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 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.,

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

<table>
<thead>
<tr>
<th>MEMARY LAYOUT</th>
<th>CLASS RELATIONSHIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>use</td>
</tr>
<tr>
<td></td>
<td>op</td>
</tr>
<tr>
<td></td>
<td>num</td>
</tr>
<tr>
<td></td>
<td>unary</td>
</tr>
<tr>
<td></td>
<td>binary</td>
</tr>
</tbody>
</table>

- 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.,

```c
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.l);
        cout << ")"; break;
    default: cerr << "error, unknown type
"; 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 switching on type tags...
  - Solution wastes space by making worst-case assumptions wrt structs and unions
    - 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 IntNode: used for implicitly converting int to Tree node
      - class UnaryNode: handles unary operators, e.g., −10, +10, 1a, or “foo”, etc.
      - class BinaryNode: 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

```c
#ifndef _NODE_H
#define _NODE_H
#include <ostream.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 Node (void)[]): // important to make virtual!
    private:
        int use; /* reference counter */
};
#endif
```

C++ Tree Interface

• // tree.h

```c
#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 &);
    Tree (char *, Tree &);
    Tree (char *, Tree &, Tree &);
    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

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

• // unary-node.h

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

C++ Binary_Node Interface

• // binary-node.h

```c
#ifndef _BINARY_NODE_H
#define _BINARY_NODE_H
#include "node.h"
class BinaryNode: public Node {
friend class Tree;

public:
    BinaryNode (const char *op, const Tree &1, const Tree &2);
    virtual void print (ostream &stream) 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 &t1, const Tree &t2):
        ptr (new Unary_Node (op, t1)) {}

    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;
        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

```cpp
#include <iostream>
#include "tree.h"

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

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

- **Expression Tree Diagram 1**

- **Expression Tree Diagram 2**

- **Extending Solution with Ternary_Nodes**

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

```cpp
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 &)
```

- // ternary-node.h

```cpp
#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 &)
```

- #endif
### C++ Ternary_Node Implementation

- // ternary-node.C

```cpp
#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

```cpp
class Tree { // add 1 class constructor
    // Same as before
public:
    // Same as before
    Tree (const char *, const Tree &a,
          const Tree &b, const Tree &c);
    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.,
    ```cpp
    struct TreeNode {
        enum {
            NUM, UNARY, BINARY, TERNARY
        } tag;
        // same as before
        union {
            // same as before
            struct {
                TreeNode *l, *m, *r;
            } ternary;
        } c;
    };
    #define ternary ct.ternary
    - and many parts of the code, e.g.,
    ```cpp
    void print_tree (TreeNode *root) {
        // same as before
        case TERNARY: /* must be TERNARY */
            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!