Object-oriented sequence analysis: SCL-a C++ class library

Wolfgang Vahrson, Klaus Hermann, Jürgen Kleffe and Burghardt Wittig

Abstract

SCL (Sequence Class Library) is a class library written in the C++ programming language. Designed using object-oriented programming principles, SCL consists of classes of objects performing tasks typically needed for analyzing DNA or protein sequences. Among them are very flexible sequence classes, classes accessing databases in various formats, classes managing collections of sequences, as well as classes performing higher-level tasks like calculating a pairwise sequence alignment. SCL also includes classes that provide general programming support, like a dynamically growing array, sets, matrices, strings, classes performing file input/output, and utilities for error handling. By providing these components, SCL fosters an explorative programming style: experimenting with algorithms and alternative implementations is encouraged rather than punished. A description of SCL’s overall structure as well as an overview of its classes is given. Important aspects of the work with SCL are discussed in the context of a sample program.

Introduction

Commercially available software packages for sequence analysis are in general inadequate for research concerned with new algorithms or for dealing with large amounts of data. This kind of research instead depends on genuine and inventive programming. A good deal of the work involved is spent with designing, implementing, testing and bug fixing of the software needed. In earlier work (Kleffe et al., 1995) we pointed out that many routine tasks can be achieved by employing a library of routines specialized in sequence analysis, called DNAStat. DNAStat facilitates technical matters like memory management and error handling, allowing the researcher to concentrate on the subject matter.

Such a library, however, cannot address the problem of software design, i.e. how a problem translates from the problem domain (sequence analysis) into the vocabulary of the solution domain (a software program). Clearly, if the solution domain provided a vocabulary reflecting the problem domain accurately, the design process would be greatly simplified. This is exactly the goal of object-oriented programming and the related disciplines of object-oriented analysis and object-oriented design.

Object-oriented programming

Object-oriented programming aims at identifying typical entities (objects) in the problem domain and classifying them according to common properties. Examples of such classes in the field of sequence analysis are sequences, databases, vectors, but also insubstantial things like a sequence alignment algorithm. It is important to notice the difference between classes and objects: an object is always a particular thing, e.g. the DNA sequence of HIV, while a class is an abstract general description, say all DNA sequences.

The relationships between classes can be expressed as 'IS A KIND OF' and 'USES' relationships. A DNA sequence 'IS A KIND OF sequence of residues. 'IS A KIND OF' relationships are modeled by inheritance. The base class provides the common aspects, while classes derived from it add special properties and behavior. An obvious example is deriving both DNA and protein sequences from a common base class we call the sequence. The sequence class captures the basic notion of an ordered collection of residues, without regard to what residues are, while the derived class DNA sequence might add the notion that residues are in fact nucleotides.

In 'USES' relationships a client object requests an action from the object it uses. For example, a sequence alignment USES a pair of sequences; it might query them for their lengths in order to allocate memory and then ask for the residues at given positions of the sequences.

The description of a class splits into two parts: a public part, the interface of the class, and a private part, the implementation. The interface declares what operations a class is capable of. It reveals everything needed to use objects of this class, without exposing the internal structure of the class or the particular way an operation is implemented. For example, a sequence class might declare an operation Length() returning its length. A client using it does not have to know how it is implemented, but can rely on the fact that it will get a meaningful value for the length of the sequence.
Hiding the details of implementation is called encapsulation. It is one of the most important aspects of object-oriented programming: it makes modifying implementations easy, because if the implementation is changed only one class is affected, but not its clients. Furthermore, it makes thinking about classes and their relationships easier. One can think about classes without paying attention to their actual representation and implied technical details.

For further reading on object-oriented programming we recommend some of the following excellent introductory books: Cox (1986); Meyer (1988); Booch (1994) and Gamma et al. (1994).

While this paper was under review, an in-depth discussion of applying object-oriented programming to biology was published by Wiechert et al. (1995).

Class libraries

A collection of conceptually related classes is called a class library. It is superior to libraries of conventional programming languages by not only supplying isolated routines to manipulate data but in providing a model for the problem domain (Stroustrup, 1994). Examples of class libraries for general-purpose programming are NIHCL (Gorlen et al., 1990), lib++ (Lea, 1993), LEDA (Nher, 1990) and STL (Stein and Lee, 1994). In the field of biology, Chang et al. (1994) described PDBlib, a class library that models three-dimensional structures of macromolecules. Closest to our work is the molbio++ class library by Keith Robison (unpublished; available from ftp://golgi.harvard.edu/pub/CONTRIBUTIONS/molbio++) that gives basic support for sequence analysis.

Choosing C++

We used C++ for the implementation of SCL for two reasons. First, C++ is rapidly becoming the de facto standard for commercial programming. In consequence it is available on most platforms. For Unix users the excellent gcc compiler of the Free Software Foundation is available free of charge. Secondly, C++ spans a wide range between high-level and low-level programming, allowing the performance of a program to be tuned if necessary. Many scientific applications are very demanding for runtime efficiency that cannot be achieved in higher-level object oriented programming languages such as Smalltalk.

Literature on C++ is abundant; see, for instance, Lippman (1991), Stroustrup (1991), Cargill (1992), Coplien (1992), Stroustrup (1994) and references therein.

System and methods

Implementation and portability

SCL is designed to be portable across different platforms and operating systems. To achieve this, SCL uses external libraries sparingly; in fact only functions of the ANSI-C standard libraries are required (ctype.h, math.h, stdarg.h, stdio.h, stdlib.h, string.h, time.h, types.h). It does not use any C++-specific libraries (like iostreams). In principle SCL should be portable to any system fulfilling the following requirements:

- continuous memory model supported;
- long filenames supported;
- gcc compiler available.

SCL has been compiled and tested on the following hardware/operating system/compiler combinations:

- Apple Macintosh (both 68k and PPC)/MacOS 7.5/Symantec C++ 8.0;
- Silicon Graphics Indigo/IRIX 5.2/Free Software Foundation gcc 2.6.3;
- AXIL (Sun4m compatible)/SunOS 4.1.3/Free Software Foundation gcc 2.6.3.

Availability

The full source code, sample programs and documentation are available via anonymous ftp from ftp://molbiol-grumed.fu-berlin.de/pub/scl. A World Wide Web site (http://www.grumed.fu-berlin.de/scl) has been set up to provide online documentation and tutorial programs as well as more recent release information.

Description of classes

Overall design

SCL's classes do not descend from a common root class; they form a forest rather than a tree of classes. While the classes are conceptually related because they deal with sequence analysis, we tried to keep dependencies on the level of implementation to a necessary minimum. This design approach, known as decoupling, makes it possible to test and use individual parts of the class library without regard to the rest.

SCL has been written with the pure Unix philosophy of program execution in mind, i.e. a SCL program takes a number of arguments on the command line, reads from standard input or from files, and writes results to standard output or to files. It has no concept of graphical output or of a graphical user interface. This is in order to avoid portability problems between different platforms, as window systems differ considerably. On the other hand, if graphics is desired, SCL can be combined with class libraries providing for it.
Common features of all classes

All classes implement the minimal set of methods known as orthodox canonical form (Coplien, 1992) including the default constructor, the copy constructor, the assignment operator, and the destructor. Additionally SCL features a basic persistence mechanism: each object has `Dump()` and `Load()` methods, which write the object to, or read it from a file on disk, respectively. Note that `Dump()` and `Load()` are restricted to objects that do not store references to themselves, e.g. a circular list of objects cannot be stored.

Naming conventions

The names of all classes start with a capital ‘S’ in order to avoid name clashes with other class libraries. The prefix ‘A’ as in `SAIndexedCollection` denotes an abstract base class (a class that itself is not instantiated). The prefix ‘T’ is used to distinguish template classes (see below) from ordinary classes (`STSequence` versus `SSequence`).

Layered design

SCL’s classes can be split into three categories: (i) foundation classes, (ii) general biological classes and (iii) application-specific classes. The foundation classes provide general data structures, such as matrices, sets, strings, etc. General biological classes use foundation classes to implement concepts of sequence analysis, such as sequences, databases, sets of residues, etc. Application-specific classes use general biological classes and foundation classes to implement higher-level behavior, e.g. sequence alignment.

In the following we give an overview of the categories and of individual classes.

Foundation classes

In this category, classes are found that provide general data structures and utilities at a low level of specialization (Table I). Among them are classes managing collections of items like dynamically growing arrays (`SDynStore`, `SHeap`), associative arrays (`SALookUpTable` and derived classes), one-dimensional and two-dimensional arrays (`SVector`, `SMatrix`), and sets (`SIntSet`). Most of these are implemented as class templates, so that elements of any type (integers, floating-point numbers, user defined types) can be stored. Other classes provide access to files on disk (`SFile`, `STextFile`), parsing and validating command line arguments (`SArg`), time stamping (`SProtocol`), or access to commonly used data (`SResourceDB`, `SResourceFile`); and there is of course the inevitable string class (`SString`).

The most widely used class throughout SCL is `SDynStore`. It is SCL’s workhorse for storing all kinds of elements. It performs dynamic allocation of memory in a manner completely transparent to the user, growing and shrinking as needed; it allows its elements to be accessed

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SAIndexedCollection</code></td>
<td>abstract collection accessible by index</td>
</tr>
<tr>
<td><code>SDynStore &lt; element &gt;</code></td>
<td>dynamic storage of elements; all elements must be of same size</td>
</tr>
<tr>
<td><code>SHeap &lt; element &gt;</code></td>
<td>dynamic storage of elements; elements may differ in size</td>
</tr>
<tr>
<td><code>SStringHeap</code></td>
<td>dynamic storage of C-style character strings</td>
</tr>
<tr>
<td><code>SBitArray</code></td>
<td>space-efficient storage of integers in the range of 0...1</td>
</tr>
<tr>
<td><code>S2BitArray</code></td>
<td>space-efficient storage of integers in the range of 0...3</td>
</tr>
<tr>
<td><code>SALookUpTable &lt; key,element &gt;</code></td>
<td>store elements in a table; make them accessible by key</td>
</tr>
<tr>
<td><code>SFixedLUT &lt; key,element &gt;</code></td>
<td>... all elements must be of same size</td>
</tr>
<tr>
<td><code>SCharTable</code></td>
<td>elements and keys are characters</td>
</tr>
<tr>
<td><code>LetterCodeTable</code></td>
<td>used by <code>SAminoAcid3LC</code></td>
</tr>
<tr>
<td><code>SDynLut &lt; key,element &gt;</code></td>
<td>... elements may differ in size</td>
</tr>
<tr>
<td><code>SDynStoreFinder &lt; element &gt;</code></td>
<td>find an element in a <code>SDynStore &lt; element &gt;</code></td>
</tr>
<tr>
<td><code>SFile</code></td>
<td>file handling (wrapper of stdio)</td>
</tr>
<tr>
<td><code>STextFile</code></td>
<td>automatic recognition of Unix, Macintosh and MS-DOS text format</td>
</tr>
<tr>
<td><code>SStdIn</code></td>
<td>standard input stream</td>
</tr>
<tr>
<td><code>SStdOut</code></td>
<td>standard output stream</td>
</tr>
<tr>
<td><code>SIntSet</code></td>
<td>set whose members can be represented as integers</td>
</tr>
<tr>
<td><code>SNumSet</code>, <code>SNucSet</code>, etc.</td>
<td>various sets derived from <code>SIntSet</code> are predefined</td>
</tr>
<tr>
<td><code>SMatrix &lt; element &gt;</code></td>
<td>2-d array of elements</td>
</tr>
<tr>
<td><code>SVector &lt; element &gt;</code></td>
<td>1-d array of elements</td>
</tr>
<tr>
<td><code>SProtocol</code></td>
<td>notify user, time stamps, stop watch</td>
</tr>
<tr>
<td><code>SResourceDB</code></td>
<td>give access to files of general interest, e.g. PAM-matrices</td>
</tr>
<tr>
<td><code>SString</code></td>
<td>dynamic string of characters</td>
</tr>
</tbody>
</table>

Derived classes are marked by indentation with respect to their base class.
Fig. 1. Genealogy of the sequence classes. Sequences of any type are generated by expanding the class template STSequence<type>. SDNASequence and SProteinSequence are derived from SSequence.

by index like an array or in a list-like fashion, and it can sort its contents based on user-supplied ordering criteria. With the help of an associated helper class, SDynStoreFinder, elements in a SDynStore can be searched for in any way specified by the user.

The foundation classes implement many things one would wish a standard C++ library to do, and in fact our first thought was to use existing libraries. However, we had to learn that -at least at that time- there was no common library among all our target systems, forcing us to roll our own. The foundation classes should not be seen as competing with more general libraries like libg++ or STL; they have been tailored for use with SCL in the first place.

General biological classes

The classes of the second category, general biological classes, provide concepts commonly used in programs dealing with biological sequence data, which is the reason why SCL was written in the first place (Table II). The most prominent member of this category is naturally the sequence class, STSequence. It conveys the notion of a collection of elements in sequential order. It allows one to append, insert, edit and delete individual elements, growing dynamically to hold any sequence that fits into RAM. The order of sequence elements can be reversed or

Table II. General biological and application-specific classes

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAlignment</td>
<td>pairwise alignment of two DNA or protein sequences</td>
</tr>
<tr>
<td>SDescriptiveStatistics</td>
<td>statistics of values in a collection class</td>
</tr>
<tr>
<td>STSequence&lt;element&gt;</td>
<td>a sequence of elements</td>
</tr>
<tr>
<td>SDNASequence</td>
<td>a sequence of nucleotides</td>
</tr>
<tr>
<td>SProteinSequence</td>
<td>a sequence of amino acids</td>
</tr>
<tr>
<td>STSlidingWindow&lt;element&gt;</td>
<td>iterator for a sequence of elements</td>
</tr>
<tr>
<td>SSlidingWindow&lt;char&gt;</td>
<td>iterator for a sequence of characters</td>
</tr>
<tr>
<td>SADB</td>
<td>abstract database class</td>
</tr>
<tr>
<td>STaggedDB</td>
<td>read and write in a database file in a tagged line format</td>
</tr>
<tr>
<td>SEMBLDB</td>
<td>… adapted for EMBL-style</td>
</tr>
<tr>
<td>SGenBankDB</td>
<td>each line is a record, fields are separated by white space</td>
</tr>
<tr>
<td>SLINEDB</td>
<td>abstract numbered collection of sequences</td>
</tr>
<tr>
<td>SASetOfSequences</td>
<td>sequences reside in an EMBL-style database</td>
</tr>
<tr>
<td>SSoSEMBL</td>
<td>sequences reside in a GenBank-style database</td>
</tr>
<tr>
<td>SSoSGenBank</td>
<td>… sequences reside in individual files</td>
</tr>
<tr>
<td>SSoSFiles</td>
<td>… sequences reside in RAM</td>
</tr>
<tr>
<td>SSoSRAM</td>
<td>map amino acids from one-letter code to three-letter code</td>
</tr>
<tr>
<td>SAminoAcid3LC</td>
<td>transform sequences into other alphabets (hash)</td>
</tr>
<tr>
<td>SSequenceEncoder</td>
<td>input and output for sequences of characters</td>
</tr>
<tr>
<td>SSequenceIO</td>
<td>input and output specific for nucleic acids sequences</td>
</tr>
<tr>
<td>SDNASEquenceIO</td>
<td>input and output specific for protein sequences</td>
</tr>
<tr>
<td>SProteinSequenceIO</td>
<td>input and output specific for alignments; see SAlignmentSequenceIO</td>
</tr>
<tr>
<td>SAlignmentSequenceIO</td>
<td>translate DNA into Protein sequences, and vice versa</td>
</tr>
</tbody>
</table>

Derived classes are marked by indentation with respect to their base class.

122
randomly scrambled. Single elements or subsequences can be extracted. It is noteworthy that by virtue of the C++ template mechanism, elements need not be characters. In fact, an element of a sequence can be of any type, like an integer or a floating-point number or a user-defined class. This is a very powerful feature for analyses that examine sequences using higher-order alphabets (e.g. hexamers instead of single residues) where elements cannot be represented as characters. For convenience, the very frequently used class SSequence is declared as a SSequence(char), a sequence of characters. Derived from it are SDNASequence and SProteinSequence. SDNASequence adds behavior for handling circular sequences like plasmids and for forming the reverse-complement of a sequence. SProteinSequence does not add special behavior at the moment but exists for future extensions. Figure 1 illustrates the genealogy of sequence classes.

Grouped around the SSequence class are classes cooperating with it. Figure 2 gives an overview of the interactions between these classes. SSequenceIO performs input and output operations on sequence objects. It takes care of alphabet checking when reading sequences and prints formatted human-readable output with a number of formatting options. It can read from and write to both files on disk and character arrays in memory. Separating SSequenceIO from the SSequence class itself reduces the overhead of using a SSequence object.

The SSLidingWindow class implements a behavior that is needed in many algorithms dealing with sequences: it splits a sequence into consecutive and possibly overlapping fixed-size subsequences. The picture we had in mind when naming the class is that of a window sliding over the sequence, bringing each time another part of the sequence into view. In the terminology of object-oriented programming such a class is called an 'iterator'. Having a separate iterator class is advantageous for two reasons: the iterator is created only when needed, hence reducing overhead, and independent iterators can be created operating on the same sequence.

The SSequenceEncoder class transforms sequences into other alphabets. For example, a very common transformation is converting the character representation of nucleotides (a, c, g, t) of a DNA sequence into integers (0, 1, 2, 3).

A very useful family of classes, the SoS-classes, is derived from the abstract base class SASetOfSequences. SoS-classes embody the concept of a collection of sequences, each of which can be accessed sequentially or by index. The key point is that SoS-classes completely hide from clients the way in which sequences are stored physically, i.e. whether they exist as individual files or as a single database file or as an array of sequences in RAM. The benefit of this polymorphism is that it is very easy to adapt a program for a different kind of input, simply by changing the SoS-class used.

Another group of classes deals with reading and writing databases. The abstract base class SADB provides the model for databases in general: a database is associated with a file, it is divided into records, and records can be written and read. The actual implementation of this model is deferred to derived classes. STaggedDB is derived from SADB and implements a special flavor of database. It reads and writes databases with a tagged line format. This format consists of lines of text carrying a tag at the beginning describing the line's contents and is very popular for flat file distribution of biological databases. STaggedDB is quite flexible on what exactly is considered a tag, it can cope with fixed-size as well as with variable-length tags. It also provides basic indexing facilities so that individual entries in the database can be accessed directly.

SEMBLDB and SGenBankDB are classes derived from STaggedDB and add functionality specific for the EMBL and the GenBank database format. They also allow direct access to an entry's sequence or a portion of it.

A different kind of database is implemented by SLineDB, a class also derived from SADB. It writes and reads databases in which a record consists of a single line of text and where fields are separated by spaces or tabs. This simple format is very handy for constructing databases on the fly. As an intended side-effect it is identical to the format used by the standard Unix tool awk (Aho et al., 1988).

Application specific classes

The third category finally covers application-specific classes used to implement a particular algorithm or behavior. Application-specific classes can be regarded as being on a higher level of specialization than general biological classes, though the distinction is somewhat arbitrary. Examples are the SAlignment class, which calculates the alignment between two sequences using adaptations of the Smith-Waterman and the Needleman-Wunsch algorithms as presented by Pearson and Miller (1992); the SDescriptiveStatistics class, which gathers simple statistics about collections (mean, median, variance, standard deviation, etc.); and the STranslation class, which translates nucleic acids sequences into protein sequences and vice versa (Table II).

Implementation

Use of C++ language features

C++ provides many very advanced language elements such as multiple inheritance, operator overloading, exception handling and template classes (parameterized types). While these features are powerful when used
appropriately, they also pose the danger of cramming a program beyond comprehension when used excessively or inconsiderately. We tried to keep SCL as simple as possible by sticking to the policy outlined below:

- Multiple inheritance is not used.
- Operator overloading is used sparingly. In general we used it only when the semantics of the overloaded operator are well established. Overloading the \( += \) operator in the SString class meaning concatenation or overloading the \( (\ ) \) operator to access elements in the SMatrix class are obvious examples. We refrained from seemingly clever overloading, such as overloading the \(+ +\) operator to mean 'next record' in the database classes or the like.
- Exception handling as described by Stroustrup (1991) is not used—mainly because there were no C++ compilers available implementing it. See below for how SCL handles errors.
- Template classes (parameterized types) are used a lot in SCL and are central to its flexibility. Template classes are classes parameterized with respect to some of their properties. For example, the class SDynStore is parameterized regarding the type of element it stores. At compile time, when the parameters are supplied, this general template class expands to the concrete class. For example, a SDynStore<char> is a concrete expansion of SDynStore, storing characters.

Error handling

SCL features a simple yet flexible error-handling scheme. It consists of a single C-preprocessor macro called \texttt{Test(condition)}. If \texttt{condition} evaluates to zero, \texttt{Test(\ )} calls the global function \_SCLErrorHandler( ). By default the error handler prints out the filename and line number of the source code where the error occurred and then terminates the program. This behavior is modeled after the ANSI \texttt{C} \texttt{assert(\ )} macro. It can be altered by supplying another error handler at any point of the program. \texttt{Test(\ )} is used continuously throughout the class library. Though it is simple, it has proved to be an invaluable tool for detecting runtime errors that would have gone by unnoticed and otherwise perpetuated. It adds only very little overhead to a program. If processing time is at a premium, however, the \texttt{Test(\ )} mechanism can be switched off either for a compilation unit (source code file) or as a whole.

Discussion

Sample program

We present a simple but non-trivial program named \textit{OligoCount} that illustrates some of SCL's features. The discussion assumes a reading understanding of C++.

OligoCount's purpose is to read through an EMBL-style database, count the occurrences of all oligomers of a given length and report their frequencies. OligoCount is invoked from the command line with the length of oligomers to be examined and the name of the database file, e.g.

\texttt{OligoCount 2 pri.dat.}

The following is a step-by-step discussion of OligoCount (Listing 1). The numbers correspond to the numbering in the listing:

In the main( ) function of the program (code not shown), SCL is initialized, the command line arguments are parsed and function \texttt{AnalyzeDatabase()} is called with the arguments \texttt{filename} and \texttt{len}.

\texttt{(1) \texttt{AnalyzeDatabase(\ )}}

In \texttt{AnalyzeDatabase()} several objects are first created: \texttt{db} is the SEMBLDB object that reads through the database. \texttt{dna} is a SDNASequence object that holds the sequence of the current database entry. The SSequenceEncoder object \texttt{encoder} transforms each oligonucleotide into an integer number in the range \(0..4^{\text{length}}-1\). For instance, if \texttt{length = 2}, the codes will be in the range of \(0..15\).

\texttt{trf} is a sequence whose elements are 32-bit integers; it holds the encoded sequence. If the original oligomer contains undetermined bases (like 'N'), the corresponding element in \texttt{trf} is set to a special value, \texttt{knInvalidSeqWordCode}, and is said to be invalid.

The SVector object \texttt{frq} is a vector of integers; each of its elements records the frequency of an oligonucleotide. It is constructed to be big enough to hold elements for all possible codes as well as an additional one for invalid codes.

\texttt{(2) \texttt{The database file is opened; the Test(\ ) statement checks if this operation was successful and exits with an error message if not (in a more elaborate version we could print a more detailed description of the problem).}}

In the do loop, each entry in the database is visited in turn by successive calls to \texttt{db.NextEntry(\ )}. Note how \texttt{db} hides the details of file handling; except for method \texttt{Open()} taking a filename as argument, we are not bothered with files when using \texttt{db}. Even more important, all knowledge of the EMBL file format is encapsulated in \texttt{db} as well; we do not have to fiddle with it here. If we want to adapt the program for a different database format, we only have to plug in an object of the appropriate database class.

For each entry in the database the contents of both \texttt{dna} and \texttt{trf} are cleared by calling the method \texttt{Purge(\ )}. The
Listing 1. The OligoCount program. A detailed discussion is found in the text.

```
// (1)
void AnalyzeDatabase( const char *filename, int len)  
    // in an EMBL-style database specified by filename,
    // iterate over all entries,
    // determine the frequencies of oligomers of length len
    // and report them
{
    SEMBLDB db;
    SDNASequence dna;
    SSequenceEncoder (SSequenceEncoder::kACGT, len);
    STSequence < SeqWordCode > trf;
    SVector < int > frq( encoder.GetNumSeqWords ( ) +1);

    // (2)
    Test( db.Open( filename ));
    do {
        dna.Purge( );
        trf.Purge( );
        db.ReadSequence( dna);
        encoder.Encode( dna, trf);
        DetermineFrequencies( trf, frq);
    } while (db.NextEntry( ));
    ReportFrequencies( frq, encoder );
}

// (3)
void DetermineFrequencies ( STSequence < SeqWordCode > &trf, SVector < int > &frq)  
    // count the occurrences of numbers in trf and store them in frq
{
    for ( SeqPosition i = 1; i <= trf.Length( ); i ++) {
        SeqWordCode code= trf.Get (i);
        if (code != SSequenceEncoder::kInvalidSeqWordCode) 
            frq[code]++;
        else 
            frq[ frq.Howmany( )-1 ]++;
    }
}

// (4)
void ReportFrequencies( SVector < int > &frq, const SSequenceEncoder & encoder)  
    // report the frequencies of oligomers in the database
{
    SString oligo;
    for (int code= 0; code < frq.Howmany( )-1; code++) {
        encoder.Decode( oligo, code);
        printf( "%s %d \n", (const char*) oligo, frq[code] );
    }
    printf( "other %d \n", frq[ frq.Howmany( )-1 ] );
```

DNA sequence is read from the database and deposited in dna. The size of the sequence read is of no concern to us; dna grows and shrinks automatically to accommodate it. dna is encoded by the encoder object, resulting in the transformed sequence trf. trf is passed to the function DetermineFrequencies( ).

(3) DetermineFrequencies( )

DetermineFrequencies() runs over the whole transformed sequence trf, reading each of its elements and assigning it to code. If code is in the range of valid codes it is taken as an index into the frq object and the respective element is incremented. All invalid codes are summed up in the last element of frq. Note how operator [ ] is overloaded, giving access to the elements in frq in a manner identical to the way plain C++ arrays are accessed. In contrast to those, however, frq catches attempts to access elements out of the array's bounds.

Finally, when all database entries have been examined, the result is reported in the function ReportFrequencies( ).

(4) ReportFrequencies( )

The for loop iterates over all but the last elements in frq. The code is decoded into the original oligomer (recall that the code of the respective oligomer had been taken as index into frq) and stored as the SString object oligo. Note that oligo handles strings of any size; we do not have to pass the original length of the oligomers to ReportFrequencies( ).

oligo and its corresponding frequency are printed out using the ANSI-C function printf( ). The last element of frq holds the frequency of oligomers that contain undetermined bases and is printed out last.

Table III shows the results of running OligoCount on the primate division of the EMBL database, release 41 (~32 600 entries, 34 000 000 nucleotides). The calculation took ~5 min on a SGI Indigo R4400/150 MHz. To save space, only the frequency of dinucleotides is shown; results for longer oligomers are analogous.

It should be noted that the implementation of OligoCount as presented here has been chosen for brevity and clarity, not for optimal performance. It is not necessary, for instance, to read in and transform the DNA sequence as a whole; it would rather suffice to process only one oligomer at a time.

It took only ~1 h to implement the program from the initial idea to the correctly running program. More complex sample programs are distributed together with SCL or can be accessed via the WWW site mentioned above.

Comparison of SCL and molbio++

The molbio++ class library by Keith Robison, while unpublished, is quite widely known. Its main strength compared to SCL is its maturity: it contains a lot of specialized classes and supports more file formats than SCL currently does. It also contains an interface for plotting values against sequences, while SCL does not deal with graphics.

The strength of SCL, on the other hand, is flexibility, achieved by the use of template classes and by the layered design of classes, where primitive general classes are used to implement more complex behavior. Additionally, it contains classes for concepts absent in molbio++, namely the family of database classes, the classes for sets of sequences and the foundation classes.

Conclusion

The benefits of using SCL are twofold. First, on a pure technical level, SCL takes many details off the programmers' shoulders, such as dynamic memory allocation, checking bounds of arrays, etc. While this is helpful in its own right, it could also be accomplished by conventional procedural programming and does not necessarily ask for a class library. Second, on the conceptual level SCL provides a solid framework for doing sequence analysis. It models the basic entities in this field and their relations and allows programs to be assembled quickly from prebuilt components. This enormously stimulates experimentation with programs and makes it easy to adapt them for new purposes. By inviting such an explorative programming style, SCL is well equipped to deal with the rapidly developing field of sequence analysis and interpretation.

However, the benefits of SCL come at the price of learning C++. C++ is a complex and still evolving language. Its powerful features are traded off by certain
subtle facets which make the language hard to master, especially for a beginner or casual programmer. Moreover, the way in which C++ is implemented by a given compiler may cause unexpected problems.

In its current version SCL is somewhat minimalist. It still lacks many desirable features, e.g. support of ASN.1 formatted sequence data, or sequence pattern matching. We think it is nevertheless useful because it can be easily extended, unlike conventional libraries, by deriving or composing new classes to meet particular needs. In future we will add application-specific classes according to our research interests. We would like to share SCL with other researchers and would greatly appreciate comments and contributions.

Acknowledgements

W.V. would like to thank Helge Horch and Wilhelm Vahrson for critically reading and discussing the manuscript. K.H. and J.K. are supported by Deutsche Forschungsgemeinschaft

References


Robison,K. (unpublished) molbio++ is available from ftp://golgi.harvard.edu/pub/CONTRIBUTIONS/molbio++


Received on September 9, 1995; revised and accepted on January 17, 1996