The NCBI C++ Toolkit

15: Biological Object Manager

Last Update: January 30, 2012.

Overview

Introduction

The Object Manager[Library xobjmgr: include | src]

The Object Manager is a library, working in conjunction with the serializable object classes (see above) to facilitate access to biological sequence data. The Object Manager has been designed to coordinate the use of these objects, particularly the management of the details of loading data from one or more potentially heterogeneous data sources. The goal is to present a consistent, flexible interface to users that minimizes their exposure to the details of interacting with biological databases and their underlying data structures.

Most of the major classes in this library have a short definition in addition to the descriptions and links below. Handles are the primary mechanism through which users access data; details of the retrieval are managed transparently by the Object Manager.

See the usage page to begin working with the Object Manager. An example and sample project have been created to further assist new users and serve as a template for new projects. We have also compiled a list of common problems encountered when using the Object Manager.

Object Manager [include/objmgr | src/objmgr]

Chapter Outline

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

Preface

Molecular biology is generating a vast multitude of data referring to our understanding of the processes which underlie all living things. This data is being accumulated and analyzed in thousands of laboratories all over the world. Its raw volume is growing at an astonishing rate.

In these circumstances the problem of storing, searching, retrieving and exchanging molecular biology data cannot be underestimated. NCBI maintains several databases for storing biomedical information. While the amount of information stored in these databases grows at an exponential rate, it becomes more and more important to optimize and improve the data retrieval software tools. Object Manager is a tool specifically designed to facilitate data retrieval.

The NCBI databases and software tools are designed around a particular model of biological sequence data. The nature of this data is not yet fully understood, and its fundamental properties and relationships are constantly being revised. So, the data model must be very flexible. NCBI uses Abstract Syntax Notation One (ASN.1) as a formal language to describe biological sequence data and its associated information.

Requirements

Clients must be able to analyze biological sequence data, which come from multiple heterogeneous data sources. As for 'standard' databases, we mean only NCBI GenBank. 'Nonstandard' data sources may include but are not limited to reading data from files or constructing bio sequences 'manually'.

A biologist's goal could be to investigate different combinations of data. The system should provide for transparent merging of different pieces of data, as well as various combinations of it. It is Important to note that such combinations may be incorrect or ambiguous. It is one of the possible goals of the client to discover such ambiguities.

The bio sequence data may be huge. Querying this vast amount of data from a remote database may impose severe requirements on communication lines and computer resources - both client and server. The system should provide for partial data acquisition. In other words, the system should only transmit data that is really needed, not all of it at once. At the same time this technology should not impose additional (or too much) restrictions on a client system. The process, from a client point of view, should be as transparent as possible. When and if the client needs more information, it should be retrieved 'automatically'.

Different biological sequences can refer to each other. One example of such a reference may be in the form 'the sequence of amino acids here is the same as the sequence of amino acids there' (the meaning of here and there is a separate question). The data retrieval system should be able to resolve such references automatically answering what amino acids (or nucleic acids) are actually here. At the same time, at the client's request, such automatic resolution may be turned off. Probably, the client's purpose is to investigate such references.

Biological sequences are identified by Seq-id, which may have different forms. Information about specific sequence stored in the database can be modified at any time. Sometimes, if changes are minor, this only results in creating a new submission of an existing bio sequence and assigning a new *revision* number to it. In the case of more substantial changes, a new *version* number can be assigned. From the client's point of view, the system should remain consistent when data change. Possible scenarios include:

- Database changes during client's session. Client starts working and retrieves some data from the database, the data in database then change. When client then asks for an additional data, the system should retrieve original bio sequence submission data, not the most recent one.
- Database changes between client's sessions. Client retrieves some data and ends work session. Next time the most recent submission data is retrieved, unless the client asks for a specific version number.

The system must support multithreading. It should be possible to work with bio sequence data from multiple threads.

Use cases

Biological sequence data and its associated information are specified in the NCBI data model using Abstract Syntax Notation One (ASN.1). There is a tool which, based on these specifications, generates corresponding data objects. The Object Manager manipulates these objects, so they are referenced in this document without further explanation.

The most general container object of bio sequence data, as defined in the NCBI data model, is Seq-entry. In general, Seq-entry is defined recursively as a tree of Seq-entry's (one entry refers to another one etc), where each node contains either a Bioseq or a list of other Seq-entry's plus some additional data like sequence description, sequence annotations etc. Naturally, in any such tree there is only one top-level Seq-entry (TSE).

The client must be able to define a scope of visibility and reference resolution. Such a scope is defined by the sources of data - the system uses only 'allowed' sources to look for data. Such scopes may, for instance, contain several variants of the same bio sequence (Seq-entry). Since sequences refer to each other, the scopes practically always intersect. In this case changing some data in one scope should be somehow reflected in all other scopes, which 'look' at the same data - there is a need for some sort of communication between scopes.

A scope may contain multiple top-level Seq-entry's and multiple sources of data.

Once a scope is created, a client should be able to:

- Add an externally created top-level Seq-entry to it.
- Add a data loader to it. A data loader is a link between an out-of-process source of bio sequence data and the scope; it loads data when and if necessary.
- Edit objects retrieved from the scope. Data fetched from external sources through loaders can not be modified directly. Instead, an object may be detached from its original source and the new copy provided for editing. Editing includes:
  - moving existing data from one object to another;
  - adding new data to an object; and
  - removing data from an object.

Once the scope is populated with data, a client should be able to:

- Find a Bioseq with a given Seq_id, loading the Seq-entry if necessary.
- Find a top-level Seq-entry for a sequence with a given Seq_id.

- Retrieve general information about the sequence (type, length etc., without fetching sequence data) by Seq_id.
- Obtain sequence data - actual sequence data (by Seq_id) in a specified encoding.
- Enumerate sequence descriptions and sequence annotation data, namely: features, graphs, and alignments. The annotation iterators may be fine-tuned to restrict annotation types, locations, depth of search, etc.

Multithreading. There are two scenarios:

- Several threads work with the same scope simultaneously. The scope is given to them from the outside, so this external controller is responsible for waiting for thread termination and deleting the scope.
- Different threads create their own scopes to work with the same data source. That is, the data source is a shared resource.

Classes

Definition

Here we define Object Manager's key classes and their behavior:

- Object manager
- Scope
- Data loader
- Data source
- Handles
- Seq-map
- Seq-vector
- Iterators
- CFeatTree

Object manager

Object manager manages data objects, provides them to Scopes when needed. It knows all existing Data sources and Data loaders. When a Scope needs one, it receives a data object from the Object Manager. This enables sharing and reusing of all relevant data between different Scopes. Another function of the Object Manager is letting Scopes know each other, letting Scopes to communicate. This is a barely visible entity.

Scope

Scope is a top-level object available to a client. Its purpose is to define a scope of visibility and reference resolution and provide access to the bio sequence data.

Data loader

Data loader is a link between in-process data storage and remote, out-of process data source. Its purpose is to communicate with a remote data source, receive data from there, and understand what is already received and what is missing, and pass data to the local storage (Data source). Data loader maintains its own index of what data is loaded already and references that data in the Data source.

Data source

Data source stores bio sequence data locally. Scope communicates with this object to obtain any sequence data. Data source creates and maintains internal indices to facilitate information search. Data source may contain data of several top-level Seq-entry's. In case client pushes an externally constructed Seq-entry object in the Scope, such object is stored in a separate Data source. In this case, Data source has only one top-level Seq-entry. From the other side, when Data source is linked to a Data loader, it will contain all top-level Seq-entry's retrieved by that loader.

Handles

Most objects received from the Object Manager are accessed through handles. One of the most important of them is Bioseq handle, a proxy for CBioseq. Its purpose is to facilitate access to Bioseq data. When client wants to access particular biological sequence, it requests a Bioseq handle from the Scope. Another important class is Seq-id handle which is used in many places to optimize data indexing. Other handles used in the Object Manager are:

- Bioseq-set handle
- Seq-entry handle
- Seq-annot handle
- Seq-feat handle
- Seq-align handle
- Seq-graph handle

Most handles have two versions: simple read-only handle and edit handle, which may be used to modify the data.

Seq-map

Seq-map contains general information about the sequence structure: location of data, references gaps etc.

Seq-vector

Seq-vector provides sequence data in the selected coding.

Iterators

Many objects in the Object Manager can be enumerated using iterators. Some of the iterators behave like usual container iterators (e.g. Seq-vector iterator), others have more complicated behavior depending on different arguments and flags.

Description iterators traverse bio sequence descriptions (Seq-descr and Seqdesc) in the Seq-entry. They start with the description(s) of the requested Bioseq or Seq-entry and then retrieve all descriptions iterating through the tree nodes up to the top-level Seq-entry. Starting Bioseq is defined by a Bioseq handle. Descriptions do not contain information about what Bioseq they describe, so the only way to figure it out is by description location on the tree.

Annotation iterators are utility classes for traversing sequence annotation data. Each annotation contains a reference to one or more regions on one or more sequences (Bioseq). From one point of view this is good, because we can always say which sequences are related to the given annotation. On the other hand, this creates many problems, since an annotation referencing a sequence may be stored in another sequence/Seq-entry/tree. The annotation iterators attempt to find all objects related to the given location in all Data sources from the current Scope. Data sources create indexes for all annotations by their locations. Another useful feature of the annotation iterators is location mapping: for segmented sequences the iterators can collect annotations defined on segments and adjust their locations to point to the master sequence.

There are several annotation iterator classes; some specialized for particular annotation types:

- Seq-annot iterator - traverses Seq-annot objects starting from a given Seq-entry/Bioseq up to the top-level Seq-entry (The same way as Descriptor iterators do) or down to each leaf Seq-entry. (Seq-annot);
- Annot iterator -traverses Seq-annot objects (Seq-annot) rather than individual annotations;
- Feature iterator - traverses sequence features (Seq-feat);
- Alignment iterator - traverses sequence alignments descriptions (Seq-align).
- Graph iterator - traverses sequence graphs (Seq-graph);

Tree iterators include Bioseq iterator and Seq-entry iterator, which may be used to visit leafs and nodes of a Seq-entry tree.

Seq-map iterator iterates over parts of a Bioseq. It is used mostly with segmented sequences to enumerate their segments and check their type without fetching complete sequence data.

Seq-vector iterator is used to access individual sequence characters in a selected coding.

CFeatTree

The CFeatTree class builds a parent-child feature tree in a more efficient way than repeatedly calling GetParentFeature() for each feature. The algorithm of a parent search is the same as the one used by GetParentFeature().

The class CFeatTree works with a set of features specified by calling AddFeature() or AddFeatures(). The actual tree is built the first time method GetParent() or GetChildren() is called after adding new features. Features can be added later, but the parent information is cached and will not change if parents were found already. However, features with no parent will be processed again in attempt to find parents from the newly added features.

Here's a sample code snippet that constructs a CFeatTree based on selected features:

```
// Construct the Seq-loc to get features for.
CSeq_loc seq_loc;
seq_loc.SetWhole().SetGi(src.gi);

// Make a selector to limit features to those of interest.
SAnnotSelector sel;
sel.SetResolveAll();
sel.SetAdaptiveDepth(true);
sel.IncludeFeatType(CSeqFeatData::e_Gene)
 .IncludeFeatType(CSeqFeatData::e_Cdregion)
 .IncludeFeatType(CSeqFeatData::e_Rna);

// Exclude SNP's and STS's since they won't add anything interesting
// but could significantly degrade performance.
sel.ExcludeNamedAnnots("SNP");
sel.ExcludeNamedAnnots("STS");
```

```
// Use a CFeat_CI iterator to iterate through all selected features.
CFeat_CI feat_it(CFeat_CI(*gscope, seq_loc, sel));

// Create the feature tree and add to it the features found by the
// feature iterator.
feature::CFeatTree feat_tree;
feat_tree.AddFeatures(feat_it);
```

The CFeatTree class can also improve the performance of the feature::GetBestXxxForYyy() functions, such as GetBestGeneForMrna(). Simply create the CFeatTree and pass it to the GetBestXxxForYyy() functions.

Note: There are "old" and "new" GetBestXxxForYyy() functions. The "new" functions are in the feature namespace, are located in include/objmgr/util/feature.hpp, and should be used for new development, as they are more efficient. The "old" functions are in the sequence namespace and are located in include/objmgr/util/sequence.hpp.

Attributes and Operations

- Object manager
- Scope
- Data loader
- Handles:
  - Bioseq handle
  - Bioseq-set handle
  - Seq-entry handle
  - Seq-annot handle
  - Seq-feat handle
  - Seq-align handle
  - Seq-graph handle
- Seq-map
- Seq-vector
- Iterators:
  - Bioseq iterator
  - Seq-entry iterator
  - Seq-descr iterator
  - Seqdesc iterator
  - Seq-annot iterator
  - Annot iterator
  - Feature iterator
  - Alignment iterator
  - Graph iterator
  - Seq-map iterator
  - Seq-vector iterator

Object manager

Before being able to use any Scopes, a client must create and initialize the Object Manager (CObjectManager). Initialization functions include registration of Data loaders, some of which may be declared as default ones. All default Data loaders are added to a Scope when the latter asks for them. All Data loaders are named, so Scopes may refer to them by name. Another kind of data object is CSeq_entry - it does not require any data loader, but also may be registered with the Object Manager. Seq-entry may not be a default data object.

CObjectManager is a singleton, which means at any moment you may have only one instance of the class using static method CObjectManager::GetInstance(void). The method returns CRef<CObjectManager> and this CRef<> should not be released until you finish using the Object Manager. Otherwise the Object Manager may be deleted and the next call to GetInstance will return a new object.

Most other CObjectManager methods are used to manage Data loaders.

CObjectManager important methods

- GetInstance - returns the object manager singleton (creating it if necessary). The returned CRef should be kept alive while the object manager is used.
- RegisterDataLoader - creates and registers data loader specified by driver name using plugin manager.
- FindDataLoader - finds data loader by its name. Returns pointer to the loader or null if no loader was found.
- GetRegisteredNames - fills vector of strings with the names of all registered data loaders.
- void SetLoaderOptions - allows to modify options (default flag and priority) of a registered data loader.
- bool RevokeDataLoader - revokes a registered data loader by pointer or name. Returns false if the loader is still in use. Throws exception if the loader is not registered.

See the CObjectManager API reference for an up-to-date list of all methods.

Scope

The Scope class (CScope) is designed to be a lightweight object, which could be easily created and destroyed. Scope may even be created on the stack - as an automatic object. Scope is populated with data by adding data loaders or already created Seq-entry's to it. Data loaders can only be added by name, which means it must be registered with the Object Manager beforehand. Once an externally created Seq-entry is added to a Scope, it should not be modified any more.

The main task of a scope is to cache resolved data references. Any resolved data chunk will be locked by the scope through which it was fetched. For this reason retrieving a lot of unrelated data through the same scope may consume a lot of memory. To clean a scope's cache and release the memory you can use ResetHistory or just destroy the scope and create a new one. Note: When a scope is destroyed or cleaned any handles retrieved from the scope become invalid.

CScope important methods

- AddDefaults - adds all loaders registered as default in the object manager.

- AddDataLoader - adds a data loader to the scope using the loader's name.
- AddScope - adds all loaders attached to another scope.
- AddTopLevelSeqEntry - adds a TSE to the scope. If the TSE has been already added to some scope, the data and indices will be re-used.
- AddBioseq - adds a Bioseq object wrapping it to a new Seq-entry.
- AddSeq_annot - adds a Seq-annot object to the scope.
- GetBioseqHandle - returns a Bioseq handle for the requested Bioseq. There are several versions of this function accepting different arguments. A bioseqs can be found by its Seq-id, Seq-id handle or Seq-loc. There are special flags which control data loading while resolving a Bioseq (e.g. you may want to check if a Bioseq has been already loaded by any scope or resolved in this particular scope).
- GetBioseqHandleFromTSE - allows getting a Bioseq handle restricting the search to a single top-level Seq-entry.
- GetSynonyms - returns a set of synonyms for a given Bioseq. Synonyms returned by a scope may differ from the Seq-id set stored in Bioseq object. The returned set includes all ids which are resolved to the Bioseq in this scope. An id may be hidden if it has been resolved to another Bioseq. Several modifications of the same id may appear as synonyms (e.g. accession.version and accession-only may be synonyms).
- GetAllTSEs - fills a vector of Seq-entry handles with all resolved TSEs.
- GetIds - fetches complete list of IDs for a given Seq-id without fetching the Bioseq (if supported by loader).

See the CScope API reference for an up-to-date list of all methods.

All data sources (data loaders and explicitly added data) have priorities. For example, if you call AddScope() and specify a non-default priority, the scope scans data sources in order of increasing priority to find the sequence you've requested. By default, explicitly added data have priority 9 and data loaders have priority 99, so the scope will first look in explicit data, then in data loaders. If you have conflicting data or loaders (e.g. GenBank and BLAST), you may need different priorities to make the scope first look, for example, in BLAST, and then in GenBank if the sequence is not found.

Note: the priority you've specified for a data loader at registration time (RegisterInObjectManager()) is a new default for it, and can be overridden when you add the data loader to a scope.

Data loader

The Data loader base class (CDataLoader) is almost never used by a client application directly. The specific data loaders (like GenBank data loader) have several static methods which should be used to register loaders in the Object Manager. Each of RegisterInObjectManager methods constructs a loader name depending on the arguments, checks if a loader with this name is already registered, creates and registers the loader if necessary. GetLoaderNameFromArgs methods may be used to get a potential loader's name from a set of arguments. RegisterInObjectManager returns a simple structure with two methods: IsCreated, indicating if the loader was just created or a registered loader with the same name was found, and GetLoader, returning pointer to the loader. The pointer may be null if the RegisterInObjectManager function fails or if the type of the already registered loader can not be casted to the type requested.

Bioseq handle

When a client wants to access a Bioseq data, it asks the Scope for a Bioseq handle (CBioseq_Handle). The Bioseq handle is a proxy to access the Bioseq data; it may be used to iterate over annotations and descriptors related to the Bioseq etc. Bioseq handle also takes care of loading any necessary data when requested. E.g. to get a sequence of characters for a segmented Bioseq it will load all segments and put their data in the right places.

Most methods of CBioseq for checking and getting object members are mirrored in the Bioseq handle's interface. Other methods are described below.

CBioseq_Handle important methods

- GetSeqId - returns Seq-id which was used to obtain the handle or null (if the handle was obtained in a way not requiring Seq-id).
- GetSeq_id_Handle - returns Seq-id handle corresponding to the id used to obtain the handle.
- IsSynonym - returns true if the id resolves to the same handle.
- GetSynonyms - returns a list of all Bioseq synonyms.
- GetParentEntry - returns a handle for the parent Seq-entry of the Bioseq.
- GetTopLevelEntry - returns a handle for the top-level Seq-entry.
- GetBioseqCore - returns TBioseqCore, which is CConstRef<CBioseq>. The Bioseq object is guaranteed to have basic information loaded (the list of Seq-ids, Bioseq length, type etc.). Some information in the Bioseq (descriptors, annotations, sequence data) may be not loaded yet.
- GetCompleteBioseq - returns the complete Bioseq object. Any missing data will be loaded and put in the Bioseq members.
- GetComplexityLevel and GetExactComplexityLevel - allow finding a parent Seq-entry of a specified class (e.g. nuc-prot). The first method is more flexible since it considers some Seq-entry classes as equivalent.
- GetBioseqMolType - returns molecule type of the Bioseq.
- GetSeqMap - returns Seq-map object for the Bioseq.
- GetSeqVector - returns Seq-vector with the selected coding and strand.
- GetSequenceView - creates a Seq-vector for a part of the Bioseq. Depending on the flags the resulting Seq-vector may show all intervals (merged or not) on the Bioseq specified by Seq-loc, or all parts of the Bioseq not included in the Seq-loc.
- GetSeqMapByLocation - returns Seq-map constructed from a Seq-loc. The method uses the same flags as GetSequenceView.
- MapLocation - maps a Seq-loc from the Bioseq's segment to the Bioseq.

See the CBioseq_Handle API reference for an up-to-date list of all methods.

Bioseq-set handle

The Bioseq-set handle class (CBioseq_set_Handle) is a proxy class for Bioseq-set objects. Like in Bioseq handle, most of its methods allow read-only access to the members of CBioseq_set object. Some other methods are similar to the Bioseq handle's interface.

CBioseq_set_Handle important methods

- GetParentEntry - returns a handle for the parent Seq-entry of the Bioseq.
- GetTopLevelEntry - returns a handle for the top-level Seq-entry.
- GetBioseq_setCore - returns core data for the Bioseq-set. The object is guaranteed to have basic information loaded. Some information may be not loaded yet.
- GetCompleteBioseq_set - returns the complete Bioseq-set object. Any missing data will be loaded and put in the Bioseq members.
- GetComplexityLevel and GetExactComplexityLevel - allow finding a parent Seq-entry of a specified class (e.g. nuc-prot). The first method is more flexible since it considers some Seq-entry classes as equivalent.

See the CBioseq_set_Handle API reference for an up-to-date list of all methods.

Seq-entry handle

The Seq-entry handle class (CSeq_entry_Handle) is a proxy class for Seq-entry objects. Most of its methods allow read-only access to the members of Seq-entry object. Other methods may be used to navigate the Seq-entry tree.

CSeq_entry_Handle important methods

- GetParentBioseq_set - returns a handle for the parent Bioseq-set if any.
- GetParentEntry - returns a handle for the parent Seq-entry.
- GetSingleSubEntry - checks that the Seq-entry contains a Bioseq-set of just one child Seq-entry and returns a handle for this entry, otherwise throws exception.
- GetTopLevelEntry - returns a handle for the top-level Seq-entry.
- GetSeq_entryCore - returns core data for the Seq-entry. Some information may be not loaded yet.
- GetCompleteSeq_entry - returns the complete Seq-entry object. Any missing data will be loaded and put in the Bioseq members.

See the CSeq_entry_Handle API reference for an up-to-date list of all methods.

Seq-annot handle

The Seq-annot handle class (CSeq_annot_Handle) is a simple proxy for Seq-annot objects.

CSeq_annot_Handle important methods

- GetParentEntry - returns a handle for the parent Seq-entry.
- GetTopLevelEntry - returns a handle for the top-level Seq-entry.
- GetCompleteSeq_annot - returns the complete Seq-annot object. Any data stubs are resolved and loaded.

See the CSeq_annot_Handle API reference for an up-to-date list of all methods.

Seq-feat handle

The Seq-feat handle class (CSeq_feat_Handle) is a read-only proxy to Seq-feat objects data. It also simplifies and optimizes access to methods of SNP features.

Seq-align handle

The Seq-align handle class (CSeq_align_Handle) is a read-only proxy to Seq-align objects data. Most of its methods are simply mapped to the CSeq_align methods.

Seq-graph handle

The Seq-graph handle class (CSeq_graph_Handle) is a read-only proxy to Seq-graph objects data. Most of its methods are simply mapped to the CSeq_graph methods.

Seq-map

The Seq-map class (CSeqMap) object gives a general description of a biological sequence: the location and type of each segment, without the actual sequence data. It provides the overall structure of a Bioseq, or can be constructed from a Seq-loc, representing a set of locations rather than a real Bioseq. Seq-map is typically used with Seq-map iterator, which enumerates individual segments. Special flags allow selecting the types of segments to be iterated and the maximum depth of resolved references.

CSeqMap important methods

- GetSegmentsCount - returns the number of segments in the Seq-map.
- GetLength - returns the length of the whole Seq-map.
- GetMol - returns the molecule type for real bioseqs.
- begin, Begin, end, End, FindSegment - methods for normal Seq-map iteration (lower case names added for compatibility with STL).
- BeginResolved, FindResolved, EndResolved - force resolving references in the Seq-map. Optional arguments allow controlling types of segments to be shown and resolution depth.
- ResolvedRangeIterator - starts iterator over the specified range and strand only.
- CanResolveRange - checks if necessary data is available to resolve all segments in the specified range.

See the CSeqMap API reference for an up-to-date list of all methods.

Seq-vector

The Seq-vector class (CSeqVector) is a convenient representation of sequence data. It uses interface similar to the STL vector but data retrieval is optimized for better performance on big sequences. Individual characters may be accessed through operator[], but better performance may be achieved with Seq-vector iterator. Seq-vector can be obtained from a Bioseq handle, or constructed from a Seq-map or Seq-loc.

CSeqVector important methods

- size - returns length of the whole Seq-vector.
- begin, end - STL-style methods for iterating over Seq-vector.
- operator[] - provides access to individual character at a given position.
- GetSeqData - copy characters from a specified range to a string.
- GetSequenceType, IsProtein, IsNucleotide - check sequence type.

- SetCoding, SetIupacCoding, SetNcbiCoding - control coding used by Seq-vector. These methods allow selecting Iupac or Ncbi coding without checking the exact sequence type - correct coding will be selected by the Seq-vector automatically.
- GetGapChar - returns character used in the current coding to indicate gaps in the sequence.
- CanGetRange - check if sequence data for the specified range is available.
- SetRandomizeAmbiguities, SetNoAmbiguities - control randomization of ambiguities in ncbi2na coding. If set, ambiguities will be represented with random characters with distribution corresponding to the ambiguity symbol at each position. Once assigned, the same character will be returned every time for the same position.

See the CSeqVector API reference for an up-to-date list of all methods.

Bioseq iterator

The Bioseq iterator class (CBioseq_CI) enumerates bioseqs in a given Seq-entry. Optional filters may be used to restrict types of bioseqs to iterate.

Seq-entry iterator

The Seq-entry iterator (CSeq_entry_CI) enumerates Seq-entry's in a given parent Seq-entry or a Bioseq-set. Note that the iterator enumerates sub-entries for only one tree level. It **does not** go down the tree if it finds a sub-entry of type 'set'.

Seq-descr iterator

The Seq-descr iterator (CSeq_descr_CI) enumerates CSeq_descr objects from a Bioseq or Seq-entry handle. The iterator starts from the specified point in the tree and goes up to the top-level Seq-entry. This provides sets of descriptors more closely related to the Bioseq/Seq-entry requested to be returned first, followed by descriptors that are more generic. To enumerate individual descriptors CSeqdesc_CI iterator should be used.

Seqdesc iterator

Another type of descriptor iterator is CSeqdesc_CI. It enumerates individual descriptors (CSeqdesc) rather than sets of them. Optional flags allow selecting type of descriptors to be included and depth of the search. The iteration starts from the requested Seq-entry or Bioseq and proceeds to the top-level Seq-entry or stops after going selected number of Seq-entry's up the tree.

Seq-annot iterator

The Seq-annot iterator (CSeq_annot_CI) may be used to enumerate CSeq_annot objects - packs of annotations (features, graphs, alignments etc.). The iterator can work in two directions: starting from a Bioseq and going up to the top-level Seq-entry, or going down the tree from the selected Seq-entry.

Annot iterator

Although returning CSeq_annot objects, CAnnot_CI searches individual features, alignments and graphs related to the specified Bioseq or Seq-loc. It enumerates all Seq-annot's containing the requested annotations. The search parameters may be fine-tuned using SAnnotSelector for feature, alignment, or graph iterators.

SAnnotSelector

SAnnotSelector is a helper class which may be used to fine-tune annotation iterator's settings. It is used with CAnnot_CI, CFeat_CI, CAlign_CI and CGraph_CI iterators. Below is the brief explanation of the class methods. Some methods have several modifications to simplify the selector usage. E.g. one can find SetOverlapIntervals() more convenient than SetOverlapType (SAnnotSelector::eOverlap_Intervals).

- SetAnnotType - selects type of annotations to search for (features, alignments or graphs). Type-specific iterators set this type automatically.
- SetFeatType - selects type of features to search for. Ignored when used with alignment or graph iterator.
- SetFeatSubtype - selects feature subtype and corresponding type.
- SetByProduct - sets flag to search features by product rather than by location.
- SetOverlapType - select type of location matching during the search. If overlap type is set to intervals, the annotation should have at least one interval intersecting with the requested ranges to be included in the results. If overlap type is set to total range, the annotation will be found even if its location has a gap intersecting with the requested range. The default value is intervals. Total ranges are calculated for each referenced Bioseq individually, even if an annotation is located on several bioseqs, which are segments of the same parent sequence.
- SetSortOrder - selects sorting of annotations: normal, reverse or none. The default value is normal.
- SetResolveMethod - defines method of resolving references in segmented bioseqs. Default value is TSE, meaning that annotations should only be searched on segments located in the same top-level Seq-entry. Other available options are none (to ignore annotations on segments) and all (to search on all segments regardless of their location). Resolving all references may produce a huge number of annotations for big bioseqs, this option should be used with care.
- SetResolveDepth - limits the depth of resolving references in segmented bioseqs. By default the search depth is not limited (set to kMax_Int).
- SetAdaptiveDepth, SetAdaptiveTrigger - set search depth limit using a trigger type/subtype. The search stops when an annotation of the trigger type is found on some level.
- SetMaxSize - limits total number of annotations to find.
- SetLimitNone, SetLimitTSE, SetLimitSeqEntry, SetLimitSeqAnnot - limits the search to a single TSE, Seq-entry or Seq-annot object.
- SetUnresolvedFlag, SetIgnoreUnresolved, SetSearchUnresolved, SetFailUnresolved - define how the iterators should behave if a reference in a sequence can not be resolved. Ignore (default) will ignore missing parts, Fail will throw CAnnotException. Search may be used to search by known ID on missing parts, but will work only if limit object is also set, since the iterator needs to know where to look for the annotations.
- SetSearchExternal - sets all flags to search for external annotations. Such annotations are packed with special bioseqs, (e.g. gnl|Annot:CDD|6 references gi 6 and contains CDD features for the gi). If SetSearchSpecial is called with the Bioseq handle for this special sequence or its TSE handle, only external CDD features from this TSE will be found. The method calls SetResolveTSE, sets limit object to the same TSE and sets SearchUnresolved flag.

- SetNoMapping - prevents the iterator from mapping locations to the top-level Bioseq. This option can dramatically increase iterators' performance when searching annotations on a segmented Bioseq.

Feature iterator

The Feature iterator (CFeat_CI) is a kind of annotation iterator. It enumerates CSeq_feat objects related to a Bioseq, Seq-loc, or contained in a particular Seq-entry or Seq-annot regardless of the referenced locations. The search parameters may be set using SAnnotSelector (preferred method) or using constructors with different arguments. The iterator returns CMappedFeat object rather than CSeq_feat. This allows accessing both the original feature (e.g. loaded from a database) and the mapped one, with its location adjusted according to the search parameters. Most methods of CMappedFeat are just proxies for the original feature members and are not listed here.

CMappedFeat important methods

- GetOriginalFeature - returns the original feature.
- GetSeq_feat_Handle - returns handle for the original feature object.
- GetMappedFeature - returns a copy of the original feature with its location/product adjusted according to the search parameters (e.g. id and ranges changed from a segment to the parent Bioseq). The mapped feature is not created unless requested. This allows improving the iterator's performance.
- GetLocation - although present in CSeq_feat class, this method does not always return the original feature's location, but first checks if the feature should be mapped, creates the mapped location if necessary and returns it. To get the unmapped location use GetOriginalFeature().GetLocation() instead.
- GetAnnot - returns handle for the Seq-annot object, containing the original feature.

See the CMappedFeat API reference for an up-to-date list of all methods.

Alignment iterator

The Alignment iterator (CAlign_CI) enumerates CSeq_align objects related to the specified Bioseq or Seq-loc. It behaves much like CFeat_CI. operator* and operator-> return a mapped CSeq_align object. To get the original alignment you can use GetOriginalSeq_align or GetSeq_align_Handle methods. The objects iterated over may be selected by using SAnnotSelector in the constructor.

Graph iterator

The Graph iterator (CGraph_CI) enumerates CSeq_graph objects related to a specific Bioseq or Seq-loc. It behaves much like CFeat_CI, returning CMappedGraph object which imitates the interface of CSeq_graph and has additional methods to access both original and mapped graphs. The objects iterated over may be selected by using SAnnotSelector in the constructor.

Seq-map iterator

The Seq-map iterator (CSeqMap_CI) is used to enumerate Seq-map segments. The segments to be iterated are selected through a SSeqMapSelector.

CSeqMap_CI important methods

- GetType - returns type of the current segment. The allowed types are eSeqGap, eSeqData, eSubMap, eSeqRef, and eSeqEnd, and eSeqChunk.

- GetPosition - returns start position of the current segment.
- GetLength - returns length of the current segment.
- IsUnknownLength - returns whether the length of the current segment is known.
- GetEndPosition - returns end position (exclusive) of the current segment.
- GetData - returns sequence data (CSeq_data). The current segment type must be eSeqData.
- GetRefSeqId - returns referenced Seq-id for segments of type eSeqRef.
- GetRefData - returns sequence data for any segment which can be resolved to a real sequence. The real position, length and strand of the data should be checked using other methods.
- GetRefPosition - returns start position on the referenced Bioseq for segments of type eSeqRef.
- GetRefEndPosition - returns end position (exclusive) on the referenced Bioseq for segments of type eSeqRef.
- GetRefMinusStrand - returns true if referenced Bioseq's strand should be reversed. If there are several levels of references for the current segment, the method checks strands on each level.

See the CSeqMap_CI API reference for an up-to-date list of all methods.

Note: Some methods will throw exceptions if called inappropriately, so you should either check for the appropriate conditions before calling these methods or catch the exceptions. The methods that throw and the appropriate conditions for calling them are:

Method	**Calling Condition**
GetData	Type must be eSeqGap or eSeqData. If type is eSeqData then GetRefPosition must return zero and GetRefMinusStrand must return false. If the data must be modified (e.g. for a delta sequence) then GetRefData should be called rather than GetData.
GetRefSeqid	Type must be eSeqRef.
GetRefData	Type must be eSeqGap or eSeqData.

Note: Some other methods will not throw exceptions if called inappropriately, and will instead return invalid data. Therefore you must check for the appropriate conditions before calling these methods or using their data:

Method	**Calling Condition**
GetLength	IsUnknownLength must return false.
GetEndPosition	IsUnknownLength must return false.
GetRefEndPosition	Type must be eSeqRef and IsUnknownLength must return false.

SSeqMapSelector

SSeqMapSelector is a helper class which may be used to fine-tune the Seq-map iterator's settings. Below is a brief description of its main class methods.

SSeqMapSelector important methods

- SSeqMapSelector - there is a constructor that takes flags (CSeqMap::Tflags) and a resolve count. The flags can determine which types of segments are included, while the resolve count determines how many levels over which references are resolved.
- SetPosition - selects segments containing this position.
- SetRange - selects segments within this range.
- SetStrand - selects segments matching a strand constraint.
- SetResolveCount - limits the depth of resolved references.
- SetLinkUsedTSE - limits the TSE to resolve references.
- SetFlags - selects segments matching these flags.
- SetByFeaturePolicy - a convenience method equivalent to SetFlags (my_selector.GetFlags() | CSeqMap::fByFeaturePolicy).

See the SSeqMapSelector API reference for an up-to-date list of all methods.

Here is some code that illustrates:

- iterating over data, gaps, and references;
- resolving up to 3 levels of references;
- avoiding exceptions and invalid data; and
- calling various API methods on the iterated segments.

```
// Create a new scope ("attached" to our OM).
// Add default loaders to the scope.
CScope scope(*m_ObjMgr);
scope.AddDefaults();

// Create a Seq-id.
CSeq_id seq_id;
seq_id.SetGi(123456);

// Create a bioseq handle for this seqid.
CBioseq_Handle handle = scope.GetBioseqHandle(seq_id);

// Create the selector, resolving up to 3 levels of references.
SSeqMapSelector sel(CSeqMap::fFindAny, 3);

// Iterate over the segments, printing relevant data:
for ( CSeqMap_CI map_it(handle, sel); map_it; ++map_it ) {
 CSeqMap::ESegmentType segtype = map_it.GetType();

 bool getData = ( ( segtype == CSeqMap::eSeqGap ) ||
 ( segtype == CSeqMap::eSeqData &&
 map_it.GetRefPosition() == 0 &&
 ! map_it.GetRefMinusStrand() ) );
 bool getPos = true;
 bool getLen = ( ! map_it.IsUnknownLength() );
 bool getEndPos = ( ! map_it.IsUnknownLength() );
 bool getRefSeqid = ( segtype == CSeqMap::eSeqRef );
 bool getRefData = ( segtype == CSeqMap::eSeqGap ||
 segtype == CSeqMap::eSeqData );
 bool getRefPos = ( segtype == CSeqMap::eSeqRef );
 bool getRefEndPos = ( segtype == CSeqMap::eSeqRef &&
 ! map_it.IsUnknownLength() );
 bool getRefMinus = ( segtype == CSeqMap::eSeqRef );

 cout << "Type=" << segtype;
 if ( getData ) {
 cout << " Data=";
 if ( map_it.IsSetData() ) {
 if ( segtype == CSeqMap::eSeqGap ) {
 cout << "gap";
 } else {
 const CSeq_data& data(map_it.GetData());
 cout << data.SelectionName(data.Which());
 }
 } else {
 cout << "(not set)";
 }
 }
 if ( getPos ) cout << " Pos=" << map_it.GetPosition();
 if ( getLen ) cout << " Length=" << map_it.GetLength();
 if ( getEndPos ) cout << " EndPos=" << map_it.GetEndPosition();
 if ( getRefSeqid ) cout << " Seqid=" << map_it.GetRefSeqid();
 if ( getRefData ) {
 cout << " RefData=";
 if ( segtype == CSeqMap::eSeqGap ) {
 cout << "gap";
 } else {
 const CSeq_data& refdata(map_it.GetRefData());
 cout << refdata.SelectionName(refdata.Which());
 }
 }
 if ( getRefPos ) cout << " RefPos=" << map_it.GetRefPosition();
 if ( getRefEndPos ) cout << " RefEndPos=" << map_it.GetRefEndPosition();
 if ( getRefMinus ) cout << " RefMinus=" << map_it.GetRefMinusStrand();
 cout << endl;
}
```

Seq-vector iterator

The Seq-vector iterator (CSeqVector_CI) is used to access individual characters from a Seq-vector. It has better performance than CSeqVector::operator[] when used for sequential access to the data.

CSeqVector_CI important methods

- GetSeqData - copy characters from a specified range to a string.
- GetPos, SetPos - control current position of the iterator.

- GetCoding, SetCoding - control character coding.
- SetRandomizeAmbiguities, SetNoAmbiguities - control randomization of ambiguities in ncbi2na coding. If set, ambiguities will be represented with random characters with distribution corresponding to the ambiguity symbol at each position. Once assigned, the same character will be returned every time for the same position.

See the CSeqVector_CI API reference for an up-to-date list of all methods.

Request history and conflict resolution

There are several points of potential ambiguity:

1. the client request may be incomplete;
2. the data in the database may be ambiguous;
3. the data stored by the Object Manager in the local cache may be out of date (in case the database has been updated during the client session);
4. the history of requests may create conflicts (when the Object Manager is unable to decide what exactly is the meaning of the request).

Incomplete Seq-id

Biological sequence id (Seq-id) gives a lot of freedom in defining what sequence the client is interested in. It can be a Gi - a simple integer assigned to a sequence by the NCBI 'ID' database, which in most cases is unique and univocal (Gi does not change if only annotations are changed), but it also can be an accession string only (without version number or release specification). It can specify in what database the sequence data is stored, or this information could be missing.

The Object Manager's interpretation of such requests is kind of arbitrary (yet reasonable, e.g. only the latest version of a given accession is chosen). That is, the sequence could probably be found, but only one sequence, not the list of 'matching' ones. At this point the initially incomplete Seq-id has been resolved into a complete one. That is, the client asked the Scope for a BioseqHandle providing an incomplete Seq-id as the input. The Scope resolved it into a specific complete Seq-id and returned a handle. The client may now ask the handle about its Seq-id. The returned Seq-id differs from the one provided initially by the client.

History of requests

Once the Seq-id has been resolved into a specific Seq-entry, the Object Manager keeps track of all data requests to this sequence in order to maintain consistency. That is, it is perfectly possible that few minutes later this same Seq-id could be resolved into another Seq-entry (the data in the database may change). Still, from the client point of view, as long as this is the same session, nothing should happen - the data should not change.

By 'session' we mean here the same Scope of resolution. That is, as long as the data are requested through the same Scope, it is consistent. In another Scope the data could potentially be different. The Scope can be made to forget about previous requests by calling its ResetHistory() method.

Ambiguous requests

It is possible that there are several Seq-entry's which contain requested information. In this case the processing depends on what is actually requested: sequence data or sequence annotations. The Bioseq may be taken from only one source, while annotations - from several Seq-entry's.

Request for Bioseq

Scopes use several rules when searching for the best Bioseq for each requested Seq-id. These rules are listed below in the order they are applied:

1. Check if the requested Seq-id has been already resolved to a Seq-entry within this scope. This guarantees the same Bioseq will be returned for the same Seq-id.
2. If the Seq-id requested is not resolved yet, request it from Data sources starting from the highest priority sources. Do not check lower-priority sources if something was found in the higher-priority ones.
3. If more than one Data source of the same priority contain the Bioseq or there is one Data source with several versions of the same Seq-id, ask the Data source to resolve the conflict. The Data source may take into account whether the Bioseq is most recent or not, what Seq-entry's have been already used by the Scope (preferred Seq-entry's), etc.

Request for annotations

Annotation iterators start with examining all Data Sources in the Scope to find all top-level Seq-entry's that contain annotations pointing to the given Seq-id. The rules for filtering annotations are slightly different than for resolving Bioseqs. First of all, the scope resolves the requested Seq-id and takes all annotations related to the Seq-id from its top-level Seq-entry. TSEs containing both sequence and annotations with the same Seq-id are ignored, since any other Bioseq with the same id is considered an old version of the resolved one. If there are external annotations in TSEs not containing a Bioseq with the requested Seq-id, they are also collected.

GenBank data loader configuration

Application configuration is stored in a file with the same name as application, and extension .ini. The file will be found either in the executable or in the user's home directory.

GenBank data loader looks for parameters in section [genbank] and its subsections.

Main GenBank data loader configuration section [genbank]

```
[genbank]

; loader_method lists GenBank readers - interfaces to GenBank server.
; They are checked by GenBank loader in the order of appearance in this list.
; For example the value "cache;id2" directs GenBank loader to look in cache
; reader first, then to look for information in id2 reader from GenBank
servers.
; Available readers are: id1, id2, pubseqos, pubseqos2, and cache.
loader_method = cache;id2

; preopen can be set to false to postpone GenBank connection until needed,
; or to true to open connections in all readers at GenBank construction time.
; By default, each reader opens its connection depending on reader settings.
preopen = true
```

GenBank readers configuration

Readers id1& id2
section [genbank/id1] or [genbank/id2]

```
[genbank/id1]
; no_conn means maximum number of simultaneous connections to ID server.
; By default it's 3 in multi-threaded application, and 1 in single-threaded.
no_conn = 2
; max_number_of_connections is a synonym for no_conn, e.g.:
; max_number_of_connections = 2

; If preopen is not set in [genbank] section, local setting of preopen
; will be used to determine when to open ID connection.
; If preopen is set to false, ID reader will open connection only when
needed.
; If the value is true the connection will be opened at GenBank
; construction time.
preopen = false

; ID1/ID2 service name, (default: ID1 or ID2 correspondingly)
service = ID1_TEST

; ID1/ID2 connection timeout in seconds, (default: 20 for ID1 and ID2)
timeout = 10

; ID1/ID2 connection timeout in seconds while opening and initializing,
(default: 5 for ID1 and ID2)
open_timeout = 5

; number of connection retries in case of error (default: 5)
retry = 3
```

Readers pubseqos & pubseqos2
section [genbank/pubseqos] or [genbank/pubseqos2]

```
[genbank/pubseqos]

; no_conn means maximum number of simultaneous connections to PubSeqOS
server.
; By default it's 2 in multi-threaded application, and 1 in single-threaded.
no_conn = 1

; If preopen is not set in [genbank] section, local setting of preopen will
be used
; to determine when to open PubSeqOS connection.
; If preopen is set to false, PubSeqOS reader will open connection only when
needed.
; If the value is true the connection will be opened at GenBank construction
time.
preopen = false

; PubSeqOS server name, (default: PUBSEQ_OS)
server = PUBSEQ_OS_PUBLIC
```

```
; PubSeqOS connection login name, (default: myusername)
user = myusername

; PubSeqOS connection password, (default: mypassword)
password = mypassword

; number of connection retries in case of error (default: 3)
retry = 3
```

Reader cache section [gebank/cache]

GenBank loader cache consists of two parts, **id_cache** for storing small information, and **blob_cache** for storing large sequence data. Parameters of those caches are similar and stored in two sections, **[genbank/cache/id_cache]** and **[genbank/cache/blob_cache].**

The only parameter in those sections is **driver**, which can have values: **bdb** for a cache in a local BerkeleyDB database, **netcache** for a cache in netcached. Then parameters of corresponding **ICache** plugins are stored either in **netcache** or in **bdb** subsections.

Usually, both caches use the same interface with the same parameters, so it makes sense to put interface parameters in one section and include it in both places.

For example:

```
[genbank/cache/id_cache]

driver=netcache


[genbank/cache/id_cache/netcache]

.Include = netcache


[genbank/cache/blob_cache]

driver=netcache


[genbank/cache/blob_cache/netcache]

.Include = netcache


[netcache]

; Section with parameters of netcache interface.
; Name or IP of the computer where netcached is running.
host = localhost
```

```
; Port of netcached service.
port = 9000

; Display name of this application for use by netcached in its logs and
diagnostics.
client = objmgr_demo
```

Configuring NetCached to cache GenBank data

NetCached configuration is stored in netcached.ini file either in the executable or in the user's home directory.

Configuration parameters in the file are grouped in several sections.

Section **[server]** describes parameters of the server not related to storage.

Section **[bdb]** describes parameters of BerkeleyDB database for main NetCache storage.

One or more **[icache_???]** sections describe parameters of ICache instances used by GenBank loader.

Server configuration section [server]

```
[server]

; port number server responds on
port=9000

; maximum number of clients(threads) can be served simultaneously
max_threads=25

; initial number of threads created for incoming requests
init_threads=5

; directory where server creates access log and error log
log_path=

; Server side logging
log=false

; Use name instead of IP address in keys, false by default
;use_hostname=false

; Size of thread local buffer (65536 should be fine)
tls_size=65536

; when true, if database cannot be open (corrupted) server
; automatically drops the db directory (works only for BDB)
; and creates the database from scratch
; (the content is going to be lost)
; Directory reinitialization can be forced by 'netcached -reinit'
drop_db=true
```

```
; Network inactivity timeout in seconds
network_timeout=20

; Switch for session management API
; when turned on if the last customer disconnects server shutdowns
; after waiting for 'session_shutdown_timeout'
session_mng=false

; application shuts itself down if no new sessions arrive in the
; specified time
session_shutdown_timeout=30
```

Main BerkeleyDB database configuration section [bdb]

```
[bdb]

; directory to keep the database. It is important that this
; directory resides on local drive (not NFS)
;
; WARNING: the database directory sometimes can be recursively deleted
;(when netcached started with -reinit).
;DO NOT keep any of your files(besides the database) in it.
path=e:/netcached_data

; Path to transaction log storage. By default transaction logs are stored
; at the same location as main database, but to improve performance it's
; best to put it to a dedicated fast hard drive (split I/O load)
;
transaction_log_path=

; cache name
name=nccache

; use syncronous or asyncromous writes (used with transactions)
write_sync=false

; Direct IO for database files
direct_db=false

; Direct IO for transaction logs
direct_log=false

; when 'true' the database is transaction protected
use_transactions=true

; BLOB expiration timeout in seconds
timeout=3600

; onread - update BLOB time stamp on every read
;(otherwise only creation time will taken into account)
```

```
; purge_on_startup - delete all deprecated BLOBs when startind netcached
; (may significantly slow down startup propcess)
; check_expiration - check if BLOB has expired (on read) and if it is
; return 'not found'. Otherwise BLOB lives until
; it is deleted by the internal garbage collector
timestamp=onread
# purge_on_startup check_expiration

; do not change this
keep_versions=all

; Run background cleaning thread
; (Pretty much mandatory parameter, turn it off only if you want
; to keep absolutely everything in the database)
purge_thread=true

; Delay (seconds) between purge(garbage collector) runs.
purge_thread_delay=30

; maintanance thread sleeps for specified number of milliseconds after
; each batch. By changing this parameter you can adjust the purge
; thread priority
purge_batch_sleep=100

; maintanance thread processes database records by chunks of specified
; number. If you increase this number it also increases the performance
; of purge process (at the expense of the online connections)
purge_batch_size=70

; amount of memory allocated by BerkeleyDB for the database cache
; Berkeley DB page cache) (More is better)
mem_size=50M

; when non 0 transaction LOG will be placed to memory for better performance
; as a result transactions become non-durable and there is a risk of
; loosing the data if server fails
; (set to at least 100M if planned to have bulk transactions)
;
log_mem_size=0

; Maximum size of the transaction log file
log_file_max=200M

; Percent of pages NC tries to keep available for read
; 0 - means no background dirty page write
;
memp_trickle=10

; Number of times Berkeley DB mutex spins before sleeping
; for some reason values beyond 75 somehow disable memp_trickle
;
```

```
tas_spins=200

; Specifies how often cache should remove the Berkeley DB LOG files
; Removal is triggered by the purge thread. Value of 2 means LOG is
; cleaned every second purge
purge_clean_log=2

; Call transaction checkpoint every 'checkpoint_bytes' of stored data
checkpoint_bytes=10M

; BLOBs < 10M stored in database
overflow_limit=10M

; This parameter regulates BLOB expiration. If client constantly reads
; the BLOB and you do not want it to stuck in the database forever
; (timestamp=onread), set this parameter.
; If timeout is 3600 and ttl_prolong is 2, maximum possible timeout for
; the BLOB becomes 3600 * 2 = 7200 seconds.
ttl_prolong=3

; Maximum allowed BLOB size (for a single BLOB). 0 - no restriction
max_blob_size=0

; Number of round robin volumes. 0 - no rotation
; Cache opens approx 7 files per RR volume.
rr_volumes=3
```

ICache instances configuration sections [icache_*]

Each ICache instance has an interface name which is used by clients to select the instance.

The name of the section with the ICache instance's configuration is a concatenation of the string **icache_** and the name of the instance.

For example, the parameters of an ICache instance named **ids** are stored in the section **[icache_ids]**.

The parameters inside the section are the same as the parameters in the **[bdb]** section with some exceptions.

If the **path** parameter has the same value as **path** in main **[bdb]** section, then both databases will be stored in the same directory and share the same BerkeleyDB environment.

As a result, all parameters of the BerkeleyDB environment have no meaning in an ICache section and are taken from the [**bdb**] section instead. To avoid a database conflict, all sections with the same **path** parameter must have different **name** parameters.

The GenBank data loader requires two cache instances with slightly different parameters. The first, named **ids** by default, is used for small Seq-id resolution information. The second, named **blobs** by default, is used for large Seq-entry information. The names of those caches can be changed in the client program configuration.

Similarly, NetCached configuration should describe two instances of ICache with names matching to the names on client (**ids** and **blobs** by default).

For example:

```
[icache_ids]
name=ids
path=e:/netcached_data
write_sync=false
use_transactions=true
timeout=3600
timestamp=subkey check_expiration
keep_versions=all
purge_thread=true
purge_thread_delay=3600
purge_batch_sleep=5000
purge_batch_size=10
mem_size=0
purge_clean_log=10
checkpoint_bytes=10M
overflow_limit=1M
ttl_prolong=3
page_size=small

[icache_blobs]
name=blobs
path=e:/netcached_data
write_sync=false
use_transactions=true
timeout=3600
timestamp=subkey onread check_expiration
keep_versions=all
purge_thread=true
purge_thread_delay=3600
purge_batch_sleep=5000
purge_batch_size=10
mem_size=0
purge_clean_log=10
checkpoint_bytes=10M
overflow_limit=1M
ttl_prolong
```

Use of Local Data Storage (LDS) by Object Manager

Serializable object data can be stored locally in an SQLite database for efficient access from the Object Manager.

The required libraries are:

UNIX	LIB = ncbi_xloader_lds2 lds2 xobjread id2 id1 seqsplit sqlitewrapp creaders $(COMPRESS_LIBS) $(SOBJMGR_LIBS)
	LIBS = $(SQLITE3_LIBS) $(CMPRS_LIBS) $(DL_LIBS) $(ORIG_LIBS)

Windows	id1.lib, id2.lib, lds2.lib, sqlitewrapp.lib, sqlite3.lib, ncbi_xloader_lds2.lib, xobjread.lib

A demonstration program is available: SVN | LXR

Registering the LDS loader with the Object Manager

The CLDS2_Manager class creates (or updates) an SQLite database at the path specified in its constructor. Data files that it should manage can be specified with the AddDataFile() and/or AddDataDir() methods. AddDataFile() adds a single data file; AddDataDir() adds all data files in the specified directory and its subdirectories (by default). Recursion into the subdirectories can be disabled by passing CLDS2_Manager::eDir_NoRecurse as the second argument to the AddDataDir() call. UpdateData() synchronizes the database with all the added data files. Source data files can be in ASN.1 text, ASN.1 binary, XML, or FASTA format.

For example, the following code creates an LDS database, populates it with data, registers it with the Object Manager, and adds the LDS data loader to the scope.

```
// Create/update LDS db at given path, based on data in given directory.
CRef<CLDS2_Manager> mgr(new CLDS2_Manager(db_path));
mgr->AddDataDir(data_dir);
mgr->UpdateData();

// Register LDS with Object Manager.
CLDS2_DataLoader::RegisterInObjectManager(*object_manager, db_path);

// Explicitly add LDS to scope.
scope.AddDataLoader(CLDS2_DataLoader::GetLoaderNameFromArgs(db_path));
```

Using both the LDS and GenBank loaders

The previous example adds the LDS data loader to the scope without adding any default loaders, including GenBank. To add both the LDS and GenBank loaders (but no other default loaders) to the scope:

```
// Create/update LDS db at given path, based on data in given directory.
CRef<CLDS2_Manager> mgr(new CLDS2_Manager(db_path));
mgr->AddDataDir(data_dir);
mgr->UpdateData();

// Register LDS with Object Manager - as first priority.
CLDS2_DataLoader::RegisterInObjectManager(*object_manager, db_path, -1,
 CObjectManager::eNonDefault, 1);

// Explicitly add LDS to scope.
scope.AddDataLoader(CLDS2_DataLoader::GetLoaderNameFromArgs(db_path));

// Register GenBank with Object Manager - as second priority.
CGBDataLoader::RegisterInObjectManager(*object_manager, 0,
 CObjectManager::eNonDefault, 2);
```

```
// Explicitly add GenBank to scope.
scope.AddDataLoader(CGBDataLoader::GetLoaderNameFromArgs());
```

The scope will now include just LDS and GenBank.

CObjectManager::eNonDefault was passed to the RegisterInObjectManager() method in this example simply because it is the default value for that argument, and some value was necessary so that the next argument could be specified. It could equally well have been CObjectManager::eDefault.

The last argument to RegisterInObjectManager() is the priority. Here it was set to 1 for LDS and 2 for GenBank so the Object Manager would attempt to load data via LDS first, and only if that failed would it resort to GenBank.

In the above example, the loaders were explicitly added to the scope to ensure that they were the only loaders in the scope.

To add the LDS data loader and any other default loaders to the scope:

```
// Create/update LDS db at given path, based on data in given directory.
CRef<CLDS2_Manager> mgr(new CLDS2_Manager(db_path));
mgr->AddDataDir(data_dir);
mgr->UpdateData();

// Register LDS with Object Manager - as first priority.
CLDS2_DataLoader::RegisterInObjectManager(*object_manager, db_path, -1,
 CObjectManager::eDefault, 1);

// Register GenBank with Object Manager - as second priority.
CGBDataLoader::RegisterInObjectManager(*object_manager, 0,
 CObjectManager::eDefault, 2);

// Add default loaders to scope.
scope.AddDefaults();
```

By registering with eDefault, the LDS data loader will be added to the scope along with the default data loaders.

Known gotchas

Resolving Data References

Multiple factors determine whether data references can be resolved or not. For example, imagine that a local data store has been created from a collection of simple annotations. References between annotations might not be resolved, unless the GenBank loader is also registered with the Object Manager, or unless a flag has been set to search unresolved annotations, as in:

```
SAnnotSelector sel;
sel.SetUnresolvedFlag(SAnnotSelector::eSearchUnresolved);
```

For more information about resolving data references, see the section on SAnnot_Selector and the associated header documentation.

Setting Loader Priority

It is the caller's responsibility to ensure that the priorities are different for different loaders – or that the same sequence won't be found by both loaders. If multiple loaders are registered with the same priority, or if they are registered without specifying a priority (which results in them both getting the default priority), and if both loaders can fetch the same data, then an exception may be thrown.

In-Memory Caching in the Object Manager and Data Loaders

The following table summarizes the classes that perform short-term, in-memory caching for various objects. A custom class must be written for short-term caching of other objects or long-term caching of any objects.

Object(s)	**Caching done by**
master TSE blob	CObjectManager
id, gi, label, taxid	CGBDataLoader
blob id	CGBDataLoader

If you want in-memory caching for objects other than those listed in the table, you can implement a cache in a CDataLoader subclass. For an example implementation, see the CGBDataLoader class. CGBDataLoader actually has two Seq-id caches - one for blob id's and the other for the other small objects listed in the table. The size for both of these caches is controlled through the [GENBANK] ID_GC_SIZE configuration parameter (i.e. their sizes can't be set independently). Subclasses of CGBDataLoader can access their configuration using the CParam methods.

Short-term caching, as applied to the Object Manager and Data Loaders, means keeping data for "a little while" in a FIFO before deleting. Long-term caching means keeping objects for "a long while" – i.e. longer than they would be kept using a short-term cache. Here, "a while" is relative to the rate at which objects are discarded, not relative to elapsed time. So short-term caching means keeping at most a given number of objects, rather than keeping objects for a given amount of time.

A CDataSource object inside the Object Manager automatically performs short-term caching of blobs for master TSEs. To set the Object Manager's blob cache size, use the [OBJMGR] BLOB_CACHE configuration parameter. This configuration parameter is created by the CParam declaration "NCBI_PARAM_DECL(unsigned, OBJMGR, BLOB_CACHE)" in src/objmgr/data_source.cpp and can be set via the environment, the registry, or manipulated via the CParam API. Note: External annotation TSEs and TSEs with Delta segments are linked to one of the master TSEs.

Short-term caching is done automatically for CGBDataLoader, but not for other data loaders. If you want short-term caching for some other data loader, you'll have to add it, possibly using CGBDataLoader as an example.

Long-term caching is not done by either the Object Manager or the GenBank data loader, so to get it you will have to implement your own mechanism. Simply keeping a handle on the objects you wish to cache long-term will prevent them from being put into the short-term cache. When you no longer need the objects to be cached, just delete the handles. Note that some system of prioritization must be used to limit the number of handles kept, since keeping handles on all object would be essentially the same as increasing the short-term cache size, which presumably failed if you're trying long-term caching. You may want to see if the CSyncQueue__priority__queue class will meet your needs.

How to use it

Start working with the Object Manager

Include the necessary headers:

```
#include <objmgr/object_manager.hpp>
#include <objmgr/scope.hpp>
#include <objmgr/bioseq_handle.hpp>
#include <objmgr/seq_vector.hpp>
#include <objmgr/desc_ci.hpp>
#include <objmgr/feat_ci.hpp>
#include <objmgr/align_ci.hpp>
#include <objmgr/graph_ci.hpp>
```

Request an instance of the CObjectManager and store as CRef:

```
CRef<CObjectManager> obj_mgr = CObjectManager::GetInstance();
```

Create a CScope. The Scope may be created as an object on the stack, or on the heap:

```
CRef<CScope> scope1 = new CScope(*obj_mgr);
CScope scope2(*obj_mgr);
```

Add externally created top-level Seq-entry to the Scope

Once there is a Seq-entry created somehow, it can be added to the Scope using the following code:

```
CRef<CSeq_entry> entry(new CSeq_entry);
... // Populate or load the Seq-entry in some way
scope.AddTopLevelSeqEntry(*entry);
```

Add a data loader to the Scope

The data loader is designed to be a replaceable object. There can be a variety of data loaders, each of which would load data from different databases, flat files, etc. Each data loader must be registered with the Object Manager. One distinguishes them later by their names. One of the most popular data loaders is the one that loads data from GenBank - CGBDataLoader. Each loader has at least one RegisterInObjectManager() static method, the first argument is usually a reference to the Object Manager:

```
#include <objtools/data_loaders/genbank/gbloader.hpp>
...
CGBDataLoader::RegisterInObjectManager(*obj_mgr);
```

A data loader may be registered as a default or non-default loader. The GenBank loader is automatically registered as default if you don't override it explicitly. For other loaders you may need to specify additional arguments to set their priority or make them default (usually this can be done through the last two arguments of the RegisterInObjectManager() method). A Scope can request data loaders from the Object Manager one at a time - by name. In this case you will need to know the loader's name. You can get it from the loader using its GetName() method, or if you don't have a loader instance, you can use the desired loader's static method GetLoaderNameFromArgs():

```
scope.AddDataLoader(my_loader.GetName()); // with loader instance
scope.AddDataLoader(CGBDataLoader::GetLoaderNameFromArgs()); // without a
loader
```

A more convenient way to add data loaders to a Scope is to register them with the Object Manager as default and then add all the default loaders to the scope, for example:

```
CLDS2_DataLoader::RegisterInObjectManager(*object_manager, db_path, -1,
 CObjectManager::eDefault, 1);
scope.AddDefaults();
```

Start working with a Bioseq

In order to be able to access a Bioseq, one has to obtain a Bioseq handle from the Scope, based on a known Seq_id. It's always a good idea to check if the operation was successful:

```
CSeq_id seq_id;
seq_id.SetGi(3);
CBioseq_Handle handle = scope.GetBioseqHandle(seq_id);
if ( !handle ) {
 ... // Failed to get the bioseq handle
}
```

Access sequence data

The access to the sequence data is provided through the Seq- vector object, which can be obtained from a Bioseq handle. The vector may be used together with Seq-vector iterator to enumerate the sequence characters:

```
CSeqVector seq_vec = handle.GetSeqVector(CBioseq_Handle::eCoding_Iupac);
for (CSeqVector_CI it = seq_vec.begin(); it; ++it) {
 NcbiCout << *it;
}
```

Seq-vector
class provides much more than the plain data storage. It rather 'knows where to find' the data.

As a result of a query, it may initiate reference resolution process, send requests to the source database for more data etc.

There is another useful object, which describes sequence data - Sequence map. It is a collection of segments, which describe sequence parts in general - location and type only, without providing any real data. To obtain Sequence map from a Bioseq handle:

```
CConstRef<CSeqMap> seqmap(&handle.GetSeqMap());
```

It is possible then to enumerate all the segments in the map asking their type, length or position. Note that in this example the iterator is obtained using begin() method and will enumerate only top level segments of the Seq-map:

```
int len = 0;
for (CSeqMap::const_iterator seg = seqmap->begin() ; seg; ++seg) {
 switch (seg.GetType()) {
 case CSeqMap::eSeqData:
 len += seg.GetLength();
 break;
 case CSeqMap::eSeqRef:
 len += seg.GetLength();
 break;
 case CSeqMap::eSeqGap:
 len += seg.GetLength();
 break;
 default:
 break;
 }
}
```

Enumerate sequence descriptions

Description iterator may be initialized with a Bioseq handle or Seq-entry handle. It makes it possible to enumerate all CSeqdesc objects the Bioseq or the Seq-entry refers to:

```
for (CSeqdesc_CI desc_it(handle); desc_it; ++desc_it) {
 const CSeqdesc& desc = *desc_it;
 ... // your code here
}
```

Another type of descriptor iterator iterates over sets of descriptors rather than individual objects:

```
for (CSeq_descr_CI descr_it(handle); descr_it; ++descr_it) {
 const CSeq_descr& descr = *descr_it;
 ... // your code here
}
```

Enumerate sequence annotations

Annotation iterators may be used to enumerate annotations (features, alignments and graphs) related to a Bioseq or a Seq-loc. They are very flexible and can be fine-tuned through Annot-selector structure:

```
// Search all TSEs in the Scope for gene features
SAnnotSelector sel;
sel.SetFeatType(CSeqFeatData::e_Gene);
/// both start and stop are 0 - iterate the whole bioseq
CFeat_CI feat_it(handle, 0, 0, sel);
for (; feat_it; ++feat_it) {
 const CSeq_loc& loc = feat_it->GetLocation();
 ... // your code here
}
```

The next example shows slightly more complicated settings for the feature iterator. The selector forces resolution of all references, both near (to Bioseqs located in the same TSE) and far. The features will be collected from all segments resolved. Since this may result in loading a lot of external Bioseqs, the selector is set to restrict the depth of references to 2 levels:

```
SAnnotSelector sel;
sel.SetFeatType(CSeqFeatData::e_Gene)
 .SetReaolveAll()
 .SetResolveDepth(2);
CFeat_CI feat_it(handle, 0, 0, sel);
for (; feat_it; ++feat_it) {
 const CSeq_loc& loc = feat_it->GetLocation();
 ... // your code here
}
```

Usage of alignment and graph iterators is similar to the feature iterator:

```
CAlign_CI align_it(handle, 0, 0);
...
CGraph_CI graph_it(handle, 0, 0);
...
```

All the above examples iterate annotations in a continuous interval on a Bioseq. To specify more complicated locations a Seq-loc may be used instead of the Bioseq handle. The Seq-loc may even reference different ranges on several Bioseqs:

```
CSeq_loc loc;
CSeq_loc_mix& mix = loc.SetMix();
... // fill the mixed location
for (CFeat_CI feat_it(scope, loc); feat_it; ++feat_it) {
 const CSeq_loc& feat_loc = feat_it->GetLocation();
 ... // your code here
}
```

Use the CPrefetchManager class

Suppose you want to retrieve all the features for several hundred thousand protein sequences. Features don't consume much memory and protein sequences typically have a small number of features, so it should be feasible to simultaneously load all the features into memory.

The CPrefetchManager class was designed to improve the efficiency of this type of data retrieval, as illustrated here:

```
// Set up all the object manager stuff.
m_ObjMgr = CObjectManager::GetInstance();
CGBDataLoader::RegisterInObjectManager(*m_ObjMgr);
CScope scope(*m_ObjMgr);
scope.AddDefaults();
SAnnotSelector sel(CSeqFeatData::e_not_set);
sel.SetResolveMethod(sel.eResolve_All);

// Create a vector for IDs.
vector<CSeq_id_Handle> m_Ids;
PopulateTheIdVectorSomehow(&m_Ids);

// Create the prefetch manager.
m_PrefMgr = new CPrefetchManager();

// Create a prefetch sequence, using the prefetch manager and based on a
// feature iterator action source (in turn based on the scope, IDs, and
// feature selector).
// Note: CPrefetchSequence is designed to take ownership of the action
// source, so do not delete it or use auto_ptr etc to manage it.
CRef<CPrefetchSequence> prefetch;
prefetch = new CPrefetchSequence(*m_PrefMgr,
new CPrefetchFeat_CIActionSource(CScopeSource::New(scope),
m_Ids, sel));

// Fetch data for each ID.
for ( size_t i = 0; i < m_Ids.size(); ++i ) {

// Get a feature iterator that uses the prefetch.
CRef<CPrefetchRequest> token = prefetch->GetNextToken();
CFeat_CI it = CStdPrefetch::GetFeat_CI(token);

// Do something with the fetched features.
for ( ; it; ++it ) {
DoSomethingInterestingWithTheFeature(it->GetOriginalFeature());
}
}
```

Note: Error handling was removed from the above code for clarity - please see the Object Manager test code for examples of appropriate error handling.

Educational Exercises

Setup the framework for the C++ Object Manager learning task

Starting point

To jump-start your first project utilizing the new C++ Object Manager in the C++ Toolkit framework on a UNIX platform, we suggest using the new_project shell script, which creates a sample application and a makefile:

1. Create a new project called task in the folder task using the new_project shell script (this will create the folder, the source file and the makefile):

```
new_project task app/objmgr
```

2. Build the sample project and run the application:

   ```
   cd task
   make -f Makefile.task_app
   ./task -gi 333
   ```

   The output should look like this (although for technical reasons the following output sample is missing most of its whitespace):

   ```
   First ID: emb|CAA23443.1|
   Sequence: length=263, data=MARFLGLCTW
   # of descriptions: 6
   # of features:
   [whole] Any: 2
   [whole] Genes: 0
   [0..9] Any: 2
   [0..999, TSE] Any: 1
   # of alignments:
   [whole] Any: 0
   Done
   ```

3. Now you can go ahead and convert the sample code in the task.cpp into the code that performs your learning task.

The new_project script can also be used to create a new project on Windows, and the usage is the same as on UNIX.

How to convert the test application into CGI one?

In order to convert your application into CGI one:

1. Create copy of the source (task.cpp) and makefile (Makefile.task_app)

   ```
   cp task.cpp task_cgi.cpp
   cp Makefile.task_app Makefile.task_cgiapp
   ```

2. Edit the makefile for the CGI application (Makefile.task_cgiapp): change application name, name of the source file, add cgi libraries:

   ```
   APP = task.cgi
   SRC = task_cgi

   LIB = xobjmgr id1 seqset $(SEQ_LIBS) pub medline biblio general \
   xser xhtml xcgi xutil xconnect xncbi
   LIBS = $(NCBI_C_LIBPATH) $(NCBI_C_ncbi) $(FASTCGI_LIBS) \
   $(NETWORK_LIBS) $(ORIG_LIBS)
   ```

3. Build the project (at this time it is not a CGI application yet):

   ```
   make -f Makefile.task_cgiapp
   ```

4. Convert task_cgi.cpp into a CGI application.

Please also see the section on FCGI Redirection and Debugging CGI Programs for more information.

Convert CGI application into Fast-CGI one

In the LIB=... section of Makefile.task_cgiapp, just replace xcgi library by xfcgi:

```
LIB = xobjmgr id1 seqset $(SEQ_LIBS) pub medline biblio general \
 xser xhtml xfcgi xutil xconnect xncbi
```

Task Description

We have compiled here a list of teaching examples to help you start working with the C++ Object Manager. Completing them, getting your comments and investigating the problems encountered would let us give warnings of issues to deal with in the nearest future, better understand what modifications should be made to this software system.

The main idea here is to build one task on the top of another, in growing level of complexity:

1. having a Seq-id (GI), get the Bioseq;
2. print the Bioseq's title descriptor;
3. print the Bioseq's length;
4. dump the Seg-map structure;
5. print the total number of cd-region features on the Bioseq;
6. calculate percentage of 'G' and 'C' symbols in the whole sequence;
7. calculate percentage of 'G' and 'C' symbols within cd-regions;
8. calculate percentage of 'G' and 'C' symbols for regions outside any cd-region feature;
9. convert the application into a CGI one;
10. convert the application into a FCGI one.

Test Bioseqs

Below is the list of example sequences to use with the C++ toolkit training course. It starts with one Teaching Example that has one genomic nucleic acid sequence and one protein with a cd-region. Following that is the list of Test Examples. Once the code is functioning on the Teaching Example, we suggest running it through these. They include a bunch of different conditions: short sequence with one cd-region, longer with 6 cd-regions, a protein record (this is an error, and code should recover), segmented sequence, 8 megabase genomic contig, a popset member, and a draft sequence with no cd-regions.

Teaching example

IDs and description of the sequence to be used as a simple teaching example is shown in Table 1.

The application should produce the following results for the above Bioseq:

```
ID: emb|AJ438945.1|HSA438945 + gi|19584253
Homo sapiens SLC16A1 gene for monocarboxylate transporter isoform 1, exons
2-5
Sequence length: 17312
Sequence map:
 Segment: pos=0, length=17312, type=DATA
Total: 40.29%
 cdr0: 46.4405%
```

```
Cdreg: 46.4405%
Non-Cdreg: 39.7052%
```

Test examples

More complicated test Bioseqs are listed in Table 2.

Correct Results

Below are shown the correct results for each of the test Bioseqs. You can use them as reference to make sure your application works correctly.

```
ID: gb|J01066.1|DROADH + gi|156787
D.melanogaster alcohol dehydrogenase gene, complete cds.
Sequence length: 2126
Sequence map:
 Segment: pos=0, length=2126, type=DATA
Total: 45.8137%
 cdr0: 57.847%
Cdreg: 57.847%
Non-Cdreg: 38.9668%

ID: gb|U01317.1|HUMHBB + gi|455025
Human beta globin region on chromosome 11.
Sequence length: 73308
Sequence map:
 Segment: pos=0, length=73308, type=DATA
Total: 39.465%
 cdr0: 52.9279%
 cdr1: 53.6036%
 cdr2: 53.6036%
 cdr3: 49.2099%
 cdr4: 54.5045%
 cdr5: 56.3063%
 cdr6: 56.7568%
Cdreg: 53.2811%
Non-Cdreg: 38.9403%

ID: emb|AJ293577.1|HSA293577 + gi|14971422
Homo sapiens partial MOCS1 gene, exon 1 and joined CDS
Sequence length: 913
Sequence map:
 Segment: pos=0, length=913, type=DATA
Total: 54.655%
 cdr0: 58.3765%
Cdreg: 58.3765%
Non-Cdreg: 51.5837%

ID: gb|AH011004.1|SEG_Y043402S + gi|19550966
Mus musculus light ear protein (le) gene, complete cds.
Sequence length: 5571
Sequence map:
 Segment: pos=0, length=255, type=DATA
 Segment: pos=255, length=0, type=GAP
 Segment: pos=255, length=306, type=DATA
 Segment: pos=561, length=0, type=GAP
 Segment: pos=561, length=309, type=DATA
 Segment: pos=870, length=0, type=GAP
 Segment: pos=870, length=339, type=DATA
 Segment: pos=1209, length=0, type=GAP
 Segment: pos=1209, length=404, type=DATA
 Segment: pos=1613, length=0, type=GAP
 Segment: pos=1613, length=349, type=DATA
 Segment: pos=1962, length=0, type=GAP
 Segment: pos=1962, length=361, type=DATA
 Segment: pos=2323, length=0, type=GAP
 Segment: pos=2323, length=369, type=DATA
 Segment: pos=2692, length=0, type=GAP
 Segment: pos=2692, length=347, type=DATA
 Segment: pos=3039, length=0, type=GAP
 Segment: pos=3039, length=1066, type=DATA
 Segment: pos=4105, length=0, type=GAP
 Segment: pos=4105, length=465, type=DATA
 Segment: pos=4570, length=0, type=GAP
 Segment: pos=4570, length=417, type=DATA
 Segment: pos=4987, length=0, type=GAP
 Segment: pos=4987, length=584, type=DATA
Total: 57.2305%
 cdr0: 59.5734%
Cdreg: 59.5734%
Non-Cdreg: 55.8899%

ID: ref|NT_017168.8|HS7_17324 + gi|18565551
Homo sapiens chromosome 7 working draft sequence segment
Sequence length: 8470605
Sequence map:
 Segment: pos=0, length=29884, type=DATA
 Segment: pos=29884, length=100, type=GAP
 Segment: pos=29984, length=20739, type=DATA
 Segment: pos=50723, length=100, type=GAP
 Segment: pos=50823, length=157624, type=DATA
 Segment: pos=208447, length=29098, type=DATA
 Segment: pos=237545, length=115321, type=DATA
 Segment: pos=352866, length=25743, type=DATA
 Segment: pos=378609, length=116266, type=DATA
 Segment: pos=494875, length=144935, type=DATA
 Segment: pos=639810, length=108678, type=DATA
 Segment: pos=748488, length=102398, type=DATA
 Segment: pos=850886, length=149564, type=DATA
 Segment: pos=1000450, length=120030, type=DATA
 Segment: pos=1120480, length=89411, type=DATA
 Segment: pos=1209891, length=51161, type=DATA
 Segment: pos=1261052, length=131072, type=DATA
 Segment: pos=1392124, length=118395, type=DATA
```

```
Segment: pos=1510519, length=70119, type=DATA
Segment: pos=1580638, length=59919, type=DATA
Segment: pos=1640557, length=131072, type=DATA
Segment: pos=1771629, length=41711, type=DATA
Segment: pos=1813340, length=131072, type=DATA
Segment: pos=1944412, length=56095, type=DATA
Segment: pos=2000507, length=93704, type=DATA
Segment: pos=2094211, length=82061, type=DATA
Segment: pos=2176272, length=73699, type=DATA
Segment: pos=2249971, length=148994, type=DATA
Segment: pos=2398965, length=37272, type=DATA
Segment: pos=2436237, length=96425, type=DATA
Segment: pos=2532662, length=142196, type=DATA
Segment: pos=2674858, length=58905, type=DATA
Segment: pos=2733763, length=94760, type=DATA
Segment: pos=2828523, length=110194, type=DATA
Segment: pos=2938717, length=84638, type=DATA
Segment: pos=3023355, length=94120, type=DATA
Segment: pos=3117475, length=46219, type=DATA
Segment: pos=3163694, length=7249, type=DATA
Segment: pos=3170943, length=118946, type=DATA
Segment: pos=3289889, length=127808, type=DATA
Segment: pos=3417697, length=51783, type=DATA
Segment: pos=3469480, length=127727, type=DATA
Segment: pos=3597207, length=76631, type=DATA
Segment: pos=3673838, length=81832, type=DATA
Segment: pos=3755670, length=21142, type=DATA
Segment: pos=3776812, length=156640, type=DATA
Segment: pos=3933452, length=117754, type=DATA
Segment: pos=4051206, length=107098, type=DATA
Segment: pos=4158304, length=15499, type=DATA
Segment: pos=4173803, length=156199, type=DATA
Segment: pos=4330002, length=89478, type=DATA
Segment: pos=4419480, length=156014, type=DATA
Segment: pos=4575494, length=105047, type=DATA
Segment: pos=4680541, length=120711, type=DATA
Segment: pos=4801252, length=119796, type=DATA
Segment: pos=4921048, length=35711, type=DATA
Segment: pos=4956759, length=131072, type=DATA
Segment: pos=5087831, length=1747, type=DATA
Segment: pos=5089578, length=38864, type=DATA
Segment: pos=5128442, length=131072, type=DATA
Segment: pos=5259514, length=97493, type=DATA
Segment: pos=5357007, length=125390, type=DATA
Segment: pos=5482397, length=96758, type=DATA
Segment: pos=5579155, length=1822, type=DATA
Segment: pos=5580977, length=144039, type=DATA
Segment: pos=5725016, length=58445, type=DATA
Segment: pos=5783461, length=158094, type=DATA
Segment: pos=5941555, length=4191, type=DATA
Segment: pos=5945746, length=143965, type=DATA
```

```
 Segment: pos=6089711, length=107230, type=DATA
 Segment: pos=6196941, length=158337, type=DATA
 Segment: pos=6355278, length=25906, type=DATA
 Segment: pos=6381184, length=71810, type=DATA
 Segment: pos=6452994, length=118113, type=DATA
 Segment: pos=6571107, length=118134, type=DATA
 Segment: pos=6689241, length=92669, type=DATA
 Segment: pos=6781910, length=123131, type=DATA
 Segment: pos=6905041, length=136624, type=DATA
 Segment: pos=7041665, length=177180, type=DATA
 Segment: pos=7218845, length=98272, type=DATA
 Segment: pos=7317117, length=22979, type=DATA
 Segment: pos=7340096, length=123747, type=DATA
 Segment: pos=7463843, length=13134, type=DATA
 Segment: pos=7476977, length=156146, type=DATA
 Segment: pos=7633123, length=59501, type=DATA
 Segment: pos=7692624, length=107689, type=DATA
 Segment: pos=7800313, length=29779, type=DATA
 Segment: pos=7830092, length=135950, type=DATA
 Segment: pos=7966042, length=71035, type=DATA
 Segment: pos=8037077, length=129637, type=DATA
 Segment: pos=8166714, length=80331, type=DATA
 Segment: pos=8247045, length=49125, type=DATA
 Segment: pos=8296170, length=131072, type=DATA
 Segment: pos=8427242, length=25426, type=DATA
 Segment: pos=8452668, length=100, type=GAP
 Segment: pos=8452768, length=16014, type=DATA
 Segment: pos=8468782, length=100, type=GAP
 Segment: pos=8468882, length=1723, type=DATA
Total: 37.2259%
 cdr0: 39.6135%
 cdr1: 38.9474%
 cdr2: 57.362%
 cdr3: 59.144%
 cdr4: 45.4338%
 cdr5: 37.6812%
 cdr6: 58.9856%
 cdr7: 61.1408%
 cdr8: 51.2472%
 cdr9: 44.2105%
 cdr10: 49.1071%
 cdr11: 43.6508%
 cdr12: 38.3754%
 cdr13: 39.1892%
 cdr14: 42.2222%
 cdr15: 49.5763%
 cdr16: 44.4034%
 cdr17: 42.9907%
 cdr18: 47.619%
 cdr19: 47.3684%
 cdr20: 47.973%
```

```
cdr21: 38.6544%
cdr22: 45.3052%
cdr23: 37.7115%
cdr24: 36.1331%
cdr25: 61.4583%
cdr26: 51.9878%
cdr27: 47.6667%
cdr28: 45.3608%
cdr29: 38.7387%
cdr30: 37.415%
cdr31: 40.5405%
cdr32: 41.1819%
cdr33: 42.6791%
cdr34: 43.7352%
cdr35: 44.9235%
cdr36: 38.218%
cdr37: 34.4928%
cdr38: 44.3137%
cdr39: 37.9734%
cdr40: 37.0717%
cdr41: 48.6772%
cdr42: 38.25%
cdr43: 48.8701%
cdr44: 46.201%
cdr45: 46.7803%
cdr46: 55.8405%
cdr47: 43.672%
cdr48: 50.3623%
cdr49: 65.4835%
cdr50: 52.6807%
cdr51: 45.7447%
cdr52: 53.7037%
cdr53: 49.6599%
cdr54: 38.5739%
cdr55: 63.3772%
cdr56: 37.6274%
cdr57: 38.0952%
cdr58: 39.6352%
cdr59: 39.6078%
cdr60: 58.4795%
cdr61: 49.4987%
cdr62: 47.0968%
cdr63: 45.0617%
cdr64: 41.5133%
cdr65: 40.2516%
cdr66: 39.6208%
cdr67: 40.4412%
cdr68: 43.0199%
cdr69: 40.5512%
cdr70: 54.7325%
cdr71: 45.3034%
```

```
 cdr72: 55.6634%
 cdr73: 43.7107%
 cdr74: 45.098%
 cdr75: 43.8406%
 cdr76: 49.4137%
 cdr77: 44.7006%
 cdr78: 44.6899%
 cdr79: 56.4151%
 cdr80: 36.1975%
 cdr81: 34.8238%
 cdr82: 38.5447%
 cdr83: 44.0451%
 cdr84: 45.6684%
 cdr85: 45.1696%
 cdr86: 40.9462%
 cdr87: 56.044%
 cdr88: 46.2366%
 cdr89: 41.1765%
 cdr90: 42.9698%
 cdr91: 47.8261%
 cdr92: 43.2234%
 cdr93: 49.7849%
 cdr94: 43.3755%
 cdr95: 51.2149%
Cdreg: 44.397%
Non-Cdreg: 37.1899%

ID: gb|AF022257.1| + gi|2415435
HIV-1 patient ACH0039, clone 3918C6 from The Netherlands, envelope
glycoprotein V3 region (env) gene, partial cds.
Sequence length: 388
Sequence map:
 Segment: pos=0, length=388, type=DATA
Total: 31.9588%
 cdr0: 31.9588%
Cdreg: 31.9588%
Non-Cdreg: 0%

ID: gb|AC116052.1| + gnl|WUGSC|RP23-291E18 + gi|19697559
Sequence length: 18561
Sequence map:
 Segment: pos=0, length=1082, type=DATA
 Segment: pos=1082, length=100, type=GAP
 Segment: pos=1182, length=1086, type=DATA
 Segment: pos=2268, length=100, type=GAP
 Segment: pos=2368, length=1096, type=DATA
 Segment: pos=3464, length=100, type=GAP
 Segment: pos=3564, length=1462, type=DATA
 Segment: pos=5026, length=100, type=GAP
 Segment: pos=5126, length=1217, type=DATA
 Segment: pos=6343, length=100, type=GAP
 Segment: pos=6443, length=1450, type=DATA
 Segment: pos=7893, length=100, type=GAP
 Segment: pos=7993, length=1086, type=DATA
 Segment: pos=9079, length=100, type=GAP
 Segment: pos=9179, length=1127, type=DATA
 Segment: pos=10306, length=100, type=GAP
 Segment: pos=10406, length=1145, type=DATA
 Segment: pos=11551, length=100, type=GAP
 Segment: pos=11651, length=1257, type=DATA
 Segment: pos=12908, length=100, type=GAP
 Segment: pos=13008, length=1024, type=DATA
 Segment: pos=14032, length=100, type=GAP
 Segment: pos=14132, length=1600, type=DATA
 Segment: pos=15732, length=100, type=GAP
 Segment: pos=15832, length=2729, type=DATA
Total: 43.9253%
No coding regions found

ID: sp|Q08345|DDR1_HUMAN + gi|729008
Epithelial discoidin domain receptor 1 precursor (Tyrosine kinase DDR)
(Discoidin receptor tyrosine kinase) (Tyrosine-protein kinase CAK)
(Cell adhesion kinase) (TRK E) (Protein-tyrosine kinase RTK 6)
(CD167a antigen) (HGK2).
Sequence length: 913
Sequence map:
 Segment: pos=0, length=913, type=DATA
Not a DNA
```

Common problems

How to construct Seq_id by accession?

CSeq_id class has constructor, accepting a string, which may contain a Bioseq accession, or accession and version separated with dot. If no version is provided, the Object Manager will try to find and fetch the latest one.

What is the format of data CSeqVector returns?

GetSeqVector method of CBioseq_Handle has optional argument to select data coding. One of the possible values for this argument is CBioseq_Handle::eCoding_Iupac. It forces the resulting Seq-vector to convert data to printable characters - either Iupac-na or Iupac-aa, depending on the sequence type. Gaps in the sequence are coded with special character, which can be received using CSeqVector::GetGapChar, for nucleotides in Iupac coding it will be 'N' character. Note that when calculating the percentage of 'G' /'C' in a sequence you need to ignore gaps.

What to pay attention to when processing cd-regions?

When looking for cd-regions on a sequence, you get a set of features, which locations describe their position on the sequence. Please note, that these locations may, and do overlap, which makes calculating percentage of 'G'/'C' in the cd-regions much more difficult. To simplify this part of the task you can merge individual cd-region locations using CSeq_loc methods (do not forget to sort the Seq-locs for correct merging) and use the resulting Seq-loc to initialize a Seq-vector. To calculate percentage of 'G'/'C' for non-cdr parts of a sequence create a new Seq-loc with CSeq_loc::Subtract() method.

Table 1. Teaching Example: Sequence

Accession	Version	Gi	Definition
AJ438945	AJ438945.1	19584253	Homo sapiens SLC16A1 gene...

Table 2. Test Examples: Sequences

Accession	Version	Gi	Definition
J01066	J01066.1	156787	D.melanogaster alcohol dehydrogenase gene, complete cds
U01317	U01317.1	455025	Human beta globin region on chromosome 11.
AJ293577	AJ293577.1	14971422	Homo sapiens partial MOCS1 gene, exon 1 and joined CDS
AH01100	AH011004.1	19550966	Mus musculus light ear protein (le) gene, complete cds
NT_017168	NT_017168.8	18565551	Homo sapiens chromosome 7 working draft sequence segment
AF022257	AF022257.1	2415435	HIV-1 patient ACH0039, clone 3918C6 from The Netherlands...
AC116052	AC116052.1	19697559	Mus musculus chromosome UNK clone
Q08345	Q08345.1	729008	Epithelial discoidin domain receptor 1 precursor...