A Tutorial Example

Introduction

It is easiest to understand how the GNU cgicc library might be used by first looking at an example. Suppose you want an HTML form on your web site asking a user to enter their name, age, and sex, perhaps as part of a user-registration procedure, and you wish to write a CGI script using cgicc to process the form in some meaningful way.

You would begin by creating an HTML form containing the HTML fragment

```
<form method="post" action="http://change_this_path/cgi-bin/foo.cgi">
Your name : <input type="text" name="name" /><br />
Your age : <input type="text" name="age" /><br />
Your sex : <input type="radio" name="sex" value="male"checked="checked" />Male
<input type="radio" name="sex" value="female" />Female <br />
</form>
```

Then, on to the CGI application. Applications written using cgicc, like all other applications, begin with a `main` function:

```
int main(int argc, char **argv)
{
   // CGI processing goes here
}
```

Initialization

The three main classes of cgicc you will use to process the submitted data are **cgicc::Cgicc**, **cgicc::CgiEnvironment**, and **cgicc::FormEntry**. These classes will be explained in detail later; for now, it is sufficient to know that:

- The class **cgicc::Cgicc** is used for retrieving information on the submitted form elements.
- The class **cgicc::CgiEnvironment** is used to retrieve information on environment variables passed from the HTTP server.
- The class **cgicc::FormEntry** is used to extract various types of data from the submitted form elements.

All of cgicc's functionality is accessed through class **cgicc::Cgicc**. Thus, the first step in CGI processing is to instantiate an object of type **cgicc::Cgicc**:

```
cgicc::Cgicc cgi;
```

or

```
using namespace cgicc;
```

```
Cgicc cgi;
```

Upon instantiation, the class **cgicc::Cgicc** parses all data passed to the CGI script by the HTTP server.

Since errors are handled using exceptions, you may wish to wrap your CGI code in a `try` block to better handle unexpected conditions:

```
try {
   cgicc::Cgicc cgi;
}

catch(exception& e) {
   // Caught a standard library exception
}
```

Extracting Form Information

Each element of data entered by the user is parsed into a **cgicc::FormEntry**. A **cgicc::FormEntry** contains methods for accessing data as strings, integers, and doubles. In the form mentioned above, a user would enter their name, age, and sex. Regardless of the type of value, the data is accessed using **cgicc::FormEntry** (this is not entirely true. For uploaded files, the data is accessed via the class **cgicc::FormFile**). You obtain **cgicc::FormEntry** objects via **cgicc::Cgicc**'s `getElement` methods, all of which return typedefs of C++ standard template library (STL) iterators:

```
cgicc::form_iterator name = cgi.getElement("name");
```

If the item is not found, the iterator will refer to an invalid element, and should not be dereferenced using `operator*` or `operator->`. **cgicc::Cgicc** provides methods for determining whether an iterator refers to a valid element:

```
if(name != cgi.getElements().end()) {
   // iterator refers to a valid element
}
```

The **cgicc::FormEntry** class provides methods for extracting data as numbers, removing line breaks, etc. If you are not interested in performing any data validation or modification, but simply want to access a string representaion of the data, the simplest case is streamlined:

```
std::string name = cgi("name");
```

Output of Form Data

Once you have a valid element, you will more than likely want to do something with the data. The simplest thing to do is just echo it back to the user. You can extract a `basic_string` from a **cgicc::FormEntry** by calling the `getValue` method. Since `ostream` has an overload for writing `basic_string` objects, it is trivial to output objects of this type:

```
cout << "Your name is " << name->getValue() << endl;
```

Since both `iterator` and **cgicc::FormEntry** overload `operator*`, the code given above may also be written as:

```
cout << "Your name is " << **name << endl;
```

The first * returns an object of type **cgicc::FormEntry**, and the second * returns an object of type `basic_string`.

As mentioned above, if you simply want to output a string without validating or modifying the data, the simplest case is streamlined:

```
cout << "Your name is " << cgi("name") << endl;
```

The HTTP Response

A CGI response will generally consist of an HTML document. The HTTP protocol requires that a certain set of headers precede all documents, to inform the client of the size and type of data being received, among other things. In a normal CGI response, the HTTP server will take care of sending many of these headers for you. However, it is necessary for the CGI script to supply the type of content it is returning to the HTTP server and the client. This is done by emitting a `Content-Type` header. If you're interested, the full HTTP 1.1 specification may be found in RFC 2068 at http://www.w3.org/Protocols/rfc2068/rfc2068

cgicc provides several classes for outputting HTTP headers, all of which begin with `HTTP`. A standard HTML 4.0 document need only output a single header:

```
cout << cgicc::HTTPHTMLHeader() << endl;
```

This will generate the output

```
Content-Type: text/html\n\n
```

Simple HTML Output

cgicc provides one class for every HTML tag defined in the HTML 4.0 standard in the header file `"cgicc/HTMLClasses.h"`. These classes have the same name as the HTML tags. For example, in HTML, to indicate the start of a document you write `<html>` ; this can be accomplished using cgicc by writing

```
cout << html() << endl;
```

The class `html` keeps state internally, so the code above will produce as output `<html>`; conversely, the code

```
cout << html() << "html text!" << html() << endl;
```

will produce as output `<html>html text!</html>`.

All of cgicc's HTML output classes are subclasses of the abstract class **cgicc::HTMLElement**. You can embed the text for the element directly in the constructor:

```
cout << html("html text!") << endl;
```

Furthermore, it is possible to embed one **cgicc::HTMLElement** in another:

```
cout << head(title("Title")) << endl;
```

This produces as output

```
<head><title>Title</title></head>
```

And, if you wish be more specific about the type of HTML 4.0 you are going to return (strict, transitional, or frameset), you can use the class **cgicc::HTMLDoctype** before the cgicc::html tag:

```
cout << HTMLDoctype(HTMLDoctype::eStrict) << endl;
```

which produces

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN" "http://www.w3.org/TR/REC-html40/strict.
```

More Complex HTML Output

In real HTML, most tags possess a set of attributes. For example, the HTML `<img>` tag requires certain attributes specifying the source image file, the image width, height, and so on. There are a bewildering number of possible attributes in HTML 4.0. For a definitive list, see the HTML 4.0 specification at http://www.w3.org/TR/REC-html40/ A typical `<img>` tag might look like:

```
<img src="file.jpg" width="100" height="100" alt="description" />
```

This tag has four attributes: `src`, `width`, `height`, and `alt`, with the values `file.jpg`, `100`, `100`, and `description`, respectively. Attributes in HTML tags are represented by the class **cgicc::HTMLAttribute**, which essentially is a name/value pair. To build an **cgicc::HTMLElement** containing **cgicc::HTMLAttribute** objects, use the `set` method on **cgicc::HTMLElement**. To generate the `<img>` tag given above:

```
cout << img().set("src", "file.jpg")
             .set("width", "100").set("height", "100")
             .set("alt", "description") << endl;
```

In a similar way, multiple **cgicc::HTMLElement** objects may be embedded at the same level inside another **cgicc::HTMLElement**. To build an **cgicc::HTMLElement** containing multiple embedded **cgicc::HTMLElement** objects, use the `add` method on **cgicc::HTMLElement**:

```
cout << tr().add(td("0")).add(td("1")).add(td("2")) << endl;
```

This produces as output

```
<tr><td>0</td><td>1</td><td>2</td></tr>
```

Notes on Output

All of cgicc's output is written to a C++ standard output stream, usually `cout`. It is not necessary to use cgicc's HTML output classes; they are provided as a convenience. If you prefer, you may output the HTML code directly to `cout`.

The Complete Example

The code below is a complete CGI program that synthesizes all the sample code given above.

```
#include <iostream>
#include <vector>
#include <string>

#include "cgicc/Cgicc.h"
#include "cgicc/HTTPHTMLHeader.h"
#include "cgicc/HTMLClasses.h"
```

```
using namespace std;
using namespace cgicc;

int
main(int argc,
     char **argv)
{
   try {
      Cgicc cgi;

      // Send HTTP header
      cout << HTTPHTMLHeader() << endl;

      // Set up the HTML document
      cout << html() << head(title("cgicc example")) << endl;
      cout << body() << endl;

      // Print out the submitted element
      form_iterator name = cgi.getElement("name");
      if(name != cgi.getElements().end()) {
         cout << "Your name: " << **name << endl;
      }

      // Close the HTML document
      cout << body() << html();
   }
   catch(exception& e) {
      // handle any errors - omitted for brevity
   }
}
```

Previous: **Library Overview** | Current: **A Tutorial Example** | Next: **GNU cgicc Demos**

GNU cgicc - A C++ class library for writing CGI applications
Copyright © 1996 - 2004 Stephen F. Booth
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with no Invariant Sections, with no Front Cover Texts, and with no Back-Cover Texts.
Documentation generated Sat Jan 19 21:15:59 2008 for cgicc by doxygen 1.5.1