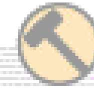
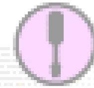
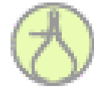




The NCBI C++ Toolkit

"The time has come
to talk of many things."
— Lewis Carrol

Without whom none of this…

- **Denis Vakatov**- Manager, corelib, configure, cgi, connect and Doc.
- Eugene Vasilchenkov- corelib, cgi, HTML, serial, datatool
- Anton Lavrentiev- connect, ctool, MSVC++ ,doc.
- Aleksey Grichenko- corelib, serial, datatool, objmgr
- Daina Zimmerman- Documentation.
- Vladimir Ivanov- HTML, corelib, util.
- Michael Kholodov, Vladimir Soussov- Database API, DB driver, serial, datatool.
- Vsevolod Sandomirskiy- corelib ,cgi.
- Anton Butanaev- high level DBAPI.
- Mikhael Kimelman- objmgr in progress.
- And many more…

The NCBI C++ Toolkit

- Contains more biological classes than any other toolkit
- An extensive set of functions to create, store and compute biological information
- Code generation utility that converts data structure definitions to working code
- Cross platform
- Easy to use utilities for writing stand-alone and cgi applications
- Easy to integrate individual projects with existing NCBI code
- Well-documented and easy to install
- **It’s free!**

License

“With the exception of certain third-party files …, this software is a "United States Government Work" under the terms of the United States Copyright Act. It was written as part of the authors' official duties as United States Government employees and thus cannot be copyrighted. This software is freely available to the public for use. The National Library of Medicine and the U.S. Government have not placed any restriction on its use or reproduction.”

The Menu

- Modules overview
- Datatool
- Writing a simple application
- Serialization
- DBPAI
- Object Manager
- CGI and HTML

The NCBI C++ Modules

- CORELIB- Platform code, application frame work, argument processing, diagnostics and exception, templates utilities, threads, Object and Ref classes and much more.
- ALGORITHM- Needleman-Wunsch, cDNA/mRNA-to-genomic algorithm and BLAST C++ API.
- CGI- Defines a CGI-specific application, deals with incoming CGI requests, posting responses, CGI-context (for FAST CGI), and cookies read/write.

More Modules

- HTML- Generating HTML pages from a program, HTML tag classes, interfacing with CGI applications, support for using template HTML pages.
- CONNECT- A C++ interface to the standard sockets SOCK API.
- CTOOLS- Bridging between the current C++ and old C toolkits.
- GUI- Basic functionality to display, navigate and highlight sequences. Uses OpenGL and FLTK third party graphic libs.

More Modules

- DBAPI- Provides a common interface to different RDBMS. Models a database as a data source that can be accessed and queried through SQL. Supports Sybase (CTLIB/DBLIB), Microsoft DBLIB, FreeTDS and ODBC.
- SERIAL- ASN.1 serialization. Deals with all aspects for reading, writing, and transferring between independent processes.
- OBJECT MANAGER- facilitate access to biological sequence data. It can transparently download data from the GenBank database, investigate biological sequence data structure, retrieve sequence data, descriptions and annotations.
- UTIL- Useful miscellaneous classes such as Checksum, Console debug dump, Lightweight string, Random number generator, regexp matching and more.

In the beginning there was ASN.1

Like in the C toolkit, the C++ has a tool to generate C++ code from ASN.1 specification called datatool (/src/serial/datatool). Datatool can:

1. Read ASN.1 data specification
2. Read/write ASN.1 objects or XML data
3. Write ASN.1 or DTD data specification
4. Generate C++ classes to serialize (read/write) the data

In addition to creating the read and write operation for every class, datatool will also implement *GetXX()*, *SetXX()*, *ResetXX()* and sometimes *IsSetXX()* functions for every variable in the module. Ready to use class; no need to write parsers.

For every ASN.1 module there are two classes generated: *Cmodulename_Base* and *Cmodule*.

Lets start with a small example…

```
Biostruc ::= SEQUENCE {
      id SEQUENCE OF Biostruc-id,
      descr SEQUENCE OF Biostruc-descr OPTIONAL,
      chemical-graph Biostruc-graph,
      features SEQUENCE OF Biostruc-feature-set
                        OPTIONAL,
      model SEQUENCE OF Biostruc-model OPTIONAL
 }
```

Now we run datatool

```
$datatool -m biostruc.asn –M external modules -od biostruc.def \
      -oA
```

```
class CBiostruc_Base : public CObject {
 public:
     // type definitions
     typedef list< CRef<CBiostruc_id> > TId;
     typedef list< CRef<CBiostruc_descr> > TDescr;
     typedef list< CRef<CBiostruc_feature_set> > TFeatures;
     typedef list< CRef<CBiostruc_model> > TModel;
     typedef CBiostruc_graph TChemical_graph;
     // Get() members
     const TId& GetId(void) const;
     const TDescr& GetDescr(void) const;
     const TChemical_graph& GetChemical_graph(void) const;
     const TFeatures& GetFeatures(void) const;
     const TModel& GetModel(void) const;
     // Set() members
     TId& SetId(void);
     TDescr& SetDescr(void);
     TChemical_graph& SetChemical_graph(void);
     TFeatures& SetFeatures(void);
     TModel& SetModel(void);
 private:
     TId m_Id;
     TDescr m_Descr;
     TChemical_graph m_Chemical_graph;
     TFeatures m_Features;
     TModel m_Model;
 };
```

Generated code

For every ASN.1 module there are 4 files created, two *.hpp and two *.cpp. The *base class* contains the class declaration(*_.hpp) and definition (*_.cpp). *User classes* are empty wrapper classes that do not add any functionality to the base class. They are simply provided as a platform for development. *Base classes* should never be used directly and are not intended for public use.

It is not recommended that you add or change functions to the NCBI ASN.1 types. In your own type, however, you can add and change functions.

N.B. Be careful, unless you check in the generated code into CVS you might overwrite any additional code you added to the *user class* by running datatool. Often, ASN.1 modules undergo changes and the code is regenerated.

Fine tuning

- It is possible to determine what data types datatool is going to use for variables. Very useful in container types, or namespace. This is done in a definition file specified by the *–od* argument when running datatool.
- The file structure is similar to *.ini files.
- In the Biostruc example, to generate a vector container instead of list we can write:

```
[Biostruc]

 descr._type=vector
```

- There are many more options that can control the generated code. Read documentation.

ASN.1 choice elements

```
Object-id ::= CHOICE {
        id INTEGER,
        str VisibleString }
```

asntool

```
typedef struct valnode
{
        unsigned choice;
         DataVal data;
         struct valnode
*next;
 } ValNode;
```

datatool

```
enum E_Choice {
        e_not_set,
        e_Id,
        e_Str    };
    ...
    typedef int TId;
    typedef string TStr;
    ...
    union {
        TId m_Id;
        string *m_string;
        };
```

C++ choice objects

- For each choice variant there is an enumeration value (in *E_Choice*), a typedef is defined and union data member.
- Private data members store information about the currently selected choice variant: *m_choice* holds the enumeration value and *m_Xxx* holds data value.
- For each choice option there is a *GetXxx()* and *SetXxx()*.
- *Which()* returns which enumeration (*E_Choice enum* value) type is set in this object.
- Each *GetXxx()* function throws an exception if the data type for that method is not the current selection type. Unlike in the C toolkit, it is not possible to get the incorrect type of choice variant.

Example

```
void Visit (const CBiostruc::TId& idList)
 {
    for (CBiostruc::TId::const_iterator i = idList.begin();
     i != idList.end(); ++i) {
         // dereference the iterator to get to the id object
         const CBiostruc_id& thisId = **i;
         CBiostruc_id::E_Choice choice = thisId.Which();
         cout << "choice = " << choice;
         // select id's get member function depending on choice
         switch (choice) {
         case CBiostruc_id::e_Mmdb_id:
         cout << " mmdbId: " << thisId.GetMmdb_id().Get() << endl;
         break;
         case CBiostruc_id::e_Local_id:
         cout << " Local Id: " << thisId.GetLocal_id().GetId() << endl;
         break;
         case CBiostruc_id::e_Other_database:
         cout << " Other DB Id: " << thisId.GetOther_database().GetDb() <
         break;
        default:
         cout << "Choice not set or unrecognized" << endl;
        }
     }
 }
```

NCBI application classes

Five fundamental classes form the foundation of an NCBI C++ toolkit application.

1. *CNcbiApplication* – the Big Kahuna!
   - Provides the mechanism to execute the application.
   - Contains a data structure to get, hold and validate the program command-line arguments.
   - Contains a data structure to hold environment variables.
   - When and where errors are reported.
   - Contains methods to read, and modify config files.

The other four..

2. *CNcbiArguments* – holding the application's command-line arguments, along with methods for accessing and modifying them.
3. *CNcbiEnvironment* – storing, accessing, and modifying the environment variables accessed by the C library routine getenv().
4. *CNcbiRegistry* – load, access, modify and store runtime information read from a configuration file.
5. *CNcbiDiag* – class implements much of the functionality of the NCBI C Toolkit error processing mechanisms.

Step by step…

```
// File name: justApp.hpp
#ifndef JUST_APP__HPP
#define JUST_APP__HPP
#include <corelib/ncbiapp.hpp>
BEGIN_NCBI_SCOPE

class CTestApp : public CNcbiApplication
{
 public:
     virtual int Run(void);
};
END_NCBI_SCOPE
#endif
```

```
// file justApp.cpp
 #include "justApp.hpp"

 BEGIN_NCBI_SCOPE

 int CTestApp::Run()
 {
     cout << "Executing CTestApp::Run()!"
                       << endl;
     return 0;
 }

 END_NCBI_SCOPE

 USING_NCBI_SCOPE;

 int main(int argc, const char* argv[])
 {     CTestApp theTestApp;
     return theTestApp.AppMain(argc, argv);
 }
```

Look at *applic.cpp* in the documentation for a more elaborate example of using the *CNcbiApplication* class with the *CArgDescriptions*, *CArgs*, and *CNcbiDiag* classes.

Command-line arguments

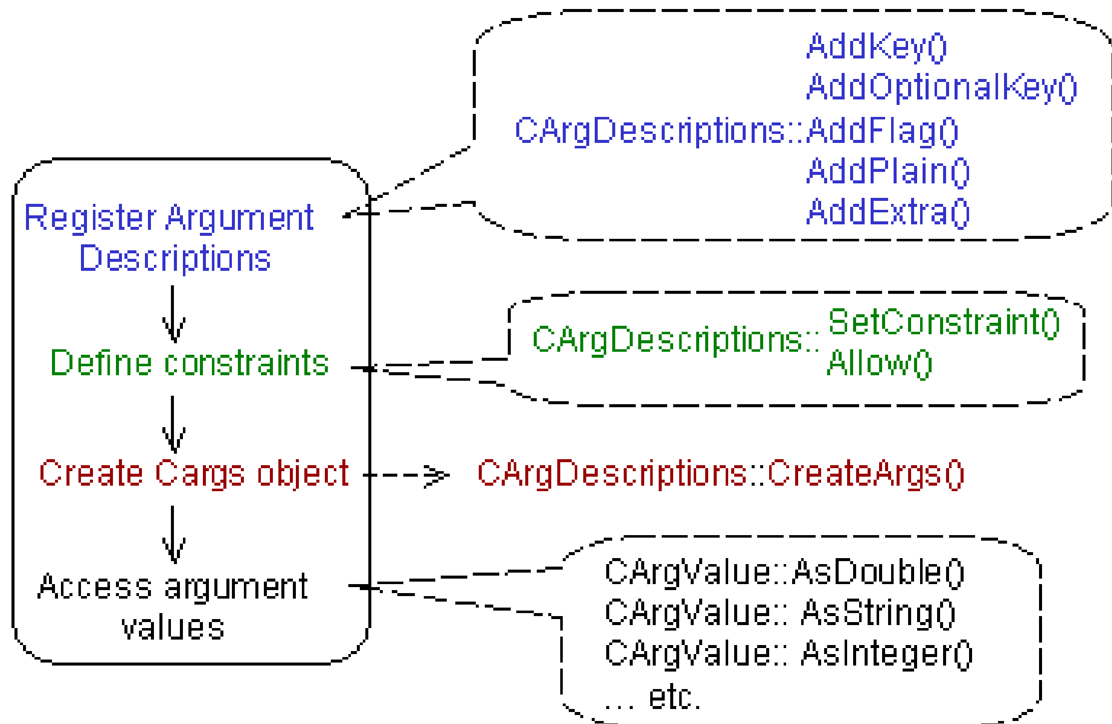
Example:

```
CArgDescriptions descrip;
descrip.AddKey("f", "float",
                "0 < f < 1", eFloat, 0);

SetConstraint(eEqual, 0);

CArgs myArgs = descrip.CreateArgs
                (GetArguments());


float f = myArgs["f"].AsDouble();
```



Diagnostic Stream

The CNcbiDiag class implements the functionality of an output stream for error posting. A CNcbiDiag is not an output stream but rather provides an interface to a stream that allow multiple threads to write to the same output.

In main()

```
//initialize diagnostic stream
 CNcbiOfstream diag(“myprogram.log");
 SetDiagStream(&diag, true);
 SetDiagPostLevel(eDiag_Info);
 // set the severity level where execution is aborted
 SetDiagDieLevel(eDiag_Error);
 SetDiagTrace(eDT_Enable);
 SetThrowTraceAbort(true);
```

In the program we can write…

```
ERR_POST(“ error message”);
```

Will write “foo.cpp, line xyz: Info: error message” to myprogram.log or write

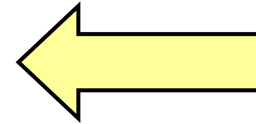
```
ERR_POST(Warning<< “ warning message”);
```

Will write “foo.cpp, line xyz: Warning: warning message” to myprogram.log

ASN.1 Objects I/O

a {
short-label "PDGF-B" ,
id
protein {
gi 4505681 ,
di 0 } ,
origin
org {
org {
taxname "Homo sapiens" ,
db {
{
db "taxon" ,
tag
id 9606 } } } } ,
descr "platelet derived growth factor B-chain. Aliases: PDGF B-chain,
PDGF-2, becaplermin. Note that the listed GI refers to an incomplete description of this molecule: [mat_peptide: 82-190]" ,
user-id 0 } ,

Read binary ASN.1



0110011010111110

Write in XML

<BIND-object>
<BIND-object_short-label>PDGF-B</BIND-object_short-label>
<BIND-object_id>
<BIND-object-type-id>
<BIND-object-type-id_protein>
<BIND-id>
<BIND-id_gi>
<Geninfo-id>4505681</Geninfo-id>
</BIND-id_gi>
<BIND-id_di>
<Domain-id>0</Domain-id>
</BIND-id_di>
</BIND-id>
</BIND-object-type-id_protein>
</BIND-object-type-id>

Format Specific Streams: The *CObject[IO]Stream* classes

- For read/write operation of ASN.1– derived classes we need two things:
  - Format-specific parsing and encoding scheme.
  - Object-specific internal structure.
- The C++ TK implemented serial IO classes that allow any *CObject*-derived class to be w/r in any format.
- *CObjectIStream* and *CObjectOStream* are the two bases classes that form the framework for this mechanism. Subclasses allow for r/w in XML, and binary/text ASN.1.
- Each *Datatool*-derived class has "Runtime Object Type Information" that is used by the the *CObject[IO]Stream* subclasses to r/w the data.

The *CObjectIStream* classes

- Acts as a virtual base class to the *XML, ASN and BinaryASN* readers.
- Several *open()* functions, most are static.

  auto_ptr<CObjectIStream> xml_in(CObjectIStream::Open(filename, eSerial_Xml));

- Can choose to load nested objects as part of the top level objects or as temporary objects using *ReadObject()* and *ReadSeperateObject().* By default the *Read()* function will read the entire struc.
- Similarly we can choose to skip over certain objects by using the *SkipObject().*

Output and Copy

- *CObjectOStream* classes are parallel to the input classes but uses write instead of read.
- There is a *CObjectStreamCopier* set of classes that allow the user to convert an object between types without storing the intermediate. Therefore, conversion between ASN.1 text to ASN.1 binary or XML to binary can be done in one line.
- For the more advanced users, you can write your own specialized I/O functions for serializable objects which will be used whenever that object is r/w. They are called Read/Write Hooks.

Examples

```
int CTestAsn::Run() {
     auto_ptr<CObjectIStream>
     xml_in(CObjectIStream::Open("1001.xml",
    eSerial_Xml));

    auto_ptr<CObjectOStream>
    txt_out(CObjectOStream::Open("1001.asntxt",
    eSerial_AsnText));

    auto_ptr<CObjectOStream>
    bin_out(CObjectOStream::Open("1001.asnbin",
    eSerial_AsnBinary));

    CBiostruc bs;
    *xml_in >> bs;
    *txt_out << bs;
    *bin_out << bs;
    return 0;
 }
```

Copy example

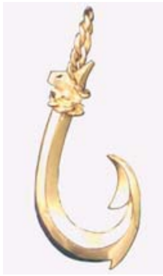
```
int CTestAsn::Run() {
     auto_ptr<CObjectIStream>
        xml_in(CObjectIStream::Open("1001.xml", eSerial_Xml));

     auto_ptr<CObjectOStream>
           txt_out(CObjectOStream::Open("1001.asntxt",
                                             eSerial_AsnText));

    CObjectStreamCopier txt_copier(*xml_in, *txt_out);
        txt_copier.copy(CBiostruc::GetTypeInfo());

     auto_ptr<CObjectOStream>
           bin_out(CObjectOStream::Open("1001.asnbin",
        eSerial_AsnBinary));

     CObjectStreamCopier bin_copier(*xml_in, *bin_out);
        bin_copier.copy(CBiostruc::GetTypeInfo());
        return 0;
    }
```



Hook classes

- Hook classes provide a mechanism to apply specific read and write operations for nested objects.
- They can be applied globally- all streams or locally, where they will be applied to selected streams.
- Typically used to skip, or write, specific data members.

Hook example

```
//hook class
class CDescrSkip : public CReadObjectHook{
  public: void ReadObject(CObjectIStream& in, const
                                CObjectInfo& object)
 {
      in.SkipObject(object); }
 };
 ...

      auto_ptr<CObjectIStream> in(CObjectIStream::Open(file,
                                      eSerial_AsnText));
      // setting hook on Seq-descr type
      CObjectTypeInfo type = CType<CSeq_descr>();
      type.SetLocalReadHook(*in, new CDescrSkip);
      CSeq_entry seq;
      *in>>seq;
```

Using Objects

- You need to know the internal structure of the object. Find the appropriate base class definition in the include dir or use the source browser.
- Every variable has a *GetXxx()* and *SetXxx()* function.
- All SEQUECE_OF and SET_OF types in ASN.1 are translates to STL containers. You can determine the type of container datatool uses in the Definition file.
- `ContainerType ContainerName;`

```
for (ContainerType::IteratorType i =
ContainerName.begin();i != ContainerName.end(); ++i)
{      ObjectType ObjectName = *i;
   // ...
 }
```

Smart Pointers *auto_ptr<>* and *CRef*

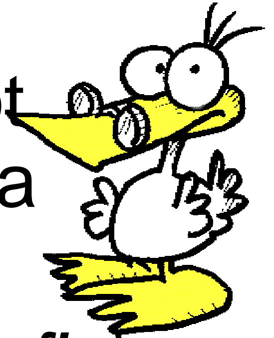
- auto_ptr<> are smart pointer that free the memory they are pointing to when the pointer comes out of scope. Great for memory management but…
- Two problems here:

```
int* ip = new int(5);
 {
   auto_ptr<int> a1 = ip;
   auto_ptr<int> a2 = ip;
 }
  *ip = 10/(*ip);
```

- Solution is a *CRef*

CRef

- A template class that provides a pointer interface to *CObject* classes.
- Walks and Quacks like a pointer but is not. However, you can use it exactly as if it is a pointer.



- It keeps count of how many pointers *(CRef's)* are pointing to a specific object. Only when the last pointer comes out of scope the object is freed.
- *CObject* is the actual class that holds the counter information. It acts as the base class for all objects that require a reference count.

The NCBI C++ Iterators

- Similar to the STL iterators, the NCBI C++ TK implemented type specific iterators that can be used to traverse ASN.1 object.
- This mechanism allows for easy access of specific data members nested within a given object. The iterator will extract all the data elements of a specific type that are nested several levels deep.
- Iteration can also be over a **set** of data members. For example, one can collect all the sequence id's and feature tables for a number of sequences in one iteration.

```
BIND-Pathway::={ pub {…},
        interactions{
            { iid, 118, pub { ..},
              a,  {egf,    pub { ..}},
              b, {egfr,    pub { ..}},
              descr{
                    { place
                          start membrane,
                          end cytoplasm
                    }
              }
            }
            { iid, 220 , pub { ..},
              a, {egf-egfr,  pub { ..}},
              b, {ATP,       pub { ..}},
              descr{
                    { place
                          start cytoplasm,
                          end  cytoplasm
                    }
              }
            }
            { iid, 238, pub { ..},
              a, {GRB2,   pub { ..}},
              b, {SOS1,   pub { ..}},
              descr{
                    { place
                          start cytoplasm,
                          end  nucleus
              }}}}
```

Traversing through objects

```
CBIND_pathway pathway;

CTypeIterator<CPub> publication;

for( publication=Begin(pathway);
        publication; ++publication)
{

    PrintAuthorList(*publication);

}
```

The DBAPI module

- The DBAPI module provides a unified access and query to databases by implementing a general database interface that is independent of the underlying relational database management system.
- The module is divided into two parts:
  - User classes that provide general database functionality, such as connection, query and results.
  - Driver classes that provide a uniform interface to the user classes so that the database driver can be changed to connect to a different database without affecting the code that makes use of the user classes.

DBAPI User Classes

- The interface to databases is provided by a number of classes (***CDataSource, CDbConnection, CStatement, CCallableStatement…***) that implement a general interface. The database drivers are responsible to support these functionalities for each specific RDBMS.
- To access a specific database the user registers the driver using the ***CDriveManager*** class.
- Once the driver is registered the application can access the database by creating a connection to the DB (using ***Connect()***) and executing queries (using ***Execute()***).
- ***GetResultSet()*** returns the results which can be traversed and checked for variant type.

DBAPI Driver Classes

- The driver classes are divided to RDBMS-dependent and RDBMS-independent classes.
- The most important RDBMS-dependent object is the "Driver Context" object that is responsible to create a connection client with the RDBMS. All driver contexts implement the same interface defined in the ***I_DriverContext*** class.
- To provide RDBMS independence, the connection information is wrapped in an RDBMS independent object ***CDB_Connection***. The SQL query commands and the results functions are also wrapped in an RDBMS independent object.

Supported DBAPI Drivers

- Sybase CTLIB
- Sybase DBLIB
- Microsoft DBLIB
- FreeTDS 0.60
- ODBC
- MySQL

The Object Manager

- The ObjectManager facilitates access and manipulation of sequence data. It provides a transparent interface to download sequences from GenBank and manipulate their sequence and annotation.
- A very useful tool for programs that need to access and manipulate biological sequences.
- Takes advantage of the NCBI extensive ASN.1 definitions (Seq_entry and Seq_id) to access any part of a sequence definition without retrieving the entire object.

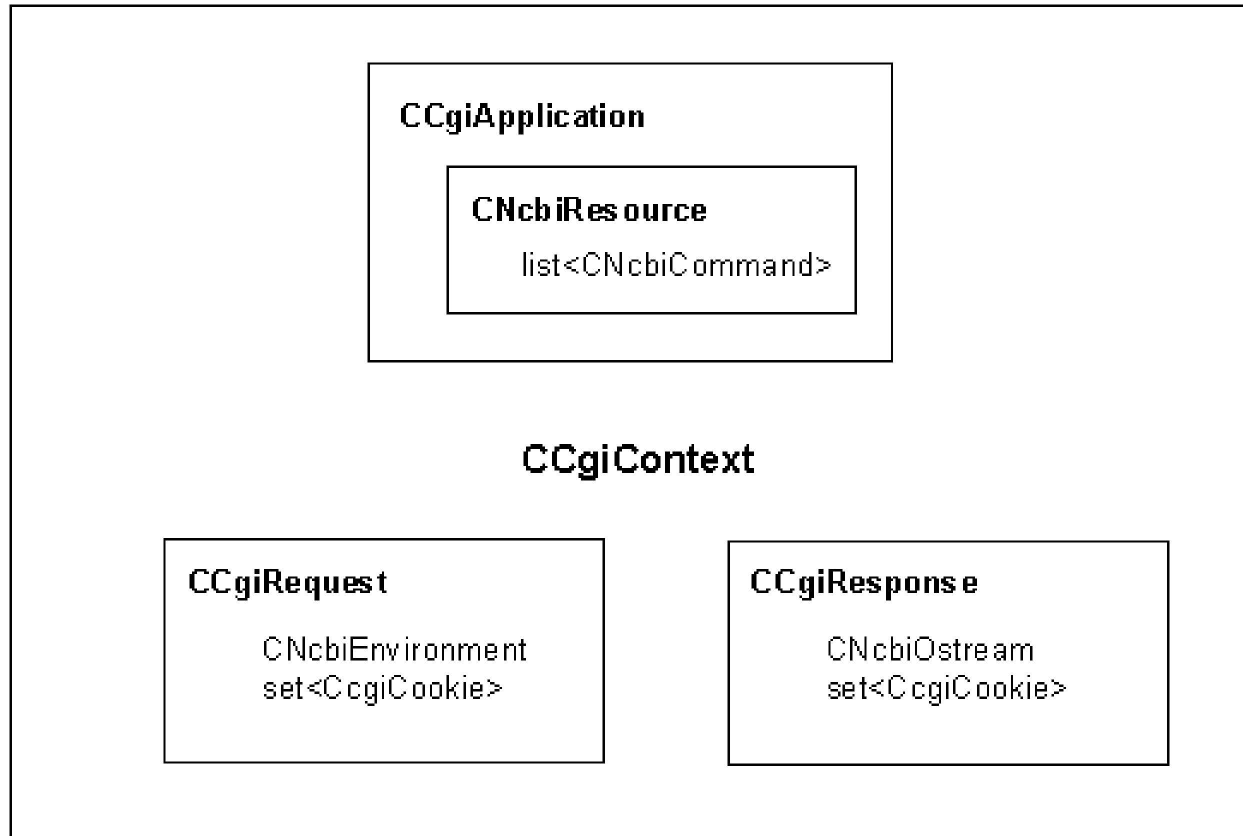
How To Use the ObjectManager

```
//Create an ObjectManager object pointer
    CRef<CObjectManager> obj_mgr = new CObjectManager;
//Create a Scope object.
    CScope scope2(*obj_mgr);
//Add a data loader to the scope object
    obj_mgr->RegisterDataLoader(*(new GBDataLoader("GENBANK")),
         CObjectManager::eDefault);
    scope.AddDataLoader("GENBANK");  or scope.AddDefaults();
//Retrieve a Seq_entry from GenBank
    CSeq_id seqid;
    seqid.SetGi(3);
    CBioseq_Handle handle = scope.GetBioseqHandle(seqid);
//Access sequence data
    CSeqVector seq_vec = handle.GetSeqVector();
```

More on Object Manager

- One can easily create and modify a biological sequences object (using Seq_Entry) including sequence information, coding regions and annotation.
- There are specific iterators to traverse certain parts of the sequence object. Such as annotation and description iterators.
- There are utility functions to write GenBank flatfiles, compute sequence weights, and convert sequence encoding.

CGI Classes



CCgiContext

CCgiRequest

CCgiResponse

CCgiContext

CCgiRequest

CCgiResponse

CCgiApplication

CCgiContext

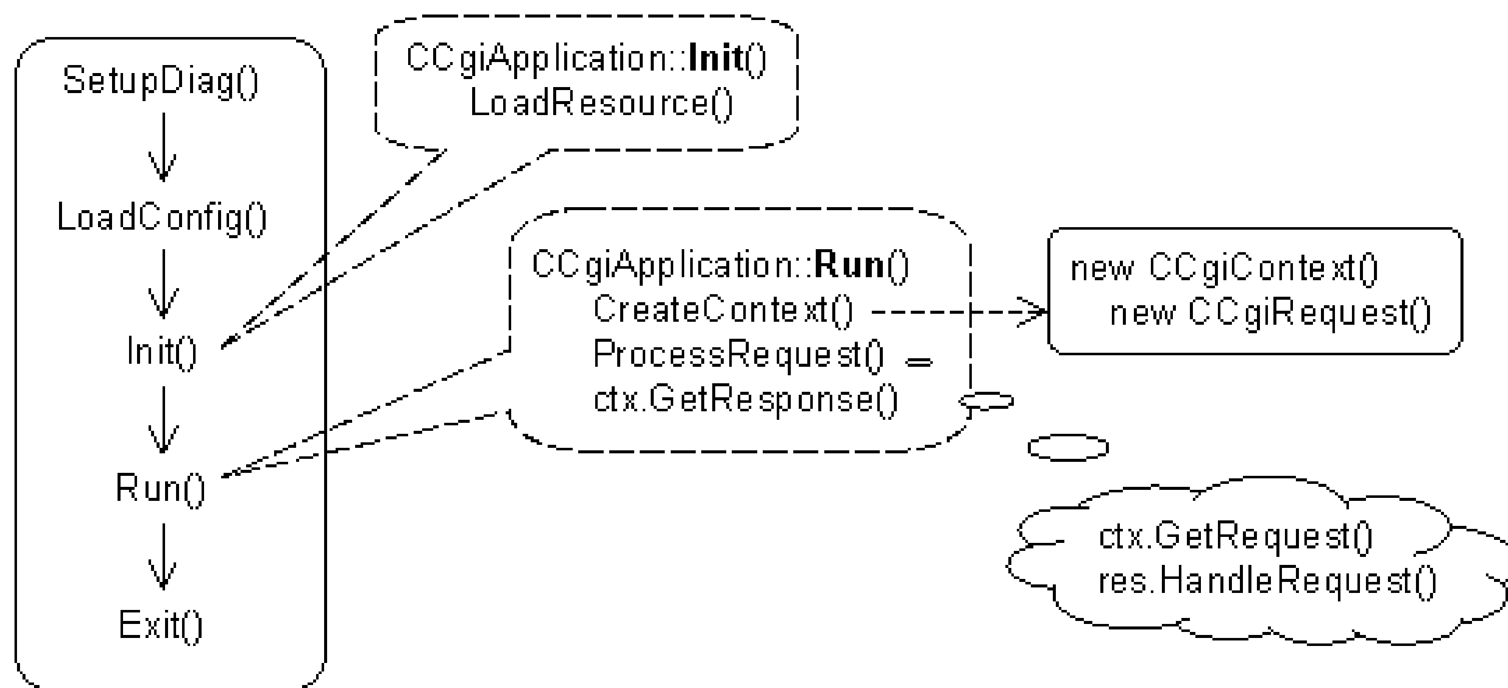
CCgiRequest

CCgiResponse

CCgiContext

CCgiRequest

CCgiResponse


CNcbiApplication.AppMain():
SetupDiag()
LoadConfig()
Init()
Run()
Exit()
CCgiApplication::Init()
LoadResource()
CCgiApplication::Run()
CreateContext()
ProcessRequest()
ctx.GetResponse()
new CCgiContext()
new CCgiRequest()
ctx.GetRequest()
res.HandleRequest()

The *CCgiApplication* class

- Derived from *CNcbiApplication*. Implements its own *Run()* function.
- The *Run()* function
  - creates a context,
  - processes the request and
  - calls the context to send the response.
- A local variable in **each** *Run* invocation (remember FAST-CGI) holds a pointer to its current context object.

CNcbiResource and *CNcbiCommand*

- The Resource class is the one that handles the request. It is created by registry object which defines data paths, resources and env. variables.
- The resource also contains a list of commands. *HandleRequest(ctx)* looks at the command list and executes the first match.
- The *CNcbiResource* can be used to implement an application that has a complex variable functionality such as, database calls, and Entrez-type systems. Generally most cgi applications implement only one function.

CCgiRequest

- Request class acts as an interface b/w the user's query and the CGI program. All constructors to this object invoke the *x_Init().*
- Caches environment, server, client and request information.
- Knows about POST and GET and will process the request accordingly.

CCgiResponse

- Provides an interface to the program output stream.
- Generates the appropriate HTML header (MIME-type).
- All cookies that are to be sent to the client are included in the header output.

HTML classes

- HTML pages can be generated independently from cgi application. The only connection is through a constructor to an HTML page which takes a *CCgiApplication* .
- Remove the annoyance of fprint's.
- Excellent support for template HTML pages. A good utility for a big project or a company.
- Supports almost all HTML tags.

CNCBINode

- The base class for all HTML class. Has three member types:
  1. *m_Name* - a *string*, used to identify the type of node or to store text data
  2. *m_Attributes* - a map<string, string> of properties for this node
  3. *m_Children* - a list of subnodes embedded (at run-time) in this node
- For HTML text nodes *m_Name* holds the actual text, for all other HTML elements, *m_Name* holds the tag name.
- Attributes can be set using *SetXxx().* Each one returns a *this* pointer. For example:

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

More about *CNCBINode*

- *M_children* are the nested children that are encapsulated by this particular tag. E.g. Table contains columns and cells.
- The *Print()* statement will recursively print all the nested tags.

NCBI Page class

- *CHMTLpage* is a container class to all the HTML tags in that we want to print.
- Two important components:
  - *m_CgiApplication* – a pointer to the *CCgiApplication*
  - *m_TagMap*- used to map strings to subcomponents of the page. E.g.

    ```
    page->AddTagMap("TABLE", new
    CHTMLTable(..));
    ```
- *<@tagname@>* are embedded within a template html file. *CHTMLpage* can load the template file and substitute the tag with real HTML tags.
- The *CHTMLpage* class in combination with template files provided a powerful way of generating application-specific web pages containing common elements.

CHTMLNode

- Classes for all the standard HTML tags (e.g. , <bt>, <a>, <img>, etc.). Up to 40 different tags are defined.
- A group of *Set/Get* attributes functions are defined for each tag (e.g. style, id, width, height, color, alignment).
- All class interface includes a number of constructors that take a tag name as a first argument and an optional second argument which could be either text or *CNCBINode.*