

NAME

Content_Node – Node in an HTML tree for HTML elements that possibly have attributes and child nodes, i.e., content.

SYNOPSIS

```
namespace HTML_Tree {

    class Content_Node : public Element_Node {
    public:
        typedef std::list< HTML_Node* > child_list;
        typedef child_list::iterator iterator;
        typedef child_list::const_iterator const_iterator;
        typedef child_list::reverse_iterator reverse_iterator;
        typedef child_list::const_reverse_iterator const_reverse_iterator;
        typedef child_list::size_type size_type;

        Content_Node();
        Content_Node( char const *name, element const&, Content_Node* = 0 );
        Content_Node(
            char const *name, element const&,
            char const *att_begin, char const *att_end, Content_Node* = 0
        );
        virtual ~Content_Node();

        HTML_Node*          back();
        HTML_Node const*    back() const;
        iterator            begin();
        const_iterator      begin() const;
        iterator            end();
        const_iterator      end() const;
        HTML_Node*          front();
        HTML_Node const*    front() const;
        bool                empty() const;
        HTML_Node*          erase( iterator const& );
        HTML_Node*          erase( HTML_Node* );
        HTML_Node*          insert( iterator const&, HTML_Node* );
        HTML_Node*          pop_back();
        HTML_Node*          pop_front();
        HTML_Node*          push_back();
        HTML_Node*          push_front();
        reverse_iterator     rbegin();
        const_reverse_iterator rbegin() const;
        reverse_iterator     rend();
        const_reverse_iterator rend() const;
        size_type            size() const;

        // overridden
        virtual void        visit( visitor&, int depth = 0 );
    protected:
        // overridden
        virtual bool        write_node( std::ostream&, int, bool ) const;
    };

}
```

DESCRIPTION

Content_Node is an Element_Node and an Content_Node that contains attributes and child nodes, i.e., content. For example, the SELECT element below is a parent of the newline Text_Node after the SELECT and all of the OPTION elements:

```

<SELECT NAME="menu">           parent child
  <OPTION>Blueberry           child grandchild
  <OPTION>Chocolate           child grandchild
  <OPTION>Raspberry           child grandchild
</SELECT>

```

whereas an element such as IMG can have no child nodes.

Public Interface**Constructors**

These are the same as those for Element_Node.

Destructor

In addition to destroying itself, the destructor also destroys all of its child nodes, if any.

```
HTML_Node* back()
```

```
HTML_Node const* back() const
```

Returns a pointer to the last child node.

```
iterator begin()
```

```
const_iterator begin() const
```

Returns an iterator positioned at the first child node.

```
iterator end()
```

```
const_iterator end() const
```

Returns an iterator positioned at one past the last child node.

```
HTML_Node* front()
```

```
HTML_Node const* front() const
```

Returns a pointer to the first child node.

```
bool empty() const
```

Returns true only if there are no child nodes.

```
HTML_Node* erase( iterator const& )
```

Erases the child node at the position of the given iterator and returns a pointer to said node. Note that the child node itself is not deleted; rather it (any its child nodes, if any) are merely “detached.”

```
HTML_Node* erase( HTML_Node* )
```

Erases the given child node and returns a pointer to it; returns null if the given node isn’t actually a child node of the current node. Note that the child node itself is not deleted; rather it (any its child nodes, if any) are merely “detached.”

```
HTML_Node* insert( iterator const&, HTML_Node* )
```

Inserts the given node before the position of the given iterator removing it from its current parent node, if any. Returns a pointer to the node only if it was actually inserted, i.e., the pointer wasn’t null and it wasn’t already a child node.

```
HTML_Node* pop_back()
```

```
HTML_Node* pop_front()
```

These are special cases of erase() for the back and front, respectively.

```
HTML_Node* push_back( HTML_Node* )
```

```
HTML_Node* push_front( HTML_Node* )
```

These are special cases of `insert()` for the back and front, respectively.

```
reverse_iterator rbegin()
```

```
const_reverse_iterator rbegin() const
```

Returns an iterator positioned at the first child node.

```
reverse_iterator rend()
```

```
const_reverse_iterator rend() const
```

Returns a reverse iterator positioned at one past the last child node.

```
size_type size() const
```

Returns the number of child nodes.

```
virtual void visit( visitor &v, int depth = 0 )
```

This member function overrides `visit()`. First, it visits this node passing `false` for `is_end_tag`. If the visitor's `operator()` returns `false`, return immediately. Otherwise, call `Content_Node::visit()` that visits each child node in order passing `depth + 1`, then visit this node again passing `true` for `is_end_tag`. If the visitor's `operator()` returns `false`, return immediately. Otherwise repeat the entire visit cycle. In pseudo-code:

```
do {
    if ( not_root_node && !v( this, depth, false ) )
        break;
    for ( every child )
        child->visit( v, depth + not_root_node );
} while ( not_root_node && v( this, depth, true ) );
```

SEE ALSO

`Element_Node(3)`, `HTML_Node(3)`.

AUTHOR

Paul J. Lucas <pauljlucas@mac.com>