12: HTML

Last Update: February 22, 2011.

The HTML API [Library xhtml: include | src]

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

This C++ HTML generation API is slowly but surely going out of fashion. Nowadays, it's recommended to use mainstream XML/XSLT approach to prepare HTML pages; in particular, the XmlWrapp API.

NB Don't confuse it with the C++ CGI framework API **-- which is alive and well!**

The HTML module can be used to compose and print out a HTML page by using a static HTML template with embedded dynamic fragments. The HTML module provides a rich set of classes to help build the dynamic fragments using HTML tag nodes together with text nodes arranged into a tree-like structure.

This chapter provides reference material for many of the HTML facilities. You can also see the quick reference guide, a note about using the HTML and CGI classes together and an additional class reference document. For an overview of the HTML module please refer to the HTML section in the introductory chapter on the C++ Toolkit.

Chapter Outline

The following is an outline of the topics presented in this chapter:

NCBI C++ HTML Classes

The NCBI C++ HTML classes are intended for use in CGI programs that generate HTML. By creating a structured method for creating HTML, these classes allow for reuse of HTML generating code and simplifies laborious tasks, such as creating and maintaining tables.

A good resource for the use of HTML is the **HTML Sourcebook** by Ian Graham.

Using these classes, the in-memory representation of an HTML page is of a graph: each element on the page can have other elements as children. For example, in

```
<HTML><BODY>hello</BODY></HTML>
```

the body tag is a child of the html tag and the text "hello" is a child of the body tag. This graph structure allows for the easy addition of components as well as reuse of code among components since they share the same base classes.

A sample program, htmldemo.cpp, can be found in internal/c++/src/html/demo.

Next, the following topics are discussed:

- Basic Classes
- Specialized Tag Classes used in Forms
- Specialized Tag Classes used in Lists
- Other Specialized Tag Classes

Basic Classes

There are several basic classes for the html library. The most basic class is CNCBINode, which is a node that knows how to contain and manipulate child CNCBINodes. Two main types of classes are derived from CNCBINode, text nodes and tag (or "element") nodes. The text nodes (CHTMLText and CHTMLPlainText) are intended to be used directly by the user, whereas the basic tag nodes (CHTMLNode, CHTMLElement, CHTMLOpenElement, and CHTMLListElement) are base classes for the nodes actually used to construct a page, such as CHTML_form.

CHTMLText and CHTMLPlainText are both used to insert text into the generated html, with the latter class performing HTML encoding before generation.

CHTMLNode is the base class for CHTMLElement (tags with close tags, like FORM), CHTMLOpenElement (tags without end tags, like BR) and CHTMLListElement (tags used in lists, like OL).

The following basic classes are discussed in more detail, next:

- CNCBINode
- CHTMLText
- CHTMLPlainText
- CHTMLNode
- CHTMLElement
- CHTMLOpenElement
- CHTMLListElement

CNCBINode

CNCBINode uses the following typedefs:**typedef list<CNCBINode*> TChildList typedef map<string, string> TAttributes**

CNCBINode* AppendChild(CNCBINode* child) Add a CNCBINode* to the end the list of child nodes. Returns *this so you can repeat the operation on the same line, e.g. Node->AppendChild(new CNCBINode)->AppendChild(new CNCBINode).

CNCBINode* AppendChild(CNodeRef& ref) Add a node by reference to the end the list of child nodes. Returns *this so you can repeat the operation on the same line.

void RemoveAllChildren(void) Removes all child nodes.

TChildList::iterator ChildBegin(void) TChildList::const_iterator ChildBegin(void) const Returns the first child.

TChildList::iterator ChildEnd(void) TChildList::const_iterator ChildEnd(void) const Returns the end of the child list (this is **not** the last child).

TChildList::iterator FindChild(CNCBINode* child) Find a particular child, otherwise return 0.

virtual CNcbiOstream& Print(CNcbiOstream& out) Create HTML from the node and all its children and send it to out. Returns a reference to out.

virtual void CreateSubNodes(void) This function is called during printing when the node has not been initialized. A newly created node is internally marked as not initialized. The intent of this function is for the user to replace it with a function that knows how to create all of the subchildren of the node. The main use of this function is in classes that define whole regions of pages.

const string& GetName(void) const void SetName(const string& namein) Get and set the name of the node.

bool HaveAttribute(const string& name) const Check for an attribute. Attributes are like the href in <a href="http://www.ncbi.nlm.nih.gov">

string GetAttribute(const string& name) const Return a copy of the attribute's value

const string* GetAttributeValue(const string& name) const Return a pointer to the attribute's value

void SetAttribute(const string& name, const string& value) void SetAttribute(const string& name) void SetAttribute(const string& name, int value) void SetOptionalAttribute(const string& name, const string& value) void SetOptionalAttribute(const string& name, bool set) void SetAttribute(const char* name, const string& value) void SetAttribute(const char* name) void SetAttribute(const char* name, int value) void SetOptionalAttribute(const char* name, const string& value) void SetOptionalAttribute(const char* name, bool set) Set an attribute. SetOptionalAttribute() only sets the attribute if value contains a string or is true.

CHTMLText

CHTMLText(const string& text)

This is a text node that can contain html tags, including tags of the form <@...@> which are replaced by CNCBINode's when printing out (this is discussed further in the CHTMLPage documentation).

const string& GetText(void) const void SetText(const string& text) Get and set the text in the node.

CHTMLPlainText

CHTMLPlainText(const string& text)

This node is for text that is to be HTML encoded. For example, characters like "&" are turned into "&amp;"

const string& GetText(void) const void SetText(const string& text)

Get and set text in the node.

CHTMLNode

CHTMLNode inherits from CNCBINode is the base class for html tags.

CHTMLNode* SetWidth(int width) CHTMLNode* SetWidth(const string& width) CHTMLNode* SetHeight(int height) CHTMLNode* SetHeight(const string& width) CHTMLNode* SetAlign(const string& align) CHTMLNode* SetBgColor(const string& color) CHTMLNode* SetColor(const string& color) Sets various attributes that are in common for many tags. Avoid setting these on tags that do not support these attributes. Returns *this so that the functions can be daisy chained:

```
CHTML_table * Table = new CHTML_table;
Table->SetWidth(400)->SetBgColor("#FFFFFF");
```

void AppendPlainText(const string &) Appends a CHTMLPlainText node. A plain text node will be encoded so that it does not contain any html tags (e.g. "<" becomes "<").

void AppendHTMLText(const string &) Appends a CHTMLTextNode. This type of node can contain HTML tags, i.e. it is not html encoded.

CHTMLElement

CHTMLElement is the base class for several tags that have the constructors with the common form:**CHTMLElement() CHTMLElement(CNCBINode* node) CHTMLElement(const string& text)** The second constructor appends node. The third constructor appends CHTMLText(const string& text).

The tags derived from this class include: CHTML_html, CHTML_head, CHTML_body, CHTML_base, CHTML_isindex, CHTML_link, CHTML_meta, CHTML_script, CHTML_style, CHTML_title, CHTML_address, CHTML_blockquote, CHTML_center, CHTML_div, CHTML_h1, CHTML_h2, CHTML_h3, CHTML_h4, CHTML_h5, CHTML_h6, CHTML_hr, CHTML_p, CHTML_pre, CHTML_dt, CHTML_dd, CHTML_li, CHTML_caption, CHTML_col, CHTML_colgroup, CHTML_thead, CHTML_tbody, CHTML_tfoot, CHTML_tr, CHTML_th, CHTML_td, CHTML_applet, CHTML_param, CHTML_cite, CHTML_code, CHTML_dfn, CHTML_em, CHTML_kbd, CHTML_samp, CHTML_strike, CHTML_strong, CHTML_var, CHTML_b, CHTML_big, CHTML_i, CHTML_s, CHTML_small, CHTML_sub, CHTML_sub, CHTML_sup, CHTML_tt, CHTML_u, CHTML_blink, CHTML_map, CHTML_area

CHTMLOpenElement

This is used for tags that do not have a close tag (like img). The constructors are of the same form as CHTMLElement. The tags derived from this class include: CHTML_pnop (paragraph tag without a close tag)

CHTMLListElement

These are elements used in a list.

CHTMLListElement(void) CHTMLListElement(bool compact) CHTMLListElement (const string& type) CHTMLListElement(const string& type, bool compact) Construct the ListElement with the given attibutes: TYPE and COMPACT. Both attributes affect the way the ListElement is displayed.

CHTMLListElement* AppendItem(const string& item) CHTMLListElement* AppendItem(CNCBINode* item) These functions add CHTMLText and CNCBINode items as children of the CHTMLListElement. The tags derived from this class include: CHTML_ul, CHTML_dir, CHTML_menu.

Specialized Tag Classes used in Forms

The rest of the sections deal with tag classes that have additional members or member functions that make the tags easier to use. In addition there are helper classes, such as CHTML_checkbox, that are easier to use instances of HTML tags.

The following specialized tag classes used in forms are discussed, next:

- CHTML_form: derived from CHTMLElement
- CHTML_input: derived from CHTMLOpenElement
- CHTML_checkbox: derived from CHTML_input
- CHTML_hidden: derived from CHTML_input
- CHTML_image: derived from CHTML_input
- CHTML_radio: derived from CHTML_input
- CHTML_reset: derived from CHTML_input
- CHTML_submit: derived from CHTML_input
- CHTML_text: derived from CHTML_input
- CHTML_select: derived from CHTMLElement
- CHTML_option: derived from CHTMLElement
- CHTML_textarea: derived from CHTMLElement

CHTML_form: derived from CHTMLElement

CHTML_form(const string& action = NcbiEmptyString, const string& method = NcbiEmptyString, const string& enctype = NcbiEmptyString) Add an HTML form tag with the given attributes. NCBIEmptyString is simply a null string.

void AddHidden(const string& name, const string& value) Add a hidden value to the form.

CHTML_input: derived from CHTMLOpenElement

CHTML_input(const string& type, const string& name) Create a input tag of the given type and name. Several of the following classes are specialized versions of the input tag, for example, CHTML_checkbox.

CHTML_checkbox: derived from CHTML_input

CHTML_checkbox(const string& name) CHTML_checkbox(const string& name, bool checked, const string& description = NcbiEmptyString) CHTML_checkbox(const

string& name, const string& value) CHTML_checkbox(const string& name, const string& value, bool checked, const string& description = NcbiEmptyString) Create a checkbox with the given attributes. This is an input tag with type = "checkbox".

CHTML_hidden: derived from CHTML_input

CHTML_hidden(const string& name, const string& value) Create a hidden value with the given attributes. This is an input tag with type = "hidden".

CHTML_image: derived from CHTML_input

CHTML_image(const string& name, const string& src) Create an image submit input tag. This is an input tag with type = "image".

CHTML_radio: derived from CHTML_input

CHTML_radio(const string& name, const string& value) CHTML_radio(const string& name, const string& value, bool checked, const string& description = NcbiEmptyString) Creates a radio button. Radio buttons are input tags with type = "radio button".

CHTML_reset: derived from CHTML_input

CHTML_reset(const string& label = NcbiEmptyString) Create a reset button. This is an input tag with type = "reset".

CHTML_submit: derived from CHTML_input

CHTML_submit(const string& name) CHTML_submit(const string& name, const string& label) Create a submit button. This is an input tag with type = "submit".

CHTML_text: derived from CHTML_input

CHTML_text(const string& name, const string& value = NcbiEmptyString) CHTML_text(const string& name, int size, const string& value = NcbiEmptyString) CHTML_text(const string& name, int size, int maxlength, const string& value = NcbiEmptyString) Create a text box. This is an input tag with type = "text".

CHTML_select: derived from CHTMLElement

CHTML_select(const string& name, bool multiple = false) CHTML_select(const string& name, int size, bool multiple = false) Create a selection tag used for drop-downs and selection boxes.

CHTML_select* AppendOption(const string& option, bool selected = false) CHTML_select* AppendOption(const string& option, const string& value, bool selected = false) Add an entry to the selection box by using the option tag. Returns *this to allow you to daisy-chain calls to AppendOption().

CHTML_option: derived from CHTMLElement

CHTML_option(const string& content, bool selected = false) CHTML_option(const string& content, const string& value, bool selected = false) The option tag used inside of select elements. See CHTML_select for an easy way to add option.

CHTML_textarea: derived from CHTMLElement

CHTML_textarea(const string& name, int cols, int rows) CHTML_textarea(const string& name, int cols, int rows, const string& value)

Create a textarea tag inside of a form.

Specialized Tag Classes used in Lists

These are specialized tag classes used in lists. See "Basic Classes" for non-specialized tag classes used in list.

The following specialized tag classes used in lists are discussed, next:

- CHTML_dl: derived from CHTMLElement
- CHTML_ol: derived from CHTMLListElement

CHTML_dl: derived from CHTMLElement

CHTML_dl(bool compact = false) Create a dl tag.

CHTML_dl* AppendTerm(const string& term, CNCBINode* definition = 0) CHTML_dl* AppendTerm(const string& term, const string& definition) CHTML_dl* AppendTerm(CNCBINode* term, CNCBINode* definition = 0) CHTML_dl* AppendTerm(CNCBINode* term, const string& definition) Append a term and definition to the list by using DD and DT tags.

CHTML_ol: derived from CHTMLListElement

CHTML_ol(bool compact = false) CHTML_ol(const string& type, bool compact = false) CHTML_ol(int start, bool compact = false) CHTML_ol(int start, const string& type, bool compact = false) The last two constructors let you specify the starting number for the list.

Other Specialized Tag Classes

These tag classes that have additional members or member functions that make the tags easier to use. The following classes are discussed next:

- CHTML_table: derived from CHTMLElement
- CHTML_a: derived from CHTMLElement
- CHTML_img: derived from CHTMLOpenElement
- CHTML_font: derived from CHTMLElement
- CHTML_color: derived from CHTMLElement
- CHTML_br: derived from CHTMLOpenElement
- CHTML_basefont: derived from CHTMLElement

CHTML_table: derived from CHTMLElement

CNCBINode* Cell(int row, int column) This function can be used to specify the size of the table or return a pointer to a particular cell in the table. Throws a runtime_error exception when the children of the table are not TR or the children of each TR is not TH or TD or there are more columns than should be.

int CalculateNumberOfColumns(void) const int CalculateNumberOfRows(void) const Returns number of columns and number of rows in the table.

CNCBINode* InsertAt(int row, int column, CNCBINode* node) CNCBINode* InsertTextAt(int row, int column, const string& text) Inserts a node or text in the table. Grows the table if the specified cell is outside the table. Uses Cell() so can throw the same exceptions.

void ColumnWidth(CHTML_table*, int column, const string & width) Set the width of a particular column.

CHTML_table* SetCellSpacing(int spacing) CHTML_table* SetCellPadding(int padding) Set the cellspacing or cellpadding attributes.

CHTML_a: derived from CHTMLElement

CHTML_a(const string& href, const string& text) CHTML_a(const string& href, CNCBINode* node) Creates a hyperlink that contains the given text or node.

CHTML_img: derived from CHTMLOpenElement

CHTML_img(const string& url) CHTML_img(const string& url, int width, int height) Creates an image tag with the given attributes.

CHTML_font: derived from CHTMLElement

CHTML_font(void) CHTML_font(int size, CNCBINode* node = 0) CHTML_font(int size, const string& text) CHTML_font(int size, bool absolute, CNCBINode* node = 0) CHTML_font(int size, bool absolute, const string& text) CHTML_font(const string& typeface, CNCBINode* node = 0) CHTML_font(const string& typeface, const string& text) CHTML_font(const string& typeface, int size, CNCBINode* node = 0) CHTML_font(const string& typeface, int size, const string& text) CHTML_font(const string& typeface, int size, bool absolute, CNCBINode* node = 0) CHTML_font(const string& typeface, int size, bool absolute, const string& text) Create a font tag with the given attributes. Appends the given text or node. Note that it is cleaner and more reusable to use a stylesheet than to use the font tag.

void SetRelativeSize(int size) Set the size of the font tag.

CHTML_color: derived from CHTMLElement

CHTML_color(const string& color, CNCBINode* node = 0) CHTML_color(const string& color, const string& text) Create a font tag with the given color and append either node or text.

CHTML_br: derived from CHTMLOpenElement

CHTML_br(void) CHTML_br(int number) The last constructor lets you insert multiple BR tags.

CHTML_basefont: derived from CHTMLElement

CHTML_basefont(int size) CHTML_basefont(const string& typeface) CHTML_basefont(const string& typeface, int size) Set the basefont for the page with the given attributes.

Generating Web Pages with the HTML classes

Web applications involving interactions with a client via a complex HTML interface can be difficult to understand and maintain. The NCBI C++ Toolkit classes decouple the complexity of interacting with a CGI client from the complexity of generating HTML output by defining separate class hierarchies for these activities. In fact, one useful application of the HTML classes is to generate web pages "offline".

The chapter on Developing CGI Applications discussed only the activities involved in processing the client's request and generating a response. This section introduces the C++ Toolkit components that support the creation of HTML pages, and concludes with a brief consideration of how the HTML classes can be used in consort with a running CCgiApplication. Further discussion of combining a CGI application with the HTML classes can be found in the section on An example web-based CGI application. See also NCBI C++ HTML Classes in the Reference Manual.

The following topics are discussed in this section:

The CNCBINode (*) class

All of the HTML classes are derived from the CNCBINode class, which in turn, is derived from the CObject class. Much of the functionality of the many derived subclasses is implemented by the CNCBINode base class. The CNCBINode class has just three data members:

- m_Name - a string, used to identify the type of node or to store text data
- m_Attributes - a map<string, string> of properties for this node
- m_Children - a list of subnodes embedded (at run-time) in this node

The m_Name data member is used differently depending on the type of node. For HTML text nodes, m_Name stores the actual body of text. For CHTMLElement objects, m_Name stores the HTML tagname that will be used in generating HTML formatted output.

The m_Attributes data member provides for the encoding of specific features to be associated with the node, such as background color for a web page. A group of "Get/SetAttribute" member functions are provided for access and modification of the node's attributes. All of the "SetAttribute" methods return this - a pointer to the HTML node being operated on, and so, can be daisy-chained, as in:

```
table->SetCellSpacing(0)->SetBgColor("CCCCCC");
```

Care must be taken however, in the order of invocations, as the object type returned by each operation is determined by the class in which the method is defined. In the above example, table is an instance of CHTML_table, which is a subclass of CNCBINode - where SetBgColor () is defined. The above expression then, effectively executes:

```
table->SetCellSpacing(0);
table->SetBgColor("CCCCCC");
```

In contrast, the expression:

```
table->SetBgColor("CCCCCC")->SetCellSpacing(0);
```

would fail to compile, as it would effectively execute:

```
table->SetBgColor("CCCCCC");
(CNCBINode*)table->SetCellSpacing(0);
```

since the method SetCellSpacing() is undefined for CNCBINode() objects.

The m_Children data member of CNCBINode stores a dynamically allocated list of CNCBINode subcomponents of the node. In general, the in memory representation of each node is a graph of CNCBINode objects (or subclasses thereof), where each object may in turn contain additional CNCBINode children. For example, an unordered list is represented as a CHTML_ul (<ul>) element containing CHTML_li (<li>) subcomponents.

A number of member functions are provided to operate on m_Children. These include methods to access, add, and remove children, along with a pair of begin/end iterators (ChildBegin() and ChildEnd()), and a function to dereference these iterators (Node(i)).

Depending on flags set at compile time, m_Children is represented as either a list of CNodeRef objects, or a list of auto_ptr<CNodeRef>, where CNodeRef is a typedef for CRef<CNCBINode>. This distinction is transparent to the user however, and the important point is that the deallocation of all dynamically embedded child nodes is handled automatically by the containing class.

CNCBINode::Print() recursively generates the HTML text for the node and all of its children, and outputs the result to a specified output stream. The Print() function takes two arguments: (1) an output stream, and (2) a CNCBINode::TMode object, where TMode is an internal class defined inside the CNCBINode class. The TMode object is used by the print function to determine what type of encoding takes place on the output, and in some cases, to locate the containing parent node.

Many of the CNCBINode objects do not actually allocate their embedded subnodes until the Print() method is invoked. Instead, a kind of lazy evaluation is used, and the information required to install these nodes to m_Children is used by the CreateSubNodes() method only when output has been requested (see discussion below).

A slice of the NCBI C++ Toolkit class hierarchy rooted at the CNCBINode class includes the following directly derived subclasses:

- CNCBINode:
  - CSmallPagerBox
  - CSelection
  - CPagerBox
  - CPager
  - CHTMLText
  - CHTMLTagNode
  - CHTMLPlainText
  - CHTMLNode
  - CHTMLDualNode
  - CHTMLBasicPage

  - CButtonList

Many of these subclasses make little sense out of context, as they are designed for use as subcomponents of, for example, a CHTMLPage. Exceptions to this are the text nodes, described next.

HTML Text nodes: CHTMLText (*) and CHTMLPlainText (*)

The CHTMLText class uses the m_Name data member (inherited from CNCBINode) to store a text string of arbitrary length. No new data members are introduced, but two new member functions are defined. SetText() resets m_Name to a new string, and GetText() returns the value currently stored in m_Name. With the exception of specially tagged sections (described below), all text occurring in a CHTMLText node is sent directly to the output without further encoding.

The CHTMLPlainText class is provided for text that may require further encoding. In addition to the SetText() and GetText() member functions described for the CHTMLText class, one new data member is introduced. m_NoEncode is a Boolean variable that designates whether or not the text should be further encoded. NoEncode() and SetNoEncode() allow for access and modification of this private data member. For example:

```
(new CHTMLText("<br> testing BR <br>"))->Print(cout);
```

will generate the output:

```
testing BR
```

whereas:

```
(new CHTMLPlainText("<br> testing BR <br>"))->Print(cout);
```

will generate:

```
<br> testing BR <br>
```

The text in the CHTMLText node is output verbatim, and the web browser interprets the <br> tags as line breaks. In contrast, the CHTMLPlainText node effectively "insulates" its content from the browser's interpretation by encoding the <br> tags as "<br&gt;".

CHTMLText nodes also play a special role in the implementation of page nodes that work with template files. A tagname in the text is delimited by "<@" and "@>", as in: <@tagname@>. This device is used for example, when working with template files, to allow additional nodes to be inserted in a pre-formatted web page. The CHTMLText::PrintBegin() method is specialized to skip over the tag names and their delimiters, outputting only the text generated by the nodes that should be inserted in that tagged section. Further discussion of this feature is deferred until the section on the NCBI page classes, which contain a TTagMap.

The NCBI Page classes

The page classes serve as generalized containers for collections of other HTML components, which are mapped to the page by a tagmap. In general, subcomponents are added to a page using the AddTagMap() method (described below), instead of the AppendChild() method. The page classes define the following subtree in the C++ Toolkit class hierarchy:

- CHTMLBasicPage
  - CHTMLPage

In addition to the data members inherited from CNCBINode, three new private data members are defined in the CHTMLBasicPage class.

- m_CgiApplication - a pointer to the CCgiApplication
- m_Style - an integer flag indicating subcomponents to display/suppress (e.g., Title)
- m_TagMap (see discussion)

In effect, m_TagMap is used to map strings to tagged subcomponents of the page - some of which may not have been instantiated yet. Specifically, m_TagMap is defined as a TTagMap variable, which has the following type definition:

```
typedef map<string, BaseTagMapper*> TTagMap;
```

Here, BaseTagMapper is a base class for a set of functor-like structs. Each of the derived subclasses of BaseTagMapper has a single data member (e.g. m_Node, m_Function or m_Method), which points to either a CNCBINode, or a function that returns a pointer to a CNCBINode. The BaseTagMapper class also has a single member function, MapTag(), which knows how to "invoke" its data member.

The simplest subclass of BaseTagMapper is the ReadyTagMapper class whose sole data member, m_Node, is a CRef pointer to a CNCBINode. In this case the MapTag() function simply returns &*m_Node. Several different types of tagmappers are derived from the BaseTagMapper class in nodemap.hpp. Each of these subclasses specializes a different type of data member, which may be a pointer to a free function, a pointer to a member function, or a pointer to an object, as in the case of the ReadyTagMapper. The action taken by the tagmapper's MapTag() method in order to return a pointer to a CNCBINode is implemented accordingly.

The CHTMLBasicPage class also has a member function named MapTag(), which is used in turn, to invoke a tagmapper's MapTag() method. Specifically, CHTMLBasicPage::MapTag (tagname) first locates the installed tagmapper associated with tagname, m_TagMap[tagname]. If an entry is found, that tagmapper's MapTag() member function is then invoked, which finally returns a pointer to a CNCBINode.

A second member function, CHTMLBasicPage::AddTagMap(str, obj), provides for the insertion of a new tag string and its associated tagmapper struct to m_TagMap. Depending on the object type of the second argument, a type-specific implementation of an overloaded helper function, CreateTagMapper(), can be used to install the desired tagmapper.

In order for a new mapping to have any effect however, the tag must also occur in one of the nodes installed as a child of the page. This is because the Print() methods for the page nodes do virtually nothing except invoke the Print() methods for m_Children. The m_TagMap data member, along with all of its supporting methods, is required for the usage of template files, as described in the next section.

The primary purpose of the CHTMLBasicPage is as a base class whose features are inherited by the CHTMLPage class - it is not intended for direct usage. Important inherited features include its three data members: m_CgiApplication, m_Style, and m_TagMap, and its member functions: Get/SetApplication(), Get/SetStyle(), MapTag(), and AddTagMap(). Several of the more advanced HTML components generate their content via access of the running CGI application. For example, see the description of a CSelection node. It is not strictly necessary to specify a CGI application when instantiating a page object however, and constructors are available that do not require an application argument.

Using the CHTMLPage class with Template Files

The CHTMLPage class is derived from the CHTMLBasicPage. In combination with the appropriate template file, this class can be used to generate the standard NCBI web page, which includes:

- the NCBI logo
- a hook for the application-specific logo
- a top menubar of links to several databases served by the query program
- a links sidebar for application-specific links to relevant sites
- a VIEW tag for the application's web interface
- a bottom menubar for help links, disclaimers, etc.

The template file is a simple HTML text file with one extension -- the use of named tags (<@tagname@>) which allow the insertion of new HTML blocks into a pre-formatted page. The standard NCBI page template file contains one such tag, VIEW.

The CHTMLPage class introduces two new data members: m_Title (string), which specifies the title for the page, and m_TemplateFile (string), which specifies a template file to load. Two constructors are available, and both accept string arguments that initialize these two data members. The first takes just the title name and template file name, with both arguments being optional. The other constructor takes a pointer to a CCgiApplication and a style (type int), along with the title and template_file names. All but the first argument are optional for the second constructor. The member functions, SetTitle() and SetTemplateFile(), allow these data members to be reset after the page has been initialized.

Five additional member functions support the usage of template files and tagnodes as follows:

- CreateTemplate() reads the contents of file m_TemplateFile into a CHTMLText node, and returns a pointer to that node.
- CreateSubNodes() executes AppendChild(CreateTemplate()), and is called at the top of Print() when m_Children is empty. Thus, the contents of the template file are read into the m_Name data member of a CHTMLText node, and that node is then installed as a child in the page's m_Children.
- CreateTitle() returns new CHTMLText(m_Title).
- CreateView() is effectively a virtual function that must be redefined by the application. The CHTMLPage class definition returns a null pointer (0).
- Init() is called by all of the CHTMLPage constructors, and initializes m_TagMap as follows:
  ```
  void CHTMLPage::Init(void)
  {
  AddTagMap("TITLE", CreateTagMapper(this, &CHTMLPage::CreateTitle));
  AddTagMap("VIEW", CreateTagMapper(this, &CHTMLPage::CreateView));
  }
  ```
  As described in the preceding section, CreateTagMapper() is an overloaded function that creates a tagmapper struct. In this case, CreateTitle() and CreateView() will be installed as the m_Method data members in the resulting tagmappers. In general, the type of struct created by CreateTagMapper depends on the argument types to that function. In its usage here, CreateTagMapper is a template function, whose arguments

are a pointer to an object and a pointer to a class method:
template<class C>
BaseTagMapper* CreateTagMapper(const C*, CNCBINode* (C::*method)(void)) {
return new TagMapper<C>(method);
}
The value returned is itself a template object, whose constructor expects a pointer to a method (which will be used as a callback to create an object of type C). Here, AddTagMap() installs CreateTitle() and CreateView() as the data member for the tagmapper associated with tag "TITLE" and tag "VIEW", respectively.

An example using the NCBI standard template file should help make these concepts more concrete. The following code excerpt uses the standard NCBI template and inserts a text node at the VIEW tag position:

```
#include <html/html.hpp>
#include <html/page.hpp>
USING_NCBI_SCOPE;
int main()
{
 try {
 CHTMLPage *Page = new CHTMLPage("A CHTMLPage!", "ncbi_page.html");
 Page->AddTagMap( "VIEW",
 new CHTMLText("Insert this string at VIEW tag"));
 Page->Print(cout);
 cout.flush();
 return 0;
 }
 catch (exception& exc) {
 NcbiCerr << "\n" << exc.what() << NcbiEndl;
 }
 return 1;
}
```

The name of the template file is stored in m_TemplateFile, and no further action on that file will be taken until Page->Print(cout) is executed. The call to AddTagMap() is in a sense then, a forward reference to a tag that we know is contained in the template. Thus, although a new CHTMLText node is instantiated in this statement, it is not appended to the page as a child, but is instead "mapped" to the page's m_TagMap where it is indexed by "VIEW".

The contents of the template file will not be read until Print() is invoked. At that time, the text in the template file will be stored in a CHTMLText node, and when that node is in turn printed, any tag node substitutions will then be made. More generally, nodes are not added to the page's m_Children graph until Print() is executed. At that time, CreateSubNodes() is invoked if m_Children is empty. Finally, the actual mapping of a tag (embedded in the template) to the associated TagMapper in m_TagMap, is executed by CHTMLText::PrintBegin().

The CHTMLPage class, in combination with a template file, provides a very powerful and general method for generating a "boiler-plate" web page which can be adapted to application-specific needs using the CHTMLPage::AddTagMap() method. When needed, The user can edit the template file to insert additional <@tagname@> tags. The AddTagMap() method is defined **only** for page objects however, as they are the only class having a m_TagMap data member.

Before continuing to a general discussion of tagnodes, let's review how the page classes work in combination with a template file:

- A page is first created with a title string and a template file name. These arguments are stored directly in the page's data members, m_Title and m_TemplateFile.
- The page's Init() method is then called to establish tagmap entries for "TITLE" and "VIEW" in m_TagMap.
- Additional HTML nodes which should be added to this page are inserted using the page's AddTagMap(tagname, *node) method, where the string tagname appears in the template as "<@tagname@>". Typically, a CGI application defines a custom implementation of the CreateView() method, and installs it using AddTagMap ("VIEW", CreateView()).
- When the page's Print() method is called, it first checks to see if the page has any child nodes, and if so, assumes there is no template loaded, and simply calls PrintChildren (). If there are no children however, page->CreateSubNodes() is called, which in turn calls the CreateTemplate() method. This method simply reads the contents of the template file and stores it directly in a CHTMLText node, which is installed as the only child of the parent page.
- The page's Print() method then calls PrintChildren(), which (eventually) causes CHTMLText::PrintBegin() to be executed. This method in turn, encodes special handling of "<@tagname@>" strings. In effect, it repeatedly outputs all text up to the first "@" character; extracts the tagname from the text; searches the parent page's m_TagMap to find the TagMapper for that tagname, and finally, calls Print() on the HTML node returned by the TagMapper. CHTMLText::PrintBegin() continues in this fashion until the end of its text is reached.

NOTE: appending any child nodes directly to the page prior to calling the Print() method will make the template effectively inaccessible, since m_Children() will not be empty. For this reason, the user is advised to use AddTagNode() rather than AppendChild() when adding subcomponents.

The CHTMLTagNode (*) class

The objects and methods described to this point provide no mechanisms for dynamically adding tagged nodes. As mentioned, the user is free to edit the template file to contain additional <@tag@> names, and AddTagMap() can then be used to associate tagmappers with these new tags. This however, requires that one know ahead of time how many tagged nodes will be used. The problem specifically arises in the usage of template files, as it is not possible to add child nodes directly to the page without overriding the the template file.

The CHTMLTagNode class addresses this issue. Derived directly from CNCBINode, the class's constructor takes a single (string or char*) argument, tagname, which is stored as m_Name. The CHTMLTagNode::PrintChildren() method is specialized to handle tags, and makes a call to MapTagAll(GetName(), mode). Here, GetName() returns the m_Name of the CHTMLTagNode, and mode is the TMode argument that was passed in to PrintChildren(). In addition to an enumeration variable specifying the mode of output, a TMode object has a pointer to the parent node that invoked PrintChildren(). This pointer is used by MapTagAll(), to locate a parent node whose m_TagMap has an installed tagmapper for the tagname. The TMode object's parent pointer essentially implements a stack which can be used to retrace the dynamic chain of PrintChildren() invocations, until either a match is found or the end of the call stack is reached. When a match is found, the associated tagmapper's MapTag() method is invoked, and Print() is applied to the node returned by this function.

The following example uses an auxillary CNCBINode(tagHolder) to install additional CHTMLTagNode objects. The tags themselves however, are installed in the containing page's m_TagMap, where they will be retrieved by the MapTagAll() function, when PrintChildren() is called for the auxillary node. That node in turn, is mapped to the page's VIEW tag. When the parent page is "printed", CreateSubNodes() will create a CHTMLText node. The text node will hold the contents of the template file and be appended as a child to the page. When PrintBegin() is later invoked for the text node, MapTagAll() associates the VIEW string with the CNCBINode, and in turn, calls Print() on that node.

```
#include <html/html.hpp>
#include <html/page.hpp>
USING_NCBI_SCOPE;
int main()
{
 try {
 CHTMLPage *Page = new CHTMLPage("myTitle", "ncbi_page.html");
 CNCBINode *tagHolder = new CNCBINode();
 Page->AddTagMap( "VIEW", tagHolder);
 tagHolder->AppendChild(new CHTMLTagNode("TAG1"));
 tagHolder->AppendChild(new CHTML_br());
 tagHolder->AppendChild(new CHTMLTagNode("TAG2"));
 Page->AddTagMap( "TAG1",
 new CHTMLText("Insert this string at TAG1"));
 Page->AddTagMap( "TAG2",
 new CHTMLText("Insert another string at TAG2"));
 Page->Print(cout);
 cout.flush();
 return 0;
 }
 catch (exception& exc) {
 NcbiCerr << "\n" << exc.what() << NcbiEndl;
 }
 return 1;
}
```

The CHTMLNode (*) class

CHTMLNode is derived directly from the CNCBINode class, and provides the base class for all elements requiring HTML tags (e.g., <ul>,<br>, <img>, <table>, etc.). The class interface includes several constructors, all of which expect the first argument to specify the HTML tagname for the node. This argument is used by the constructor to set the m_Name data member. The optional second argument may be either a text string, which will be appended to the node using AppendPlainText(), or a CNCBINode, which will be appended using AppendChild().

A uniform system of class names is applied; each subclass derived from the CHTMLNode base class is named CHTML_[tag], where [tag] is the HTML tag in lowercase, and is always preceded by an underscore. The NCBI C++ Toolkit hierarchy defines roughly 40 subclasses of CHTMLNode - all of which are defined in the Quick Reference Guide at the end of this section. The constructors for "empty" elements, such as CHTML_br, which have no assigned values, are simply invoked as CHTML_br(). The Quick Reference Guide provides brief explanations of each class, along with descriptions of the class constructors.

In addition to the subclasses explicitly defined in the hierarchy, a large number of lightweight subclasses of CHTMLNode are defined by the preprocessor macro DECLARE_HTML_ELEMENT(Tag, Parent) defined in html.hpp. All of these elements have the same interface as other CHTMLNode classes however, and the distinction is invisible to the user.

A rich interface of settable attributes is defined in the base class, and is applicable to all of the derived subclasses, including those implemented by the preprocessor macros. Settable attributes include: class, style, id, width, height, size, alignment, color, title, accesskey, and name. All of the SetXxx() functions which set these attributes return a this pointer, cast as CHTMLNode*.

The CHTMLDualNode (*) class

CHTMLDualNode is derived directly from the CNCBINode class, and provides the base class for all elements requiring different means for displaying data in eHTML and ePlainText modes.

This class interface includes several constructors. The second argument in these constructors specifies the alternative text to be displayed in ePlainText mode. The first argument of these constructors expects HTML text or pointer to an object of (or inherited from) CNCBINode class. It will be appended to the node using AppendChild() method, and printed out in eHTML mode. For example:

```
(new CHTMLDualNode(new CHTML_p("text"),"\nTEXT \n"))->Print(cout);
```

will generate the output:

```
<p>text</p>
```

whereas:

```
(new CHTMLDualNode(new CHTML_p("text"),"\n TEXT \n"))
->Print(cout, CNCBINode::ePlainText);
```

will generate:

```
\n TEXT \n
```

The CHTMLPopupMenu (*) class

CHTMLPopupMenu is a class for support JavaScript-based popup menu's in the HTML framework. It is derived directly from the CNCBINode class, The HTML pages using it can be viewed only in browsers with supporting JavaScript version 1.2 (or higher) and CSS (Cascading Style Sheets).

CHTMLPopupMenu support two popup menu types (CHTMLPopupMenu::EType):

- eSmith - developed by Gary Smith;
- eKurdin - developed by Sergey Kurdin.

We use slightly modified Smith's menu ncbi_menu_dnd.js. This version have the following differences from the original:

- Added support for dynamic menu (all menues use one container);
- Added automatic menu adjustment in the browser window;

- Turned off dragging possibility;
- Fixed some errors.

The type of menu can be specified by second argument of CHTMLPopupMenu constructor:

```
CHTMLPopupMenu(const string& name, EType type = eSmith);
```

By default, the "old" (Smith's) popup menu will be used. The first argument of constructor defines name of the menu. Each menu **must** use unique name, because this name will be as name of JavaScript variable.

To add items into menu class CHTMLPopupMenu have method AddItem(). It's two first parameters are more useful and define item's title and action to be performed on click. The action must be any valid javascript code or just URL. In latter case it must begin with "http://" string.

You can change menu style using menu attributes. Each attribute have effect only for specified menu type (CHTMLPopupMenu::EType), otherwise it will be ignored.

To attach popup menu to a HTML node a method CHTMLNode::AttachPopupMenu() can be used. This method works with both menu types.

By default, menu use javascript libraries from the NCBI site. But this behaviour can be changed using method EnablePopupMenu() of classes CHTML_html and CHTMLPage. This method also forcibly enabled using popup menues on the HTML page. If we wish to use default javascript libraries than we can skip call of this function. In this case menues will be enabled automagicaly (for each type separately) if they are used on page.

An example of popup menu usage should help to make these concepts more clear. The following code creates HTML page with two different menues:

```
// Create HTML page skeleton with HEAD and BODY
CHTML_html* html = new CHTML_html;
CHTML_head* head = new CHTML_head;
CHTML_body* body = new CHTML_body;
html->AppendChild(head);
html->AppendChild(body);
// Create one menu (Smith's menu by default)
CHTMLPopupMenu* m1 = new CHTMLPopupMenu("Menu1");
m1->AddItem("Red" , "document.bgColor='red'");
m1->AddItem("White" , "document.bgColor='white'");
m1->AddSeparator();
m1->AddItem("Green", "document.bgColor='green'");
m1->SetAttribute(eHTML_PM_fontColor, "black");
m1->SetAttribute(eHTML_PM_fontColorHilite, "yellow");
// We can add menu to the BODY only!
body->AppendChild(m1);
// Create another menu
CHTMLPopupMenu* m2 = new CHTMLPopupMenu("Menu2",CHTMLPopupMenu::eKurdin);
m2->AddItem("NCBI", "http://ncbi.nlm.nih.gov");
m2->AddItem("Netscape", "http://www.netscape.com");
m2->AddItem("Microsoft", "top.location='http://www.microsoft.com'");
m2->SetAttribute(eHTML_PM_titleColor, "yellow");
```

```
m2->SetAttribute(eHTML_PM_alignV, "top");
body->AppendChild(m2);
// Add menus call
CHTML_a* anchor1 = new CHTML_a("#","Smith's Menu");
anchor1->AttachPopupMenu(m1, eHTML_EH_Click);
CHTML_a* anchor2 = new CHTML_a("#","Kurdin's Menu");
anchor2->AttachPopupMenu(m2);
body->AppendChild(anchor1);
body->AppendChild(new CHTML_p(""));
body->AppendChild(anchor2);
// Enable using popup menus (we can skip call this function)
//html->EnablePopupMenu(CHTMLPopupMenu::eSmith);
//html->EnablePopupMenu(CHTMLPopupMenu::eKurdin);

// Print page in the HTML format
html->Print(cout);
```

Note: We must add menues to a BODY only, otherwise menu not will work.

Using the HTML classes with a CCgiApplication object

The previous chapter described the NCBI C++ Toolkit's CGI classes, with an emphasis on their independence from the HTML classes. In practice however, a real application must employ both types of objects, and they must communicate with one another. The only explicit connection between the CGI and HTML components is in the HTML page classes, whose constructors accept a CCgiApplication as an input parameter. The open-ended definition of the page's m_TagMap data member also allows the user to install tagmapper functions that are under control of the application, thus providing an "output port" for the application. In particular, an application-specific CreateView() method can easily be installed as the function to be associated with a page's VIEW tag. The CGI sample program provides a simple example of using these classes in coordination with each other.

Supplementary Information

The following topics are discussed in this section:

- The CNCBINode::TMode class
- Quick Reference Guide

The CNCBINode::TMode class

TMode is an internal class defined inside the CNCBINode class. The TMode class has three data members defined:

- EMode m_Mode - an enumeration variable specifying eHTML (0) or ePlainText (1) output encoding
- CNCBINode* m_Node - a pointer to the CNCBINode associated with this TMode object
- TMode* m_Previous - a pointer to the TMode associated with the parent of m_Node

Print() is implemented as a recursive function that allows the child node to dynamically "inherit" its mode of output from the parent node which contains it. Print() outputs the current node using PrintBegin(), recursively prints the child nodes using PrintChildren(), and concludes with a call to PrintEnd(). TMode objects are created dynamically as needed, inside the Print() function. The first call to Print() from say, a root Page node, generally specifies the output stream only, and uses a default eHTML enumeration value to initialize a TMode object. The TMode constructor in this case is:

```
TMode(EMode m = eHTML): m_Mode(m), m_Node(0), m_Previous(0) {}
```

The call to Print() with no TMode argument automatically calls this default constructor to create a TMode object which will then be substituted for the formal parameter prev inside tbe Print() method. One way to think of this is that the initial print call - which will ultimately be propagated to all of the child nodes - is initiated with a "null parent" TMode object that only specifies the mode of output.

```
CNcbiOstream& CNCBINode::Print(CNcbiOstream& os, TMode prev)
{
 // ...

 TMode mode(&prev, this);

 PrintBegin(os, mode);
 try {
 PrintChildren(out, mode);
 }
 catch (...) {
 // ...
 }
 PrintEnd(os, mode); }
```

In the first top-level call to Print(), prev is the default TMode object described above, with NULL values for m_Previous and m_Node. In the body of the Print() method however, a new TMode is created for subsequent recursion, with the following constructor used to create the new TMode at that level:

```
TMode(const TMode* M, CNCBINode* N) : m_Mode(M->m_Mode),m_Node(N),
m_Previous(M) {}
```

where M is the TMode input parameter, and N is the current node.

Thus, the output encoding specified at the top level is propagated to the PrintXxx() methods of all the child nodes embedded in the parent. The CNCBINode::PrintXxx() methods essentially do nothing;PrintBegin() and PrintEnd() simply return 0, and PrintChildren() just calls Print() on each child. Thus, the actual printing is implemented by the PrintBegin() and PrintEnd() metwebpgs.html_CHTMLBasicPaghods that are specialized by the child objects.

As the foregoing discussion implies, a generic CNCBINode which has no children explicitly installed will generate no output. For example, a CHTMLPage object which has been initialized by loading a template file has no children until they are explicitly created. In this case, the Print () method will first call CreateSubNodes() before executing PrintChildren(). The use of template files, and the associated set of TagMap functions are discussed in the section on the NCBI Page classes.

Quick Reference Guide

The following is a quick reference guide to the HTML and related classes:

- CNCBINode
  - CButtonList
  - CHTMLBasicPage
    - CHTMLPage
- CHTMLNode
  - CHTMLComment
  - CHTMLOpenElement
    - CHTML_br
    - CHTML_hr
    - CHTML_img
    - CHTML_input
      - CHTML_checkbox
      - CHTML_file
      - CHTML_hidden
      - CHTML_image
      - CHTML_radio
      - CHTML_reset
      - CHTML_submit
      - CHTML_text
- CHTMLElement
  - CHTML_a
  - CHTML_basefont CHTML_button
  - CHTML_dl
  - CHTML_fieldset
  - CHTML_font
    - CHTML_color
- CHTML_form
- CHTML_label
- CHTML_legend
- CHTML_option
- CHTML_select
- CHTML_table
  - CPageList
  - CPagerView
  - CQueryBox
- CHTML_tc
- CHTML_textarea
- CHTML_tr
- CHTMLListElement

  - CHTML_dir
  - CHTML_menu
  - CHTML_ol
  - CHTML_ul
- CHTMLPlainText
- CHTMLTagNode
- CHTMLDualNode
  - CHTMLSpecialChar
- CHTMLText
- CHTMLPopupMenu
- CPager
- CPagerBox
- CSelection
- CSmallPagerBox
- CButtonList (Custom feature not for general use.) Derived from CNCBINode; defined in components.hpp. An HTML select button with a drop down list; used in CPagerBox. The constructor takes no arguments, and child nodes (options) are added using method CbuttonList::CreateSubNodes()
- CHTML_a Derived from CHTMLElement, defined in html.hpp - an HTML anchor element, as used in <*a href="..."*>. The constructor takes the URL string as the argument, and optionally, a CNCBINode to be appended as a child node. The label inserted before the closing tag (</a>) can thus be specified by providing a CHTMLText node to the constructor, or by using the AppendChild() after the anchor has been created.
- CHTML_basefont Derived from CHTMLElement, defined in html.hpp - an HTML basefont element used to define the font size and/or typeface for text embedded in this node by AppendChild(). The constructor expects one to two arguments specifying size, typeface, or both.
- CHTML_br Derived from CHTMLOpenElement, defined in html.hpp - the HTML component used to insert line breaks. The constructor takes no arguments.
- CHTML_checkbox Derived from CHTML_input, defined in html.hpp - can only be used inside a CHTML_form; the HTML component for a checkbox. The constructor takes up to four arguments specifying the name (string), value (string), state (bool), and description (string) for the node.
- CHTML_color Derived from CHTML_font, defined in html.hpp - an HTML font color element. Two constructors are available, and both expect string color as the first argument. If no other argument is provided, a NULL CNCBINode is assumed for the second argument, and text can be added to the node using AppendChild(). An alternative constructor accepts a simple string text argument.
- CHTML_dir Derived from CHTMLListElement, defined in html.hpp - the HTML component used to insert a dir list. The constructor takes zero to two arguments; if no arguments are provided, the compact attribute is by default false, and the type attribute is left to the browser. CHTML_dir("square", true) will create a compact dir element with square icons. Items can be added to the list using AppendChild(new CHTMLText ("<li>...").

- CHTML_dl Derived from CHTMLElement, defined in html.hpp - an HTML glossary list. The constructor takes a single bool argument; if no arguments are provided, the compact attribute is by default false. Terms are added to the list using AppendTerm ().
- CHTML_fieldset Derived from CHTMLElement, defined in html.hpp - an element that groups related form controls (such as checkboxes, radio buttons, etc.) together to define a form control group. The constructors take at most 1 argument, which may be either a string or a CHTML_legend node. If the argument is a string, then it is used to create a CHTML_legend node for the fieldset. The individual form controls to be included in the group are specified using the AppendChild() method.
- CHTML_file Derived from CHTML_input, defined in html.hpp - used only inside a CHTML_form - a form input type to create a file widget for selecting files to be sent to the server. The constructor takes a string name and an optional string value.
- CHTML_font Derived from CHTMLElement, defined in html.hpp - an HTML font element. The constructor takes up to four arguments. The first three arguments specify the font typeface and size, along with a Boolean value indicating whether the given font size is absolute or relative. The last argument is either a string or a CNCBINode containing text. Additional text should be added using the AppendChild() method.
- CHTML_form Derived from CHTMLElement, defined in html.hpp - an HTML form node with two constructors. The first takes the URL string (for submission of form data) and method (CHTML::eGet or CHTML::ePost), and the AppendChild() method is used to add nodes. The second constructor takes three arguments, specifying the URL, an HTML node to append to the form, and the enumereated get/post method.
- CHTML_hidden Derived from CHTML_input, defined in html.hpp - used only inside a CHTML_form - the HTML node for adding hidden key/value pairs to the data that will be submitted by an CHTML_form. The constructor takes a name string and a value, where the latter may be either a string or an int.
- CHTML_hr Derived from CHTMLOpenElement, defined in html.hpp - the HTML component used to insert a horizontal rule. The constructor takes up to three arguments, specifying the size, width and shading to be used in the display.
- CHTML_image Derived from CHTML_input, defined in html.hpp - used only inside a CHTML_form - the HTML component used to add an inline active image to an HTML form. Clicking on the image submits the form data to the CHTML_form's URL. The constructor takes three arguments, specifying the name of the node, the URL string for the image file, and a Boolean value (optional) indicating whether or not the displayed image should have a border.
- CHTML_img Derived from CHTMLOpenElement, defined in html.hpp - an HTML img component for adding an inline image to a web page. The constructor takes a single URL string argument for the image's src. The alternative constructor also accepts two integer arguments specifying the width and height of the displayed image.
- CHTML_input Derived from CHTMLOpenElement, defined in html.hpp - the base class for all HTML input elements to be added to a CHTML_form. The constructor takes a (char*) input type and a (string) name. The constructor for each of the subclasses has a static member sm_InputType which is passed as the first argument to the CParent's (CHTML_input) constructor.
- CHTML_label Derived from CHTMLElement, defined in html.hpp - associates a label with a form control. The constructors take a string argument which specifies the text for the label, and optionally, a second string argument specifying the FOR attribute. The FOR attribute explicitly identifies the form control to associate with this label.

- CHTML_legend Derived from CHTMLElement, defined in html.hpp - defines a caption for a CHTML_fieldset element. The constructors take a single argument which may be either a string or a CHTMLNode.
- CHTML_menu Derived from CHTMLListElement, defined in html.hpp - the HTML component used to insert a menu list. The constructor takes zero to two arguments; if no arguments are provided, the compact attribute is by default false, and the type attribute is left to the browser. CHTML_menu("square", true) will create a compact menu element with square icons. Items can be added to the list using AppendChild (new CHTMLText("<li>...").
- CHTML_ol Derived from CHTMLListElement, defined in html.hpp - the HTML component used to insert an enumerated list. The constructor takes up to three arguments, specifying the starting number, the type of enumeration (Arabic, Roman Numeral etc.), and a Boolean argument specifying whether or not the display should be compact. Items can be added to the list using AppendChild(new CHTMLText ("<li>...").
- CHTML_option Derived from CHTMLElement, defined in html.hpp - an HTML option associated with a CHTML_select component. The constructor takes a value (string), a label (string or char*), and a Boolean indicating whether or not the option is by default selected. The last two arguments are optional, and by default the option is not selected.
- CHTML_radio Derived from CHTML_input, defined in html.hpp - can only be used inside a CHTML_form; the HTML component for a radio button. The constructor takes up to four arguments specifying the name (string), value (string), state (bool), and description (string) for the node.
- CHTML_reset Derived from CHTML_input, defined in html.hpp - can only be used inside a CHTML_form; the HTML component for a reset button. The constructor takes a single optional argument specifying the button's label.
- CHTML_select Derived from CHTMLElement, defined in html.hpp - an HTML select component. The constructor takes up to three arguments, specifying the name (string) and size (int) of the selection box, along with a Boolean specifying whether or not multiple selections are allowed (default is false). Select options should be added using the AppendOption() method.
- CHTML_submit Derived from CHTML_input, defined in html.hpp - can only be used inside a CHTML_form; the HTML component for a submit button. The constructor takes two string arguments specifying the button's name and label (optional). When selected, this causes the data selections in the including form to be sent to the form's URL.
- CHTML_table Derived from CHTMLElement, defined in html.hpp - an HTML table element. The constructor takes no arguments, but many member functions are provided to get/set attributes of the table. Because each of the "set attribute" methods returns this, the invocations can be strung together in a single statement.
  Use InsertAt(row, col, contents) to add contents to table cell row, col. To add contents to the next available cell, use AppendChild (new
  <listref rid="webpgs.html_CHTML_tc" RBID="webpgs.html_CHTML_tc">
  CHTML_tc </listref>
  (tag, contents)), where tag is type char* and contents is type char*, string or CNCBINode*.
- CHTML_tc Derived from CHTMLElement, defined in html.hpp - an HTML table cell element. All of the constructors expect the first argument to be a char* tagname. The second argument, if present, may be text (char* or string) or a pointer to a CNCBINode.

- CHTML_text Derived from CHTML_input, defined in html.hpp - can only be used inside a CHTML_form; the HTML component for a text box inside a form. The constructor takes up to four arguments: name (string), size (int), maxlength (int), and value (string). Only the first argument is required.
- CHTML_textarea Derived from CHTML_input, defined in html.hpp - can only be used inside a CHTML_form; the HTML component for a textarea inside a form. The constructor takes up to four arguments: name (string), cols (int), rows (int), and value (string). Only the last argument is optional.
- CHTML_tr Derived from CHTMLElement, defined in html.hpp - an HTML table row element. The constructors take a single argument, which may be either a string or a pointer to a CNCBINode.
- CHTML_ul Derived from CHTMLListElement, defined in html.hpp - the HTML component used to insert an unordered list. The constructor takes zero to two arguments; if no arguments are provided, the compact attribute is by default false, and the type attribute is left to the browser. CHTML_menu("square", true) will create a compact list element with square icons. Items can be added to the list using AppendChild(new CHTMLText("<li>...").
- CHTMLBasicPage Derived from CNCBINode, defined in page.hpp - The base class for CHTMLPage and its descendants. The HTML page classes serve as generalized containers for collections of other HTML elements, which together define a web page. Each page has a TTagMap, which maps names (strings) to the HTML subcomponents embedded in the page. Two constructors are defined. The first takes no arguments, and the other, takes a pointer to a CCgiApplication and a style (int) argument.
- CHTMLComment Derived from CHTMLNode, defined in html.hpp - used to insert an HTML comment. The constructor takes at most one argument, which may be a char*, a string, or a CNCBINode. The constructor then uses AppendPlainText() or AppendChild(), depending on the type of argument, to append the argument to the comment node.
- CHTMLElement Derived from CHTMLOpenElement, defined in html.hpp - the base class for all tagged elements which require a closing tag of the form </tagname>. CHTMLElement specializes the PrintEnd() method by generating the end tag </m_Name> on the output, where m_Name stores the tagname of the instance's subclass. Subclasses include CHTML_a, CHTML_basefont, CHTML_dl, CHTML_font, CHTML_form, CHTML_option, CHTML_select, CHTML_table, CHTML_tc, CHTML_textarea, and CHTMLListElement.
- CHTMLListElement Derived from CHTMLElement, defined in html.hpp - the base class for CHTML_ul, CHTML_ol, CHTML_dir, and CHTML_menu lists. Arguments to the constructor include the tagname and type strings for the list, along with a Boolean indicating whether or not the list is compact.
- CHTMLNode Derived from CNCBINode, defined in html.hpp - the base class for CHTMLComment and CHTMLOpenElement. Attributes include style, id, title, accesskey, color, bgcolor, height, width, align, valign, size, name, and class. All of the constructors require a tagname argument, which may be either type char* or string. The optional second argument may be type char*, string, or CNCBINode.
- CHTMLOpenElement Derived from CHTMLNode, defined in html.hpp - the base class for all tag elements, including CHTMLElement, CHTML_br, CHTML_hr, CHTML_img, and CHTML_input. All of the constructors require a tagname argument, which may be either type char* or string. The optional second argument may be type char*, string, or CNCBINode.

- CHTMLPage Derived from CHTMLBasicPage; defined in page.hpp - the basic 3 section NCBI page. There are two constructors. The first takes a title (type string) and the name of a template file (type string). Both arguments are optional. The other constructor takes a pointer to a CCgiApplication, a style (type int), a title and a template_file name. All but the first argument are optional.
- CHTMLPlainText Derived from CNCBINode, defined in html.hpp - A simple text component, which can be used to insert text that will be displayed verbatim by a browser (may require encoding). The constructor takes two arguments: the text to be inserted (char* or string) and a Boolean (default false) indicating that the output **should** be encoded. See also CHTMLText.
- CHTMLTagNode Derived from CNCBINode; defined in html.hpp.
- CHTMLDualNode Derived from CNCBINode, defined in html.hpp - Allows the user to explicitly specify what exactly to print out in eHTML and in ePlainText modes. The constructor takes 2 arguments -- the first one is for eHTML mode output (string or a pointer to a CNCBINode), and the second one is a plain text for ePlainText mode output.
- CHTMLSpecialChar Derived from CHTMLDualNode, defined in html.hpp - A class for HTML special chars like &nbsp, &copy, etc. Elements of this class have two variants for output, for eHTML and ePlainText modes. For example: &nbsp have plain text variant - " ", and &copy - "(c)". html.hpp has several predefined simple classes, based on this class, for any special chars. It is CHTML_nbsp, CHTML_gt, CHTML_lt, CHTML_quot, CHTML_amp, CHTML_copy and CHTML_reg. Each have one optional arqument, which specify the number of symbols to output.
- CHTMLText Derived from CNCBINode, defined in html.hpp - A simple text component which can be used to install a default web page design (stored in a template file) on a CHTMLPage or to simply insert encoded text. The PrintBegin() is specialized to handle tagnodes occurring in the text. The constructor takes a single argument - the text itself - which may be of type char* or string. CHTMLPlainText should be used to insert text that does not embed any tagnodes and requires further encoding.
- CHTMLPopupMenu Derived from CNCBINode; defined in jsmenu.hpp - A class for support JavaScript-based popup menu's in the HTML framework. The pages using this libraries can be viewed only in browsers that support JavaScript version 1.2 (or higher) and CSS.
- CNCBINode Derived from CObject, defined in node.hpp - A base class for all other HTML node classes. Contains data members m_Name, m_Attributes, and m_Children. The constructor takes at most one argument, name, which defines the internal data member m_Name.
- CPageList (Custom feature not for general use.) Derived from CHTML_table; defined in components.hpp. Used by the pager box components to page between results pages; contains forward and backward URLs, the current page number, and a map<int, string> that associates page numbers with URLs.
- CPager (Custom feature not for general use.) Derived from CNCBINode, defined in html.hpp
- CPagerBox (Custom feature not for general use.) Derived from CNCBINode; defined in components.hpp. A more elaborate paging component than the CSmallPagerBox; contains pointers to a CPageList and (3) CButtonList components (left, right, and top). Additional properties include width, background color, and number of results.
- CPagerView (Custom feature not for general use.) Derived from CHTML_table; defined in pager.hpp.

- CQueryBox (Custom feature not for general use.) Derived from CHTML_table; defined in components.hpp.
- CSelection (Custom feature not for general use.) Derived from CNCBINode; defined in components.hpp. A checkbox-like component whose choices are generated (using the CreateSubNodes() method) from the TCgiEntries of a CCgiRequest object.
- CSmallPagerBox (Custom feature not for general use.) Derived from CNCBINode; defined in components.hpp. A minimal paging component that displays the number of results from the query and the current page being viewed. Has background color and width attributes and contains a pointer to a CPageList. See also CPagerBox and CPager.