

NAME

HTML_Node – Abstract base class for nodes in an HTML tree

SYNOPSIS

```
namespace HTML_Tree {

    class HTML_Node {
    public:
        class visitor {
        public:
            virtual ~visitor();
            virtual bool operator()(
                HTML_Node*, int depth, bool is_end_tag
            ) = 0;
        };

        virtual ~HTML_Node() = 0;

        class iterator : public
            std::iterator< std::forward_iterator_tag, HTML_Node > {
        public:
            iterator();

            iterator& operator++();
            iterator operator++(int);

            HTML_Node& operator* () const;
            HTML_Node* operator->() const;

            friend bool operator==(iterator const&, iterator const&);
            friend bool operator!=(iterator const&, iterator const&);
        };
        iterator begin();
        iterator end();

        class const_iterator : public
            std::iterator< std::forward_iterator_tag, HTML_Node const > {
        public:
            const_iterator();

            const_iterator& operator++();
            const_iterator operator++(int);

            HTML_Node& operator* () const;
            HTML_Node* operator->() const;

            friend bool operator==(
                const_iterator const&, const_iterator const&
            );
            friend bool operator!=(
                const_iterator const&, const_iterator const&
            );
        };
        const_iterator begin() const;
        const_iterator end() const;
    };
};
```

```

std::string as_string( int pretty_print = -1 ) const;
Content_Node* parent() const;
void parent( Content_Node *new_parent );
virtual void visit( visitor&, int depth = 0 );
std::ostream& write( std::ostream&, int pretty_print = -1 ) const;
virtual bool write_node(
    std::ostream&, int spaces, bool is_end_tag
) const = 0;

class manip {
public:
    typedef
        std::ostream& (HTML_Node::*function)(std::ostream&,int) const;

    manip( HTML_Node const&, function f, int arg );
    friend std::ostream& operator<<( std::ostream&, manip const& );
};

manip write( int pretty_print = -1 ) const;

friend bool operator==( HTML_Node const&, HTML_Node const& );
friend bool operator!=( HTML_Node const&, HTML_Node const& );
protected:
    HTML_Node( Content_Node *parent = 0 );
    virtual bool similar_to( HTML_Node const& ) const;
};

Content_Node* html_parse(
    char const *begin, char const *end, bool include_comments = false
);
}

```

DESCRIPTION

HTML_Node is an abstract base class for nodes in an HTML tree that was built by parsing an HTML file into a tree structure like the HTML DOM (Document Object Model). Once built, the nodes of the tree (elements and text from the HTML file) can be traversed either by a user-defined *visitor* class or by an iterator.

Public Interface

```
string as_string( int pretty_print = -1 ) const
```

Returns the HTML tree converted (back) to an HTML string. The `pretty_print` argument, when zero or greater, specifies that the HTML is to be “pretty-printed”: text nodes are trimmed of leading and trailing whitespace and have a single newline appended; all other nodes appear on lines by themselves indented by their depth. The indentation per line is incremented by the number of spaces given by `2 * pretty_print`.

```
iterator begin()
const_iterator begin() const
iterator end()
const_iterator end()
```

Return either an `iterator` or `const_iterator`, respectively, either at the beginning or one past the end (in STL style) of the HTML tree. The iterators can be used with all STL algorithms.

```
Content_Node* parent() const
```

Returns a pointer to the current parent node for this node, or null if this node has no parent.

```
void parent( Content_Node *new_parent )
    If this node already has a parent that is not the current parent, this node is first removed from that
    parent's list of child nodes. Then, this node's parent node is set to new_value. If new_par-
    ent is not null, adds this node to the parent's list of child nodes.

virtual void visit( visitor&, int depth = 0 )
    Performs an in-order tree traversal starting at this node. For each node, the visitor's opera-
    tor() is called once.

std::ostream& write( std::ostream&, int pretty_print = -1 ) const
    Writes the HTML text representation of the tree to the given ostream. The pretty_print
    has the same meaning as for as_string().

manip write( int pretty_print = -1 ) const
    This is a specialized version of write() above that allows this to be done:

        some_ostream << node->write();
```

i.e., writing to an ostream using “insertion style.”

```
virtual bool write_node( std::ostream&, int spaces, bool is_end_tag )
const = 0
    Write the XML text representation of the node to the given ostream preceded by the given num-
    ber of spaces. If is_end_tag is true, write the end tag for the element; otherwise the start
    tag. Returns false only if nothing was written.

friend bool operator==( HTML_Node const&, HTML_Node const& )
friend bool operator!=( HTML_Node const&, HTML_Node const& )
    Compares two HTML_Nodes (or objects of classes derived from HTML_Node) for equality or
    inequality, respectively, and returns that result.
```

Protected Interface

```
HTML_Node( Content_Node *parent = 0 )
    Default constructor. If parent is not null, sets the parent and adds this node to that parent's list
    of child nodes.

virtual bool similar_to( HTML_Node const& ) const
    Returns true only if this node is the same node as the given one, i.e., their addresses are equal.
    (This is overridden by “semantically better” functions in derived classes.)
```

Global Functions

```
Content_Node* html_parse( char const *begin, char const *end, bool
include_comments = false )
    Parses the HTML in the buffer between [begin,end) into an HTML tree and returns a pointer to
    the root node of an HTML tree.
```

Iterator Classes

The classes `iterator` and `const_iterator` are STL `forward_iterators` and can be used in the same way including in all STL algorithms.

The Visitor Class

`HTML_Node::visitor` is an abstract base class for object that “visit” nodes.

Public Interface

```
virtual ~visitor()
    Destructor. It does nothing. It's defined only to ensure it's virtual as it should be for an abstract
    base class.

virtual bool operator()( HTML_Node*, int depth, bool is_end_tag )
    The visit function. A derived class must override this since it's pure virtual. The depth indi-
    cates how “deep” the current node is in the tree. Depths start at zero. The is_end_tag
```

argument is not used by `HTML_Node`, so it always passes `false`.

Iterators vs. Visitors

The `iterator` and `visitor` classes are similar in that they can both be used to iterate over (or visit) every node in the tree. However, the differences are:

1. An `iterator` iterates over every node exactly once.
2. A `visitor` visits non-empty nodes twice: once each for the start and end tags.
3. A `visitor`, based on the visitor function's return values, can either skip nodes by not descending into portions of the tree or loop back from end tags to start tags and repeat portions of the tree.

SEE ALSO

`Comment_Node(3)`, `Content_Node(3)`, `Element_Node(3)`, `Text_Node(3)`.

World Wide Web Consortium Document Object Model Working Group. *Document Object Model*, December 1998.

<http://www.w3.org/DOM/>

AUTHOR

Paul J. Lucas <pauljlucas@mac.com>

HISTORY

The HTML parser is derived from code in SWISH++, a really fast file indexing and searching engine (also by the author).