The NCBI C++ Toolkit Book
The NCBI C++ Toolkit Book

The NCBI C++ Toolkit

11: CGI and Fast-CGI

Created: January 1, 2005.
Last Update: March 21, 2011.

Overview

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

CGI and Fast-CGI [Libraries xcgi and xfcgi: include | src]

These library classes represent an integrated framework with which to write CGI applications and are designed to help retrieve and parse an HTTP request and then to compose and deliver an HTTP response. (See also this additional class reference documentation). xfcgi is a FastCGI version of xcgi.

Hint: Requires the target executable to be linked with a third-party FastCGI library, as in:

LIBS = $(FASTCGI_LIBS) $(ORIG_LIBS).

Hint: On non-FastCGI capable platforms (or if run as a plain CGI on a FastCGI-capable platform), it works the same as a plain CGI.

CGI Interface

- Basic CGI Application Class (includes CGI Diagnostic Handling) cgiapp[.hpp | .cpp]
- CGI Application Context Classes cgictx[.hpp | .cpp]
- HTTP Request Parser ncbicgi[.hpp | .cpp]
- HTTP Cookies ncbicgi[.hpp | .cpp]
- HTTP Response Generator ncbicgir[.hpp | .cpp]
- Basic CGI Resource Class ncbires[.hpp | .cpp]

FastCGI CGI Interface

- Adapter Between C++ and FastCGI Streams fcgibuf[.hpp | .cpp]
- Fast-CGI Loop Function fcgi_run[.cpp]
- Plain CGI Stub for the Fast-CGI Loop Function cgi_run[.cpp]

Demo Cases [src/cgi/demo | C++/src/sample/app/cgi/]

Test Cases [src/cgi/test]

.

Chapter Outline

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

Developing CGI applications

- Overview of the CGI classes
- The CCgiApplication class
- The CNcbiResource and CNcbiCommand classes
- The CCgiRequest class
- The CCgiResponse class
- The CCgiCookie class
- The CCgiCookies class
- The CCgiContext class
- The CCgiUserAgent class
- Example code using the CGI classes

- CGI Registry configuration
- Supplementary Information

Although CGI programs are generally run as web applications with HTML interfaces, this section of the Programming Manual places emphasis on the CGI side of things, omitting HTML details of the implementation where possible. Similarly, the section on Generating web pages focuses largely on the usage of HTML components independent of CGI details. The two branches of the NCBI C++ Toolkit hierarchy are all but independent of one another - with but one explicit hook between them: the constructors for HTML page components accept a CCgiApplication as an optional argument. This CCgiApplication argument provides the HTML page component with access to all of the CGI objects used in the application.

Further discussion of combining a CGI application with the HTML classes can be found in the section on An example web-based CGI application . The focus in this chapter is on the CGI classes only. For additional information about the CGI classes, the reader is also referred to the discussion of NCBI C++ CGI Classes in the Reference Manual.

The CGI classes

Figure 1 illustrates the layered design of the CGI classes.

This design is best described by starting with a consideration of the capabilities one might need to implement a CGI program, including:

- A way to retrieve and store the current values of environment variables
- A means of retrieving and interpreting the client's query request string
- Mechanisms to service and respond to the requested query
- Methods and data structures to obtain, store, modify, and send cookies
- A way to set/reset the context of the application (for Fast-CGI)

The CCgiContext class unifies these diverse capabilities under one aggregate structure. As their names suggest, the CCgiRequest class receives and parses the request, and the CCgiResponse class outputs the response on an output stream. All incoming CCgiCookies are also parsed and stored by the CCgiRequest object, and the outgoing cookies are sent along with the response by the CCgiResponse object. The request is actually processed by the application's CNcbiResource. The list of CNcbiCommands stored with that resource object are scanned to find a matching command, which is then executed.

The CCgiContext object, which is a friend to the CCgiApplication class, orchestrates this sequence of events in coordination with the application object. The same application may be run in many different contexts, but the resource and defined set of commands are invariant. What changes with each context is the request and its associated response.

The CCgiApplication class is a specialization of CNcbiApplication. Figure 2 illustrates the adaptation of the Init() and Run() member functions inherited from the CNcbiApplication class to the requirements of CGI programming. Although the application is contained in the context, it is the application which creates and initializes each context in which it participates. The program arguments and environmental variables are passed along to the context, where they will be stored, thus freeing the application to be restarted in a new context, as in Fast-CGI.

The application's ProcessRequest member function is an abstract function that must be implemented for each application project. In most cases, this function will access the query and the environment variables via the CCgiContext, using ctx.GetRequest() and ctx.GetConfig

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

(). The application may then service the request using its resource's HandleRequest() method. The context's response object can then be used to send an appropriate response.

These classes are described in more detail below, along with abbreviated synopses of the class definitions. These are included here to provide a conceptual framework and are not intended as reference materials. For example, constructor and destructor declarations that operate on void arguments, and const methods that duplicate non-const declarations are generally not included here. Certain virtual functions and data members that have no meaning outside of a web application are also omitted. For complete definitions, refer to the header files via the source browsers.

The CCgiApplication Class (*)

As mentioned, the CCgiApplication class implements its own version of Init() , where it instantiates a CNcbiResource object using LoadResource(). Run() is no longer a pure virtual function in this subclass, and its implementation now calls CreateContext(), ProcessRequest (), and CCgiContext::GetResponse(). The CCgiApplication class does **not** have a CCgiContext data member, because the application object can participate in multiple CCgiContexts. Instead, a local variable in each Run() invocation stores a pointer to the context created there. The LoadServerContext() member function is used in Web applications, such as the query program, where it is necessary to store more complex run-time data with the context object. The CCgiServerContext object returned by this function is stored as a data member of a CCgiContext and is application specific.

```
class CCgiApplication : public CNcbiApplication
{
 friend class CCgiContext;

public:
 void Init(void);
 void Exit(void);
 int Run(void);

 CNcbiResource& GetResource(void);
 virtual int ProcessRequest(CCgiContext&) = 0;
 CNcbiResource* LoadResource(void);
 virtual CCgiServerContext* LoadServerContext(CCgiContext& context);

 bool RunFastCGI(unsigned def_iter=3);

protected:
 CCgiContext* CreateContext(CNcbiArguments*, CNcbiEnvironment*,
 CNcbiIstream*, CNcbiOstream*);

private: auto_ptr<CNcbiResource> m_resource;
};
```

If the program was **not** compiled as a FastCGI application (or the environment does not support FastCGI), then RunFastCGI() will return false. Otherwise, a "FastCGI loop" will be iterated over def_iter times, with the initialization methods and ProcessRequest() function being executed on each iteration. The value returned by RunFastCGI() in this case is true. Run() first calls RunFastCGI(), and if that returns false, the application is run as a plain CGI program.

The CNcbiResource (*) and CNcbiCommand (*) Classes

The resource class is at the heart of the application, and it is here that the program's functionality is defined. The single argument to the resource class's constructor is a CNcbiRegistry object, which defines data paths, resources, and possibly environmental variables for the application. This information is stored in the resource class's data member, m_config. The only other data member is a TCmdList (a list of CNcbiCommands) called m_cmd.

```
class CNcbiResource
{
public:

 CNcbiResource(CNcbiRegistry& config);

 CNcbiRegistry& GetConfig(void);
 const TCmdList& GetCmdList(void) const;
 virtual CNcbiCommand* GetDefaultCommand(void) const = 0;
 virtual const CNcbiResPresentation* GetPresentation(void) const;

 void AddCommand(CNcbiCommand* command);
 virtual void HandleRequest(CCgiContext& ctx);

protected:

 CNcbiRegistry& m_config;
 TCmdList m_cmd;
};
```

The AddCommand() method is used when a resource is being initialized, to add commands to the command list. Given a CCgiRequest object defined in a particular context ctx, HandleRequest(ctx) compares entries in the context's request to commands in m_cmd. The first command in m_cmd that matches an entry in the request is then executed (see below), and the request is considered "handled". If desired, a default command can be installed that will execute when no matching command is found. The default command is defined by implementing the pure virtual function GetDefaultCommand(). The CNcbiResPresentation class is an abstract base class, and the member function, GetPresentation(), returns 0. It is provided as a hook for implementing interfaces between information resources (e.g., databases) and CGI applications.

```
class CNcbiCommand
{
public:
 CNcbiCommand(CNcbiResource& resource);

 virtual CNcbiCommand* Clone(void) const = 0;
 virtual string GetName() const = 0;
 virtual void Execute(CCgiContext& ctx) = 0;
 virtual bool IsRequested(const CCgiContext& ctx) const;

protected:
 virtual string GetEntry() const = 0;
 CNcbiResource& GetResource() const { return m_resource; }
```

```
private:
 CNcbiResource& m_resource;
};
```

CNcbiCommand is an abstract base class; its only data member is a reference to the resource it belongs to, and most of its methods - with the exception of GetResource() and IsRequested () - are pure virtual functions. IsRequested() examines the key=value entries stored with the context's request object. When an entry is found where key==GetEntry() and value==GetName (), IsRequested() returns true.

The resource's HandleRequest() method iterates over its command list, calling CNcbiCommand::IsRequested() until the first match between a command and a request entry is found. When IsRequested() returns true, the command is cloned, and the cloned command is then executed. Both the Execute() and Clone() methods are pure virtual functions that must be implemented by the user.

The CCgiRequest Class (*)

The CCgiRequest class serves as an interface between the user's query and the CGI program. Arguments to the constructor include a CNcbiArguments object, a CNcbiEnvironment object, and a CNcbiIstream object. The class constructors do little other than invoke CCgiRequest::x_Init(), where the actual initialization takes place.

x_Init() begins by examining the environment argument, and if it is NULL, m_OwnEnv (an auto_ptr) is reset to a dummy environment. Otherwise, m_OwnEnv is reset to the passed environment, making the request object the effective owner of that environment. The environment is then used to cache network information as "gettable" properties. Cached properties include:

- server properties, such as the server name, gateway interface, and server port
- client properties (the remote host and remote address)
- client data properties (content type and content length of the request)
- request properties, including the request method, query string, and path information
- authentication information, such as the remote user and remote identity
- standard HTTP properties (from the HTTP header)

These properties are keyed to an enumeration named ECgiProp and can be retrieved using the request object's GetProperty() member function. For example, GetProperty(eCgi_HttpCookie) is used to access cookies from the HTTP Header, and GetProperty(eCgi_RequestMethod) is used to determine from where the query string should be read.

NOTE: Setting $QUERY_STRING without also setting $REQUEST_METHOD will result in a failure by x_init() to read the input query. x_init() first looks for the definition of $REQUEST_METHOD, and depending on if it is GET or POST, reads the query from the environment or the input stream, respectively. If the environment does not define $REQUEST_METHOD, then x_Init() will try to read the query string from the command line only.

```
class CCgiRequest {
public:
 CCgiRequest(const CNcbiArguments*, const CNcbiEnvironment*,
 CNcbiIstream*, TFlags);
```

```
 static const string& GetPropertyName(ECgiProp prop);
 const string& GetProperty(ECgiProp prop) const;
 size_t GetContentLength(void) const;
 const CCgiCookies& GetCookies(void) const;
 const TCgiEntries& GetEntries(void) const;
 static SIZE_TYPE ParseEntries(const string& str, TCgiEntries& entries);
private:
 void x_Init(const CNcbiArguments*, const CNcbiEnvironment*,
 CNcbiIstream*, TFlags);

 const CNcbiEnvironment* m_Env;
 auto_ptr<CNcbiEnvironment> m_OwnEnv;
 TCgiEntries m_Entries;
 CCgiCookies m_Cookies;
};
```

This abbreviated definition of the CCgiRequest class highlights its primary functions:

To parse and store the <key=value> pairs contained in the query string (stored in m_Entries).

To parse and store the cookies contained in the HTTP header (stored in m_Cookies).

As implied by the "T" prefix, TCgiEntries is a type definition, and defines m_Entries to be an STL multimap of <string,string> pairs. The CCgiCookies class (described below) contains an STL set of CCgiCookie and implements an interface to this set.

The CCgiResponse Class (*)

The CCgiResponse class provides an interface to the program's output stream (usually cout), which is the sole argument to the constructor for CCgiResponse. The output stream can be accessed by the program using CCgiResponse::GetOutput(), which returns a pointer to the output stream, or, by using CCgiResponse::out(), which returns a reference to that stream.

In addition to implementing controlled access to the output stream, the primary function of the response class is to generate appropriate HTML headers that will precede the rest of the response. For example, a typical sequence in the implementation of a particular command's execute function might be:

```
MyCommand::Execute(CCgiContext& ctx)
{
 // ... generate the output and store it in MyOutput

 ctx.GetResponse().WriteHeader();
 ctx.GetResponse().out() << MyOutput;
 ctx.GetResponse.out() << "</body></html>" << endl;
 ctx.GetResponse.Flush();
}
```

Any cookies that are to be sent with the response are included in the headers generated by the response object.

Two member functions are provided for outputting HTML headers: WriteHeader() and WriteHeader(CNcbiOstream&). The second of these is for writing to a specified stream other than the default stream stored with the response object. Thus, WriteHeader(out()) is equivalent to WriteHeader().

The WriteHeader() function begins by invoking IsRawCgi() to see whether the application is a non-parsed header program. If so, then the first header put on the output stream is an HTTP status line, taken from the private static data member, sm_HTTPStatusDefault. Next, unless the content type has been set by the user (using SetContentType()), a default content line is written, using sm_ContentTypeDefault. Any cookies stored in m_Cookies are then written, followed by any additional headers stored with the request in m_HeaderValues. Finally, a new line is written to separate the body from the headers.

```
class CCgiResponse {
public:
 CCgiResponse(CNcbiOstream* out = 0);

 void SetRawCgi(bool raw);
 bool IsRawCgi(void) const;
 void SetHeaderValue(const string& name, const string& value);
 void SetHeaderValue(const string& name, const tm& value);
 void RemoveHeaderValue(const string& name);
 void SetContentType(const string &type);
 string GetHeaderValue(const string& name) const;
 bool HaveHeaderValue(const string& name) const;
 string GetContentType(void) const;

 CCgiCookies& Cookies(void); // Get cookies set
 CNcbiOstream* SetOutput(CNcbiOstream* out); // Set default output stream
 CNcbiOstream* GetOutput(void) const; // Query output stream
 CNcbiOstream& out(void) const; // Conversion to ostream
 // to enable <<

 void Flush() const;

 CNcbiOstream& WriteHeader(void) const; // Write HTTP response header
 CNcbiOstream& WriteHeader(CNcbiOstream& out) const;
protected:
 typedef map<string, string> TMap;
 static const string sm_ContentTypeName;
 static const string sm_ContentTypeDefault;
 static const string sm_HTTPStatusDefault;
 bool m_RawCgi;
 CCgiCookies m_Cookies;
 TMap m_HeaderValues; // Additional header lines in alphabetical order
 CNcbiOstream* m_Output; // Default output stream };
```

The CCgiCookie Class (*)

The traditional means of maintaining state information when servicing a multi-step request has been to include hidden input elements in the query strings passed to subsequent URLs. The newer, preferred method uses HTTP cookies, which provide the server access to client-side

state information stored with the client. The cookie is a text string consisting of four key=value pairs:

- name (required)
- expires (optional)
- domain (optional)
- path (optional)

The CCgiCookie class provides a means of creating, modifying, and sending cookies. The constructor requires at least two arguments, specifying the name and value of the cookie, along with the optional domain and path arguments. Format errors in the arguments to the constructor (see Supplementary Information) will cause the invalid argument to be thrown. The CCgiCookie::Write(CNcbiOstream&) member function creates a Set-Cookie directive using its private data members and places the resulting string on the specified output stream:

```
Set-Cookie:
m_Name=
m_Value; expires=
m_Expires; path=
m_Path;
domain=
m_Domain;
m_Secure
```

As with the constructor, and in compliance with the proposed standard (RFC 2109), only the name and value are mandatory in the directive.

```
class CCgiCookie {
public:
 CCgiCookie(const string& name, const string& value,
 const string& domain = NcbiEmptyString,
 const string& path = NcbiEmptyString);
 const string& GetName(void) const;
 CNcbiOstream& Write(CNcbiOstream& os) const;
 void Reset(void);
 void CopyAttributes(const CCgiCookie& cookie);
 void SetValue (const string& str);
 void SetDomain (const string& str);
 void SetPath (const string& str);
 void SetExpDate(const tm& exp_date);
 void SetSecure (bool secure);
 const string& GetValue (void) const;
 const string& GetDomain (void) const;
 const string& GetPath (void) const;
 string GetExpDate(void) const;
 bool GetExpDate(tm* exp_date) const;
 bool GetSecure(void) const;
 bool operator<(const CCgiCookie& cookie) const;
 typedef const CCgiCookie* TCPtr;
 struct PLessCPtr {
 bool operator() (const TCPtr& c1, const TCPtr& c2) const {
 return (*c1 < *c2);
```

```
 }
 };
private:
 string m_Name;
 string m_Value;
 string m_Domain;
 string m_Path;
 tm m_Expires;
 bool m_Secure;
};
```

With the exception of m_Name, all of the cookie's data members can be reset using the SetXxx (), Reset(), and CopyAttributes() member functions; m_Name is non-mutable. As with the constructor, format errors in the arguments to these functions will cause the invalid argument to be thrown. By default, m_Secure is false. The GetXxx() methods return the stored value for that attribute or, if no value has been set, a reference to NcbiEmptyString. GetExpDate(tm*) returns false if no expiration date was previously set. Otherwise, tm is reset to m_Expire, and true is returned.

The CCgiCookies Class (*)

The CCgiCookies class provides an interface to an STL set of CCgiCookies (m_Cookies). Each cookie in the set is uniquely identified by its name, domain, and path values and is stored in ascending order using the CCgiCookie::PLessCPtr construct. Two constructors are provided, allowing the user to initialize m_Cookies to either an empty set or to a set of N new cookies created from the string "name1=value1; name2=value2; ...; nameN=valuenN". Many of the operations on a CCgiCookies object involve iterating over the set, and the class's type definitions support these activities by providing built-in iterators and a typedef for the set, TSet.

The Add() methods provide a variety of options for creating and adding new cookies to the set. As with the constructor, a single string of name-value pairs may be used to create and add N cookies to the set at once. Previously created cookies can also be added to the set individually or as sets. Similarly, the Remove() methods allow individual cookies or sets of cookies (in the specified range) to be removed. All of the remove functions destroy the removed cookies when destroy=true. CCgiCookies::Write(CNcbiOstream&) iteratively invokes the CCgiCookie::Write() on each element.

```
class CCgiCookies {
public:
 typedef set<CCgiCookie*, CCgiCookie::PLessCPtr> TSet;
 typedef TSet::iterator TIter;
 typedef TSet::const_iterator TCIter;
 typedef pair<TIter, TIter> TRange;
 typedef pair<TCIter, TCIter> TCRange;
 CCgiCookies(void); // create empty set of cookies
 CCgiCookies(const string& str);
 // str = "name1=value1; name2=value2; ..."
 bool Empty(void) const;
 CCgiCookie* Add(const string& name, const string& value,
 const string& domain = NcbiEmptyString,
 const string& path = NcbiEmptyString);
 CCgiCookie* Add(const CCgiCookie& cookie);
 void Add(const CCgiCookies& cookies);
```

```
 void Add(const string& str);
 // "name1=value1; name2=value2; ..."
 CCgiCookie* Find(const string& name, const string& domain,
 const string& path);
 CCgiCookie* Find(const string& name, TRange* range=0);
 bool Remove(CCgiCookie* cookie, bool destroy=true);
 size_t Remove(TRange& range, bool destroy=true);
 size_t Remove(const string& name, bool destroy=true);
 void Clear(void);
 CNcbiOstream& Write(CNcbiOstream& os) const;
private:
 TSet m_Cookies;
};
```

The CCgiContext Class (*)

As depicted in Figure 1 , a CCgiContext object contains an application object, a request object, and a response object, corresponding to its data members m_app, m_request, and m_response. Additional data members include a string encoding the URL for the context (m_selfURL), a message buffer (m_lmsg), and a CCgiServerContext. These last three data members are used only in complex Web applications, such as the query program, where it is necessary to store more complex run-time data with the context object. The message buffer is essentially an STL list of string objects the class definition of which (CCtxMsgString) includes a Write() output function. GetServCtx() returns m_srvCtx if it has been defined and, otherwise, calls the application's CCgiApplication::LoadServerContext() to obtain it.

```
class CCgiContext
{
public:
 CCgiContext(CCgiApplication& app,
 const CNcbiArguments* args = 0,
 const CNcbiEnvironment* env = 0,
 CNcbiIstream* inp = 0,
 CNcbiOstream* out = 0);
 const CCgiApplication& GetApp(void) const;
 CNcbiRegistry& GetConfig(void);
 CCgiRequest& GetRequest(void);
 CCgiResponse& GetResponse(void);
 const string& GetSelfURL(void) const;
 CNcbiResource& GetResource(void);
 CCgiServerContext& GetServCtx(void);
 // output all msgs in m_lmsg to os
 CNcbiOstream& PrintMsg(CNcbiOstream& os);
 void PutMsg(const string& msg); // add message to m_lmsg
 void PutMsg(CCtxMsg* msg); // add message to m_lmsg
 bool EmptyMsg(void); // true iff m_lmsg is empty
 void ClearMsg(void); // delete all messages in m_lmsg
 string GetRequestValue(const string& name) const;
 void AddRequestValue(const string& name, const string& value);
 void RemoveRequestValues(const string& name);
 void ReplaceRequestValue(const string& name, const string& value);
private:
```

```
 CCgiApplication& m_app;
 auto_ptr<CCgiRequest> m_request;
 CCgiResponse m_response;
 mutable string m_selfURL;
 list<CCtxMsg*> m_lmsg; // message buffer
 auto_ptr<CCgiServerContext> m_srvCtx;
 // defined by CCgiApplication::LoadServerContext()
 friend class CCgiApplication;
};
```

The CCgiUserAgent class (*)

The CCgiUserAgent class is used to gather information about the client's user agent - i.e. browser type, browser name, browser version, browser engine type, browser engine version, Mozilla version (if applicable), platform, and robot information. The default constructor looks for the user agent string first in the CCgiApplication context using the eCgi_HttpUserAgent request property, then in the CNcbiApplication instance HTTP_USER_AGENT environment variable, and finally in the operating system HTTP_USER_AGENT environment variable.

```
class CCgiUserAgent
{
public:
 CCgiUserAgent(void);
 CCgiUserAgent(const string& user_agent);

 void Reset(const string& user_agent);

 string GetUserAgentStr(void) const;
 EBrowser GetBrowser(void) const;
 const string& GetBrowserName(void) const;
 EBrowserEngine GetEngine(void) const;
 EBrowserPlatform GetPlatform(void) const;

 const TUserAgentVersion& GetBrowserVersion(void) const;
 const TUserAgentVersion& GetEngineVersion(void) const;
 const TUserAgentVersion& GetMozillaVersion(void) const;

 typedef unsigned int TBotFlags;
 bool IsBot(TBotFlags flags = fBotAll, const string& patterns = kEmptyStr)
const;

protected:
 void x_Init(void);
 void x_Parse(const string& user_agent);
 bool x_ParseToken(const string& token, int where);

protected:
 string m_UserAgent;
 EBrowser m_Browser;
 string m_BrowserName;
 TUserAgentVersion m_BrowserVersion;
 EBrowserEngine m_Engine;
```

```
 TUserAgentVersion m_EngineVersion;
 TUserAgentVersion m_MozillaVersion;
 EBrowserPlatform m_Platform;
};
```

Example Code Using the CGI Classes

The sample CGI program demonstrates a simple application that combines the NCBI C++ Toolkit's CGI and HTML classes. justcgi.cpp is an adaptation of that program, stripped of all HTML references and with additional request-processing added (see Box 1 and Box 2).

Executing

```
./cgi 'cmd1=init&cmd2=reply'
```

results in execution of only cmd1, as does executing

```
./cgi 'cmd2=reply&cmd1=init'
```

The commands are matched in the order that they are registered with the resource, not according to the order in which they occur in the request. The assumption is that only the first entry (if any) in the query actually specifies a command, and that the remaining entries provide optional arguments to that command. The Makefile (see Box 3) for this example links to both the xncbi and xcgi libraries. Additional examples using the CGI classes can be found in src/cgi/test . (For Makefile.fastcgi.app, see Box 4 .)

CGI Registry Configuration

The application registry defines CGI-related configuration settings in the [CGI] section (see this table).

FastCGI settings. [FastCGI] section (see this table).

CGI load balancing settings. [CGI-LB] section (see this table).

Supplementary Information

Restrictions on arguments to the CCgiCookie constructor.

See Table 1.

CGI Diagnostic Handling

By default, CGI applications support three query parameters affecting diagnostic output : diag-destination , diag-threshold , and diag-format . It is possible to modify this behavior by overriding the virtual function CCgiApplication::ConfigureDiagnostics . (In particular, production applications may wish to disable these parameters by defining ConfigureDiagnostics to be a no-op.)

diag-destination

The parameter diag-destination controls where diagnostics appear. By default, there are two possible values (see Table 2).

However, an application can make other options available by calling RegisterDiagFactory from its Init routine. In particular, calling

```
#include <connect/email_diag_handler.hpp>
...
RegisterDiagFactory("email", new CEmailDiagFactory);
```

and linking against xconnect and connect enables destinations of the form email:user@host, which will cause the application to e-mail diagnostics to the specified address when done.

Similarly, calling

```
#include <html/commentdiag.hpp>
...
RegisterDiagFactory("comments", new CCommentDiagFactory);
```

and linking against xhtml will enable the destination comments. With this destination, diagnostics will take the form of comments in the generated HTML, provided that the application has also used SetDiagNode to indicate where they should go. (Applications may call that function repeatedly; each invocation will affect all diagnostics until the next invocation. Also, SetDiagNode is effectively a no-op for destinations other than comments, so applications may call it unconditionally.)

Those destinations are not available by default because they introduce additional dependencies; however, either may become a standard possibility in future versions of the toolkit.

diag-threshold

The parameter diag-threshold sets the minimum severity level of displayed diagnostics; its value can be either fatal, critical, error, warning, info, or trace. For the most part, setting this parameter is simply akin to calling SetDiagPostLevel . However, setting diag-threshold to trace is **not** equivalent to calling SetDiagPostLevel(eDiag_Trace); the former reports all diagnostics, whereas the latter reports only traces.

diag-format

Finally, the parameter diag-format controls diagnostics' default appearance; setting it is akin to calling {Set,Unset}DiagPostFlag . Its value is a list of flags, delimited by spaces (which appear as "+" signs in URLs); possible flags are file, path, line, prefix, severity, code, subcode, time, omitinfosev, all, trace, log, and default. Every flag but default corresponds to a value in EDiagPostFlag , and can be turned off by preceding its name with an exclamation point ("!"). default corresponds to the four flags which are on by default: line, prefix, code, and subcode, and may not be subtracted.

NCBI C++ CGI Classes

The Common Gateway Interface (CGI) is a method used by web servers to pass information from forms displayed in a web browser to a program running on the server and then allow the program to pass a web page back. The NCBI C++ CGI Classes are used by the program running on the server to decode the CGI input from the server and to send a response. The library also supports cookies, which is a method for storing information on the user's machine. The library supports the http methods GET and POST via application/x-www-form-urlencoded, and supports the POST via multipart/form-data (often used for file upload). In the POST via multipart/form-data, the data gets read into a TCgiEntries; you also can get the filename out of it (the name of the entry is as specified by "name=" of the data-part header). For more information on CGI, see the book **HTML Sourcebook** by Ian Graham or http://hoohoo.ncsa.uiuc.edu/cgi/

There are 5 main classes:

CCgiRequest–what the CGI program is getting from the client.

CCgiResponse–what the CGI program is sending to the client.

CCgiEntry–a single field value, optionally accompanied by a filename.

CCookie–a single cookie

CCookies–a cookie container

Note: In the following libraries you will see references to the following typedefs: CNcbiOstream and CNcbiIstream. On Solaris and NT, these are identical to the standard library output stream (ostream) and input stream (istream) classes. These typedefs are used on older computers to switch between the old stream library and the new standard library stream classes. Further details can be found in an accompanying document (to be written).

A demo program, cgidemo.cpp, can be found in internal/c++/src/corelib/demo.

CCgiRequest

CCgiRequest is the class that reads in the input from the web server and makes it accessible to the CGI program.

CCgiRequest uses the following typedefs to simplify the code:

```
typedef map<string, string> TCgiProperties
typedef multimap<string, CCgiEntry> TCgiEntries
typedef TCgiEntries::iterator TCgiEntriesI
typedef list<string> TCgiIndexes
```

All of the basic types come from the C++ Standard library (http://www.sgi.com/Technology/STL/)

CCgiRequest(int argc, char* argv[], CNcbiIstream* istr=0, bool indexes_as_entries=true)

A CGI program can receive its input from three sources: the command line, environment variables, and an input stream. Some of this input is given to the CCgiRequest class by the following arguments to the constructor:

int argc, char* argv[] : standard command line arguments.

CNcbiIstream* istr=0 : the input stream to read from. If 0, reads from stdin, which is what most web servers use.

bool indexes_as_entries=true : if query has any ISINDEX like terms (i.e. no "=" sign), treat it as a form query (i.e. as if it had an "=" sign).

Example:

```
CCgiRequest * MyRequest = new CCgiRequest(argc, argv);
```

const TCgiEntries& GetEntries(void) const

Get a set of decoded form entries received from the web browser. So if you sent a cgi query of the form ?name=value, the multimap referenced by TCgiEntries& includes "name" as a .first member and <"value", ""> as a .second member.

TCgiEntries& also includes "indexes" if "indexes_as_entries" in the constructor was "true".

const TCgiIndexes& GetIndexes(void) const

This performs a similar task as GetEntries(), but gets a set of decoded entries received from the web browser that are ISINDEX like terms (i.e. no "=" sign),. It will always be empty if "indexes_as_entries" in the constructor was "true"(default).

const string& GetProperty(ECgiProp prop) const

Get the value of a standard property (empty string if not specified). See the "Standard properties" list below.

static const string& GetPropertyName(ECgiProp prop)

The web server sends the CGI program properties of the web server and the http headers received from the web browser (headers are simply additional lines of information sent in a http request and response). This API gets the name(not value!) of standard properties. See the "Standard properties" list below.

Standard properties:

```
eCgi_ServerSoftware ,
eCgi_ServerName,
eCgi_GatewayInterface,
eCgi_ServerProtocol,
eCgi_ServerPort, // see also "GetServerPort()"
// client properties
eCgi_RemoteHost,
eCgi_RemoteAddr, // see also "GetRemoteAddr()"
// client data properties
eCgi_ContentType,
eCgi_ContentLength, // see also "GetContentLength()"
// request properties
eCgi_RequestMethod,
eCgi_PathInfo,
eCgi_PathTranslated,
eCgi_ScriptName,
eCgi_QueryString,
// authentication info
eCgi_AuthType,
eCgi_RemoteUser,
eCgi_RemoteIdent,
// semi-standard properties(from HTTP header)
eCgi_HttpAccept,
eCgi_HttpCookie,
eCgi_HttpIfModifiedSince,
eCgi_HttpReferer,
eCgi_HttpUserAgent
```

const string& GetRandomProperty(const string& key) const

Gets value of any http header that is passed to the CGI program using environment variables of the form "$HTTP_<key>". In general, these are special purpose http headers not included in the list above.

Uint2 GetServerPort(void) const

Gets the server port used by web browser to access the server.

size_t GetContentLength(void) const

Returns the length of the http request.

const CCgiCookies& GetCookies(void) const

Retrieve the cookies that were sent with the request. Cookies are text buffers that are stored in the user's web browsers and can be set and read via http headers. See the CCookie and CCookies classes defined below.

static SIZE_TYPE ParseEntries(const string& str, TCgiEntries& entries)

This is a helper function that isn't normally used by CGI programs. It allows you to decode the URL-encoded string "str" into a set of entries <"name", "value"> and add them to the "entries" multimap. The new entries are added without overriding the original ones, even if they have the same names. If the "str" is in ISINDEX format then the entry "value" will be empty. On success, return zero; otherwise return location(1-base) of error.

static SIZE_TYPE ParseIndexes(const string& str, TCgiIndexes& indexes)

This is also a helper function not usually used by CGI programs. This function decodes the URL-encoded string "str" into a set of ISINDEX-like entries (i.e. no "=" signs in the query) and adds them to the "indexes" set. On success, return zero, otherwise return location(1-base) of error.

CCgiResponse

CCgiResponse is the object that takes output from the CGI program and sends it to the web browser via the web server.

CNcbiOstream& WriteHeader() const

CNcbiOstream& WriteHeader(CNcbiOstream& out) const

This writes the MIME header necessary for all documents sent back to the web browser. By default, this function assumes that the "Content-type" is "text/html". Use the second form of the function if you want to use a stream other that the default.

void SetContentType(const string &type)

Sets the content type. By default this is "text/html". For example, if you were to send plaintext back to the client, you would set type to "text/plain".

string GetContentType(void) const

Retrieves the content type.

CNcbiOstream& out(void) const

This returns a reference to the output stream being used by the CCgiResponse object. Example:

```
CCgiResponse Response;
Response.WriteHeader();
Response.out() << "hello, world" << flush;
```

CNcbiOstream* SetOutput(CNcbiOstream* out)

Sets the default output stream. By default this is stdout, which is what most web servers use.

CNcbiOstream* GetOutput(void) const

Get the default output stream.

void Flush() const

Flushes the output stream.

void SetRawCgi(bool raw)

Turns on non-parsed cgi mode. When this is turned on AND the name of the cgi program begins with "nph-", then the web server does no processing of the data sent back to the client. In this situation, the client must provide all appropriate http headers. This boolean switch causes some of these headers to be sent.

bool IsRawCgi(void) const

Check to see if non-parsed cgi mode is on.

void SetHeaderValue(const string& name, const string& value)

Sets an http header with given name and value. For example, SetHeaderValue("Mime-Version", "1.0"); will create the header "Mime-Version: 1.0".

void SetHeaderValue(const string& name, const tm& value)

Similar to the above, but sets a header value using a date. See time.h for the definition of tm.

void RemoveHeaderValue(const string& name)

Remove the header with name name.

string GetHeaderValue(const string& name) const

Get the value of the header with name name.

bool HaveHeaderValue(const string& name) const

Check to see if the header with name name exists.

void AddCookie(const string& name, const string& value) void AddCookie(const CCgiCookie& cookie)

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

Add a cookie to the response. This can either be a name, value pair or use the CCookie class described below.

void AddCookies(const CCgiCookies& cookies)

Add a set of cookies to the response. See the CCookies class described below.

const CCgiCookies& Cookies(void) const CCgiCookies& Cookies(void)

Return the set of cookies to be sent in the response.

void RemoveCookie(const string& name)

Remove the cookie with the name name.

void RemoveAllCookies(void)

Remove all cookies.

bool HaveCookies(void) const

Are there cookies?

bool HaveCookie(const string& name) const

Is there a cookie with the given name?

CCgiCookie* FindCookie(const string& name) const

Return a cookie with the given name.

CCgiCookie

A cookie is a name, value string pair that can be stored on the user's web browser. Cookies are allocated per site and have restrictions on size and number. Cookies have attributes, such as the domain they originated from. CCgiCookie is used by the CCgiRequest and CCgiResponse classes.

CCgiCookie(const string& name, const string& value)

Creates a cookie with the given name and value. Throw the "invalid_argument" if "name" or "value" have invalid format:

- the "name" must not be empty; it must not contain '='
- both "name" and "value" must not contain: ";, "

const string& GetName (void) const

Get the cookie name. The cookie name cannot be changed.

CNcbiOstream& Write(CNcbiOstream& os) const

Write the cookie out to ostream os. Normally this is handled by CCgiResponse.

void Reset(void)

Reset everything but the name to the default state

void CopyAttributes(const CCgiCookie& cookie)

Set all attribute values(but name!) to those from "cookie"

void SetValue (const string& str) void SetDomain (const string& str) void SetValidPath (const string& str) void SetExpDate (const tm& exp_date) void SetSecure (bool secure) // "false" by default

These function set the various properties of a cookie. These functions will throw "invalid_argument" if "str" has invalid format. For the definition of tm, see time.h.

bool GetValue (string* str) const bool GetDomain (string* str) const bool GetValidPath (string* str) const bool GetExpDate (string* str) const bool GetExpDate (tm* exp_date) const bool GetSecure (void) const

These functions return true if the property is set. They also return value of the property in the argument. If the property is not set, str is emptied. These functions throw the "invalid_argument" exception if the argument is a zero pointer.

The string version of GetExpDate will return a string of the form "Wed Aug 9 07:49:37 1994"

CCgiCookies

CCgiCookies aggregates a collection of CCgiCookie

CCgiCookies(void) CCgiCookies(const string& str)

Creates a CCgiCookies container. To initialize it with a cookie string, use the format: "name1=value1; name2=value2; ..."

CCgiCookie* Add(const string& name, const string& value)

Add a cookie with the given name, value pair. Note the above requirements on the string format. Overrides any previous cookie with same name.

CCgiCookie* Add(const CCgiCookie& cookie)

Add a CCgiCookie.

void Add(const CCgiCookies& cookies)

Adds a CCgiCookie of cookies.

void Add(const string& str)

Adds cookies using a string of the format "name1=value1; name2=value2; ..." Overrides any previous cookies with same names.

CCgiCookie* Find(const string& name) const

Looks for a cookie with the given name. Returns zero not found.

bool Empty(void) const

"true" if contains no cookies.

bool Remove(const string& name)

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

Find and remove a cookie with the given name. Returns "false" if one is not found.

void Clear(void)

Remove all stored cookies

CNcbiOstream& Write(CNcbiOstream& os) const

Prints all cookies into the stream "os" (see also CCgiCookie::Write()). Normally this is handled by CCgiResponse.

An example web-based CGI application

Introduction

The previous two chapters described the NCBI C++ Toolkit's CGI and HTML classes, with an emphasis on their independence from one another. In practice however, a real application must employ both types of objects, with a good deal of inter-dependency.

As described in the description of the CGI classes, the CNcbiResource class can be used to implement an application whose functionality varies with the query string. Specifically, the resource class contains a list of CNcbiCommand objects, each of which has a defined GetName () and GetEntry()method. The only command selected for execution on a given query is the one whose GetName() and GetEntry() values match the leading key=value pair in the query string.

The CHelloResource class has different commands which will be executed depending on whether the query string invoked an init or a reply command. For many applications however, this selection mechanism adds unnecessary complexity to the interface, as the application always performs the same function, albeit on different input. In these cases, there is no need to use a CNcbiResource object, or CNcbiCommand objects, as the necessary functionality can be encoded directly in the application's ProcessRequest() method. The example program described in this section uses this simpler approach.

Program description

The car.cgi program presents an HTML form for ordering a custom color car with selected features. The form includes a group of checkboxes (listing individual features) and a set of radio buttons listing possible colors. Initially, no features are selected, and the default color is black. Following the form, a summary stating the currently selected features and color, along with a price quote, is displayed. When the submit button is clicked, the form generates a new query string (which includes the selected features and color), and the program is restarted.

The program uses a CHTMLPage object with a template file (car.html) to create the display. The template file contains three <@tag@> locations, which the program uses to map CNCBINodes to the page, using the AddTagMap() method. Here is an outline of the execution sequence:

Create an instance of class CCar named car.

Load car with the color and features specified in the query string.

Create a CHTMLPage named page.

Generate a CHTML_form object using the features and color currently selected for car, and map that HTML form to the <@FORM@> tag in page.

Generate the summary statement and save it in a CHTMLText node mapped to the <@SUMMARY@> tag.

Generate a price quote and save it in a CHTMLText node mapped to the <@PRICE@> tag.

Output page and exit.

The CCar created in step 1 initially has the default color (black) and no features. Any features or colors specified in the query string with which the program was invoked are added to car in step 2, prior to generating the HTML display elements. In step 4, the form element is created using the set of possible features and the set of possible colors. These sets of attributes are stored as static data members in an external utility class, CCarAttr. Each feature corresponds to a CHTML_checkbox element in the form, and each color corresponds to a CHTML_radio button. The selected color, along with all currently selected features, will be displayed as selected in the form.

The summary statement uses a CHTML_ol list element to itemize the selected features in car. The price is calculated as CCar::m_BasePrice plus an additional $1000 per feature. The submit button generates a fresh page with the new query string, as the action attribute of the form is the URL of car.cgi.

Program design: Distributing the work

The program uses three classes: CCar, CCarAttr, and CCarCgi. The CCar class knows nothing about HTML nodes or CGI objects - its only functions are to store the currently selected color and features, and compute the resulting price:

```
class CCar
{
public:
 CCar(unsigned base_price = 12000) { m_BasePrice = base_price; }
 // Mutating member functions
 void AddFeature(const string& feature_name);
 void SetColor(const string& color_name);
 // Access member functions
 bool HasFeature(const string& feature_name) const;
 string GetColor(void) const;
 string GetPrice(void) const;
 const set<string>& GetFeatures() const;
private:
 set<string> m_Features;
 string m_Color;
 unsigned m_BasePrice;
};
```

Instead, the CCar class provides an interface to all of its data members, thus allowing the application to get/set features of the car as needed. The static utility class, CCarAttr, simply provides the sets of possible features and colors, which will be used by the application in generating the HTML form for submission:

```
class CCarAttr {
public:
 CCarAttr(void);
 static const set<string>& GetFeatures(void) { return sm_Features; }
 static const set<string>& GetColors (void) { return sm_Colors; }
private:
 static set<string> sm_Features;
 static set<string> sm_Colors;
};
```

Both of these classes are defined in a header file which is #include'd in the *.cpp files. Finally, the application class does most of the actual work, and this class must know about CCar, CCarAttr, HTML, and CGI objects. The CCarCgi class has the following interface:

```
class CCarCgi : public CCgiApplication
{
public:
 virtual int ProcessRequest(CCgiContext& ctx);
private:
 CCar* CreateCarByRequest(const CCgiContext& ctx);
 void PopulatePage(CHTMLPage& page, const CCar& car);
 static CNCBINode* ComposeSummary(const CCar& car);
 static CNCBINode* ComposeForm (const CCar& car);
 static CNCBINode* ComposePrice (const CCar& car);
 static const char sm_ColorTag[];
 static const char sm_FeatureTag[];
};
```

The source code is distributed over three files:

car.hpp

car.cpp

car_cgi.cpp

The CCar and CCarAttr classes are defined in car.hpp, and implemented in car.cpp. Both the class definition and implementation for the CGI application class are in car_cgi.cpp. With this design, only the application class will be affected by changes made to either the HTML or CGI class objects. The additional files needed to compile and run the program are:

car.html

Makefile.car_app

CGI Response Codes

Wherever possible the client when encountering errors should return an appropriate response code consisting of the three digits *DDD* . In the case of client error codes, these begin with a "4" (4xx). Table 7 contains a summary of these codes.

> Note that error code 404 should be reserved for situations when the requested file does not exist. It should not be used as a "catch-all" such as when the client simply uses bogus parameters.

Table 7. CGI Client Error Codes (4XX)

Error Code	Description
400	Bad request; the client erred in the request and should not reattempt it without modifications.
401	Unauthorized; the page is password protected but required credentials were not presented.
402	Payment required; reserved.
403	Forbidden; the client is not allowed here.
404	Not found; the requested resource (as indicated in the path) does not exist on the server, temporarily or permanently.
405	Method not allowed; the server must supply allowedrequest methods in "Allow:" HTTP header.
406	Not acceptable; content characteristics are unacceptable to produce the response.
407	Proxy authentication required; similar to 401, but for proxy.
408	Request timeout; the client does not furnish the entire request within the allotted time.
409	Conflict; usually means bad form submission via PUT method.
410	Gone; the resource is and will be no longer available and forwarding address is and will not be known.
411	Length required; the client must use content-length in the request.
412	Precondition failed; request header inquired for a condition that doesn't hold.
413	Request too big; self-explanatory.
414	Request too long; query-line element is exceeding the maximal size(but the body, if any, can be okay).
415	Unsupported media; resource does not support requested format.
416	Bad range; pertains to multi-part messages when the client requested a fragment that is out of allowed range.
417	Expectation failed; "Expect:" from the HTTP/1.1 header is not met.

FCGI Redirection and Debugging C++ Toolkit CGI Programs

Development, testing, and debugging of CGI applications can be greatly facilitated by making them FastCGI-capable and using an FCGI redirector script.

Applications that were written to use the C++ Toolkit CGI framework (see example above) can easily be made to run under your account, on your development machine, and in a number of ways (e.g. standalone, with special configuration, under a debugger, using a memory checker, using strace, etc.). This is accomplished by "tunneling" through a simple FCGI redirector script that forwards HTTP requests to your application and returns the HTTP responses.

The process is described in the following sections:

- Creating and debugging a sample FastCGI application
- Debugging an existing CGI or FCGI application

Creating and debugging a sample FastCGI application

If you are starting from scratch, use the new_project script to create a CGI that is already set up for FCGI redirection:

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

```
new_project foobar app/cgi
cd foobar
make
```

This creates the following files:

File	Purpose
ffoobar.fcgi	This is the FCGI version of the foobar.cgi application and should be run when debugging.
ffoobar.cgi	This is the FCGI redirector script. It is the CGI called from the browser, and it is the only file that needs to be copied to the web server for FCGI redirection.
ffoobar.ini	This can be edited as desired (e.g. setting [FCGI] Iterations = 1). Note: Unlike "plain" CGIs, FastCGI applications usually handle more than one HTTP request. While they do it sequentially, and there are no multithreading issues, additional care should still be taken to ensure that there is no unwanted interference between different HTTP requests, and that operation is correct under more strict resource (such as heap) accounting. So, to verify how a FastCGI application will work in real life you should test it with the number of iterations greater than 1.
foobar.cgi	This is the CGI application. Use it if you want to directly invoke your application as a CGI. It isn't used in the FCGI redirection process.
foobar.html	This is the web page that your CGI / FCGI applications will load. The name is the same for both CGI and FCGI. Note: By default, the HTML page must be located with your application and it must match your project name (but with the .html extension). The HTML page name can be changed in the ProcessRequest() method.

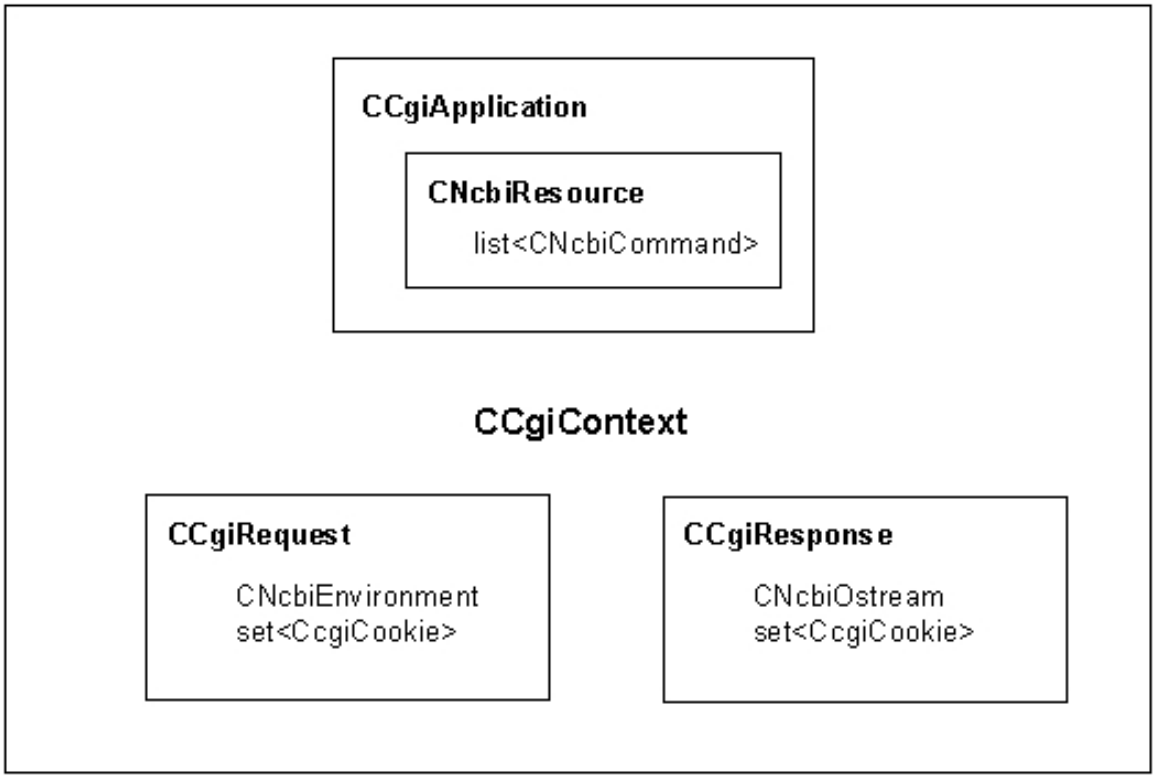
The steps to debugging your new application using FCGI redirection are:

1. Install the ffoobar.cgi redirector script on the web server.
2. Set up the application:
    a. Configure with ffoobar.ini - set [FastCGI] Iterations = 1.
    b. Set a breakpoint on ProcessRequest() and run ffoobar.fcgi under the debugger (or run under a memory checker or other tool).
3. From your web browser (or using GET/POST command-line utilities), submit a web request to ffoobar.cgi. The request/response will be tunneled to/from ffoobar.fcgi.

Debugging an existing CGI or FCGI application

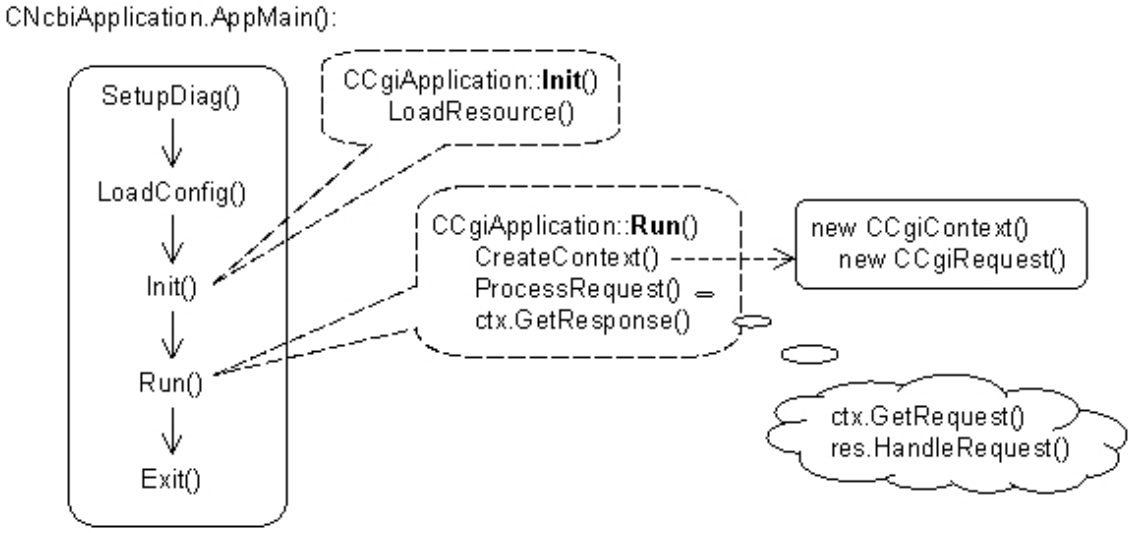
The steps to debugging an existing CGI or FCGI application using FCGI redirection are (assuming the name is foobar.cgi):

1. If it's a "plain" CGI, then make it FastCGI-capable - change the makefile to build ffoobar.fcgi instead of foobar.cgi and to link with xfcgi.lib instead of xcgi.lib. Note: the application must use the C++ Toolkit's CGI framework (as in the above example).
2. Rebuild the application.
3. Install the ffoobar.cgi redirector script on the web server (in place of the existing CGI).
4. Set up the application:
    a. Copy the application ffoobar.fcgi to a development host.
    b. Configure with ffoobar.ini - set [FastCGI] Iterations = 1.
    c. Set a breakpoint on ProcessRequest() and run ffoobar.fcgi under the debugger (or run under a memory checker or other tool).
5. From your web browser (or using GET/POST command-line utilities), submit a web request to ffoobar.cgi. The request/response will be tunneled to/from ffoobar.fcgi.



1. .

Figure 1. Layered design of the CGI classes



2. .

Figure 2. Adapting the init() and run() methods inherited from CNcbiApplication

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

Table 1. Restrictions on arguments to the CCgiCookie constructor

Field	**Restrictions**
name (required)	No spaces; must be printable ASCII; cannot contain = , or ;
value (required)	No spaces; must be printable ASCII; cannot contain , or ;
domain (optional)	No spaces; must be printable ASCII; cannot contain , or ;
path (optional)	Case sensitive

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

Table 2. Effect of setting the diag-destination parameter

value	effects
stderr	Send diagnostics to the standard error stream (default behavior)
asbody	Send diagnostics to the client in place of normal output

Box 1

```
// File name: justcgi.cpp
// Description: Demonstrate the basic CGI classes and functions
#include "justcgi.hpp"
#include <cgi/cgictx.hpp>
#include <corelib/ncbistd.hpp>
#include <corelib/ncbireg.hpp>
#include <memory>
USING_NCBI_SCOPE;
////////////////////////////////////////////////////////////////////////
// Implement the application's LoadResource() and ProcessRequest() methods
CNcbiResource* CCgiApp::LoadResource(void)
{
 auto_ptr<CCgiResource> resource(new CCgiResource(GetConfig()));
 resource->AddCommand(new CCgiBasicCommand(*resource));
 resource->AddCommand(new CCgiReplyCommand(*resource));
 return resource.release();
}
// forward declarations
void ShowCommands (const TCmdList& cmds, CCgiContext& ctx);
void ShowEntries (const TCgiEntries& entries);
int CCgiApp::ProcessRequest(CCgiContext& ctx)
{
 ShowCommands (GetResource().GetCmdList(), ctx);
 ShowEntries (const_cast<TCgiEntries&>(ctx.GetRequest().GetEntries()));
 GetResource().HandleRequest(ctx);
 return 0;
}
////////////////////////////////////////////////////////////////
// Define the resource's default command if none match queryCNcbiCommand*
CCgiResource::GetDefaultCommand(void) const
{
 cerr << " executing CCgiResource::GetDefaultCommand()" << endl;
 return new CCgiBasicCommand(const_cast<CCgiResource&>(*this));
}
////////////////////////////////////////////////////////////////
// Define the Execute() and Clone() methods for the commands
void CCgiCommand::Execute(CCgiContext& ctx)
{
 cerr << " executing CCgiCommand::Execute " << endl;
 const CNcbiRegistry& reg = ctx.GetConfig();
 ctx.GetResponse().WriteHeader();
}
```

```
CNcbiCommand* CCgiBasicCommand::Clone(void) const
{
 cerr << " executing CCgiBasicCommand::Clone()" << endl;
 return new CCgiBasicCommand(GetCgiResource());
}
CNcbiCommand* CCgiReplyCommand::Clone(void) const
{
 cerr << " executing CCgiReplyCommand::Clone" << endl;
 return new CCgiReplyCommand(GetCgiResource());
}
// Show what commands have been installed
void ShowCommands (const TCmdList& cmds, CCgiContext& ctx)
{
 cerr << "Commands defined for this application are: \n";
 for (TCmdList::const_iterator it = cmds.begin();
 it != cmds.end(); it++) {
 cerr << (*it)->GetName();
 if ((*it)->IsRequested(ctx)) {
 cerr << " (requested)" << endl;
 } else {
 cerr << " (not requested)" << endl;
 }
 }
}
// Show the <key=value> pairs in the request string
void ShowEntries (const TCgiEntries& entries)
{
 cerr << "The entries in the request string were: \n";
 for (TCgiEntries::const_iterator it = entries.begin();
 it != entries.end(); it++) {
 if (! (it->first.empty() && it->second.empty()))
 cerr << it->first << "=" << it->second << endl;
 }
}
static CCgiApp theCgiApp;
int main(int argc, const char* argv[])
{
 SetDiagStream(&cerr);
 return theCgiApp.AppMain(argc, argv);
}
```

Box 2

```
// File name: justcgi.hpp
// Description: Demonstrate the basic CGI classes and functions
#ifndef CGI_HPP
#define CGI_HPP
#include <cgi/cgiapp.hpp>
#include <cgi/ncbires.hpp>
USING_NCBI_SCOPE;
class CCgiApp : public CCgiApplication
```

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

```
{
public:
 virtual CNcbiResource* LoadResource(void);
 virtual int ProcessRequest(CCgiContext& context);
};
class CCgiResource : public CNcbiResource
{
public:
 CCgiResource(CNcbiRegistry& config)
 : CNcbiResource(config) {}
 virtual ~CCgiResource() {};
 // defines the command to be executed when no other command matches
 virtual CNcbiCommand* GetDefaultCommand( void ) const;
};
class CCgiCommand : public CNcbiCommand
{
public:
 CCgiCommand( CNcbiResource& resource ) : CNcbiCommand(resource) {};
 virtual ~CCgiCommand( void ) {};
 virtual void Execute( CCgiContext& ctx );
 virtual string GetLink(CCgiContext&) const { return NcbiEmptyString; }
protected:
 CCgiResource& GetCgiResource() const
 {
 return dynamic_cast<CCgiResource&>( GetResource() );
}
 virtual string GetEntry() const { return string("cmd"); }
};
class CCgiBasicCommand : public CCgiCommand
{
public:
 CCgiBasicCommand(CNcbiResource& resource) : CCgiCommand(resource) {};
 virtual ~CCgiBasicCommand(void) {};
 virtual CNcbiCommand* Clone( void ) const;
 virtual string GetName( void ) const { return string("init"); };
protected:
 virtual string GetEntry() const { return string("cmd1"); }
};
class CCgiReplyCommand : public CCgiBasicCommand
{
public:
 CCgiReplyCommand( CNcbiResource& resource) : CCgiBasicCommand(resource)
{};
 virtual ~CCgiReplyCommand(void) {};
 virtual CNcbiCommand* Clone( void ) const;
 virtual string GetName( void ) const { return string("reply"); };
protected:
 virtual string GetEntry() const { return string("cmd2"); }
};
```

```
#endif /* CGI_HPP */
```

Box 3

```
# Author: Diane Zimmerman
# Build CGI application "CGI"
# NOTE: see to build Fast-CGI
################################
APP = cgi
OBJ = cgiapp
LIB = xcgi xncbi
```

Box 4

```
# Author: Diane Zimmerman
# Build test Fast-CGI application "FASTCGI"
# NOTES: - it will be automagically built as a plain CGI application if
# Fast-CGI libraries are missing on your machine.
# - also, it auto-detects if it is run as a FastCGI or a plain
# CGI, and behave appropriately.
################################
APP = fastcgi
OBJ = cgiapp
LIB = xfcgi xncbi
LIBS = $(FASTCGI_LIBS) $(ORIG_LIBS)
```