNARY:
            cout << "(" << root->op[0];
            print_tree (root->unary);
            cout << ")"; break;
      case BINARY:
            cout << "(";
            print_tree (root->binary.l);
            cout << root->op[0];
            print_tree (root->binary.r);
            cout << ")"; break;
      default:
            cerr << "error, unknown type\n";
            exit (1);
      }
}
```

Limitations with C Approach

- Problems or limitations with the typical design and implementation approach include
  - Language feature limitations in C and Ada
    - ▷ *e.g.*, no support for inheritance and dynamic binding
  - Incomplete modeling of the problem domain that results in
    1. Tight coupling between nodes and edges in **union** representation
    2. Complexity being in algorithms rather than the data structures
       - ▷ *e.g.*, **switch** statements are used to select between various types of nodes in the expression trees
         - · compare with binary search!
       - ▷ Data structures are "passive" in that functions do most processing work explicitly

Limitations with C Approach (cont'd)

- Problems with typical approach (cont'd)
  - The program organization makes it difficult to extend, *e.g.*,
    - ▷ Any small changes will ripple through the entire design and implementation
      - · *e.g.*, see the ternary extension below
    - ▷ Easy to make mistakes **switch**ing on type tags..
  - Solution wastes space by making worst-case assumptions *wrt* **struct**s and **union**s
    - ▷ This not essential, but typically occurs
    - ▷ Note that this problem becomes worse the bigger the size of the largest item becomes!

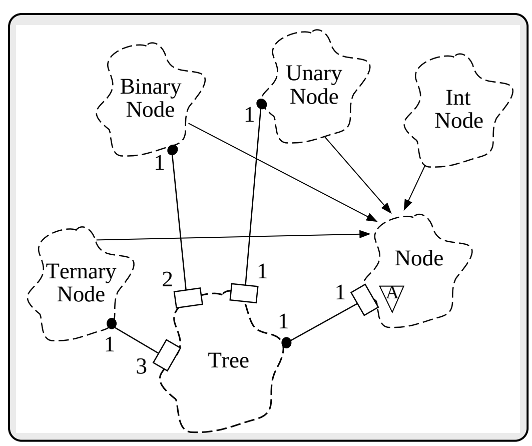
OO Alternative

- Contrast previous functional approach with an object-oriented decomposition for the same problem:
  - Start with OO modeling of the "expression tree" problem domain:
    - ▷ *e.g.*, go back to original picture
  - There are several classes involved:

    class Node: base class that describes expression tree vertices:
    - **class** Int_Node: used for implicitly converting **int** to Tree node
    - **class** Unary_Node: handles unary operators, *e.g.*, −10, +10, !a, or ~foo, etc.
    - **class** Binary_Node: handles binary operators, *e.g.*, a + b, 10 − 30, etc.

    class Tree: "glue" code that describes expression tree edges
  - Note, these classes model elements in the problem domain
    - ▷ *i.e.*, nodes and edges (or vertices and arcs)

Relationships Between Trees and Nodes



C++ Node Interface

- // node.h

```
#ifndef _NODE_H
#define _NODE_H
#include <stream.h>
#include "tree.h"

/* Describes the Tree vertices */
class Node {
friend class Tree;
friend ostream &operator << (ostream &, const Tree &);

protected: /* only visible to derived classes */
      Node (void): use (1) {}
      // pure virtual
      virtual void print (ostream &) const = 0;
      virtual ~Node (void) {}; // important to make virtual!
private:
      int use; /* reference counter */

};
#endif
```


C++ Tree Interface

- // tree.h

```
#ifndef _TREE_H
#define _TREE_H
#include "node.h"

/* Describes the Tree edges */
class Tree {
friend class Node;
friend ostream &operator << (ostream &, const Tree &);

public:
      Tree (int);
      Tree (const Tree &t);
      Tree (char *, Tree &);
      Tree (char *, Tree &, Tree &);
      void operator= (const Tree &t);
      virtual ~Tree (void); // important to make virtual
private:
      Node *ptr; /* pointer to a rooted subtree */
};
#endif
```


C++ Int_Node and Unar_Node Interface

- // int-node.h

```
#ifndef _INT_NODE_H
#define _INT_NODE_H
#include "node.h"
class Int_Node : public Node {
friend class Tree;
private:
      int num; /* operand value */
public:
      Int_Node (int k);
      virtual void print (ostream &stream) const;
};
#endif
```

- // unary-node.h

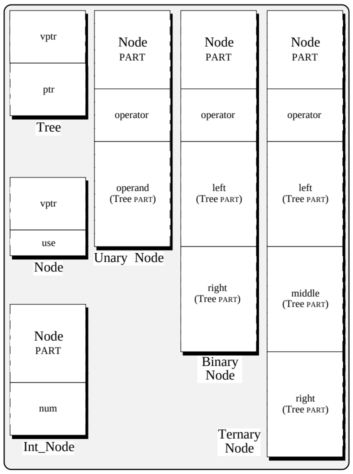
```
#ifndef _UNARY_NODE_H
#define _UNARY_NODE_H
#include "node.h"
class Unary_Node : public Node {
friend class Tree;
public:
      Unary_Node (const char *op, const Tree &t);
      virtual void print (ostream &stream) const;
private:
      const char *operation;
      Tree operand;
};
#endif
```


C++ Binary_Node Interface

- // binary-node.h

```
#ifndef _BINARY_NODE_H
#define _BINARY_NODE_H
#include "node.h"

class Binary_Node : public Node {
friend class Tree;

public:
      Binary_Node (const char *op, const Tree &t1,
                  const Tree &t2);
      virtual void print (ostream &s) const;
private:
      const char *operation;
      Tree left, right;
};
#endif
```

Memory Layout for C++ Version



- Memory layouts for different subclasses of `Node`

C++ Int_Node and Unary_Node Implementations

- // int-node.C

```
#include "int-node.h"
Int_Node::Int_Node (int k): num (k) { }

void Int_Node::print (ostream &stream) const {
    stream << this->num;
}
```

- // unary-node.C

```
#include "unary-node.h"
Unary_Node::Unary_Node (const char *op, const Tree &t1)
    : operation (op), operand (t1) { }

void Unary_Node::print (ostream &stream) const {
    stream << "(" << this->operation << " "
        << this->operand // recursive call!
        << ")";
}
```


C++ Binary_Node Implementation

- // binary-node.C

```
#include "binary-node.h"
Binary_Node::Binary_Node (const char *op, const Tree &t1,
                          const Tree &t2):
    operation (op), left (t1), right (t2) { }

void Binary_Node::print (ostream &stream) const {
    stream << "(" << this->left // recursive call
        << " " << this->operation
        << " " << this->right // recursive call
        << ")";
}
```


C++ Tree Implementation

- // tree.C

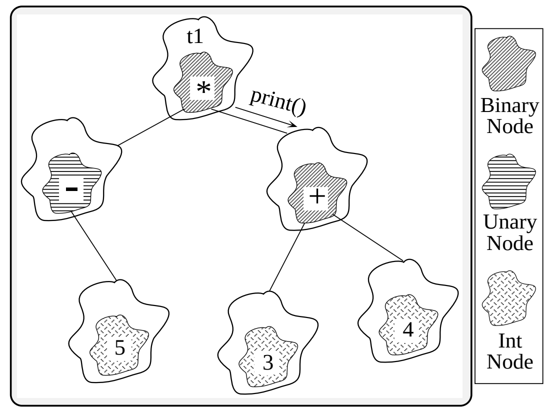
```
#include "tree.h"
#include "int-node.h"
#include "unary-node.h"
#include "binary-node.h"
#include "ternary-node.h"
Tree::Tree (int num) ptr (new Int_Node (num))
}
Tree::Tree (const Tree &t): ptr (t.ptr)
{ // Sharing, ref-counting.. ++this->ptr->use; }
Tree::Tree (const char *op, const Tree &t)
        : ptr (new Unary_Node (op, t)) {}
Tree::Tree (const char *op, const Tree &t1,
        const Tree &t2):
        : ptr (new Binary_Node (op, t1, t2)) {}
Tree::~Tree (void) { // Ref-counting, garbage collection
    if (--this->ptr->use <= 0)
        delete this->ptr;
}
void Tree::operator= (const Tree &t) {
    ++t.ptr->use;
    if (--this->ptr->use == 0) // order important
        delete this->ptr;
    this->ptr = t.ptr;
}
```

C++ Main Program

- // main.C

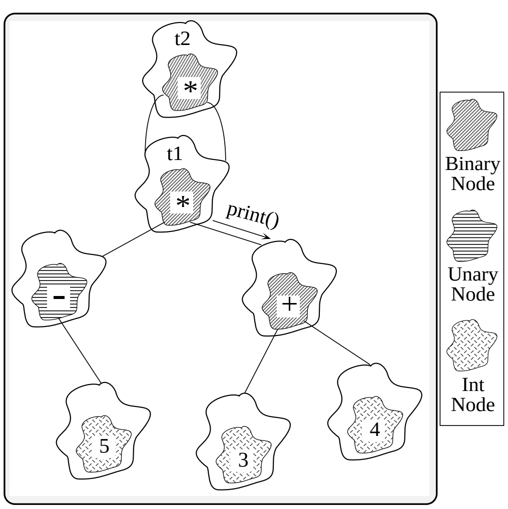
```
#include <stream.h>
#include "tree.h"

ostream &operator<< (ostream &s, const Tree &tree) {
      tree.ptr->print (s); /* Virtual call! */
      // (*tree->ptr->vptr[1]) (tree->ptr, s);
      return s;
}

int main (void) {
      const Tree t1 = Tree ("*", Tree ("-", 5),
                        Tree ("+", 3, 4));
      // Tree ("*", Tree ("-", Tree (5)),
      // Tree ("+", Tree (3), Tree (4)));

      /* prints ((−5) * (3 + 4)) */
      cout << t1 << "\n";
      const Tree t2 = Tree ("*", t1, t1);

      /* prints (((−5) * (3 + 4)) * ((−5) * (3 + 4))) */
      cout << t2 << "\n";

      /* virtual destructor recursively deletes
           entire tree leaving scope */
}
```


Expression Tree Diagram 1



- Expression tree for t1 = ((−5) * (3 + 4))

Expression Tree Diagram 2



- Expression tree for t2 = (t1 * t1)

Extending Solution with Ternary_Nodes

- Extending the existing solution to support ternary nodes is very straight forward
  - *i.e.*, just derived new class Ternary_Node

    class Ternary_Node: handles ternary operators, *e.g.*, a **==** b ? c : d, etc.

- // ternary-node.h

```
#ifndef _TERNARY_NODE
#define _TERNARY_NODE
#include "node.h"

class Ternary_Node : public Node {
friend class Tree;

private:
      const char *operation;
      Tree left, middle, right;

public:
      Ternary_Node (const char *, const Tree &,
                  const Tree &, const Tree &);
      virtual void print (ostream &) const;
};
#endif
```

C++ Ternary_Node Implementation

- // ternary-node.C

```
#include "ternary-node.h"
Ternary_Node::Ternary_Node (const char *op,
                            const Tree &a,
                            const Tree &b,
                            const Tree &c)
    : operation (op), left (a), middle (b), right (c) {}
void Ternary_Node::print (ostream &stream) const {
    stream << this->operation << "("
        << this->left // recursive call
        << "," << this->middle // recursive call
        << "," << this->right // recursive call
        << ")";
}
```

- // Modified **class** Tree

```
class Tree { // add 1 class constructor
// Same as before
public:
// Same as before
    Tree (const char *, const Tree &,
        const Tree &, const Tree &);
};
Tree::Tree (const char *op, const Tree &a,
        const Tree &b, const Tree &c):
        : ptr (new Ternary_Node (op, a, b, c)) {}
```


Differences from C Implementation

- On the other hand, modifying the original C approach requires changing:
  - The original data structures, *e.g.*,

```
struct Tree_Node {
    enum {
        NUM, UNARY, BINARY, TERNARY
    } tag;
    // same as before
    union {
        // same as before
        // add this
        struct {
            Tree_Node *l, *m, *r;
        } ternary;
    } c;
#define ternary c.ternary
};
```

  - and many parts of the code, *e.g.*,

```
void print_tree (Tree_Node *root) {
    // same as before
    case TERNARY: /* must be TERNARY */
        cout << "(";
        print_tree (root->ternary.l);
        cout << root->op[0];
        print_tree (root->ternary.m);
        cout << root->op[1];
        print_tree (root->ternary.r);
        cout << ")"; break;
    // same as before
}
```


Summary

- OO version represents a more complete modeling of the problem domain
  - *e.g.*, splits data structures into modules that correspond to "objects" and relations in expression trees
- Use of C++ language features simplify the design and facilitate extensibility
  - *e.g.*, the original source was hardly affected

Summary (cont'd)

- Potential Problems with OO approach
  - Solution is very "data structure rich"
    - ▹ *e.g.*, Requires configuration management to handle many headers and .C files!
  - May be somewhat less efficient than original C approach
    - ▹ *e.g.*, due to virtual function overhead
  - In general, however, virtual functions may be no less inefficient than large **switch** statements or **if/else** chains...
  - As a rule, be careful of micro vs. macro optimizations
    - ▹ *i.e.*, always profile your code!