An introduction to trees in C/C++

Hayo Thielecke
University of Birmingham
www.cs.bham.ac.uk/~hxt

July 2, 2014

Abstract
There are two fundamentally different ways of representing trees in C++: the C way, using struct and union, as opposed to the object-oriented way, using the a class hierarchy known as the Composite pattern. Since C++ contains both C and objects, it is one of the few languages in which both styles of programming trees are equally idiomatic. Both styles are important, but with different advantages and disadvantages.

Contents
1 Introduction 1
2 Structures and unions in C 2
3 Recursive types via pointers and structures in C 4
4 Background: parse trees 4
5 Parse trees from pointers + struct + union + enum in C 5
6 Example: parsing and evaluating expressions in C 6
7 Trees from classes + derived classes + pointers in C++ 9
8 Example: expression trees from classes in C++ 9
9 Data types versus objects 11

1 Introduction
These notes were written with the students on my second-year C/C++ course in mind. See http://www.cs.bham.ac.uk/~hxt/2014/c-plus-plus/index.shtml
In particular, students have seen in their first year:

• Java classes in Software Workshop
• Parse trees in Language and Logic
• XML trees in Information and the web
• Tree algorithms in Foundations 2
• Inductive data types in OCAML

Given this background, we can cover trees in C/C++ fairly quickly. The comparison to other languages is instructive, and goes deeper than knowing only C/C++ in isolation (let alone knowing only Java).

Parsers and parse trees are one of the best ways to demonstrate the capabilities of a programming language at a medium to advanced level. (For example, both K&R and Stroustrup use such toy parser examples for C and C++, respectively [KR88, Str12].)

C++ provides a variety of constructs and does not force a particular style on the programmer. For parse trees, we can use the C subset or the object-oriented layer of C++. They represent different programming “paradigms”.

2 Structures and unions in C

Unions are similar to the type constructor | in functional languages. However, C unions are not “tagged”. The tagging and switching needs to be dealt with explicitly by the C programmer.

For example, in OCAML, binary trees of integers can be defined as follows:

```ocaml
type intbt = Leaf of int
  | Internal of intbt * intbt;;
```

Haskell is even more concise.

The analogue in C using unions and structs is the following:

```c
struct intbt
{
    enum { isLeaf, isInternal } tag;
    union {
        // if tag == isLeaf use this:
        int Leaf;
        // if tag == isInternal use this:
        struct {
            struct intbt *left;
            struct intbt *right;
        } Internal;
    } LeafOrInternal;
};
```

Listing 1: Binary tree in C

**Exercise 2.1** Rewrite the structure intbt definition so that the enum and the inner struct declarations are made outside it.

1But be wary of “paradigms”, especially when they “shift”. The word can mean anything from marketing guff to pseudoscience.
Traversing the tree involves a `switch` on the tag and extracting the appropriate structure from the union:

```c
void traversebt(struct intbt *p)
{
    switch(p->tag)
    {
        case isLeaf:
            // use p->LeafOrInternal.Leaf;
            break;
        case isInternal:
            // use
            // p->LeafOrInternal.Internal.left
            // and
            // p->LeafOrInternal.Internal.right
            break;
        default:
            fprintf(stderr, "Invalid tag in struct intbt.\nGoodbye.\n");
            exit(1);
    }
}
```

Listing 2: Tree traversal in C

Compare and contrast the above with the convenience and type safety of pattern matching in OCAML or Haskell.

When using tagged unions we can make mistakes which are not caught by the C compiler:

```c
void wrongtag(struct intbt *p)
{
    switch(p->tag)
    {
        case isLeaf:
            p->LeafOrInternal.Internal.left = NULL; // ouch
            break;
        ...
    }
}
```

The code above will pass an integer as a pointer to a function at runtime, and most likely corrupt memory.

At this point, friends of OCAML or Haskell may be tempted to sneer at the verbosity and lack of safety of C. The slogan is that in type-safe languages like ML and Haskell, “well-typed programs cannot go wrong”. In C/C++, not so much.

To be fair to C, one should bear in mind that C dates from the early 1970s and builds on language design from the late 1960s. Moreover, data structures with pointers in C can do low-level things that would be impossible to do with trees in Haskell. (Purity comes at a price.) OCAML data structures are closer to C in this regard than Haskell, as `ref` types and `mutable` record fields in OCAML can do some of the work of C pointers (though not pointer arithmetic).
3 Recursive types via pointers and structures in C

A structure can refer to itself recursively, but only through a pointer.

The following does not work; you may get a slightly cryptic error message about an “incomplete type”. Particularly when you come from Java, you probably forgot a *

```c
struct linkedNoPointer
{
  int data;
  struct linkedNoPointer next;  /* wrong: recursion without pointer */
};
```

It looks similar to what you can write in Java for recursive classes. However, in Java pointers are invisibly added by the compiler.

While we can use recursive structures to build trees, data structures in C are a lot more flexible than merely trees, due to sharing via pointers. For instance, the following structure can be used for binary tree or doubly-linked lists:

```c
struct twoPointers
{
  int data;
  struct twoPointers *p1;
  struct twoPointers *p2;
};
```

Exercise 3.1 Draw two heaps containing three instances of `struct twoPointers`. The first heap should contain a balanced binary tree, whereas the second heap should contain a doubly linked list.

Some authors may prefer to use `typedef` when defining recursive types. To avoid complicated syntax, we will not use `typedef`, perhaps the most confusing piece of inside-out syntax in C. In C++11 `typedef` is superseded by `using`, with a syntax close to functional languages. In C++11, one writes

```c
using t = int;
```

just as one writes

```c
type t = int
```

in Haskell. The `using` is most useful in connection with C++ templates.

4 Background: parse trees

We recall some general definitions: grammars and parse trees. For more background, see any compiler textbook, e.g., [AG13]. A context-free grammar consists of

- some terminal symbols a, b, ..., +, ),...
• some non-terminal symbols $A, B, S, \ldots$
• a distinguished non-terminal start symbol $S$
• some rules of the form $A \rightarrow X_1 \ldots X_n$

where $n \geq 0$, $A$ is a non-terminal, and the $X_i$ are symbols.

For a given grammar, a parse tree is a tree of the following form: The internal nodes are labelled with nonterminals. If there is a rule $A \rightarrow X_1 \ldots X_n$, then an internal node can have the label $A$ and children $X_1, \ldots, X_n$.

The root node of the whole tree is labelled with the start symbol. The leaf nodes are labelled with terminal symbols or $\varepsilon$.

A parser is a program that reads an input string and constructs the corresponding parse tree if it exists:

$$\text{String} \xrightarrow{\text{parse}} \text{Tree} \cup \text{SyntaxError} \xrightarrow{\text{toString}} \text{String}$$

We will use the following example. We have a grammar for expressions like $1 + 2 + 3$.

$$E \rightarrow n \quad \text{where } n \text{ is an integer}$$
$$\mid E + E$$
$$\mid E - E$$
$$\mid E * E$$

5 Parse trees from pointers + struct + union + enum in C

The translation of the right-hand side of a rule $X_1 \ldots X_n$ is
struct {
    X1 x1;
    ...
    Xn xn;
} Ak;

If \( n = 1 \), the `struct` can be omitted, and we only need \( X1 x1; \).

enum Atag { A1, ..., Am };

struct A {
    enum Atag tag;
    union {
        // right-hand side of rule 1 for A
        ...
        // right-hand side of rule m for A
    } Aunion;
}

Listing 3: From grammar to C struct and union

It is slightly tedious to come up with names for all the structures, unions and enums, so that it is useful to have some naming convention.

Note that the elements of the enum and the structure member names are the same. This is purely a convention and not enforced by the language.

The point of having the tag in the structure is that it lets us pick the appropriate case from the union. Doing so is most elegantly done with a switch statement.

T traverseA(struct *A)
{
    switch(A->tag)
    {
    case A1:
        ... A->Aunion.x1 ...
        ...
        ... A->Aunion.xn ...
    case An:
        ...
    default:
        // invalid tag
    }
}

Listing 4: Traversal of trees

6 Example: parsing and evaluating expressions in C

The code is available at:
http://www.cs.bham.ac.uk/~hxt/2014/c-plus-plus/ParserTree.c
Here we are interested in the parse tree and are not concerned with the
details of how the parser works. (If you wonder why the parsing functions take
extra parameters, it is due to left recursion elimination; see [Thi13].)
We follow the construction of grammars to C from Listing 3 for our expres-
sion grammar.

```
enum Etag {
    constant, plus, minus, times
};
struct E {
    enum Etag tag;
    union {
        int constant;
        struct {
            struct E *e1;
            struct E *e2;
        } plus;
        struct {
            struct E *e1;
            struct E *e2;
        } minus;
        struct {
            struct E *e1;
            struct E *e2;
        } times;
    } Eunion;
};
```

Listing 5: Expression trees from struct + union + enum + pointers

Rather than using malloc for creating tree nodes, we manage memory our-
selves. In this instance, it is easy to do, since tree nodes need to be allocated
but never deallocated. See Listing 6. Hence we can build dynamic data struc-
tures without using the heap. This is not possible in Java, where all objects are
heap-allocated.

```
const int heapsize = 100;
struct E myheap[heapsize];
struct E *freeptr = myheap;
struct E *myalloc()
{
    if(freeptr + 1 >= myheap + heapsize) {
        fprintf(stderr, "Heap overflow.\n");
        exit(1);
    }
    return freeptr++;
}
```

Listing 6: Allocation of tree nodes in an array

```
struct E *makeconstant(int n)
```
{  
  struct E *p;
  p = myalloc();
  p->tag = constant;
  p->Eunion.constant = n;
  return p;
}

struct E *makeplus(struct E *left, struct E *right)
{
  struct E *p;
  p = myalloc();
  p->tag = plus;
  p->Eunion.plus.e1 = left;
  p->Eunion.plus.e2 = right;
  return p;
}

Evaluating an expression tree is just a traversal or “tree walk”, as in Listing 4.

```c
int eval(struct E *p)
{
  switch (p->tag) {
    case constant:
      return p->Eunion.constant;
    case plus:
      return eval(p->Eunion.plus.e1) + eval(p->Eunion.plus.e2);
    case minus:
      return eval(p->Eunion.minus.e1) - eval(p->Eunion.minus.e2);
    case times:
      return eval(p->Eunion.times.e1) * eval(p->Eunion.times.e2);
    default:
      fprintf(stderr, "Invalid tag for struct E.\n\n") ;
      exit(1);
  }
}
```

Listing 7: Evaluation function

**Exercise 6.1 (Easy)** Write a pretty-printing function that outputs an expression tree in (some form of) properly indented XML. For example, 1+2 should be printed as

```
<plus>
  <constant>
    1
  </constant>
  <constant>
    2
</plus>
```
Hint: the pretty printing function should take an integer parameter representing the current level of indentation.

**Exercise 6.2 (Harder and optional)** Modify the grammar so that operators are prefix rather than infix, and write a parser for the new grammar. Note that this parser is much simpler as it requires no left-recursion elimination.

7 Trees from classes + derived classes + pointers in C++

We translate a grammar to some mutually recursive C++ classes:

- For each non-terminal $A$ there is an abstract class $A$.
- For each rule $A \rightarrow X_1 \ldots X_n$, there is a derived class of $A$.
- It has members for all non-terminals among $X_1, \ldots, X_n$.

It is instructive to compare Listing 8 to Listing 3 above.

```cpp
class A { 
public:
    virtual T f() = 0; // for all operations we want the tree to have }
};

class A1 : public A { 
    // right-hand side of rule 1 for A }
};
...

class Ak : public A { 
    // right-hand side of rule k for A }
};
```

Listing 8: C++ classes for grammar

The above is an instance of the Composite pattern [GHJV95].

8 Example: expression trees from classes in C++

We perform the translation given in Listing 8 for the expression grammar. We use more meaningful identifiers in this case, such as `constant`, `plus`, and so on.

The code is available at: [http://www.cs.bham.ac.uk/~hxt/2014/c-plus-plus/ParserTreeOO.cpp](http://www.cs.bham.ac.uk/~hxt/2014/c-plus-plus/ParserTreeOO.cpp)
class E {
    public:
    virtual int eval() = 0;
};

Listing 9: Base class for expression trees

class constant : public E {
    int n;
    public:
    constant(int n) { this->n = n; }
    int eval();
};

Listing 10: Derived classes for expression trees

class plus : public E {
    class E *e1;
    class E *e2;
    public:
    plus(class E *e1, class E * e2) {
        this->e1 = e1;
        this->e2 = e2;
    }
    int eval();
};

The evaluation function in Listing 7 is now split into member functions of
the different classes. Notice that we do not need the enum tag and switch state-
ment. The appropriate operation is automatically chosen by dynamic dispatch
of virtual functions.

int constant::eval()
{
    return n;
}

int plus::eval()
{
    return e1->eval() + e2->eval();
}

Listing 11: Member functions for evaluation

int eval(class E *p)
{
    return p->eval();
}

Listing 12: Wrapper function for OO eval

We write a few wrapper functions, so that the parser can be used without
any changes.
class E * makeconstant (int n) {
  return new constant (n);
}

class E * makeplus (class E *left, class E *right) {
  return new plus (left, right);
}

Listing 13: Wrapper functions for OO constructors

In principle, we could avoid the use of new and allocate the objects in an array. However, the interaction between arrays and inheritance is somewhat tricky. In particular, allocating an array of the type E of the base class will not work.

9 Data types versus objects

There is a fundamental difference between the two implementations of parse trees that becomes relevant when we try to extend or evolve the code:

1. We may wish to add more cases to the grammar, say for a division operator.

2. We may wish to add more operations to the expression trees, say pretty printing or compilation to machine code.

There is a trade-off between these aims. The first is easy to do with the object-oriented implementation: we can just add another derived class, and everything works as before. The second is easy to do with structures and recursive functions processing them. For instance, a pretty printing function can be written just as the evaluation function above.

Ideally, one might hope for the best of both worlds where both new cases and new operations can be added easily and without modifying existing code. This problem is the subject of ongoing research, sometimes called the “expression problem” because parse trees of expression are the canonical example. A closely related point is the Visitor pattern [GHJV95, BT06] for traversal of trees in object-oriented style.

References


[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Boston, Massachusetts, 1995.

