RCL: A C++ Library and Programming Discipline for Research

Stephen M. Omohundro and A. Peter Blicher

NEC Research Institute
4 Independence Way
Princeton, NJ 08540
USA
om@research.nj.nec.com
blicher@research.nj.nec.com

April 30, 1997

Abstract
This report describes the design of a C++ library and programming discipline for research projects that aims to be efficient, flexible, simple, and safe. It includes classes for fundamental data structures, LAPACK-based linear algebra, random numbers and probabilistic models, image processing and vision-related tasks, and geometric structures. The programming discipline is based on a simple approach to memory management that has a number of advantages over that used by other C++ libraries. Every object has a string form which can be used for data input and output, checkpointing, debugging, inter-program communication, and cross-architecture communication.
# Contents

1 Introduction ........................................ 5

2 An example program ................................. 7

3 The memory management discipline ................. 11
   3.1 Memory management in C .......................... 11
   3.2 Memory management in C++ ....................... 12
   3.3 The library memory management discipline ....... 12
      3.3.1 Avoiding copying with take ................ 14
      3.3.2 Side effects and return values .............. 15
   3.4 Inheritance .................................... 17

4 Recurring programming constructs .................. 18
   4.1 Amortized doubling ................................ 18
   4.2 Iterators ....................................... 18
   4.3 Assertion checking .............................. 22
   4.4 Object null state ................................ 23

5 Stylistic conventions ................................ 23
   5.1 Class layout .................................... 23
   5.2 Comments ....................................... 24
   5.3 Naming conventions .............................. 24
      5.3.1 Global constants ............................ 24
      5.3.2 Class names ................................ 24
   5.4 Routine names ................................... 27

6 Typical class features ............................... 27
   6.1 Default constructor .............................. 28
   6.2 Copy constructor ................................ 28
   6.3 Other constructors ............................... 28
   6.4 Assignment operator: = .......................... 29
   6.5 String form generator: write .................... 29
   6.6 String reader: read ............................. 30
   6.7 Storage transfer: take ........................... 30
   6.8 Null state: is_null and to_null ................ 31
   6.9 Default destructor .............................. 31
   6.10 What properly written classes contain ......... 31

7 The string class: Str ................................ 32
   7.1 Empty strings ................................... 33
   7.2 Outputting Str's on streams ..................... 33
   7.3 Appending objects to strings .................... 34
   7.4 Hashing strings .................................. 34
   7.5 String iterators: StrIt .......................... 34
A Appendix. Numerical linear algebra: Blas.h and Lapack.h 70
A.1 What are BLAS and LAPACK .......................... 70
A.2 Who needs Blas.h and Lapack.h ...................... 70
A.3 What is in BLAS and LAPACK ......................... 71
A.4 BLAS and LAPACK routines provided by RCL .......... 73
A.5 BLAS and LAPACK routine naming conventions ........ 74

B Appendix. Compiling, linking, testing, and environment is-
sues. 74
B.1 Using the right C/C++ runtime ...................... 74
B.2 What to #include .................................. 75
B.3 Testing RCL ....................................... 76
B.4 Using the postscript viewer GSview with RCL ........ 76
B.5 Using the gnuplot plotting package with RCL .......... 77
B.6 Viewing and scrolling console windows ................ 77
B.7 Spurious carriage returns (ctrl-M) ................. 78
1 Introduction

This report describes RCL (Research Component Library), a C++ class library and programming discipline for research in compute-intensive areas such as machine learning, machine vision, scientific computing, algorithm design, and computer graphics. RCL provides classes to implement and support:

- Fundamental data structures
- Linear algebra
- Random numbers and probabilistic models
- Image processing and vision
- Geometric structures
- User interface
- Operating system interface

RCL also uses, supports, and enforces a C++ programming discipline designed to avoid the pitfalls and shortcomings of the language, reduce complexity, help assure bug-free programs, and facilitate rapid coding. This programming discipline is described fully later.

We describe some aspects of the library design in detail, but the header files are the primary documentation and should be consulted before using a class. Here we describe the programming discipline, the main library functionality, and some of the class interfaces. This report has been written both for the benefit of other users of the library and to document the design features that we found to be effective.

Although we developed the library primarily to support our own research, its goals and functionality should make it applicable to a wide range of related research applications. The broad aim of the library is to support the development of software which is efficient, flexible, simple, and safe.

Efficiency is critical because research projects typically push the performance limits of available hardware. The outcome of one experiment can affect the choice of the next experiment so research software is naturally interactive. This means that execution time directly affects the pace of research.

Research software must be flexible because the final form of a project is usually not known when development begins. The results of an experiment can force a complete redesign of a system. Additionally, related experiments often have related software needs. Research libraries should therefore be based on well-encapsulated reusable components.

An effective programming discipline must be simple to learn and to use. A programmer who is burdened with complex, non-orthogonal rules and constraints is unlikely to write reusable code which others can develop and maintain.

Valid research results must be based on valid software. Because research software is executed rarely and yet is modified often and used in a variety
of contexts, safety and correctness checking is especially important. Making components reusable can improve their quality because they tend to be built and tested with more care and because they are used in a wider variety of contexts.

Despite the shortcomings of C++, many believe it to be the most effective programming language for current research projects. The desire for modularity, reuse, and correctness suggests the use of an object oriented language. Efficiency requirements rule out the current implementations of languages like Smalltalk, Self and CLOS. There are experimental languages which might be more suitable than C++. For example, many of the library design principles described here first arose in the development of the object-oriented language Sather (Omohundro, 1993; Schmidt and Omohundro, 1993; Szyperski et al., 1994; Murer et al., 1996). But Sather is itself a research project which is continuing to evolve and develop. For reasons of stability, compatibility, and wide availability of tools and compatible software, the library described here is implemented in C++ (Stroustrup, 1991).

C++ is a complex language with many traps for the unwary (Meyers, 1992; Meyers, 1996; Murray, 1993; Cline and Lomow, 1995). It does, however, support modularity, reuse, and encapsulation; and high performance implementations exist on many machines. This and its compatibility with C have caused it to enjoy a wide popularity. The library described here is based on a programming style which avoids many complex constructs and which aims to be very safe and efficient. Safe and simple subsets of C++ have sometimes been called “C+-”.

There are several commercial and public domain C++ libraries available. We have studied many of these including the NIH library NIHCL (Gorlen, 1987; Gorlen et al., 1990), the Free Software Foundation’s library libg++ (Lea, 1991), the Texas Instruments library COOL, the Saarbrücken library LEDA (Näher, 1993), the Standard Template Library STL (Musser and Saini, 1996), the proposed standard C++ libraries described in (Plauger, 1995) (though the standard has since changed significantly), and libraries included with various textbooks (Budd, 1991). There is a growing literature comparing aspects of these libraries (Kunz, 1991).

Many of these libraries have interesting ideas but none of them has all of the functionality required for research or provides a programming discipline satisfying our criteria. Many levy a significant performance overhead by their space utilization, excess heap allocation and excess implicit copying. Many use non-standard mechanisms that are not portable and may be dangerous when combined with inheritance (e.g., allocating array space off the end of a struct). Most have complicated memory management and exception handling schemes that make them difficult to integrate with other code. One exception is the STL library, which has been designed to be efficient and to work well with other libraries. The library described here can easily be used in conjunction with STL.

We begin with an example program which demonstrates several features of the library. We then discuss memory management because it is a critical aspect of the programming discipline. We describe the class and routine naming conventions and features for safety and testing. We discuss the library’s use
of array-based data structures with amortized doubling. Finally we describe the design of specific classes implementing strings, tables, vectors, and other important structures.

2 An example program

We begin with a complete program using the library, to give the reader a feel for what using the library is like. The program can be modified to implement similar experiments. It also serves as a concrete example to bear in mind and refer to while reading the sections which follow.

The sample program computes a principal components decomposition of the set of $4 \times 4$ local patches of a color image. It has a graphical user interface for reading the input and output image filenames, which may be dragged and dropped. When the Compute PCA of Image button is pressed, the program reads the image from the specified file, displays it in a window, chooses 100 $4 \times 4$ patches at random, and computes the mean and covariance of those patches regarded as vectors. It then converts the eigenvectors of the computed covariance matrix to "eigenimages," doing some normalization to allow the display of possibly negative values, and creates a mosaic image of the eigenimages. It displays the mosaic, and will save it to a file if the Write PCA Image to File button is pressed. If Plot eigenvalues is depressed it will plot a graph of the associated eigenvalues on the screen and create a PostScript file containing the plot which is also displayed in a PostScript viewer. If the Print eigenvectors button is pushed, it prints the component values of each of the eigenvectors to a console window.

II PcaExample.cpp

II Tue Jan 7 20:54:26 1997 pb

#include "iostreamRCL.h"
#include "Gui.h"
#include "Gnuplot.h"

// Global variables needed for inter-callback communication:
static Imgc img; // Input image.
static Imgc tileimg; // Holds the PCA output image.
static Vec evals; // Holds the eigenvalues.
static Mat eveCS; // The columns will be the eigenvectors.

Bool read_and_display_input_img(){
    // Read the name of a color image (bmp) file from text box 0.
    // Read in the image into 'img', and display it.
    // Return 'true' if successful, 'false' if failure.
    Str filename = Gui::get_text_value(0); // get text from textbox 0
    if (filename.is_null()){

cerr << "\nError: Input file name was empty.\n" "Type a name or drag a file to input file text box." << endl;
return false;}
try{img.read_from_file(filename);} catch(FileEx fe){ // Reading from file threw an exception.
cerr << "\nFile input error. File exception description:\n" << fe.err_str << endl;
return false;}
// Display the input image:
Canvas incan = Gui::new_canvas("Input Image"); // Make a window for it.
Gui::show_img(incan, img);
return true;}

void random_patches_in(Arr<Vec>& va){
  // Choose 100 random 4x4 subimages of the image, and make Vec versions:
  va.to_size(100); // Allocate an array of 100 null vecs.
  Imgc subimg; // Temporary to hold patches as we grab them.
  subimg.to_dim(4,4); // Make the temporary be 4x4.
  for (Int i=0; i<100; i++) { // Choose 100 random patches.
    Int xmin = Rnd::int_upto(img.width() - 5); // A random x in the img.
    Int ymin = Rnd::int_upto(img.height() - 5); // A random y in the img.
    subimg.to_subimage(xmin, xmin+3, ymin, ymin+3, img); // Get a 4x4 patch.
    subimg.vec_in(va[i]);}
}

void compute_eigenvalues(const Arr<Vec>& va, Vec& mn){
  // Compute the eigenvectors and eigenvalues:
  Mat cov; // Will hold the sample covariance.
  mean_covariance_in(va, mn, cov); // Compute mean and covariance.
  cov.eigenvalues_in(evals, evecs); // Compute eigenvalues and eigenvectors.
  // Reverse order so as to start at the largest eigenvalue:
  reverse(evals); // Reverse order of eigenvalues.
  evecs.reverse_col_order(); // Reverse order of eigenvectors.
}

void eigenimages_in(Arr<Imgc>& eimgs, const Vec& mn){
  // Convert the eigenvectors to images:
  eimgs.to_size(evals.dim()+1); // Leave room for mean image.
  eimgs[0].to_dim(4,4); // Make the 0th eigenimage be 4x4.
  eimgs[0].from_vec(mn); // Put the mean into img 0.
  for (Int j=1; j<eimgs.size(); j++) { // Make the jth eigenimage be 4x4.
    eimgs[j].to_dim(4,4); // Make the jth eigenimage be 4x4.
    Vec evec; // Temporary.
    evecs.col_in(j-1, evec); // Get an eigenvector from a column.
    eimgs[j].from_vec_renormalized(evec);}// Convert it to image for display.
}

void make_and_show_tiling(const Arr<Imgc>& eimgs){


8
// Create a mosaic image of the eigenimages:
tileimg.to_tiling(eimgs); // Tile 'tileimg' with the eigenimages.
Canvas outcan = Gui::new_canvas("Output Image"); // Create a display.
Gui::show_img(outcan, tileimg); // Display the eigenimage tiling.
cout << "\nDrag image corners to zoom or shrink displayed image." << endl;

void compute_pca() {
  // Read in a color image from the file the user has entered or dropped
  // and dropped into text box 0. Compute the principal components of
  // 100 random 4x4 blocks of the image and put a mosaic image of them
  // in 'tileimg'. The first tile is the mean.
  if (!read_and_display_input_img()) // Get and show input image.
    return; // Returns to wait for more gui user input.
  // Choose 100 random 4x4 subimages of the image, and make Vec versions:
  Arr<Vec> va;
  random_patches_in(va);
  static Vec mn; // Will hold the sample mean.
  compute_eigenvalues(va, mn); // Compute the eigenvvecs, eigenvals, and mean.
  // Convert the eigenvectors to images:
  Arr<Imgc> eimgs;
  eigenimages_in(eimgs, mn);
  // Create and display a mosaic image of the eigenimages:
  make_and_show_tiling(eimgs);
}

void write_pca() {
  // Save the PCA image to the file named in text box 1.
  if (tileimg.is_null()){
    cerr << "\nError: No PCA image to write to file.\n"
    "Maybe you haven't computed the PCA yet?" << endl;
    return;
  }
  Str filename = Gui::get_text_value(l); // Get text from text box 1.
  if (filename.is_null()){
    cerr << "\nError: Output file name was empty.\n"
    "Type a name or drag a file to 'output file' text box." << endl;
    return;
  }
  try{tileimg.write_to_file(filename),}
  catch(FileEx fe){
    cerr << "\nFile output error. File exception description:\n" << fe.err_str << endl;
    return;}

void plot_evals() {
  // Plot the eigenvalues on the screen and into a postscript file.
  if (evals.is_null()){
    cerr << "\nError: No eigenvalues to plot.\n"
    "Maybe you haven't computed the PCA yet?" << endl;
  }
return;
Easyplot::plot(evals);} // Plot eigenvalues on screen and postscript.

void prt_evecs(){ // Print the eigenvectors on stdout.
    if (evecs.is_null()){
        cerr << "\nError: No eigenvectors to print. \n"
            "Maybe you haven't computed the PCA yet?" << endl;
        return;
    }
    Vec evec; // Temporary.
cout << "\nEigenvectors: " << endl;
for (Int j=0; j<evecs.ncols(); j++){
evecs.col_in(j, evec); // Get an eigenvector from a column.
    cout << "\nEigenvector " << j << ": \n"
        << str(evec).pretty() << endl;}

void main() {
    // Put up the user interface and register the callback functions.
cout << "PcaExample: " << endl; // Print to the console window.
Gui::create_dialog(4, 0, 2); // 4 buttons, 0 sliders, 2 textboxes.

    // Label the text boxes:
    Gui::set_text_label(0, "Input Image .bmp file:");
    Gui::set_text_label(1, "Output Image .bmp file:");

    // Label the buttons:
    Gui::set_button_label(0, "Compute PCA of Image");
    Gui::set_button_label(1, "Write PCA Image to File");
    Gui::set_button_label(2, "Plot eigenvalues");
    Gui::set_button_label(3, "Print eigenvectors");

    // Register callback functions for the buttons:
    Gui::set_button_fun(0, compute_pca);
    Gui::set_button_fun(1, write_pca);
    Gui::set_button_fun(2, plot_evals);
    Gui::set_button_fun(3, prt_evecs);
    Gui::win_loop(); // Start the Windows message loop.}

This sample program, PcaExample.cpp, is distributed as part of the RCL distribution. It can be found in RCL\progs\PcaExample. A Microsoft Visual C++ project to build and execute this program is also distributed with RCL in RCL\projects\RCL_projects. For more detailed instructions on how to build and run the demo, see the separate technical note “How to compile and link using the library,” distributed with the library in RCL\doc\howto.
3 The memory management discipline

By the memory management discipline we mean the rules which a programmer uses to allocate and deallocate memory for data structures. Memory management can have a tremendous effect on performance. Unfortunately, it is also a very common source of errors.

The simplest form of memory management, from the programmer’s perspective, is automatic garbage collection. Languages like Lisp, Scheme, Smalltalk, Sather and Java automatically reclaim memory which is no longer accessible. Research into garbage collection over the past 10 years (e.g., tenuring collectors) has yielded improved performance, and advocates of these languages claim that garbage collection now uses only a small percentage of the compute time. One must be aware of the two possible reasons for this type of result, however: garbage collection may indeed be insignificant or the rest of the computation may be so inefficient that it swamps the real cost of garbage collection. Mark and sweep collectors must periodically access all of the allocated memory and so can disrupt the paging and caching behavior of programs which maintain large data structures. Languages with explicit memory management, like C++, can potentially provide better performance (though this point is controversial). Even if automatic garbage collection were definitively shown to be the best approach, it is not supported by C++ (though there have been several attempts to introduce it).

Memory is always allocated in one of three regions in C and C++ programs. Static memory is a fixed space for string constants and static variables that is allocated once and for all when a program begins executing. Static memory neither grows nor shrinks. Stack memory holds the local variables, arguments, and return values of routines and grows and shrinks as routines are called and return. Heap memory is explicitly allocated and deallocated by calls to new and delete (or malloc and free in C) and is the source of most problems.

Improper memory management can lead to two common bugs. Memory leaks occur when memory is allocated but is never deallocated. Dangling pointers are references to memory which has already been deallocated and usually lead to illegal memory references, often in other code than where the bug is.

3.1 Memory management in C

In C, a programmer allocates heap space with a call to malloc and must explicitly deallocate it with a call to free. A routine which allocates an integer on the heap might look like:

```c
rout() {
  int* pi;
  pi=malloc(sizeof(int));
  *pi=foo();
  free(pi);
}
```
If the programmer had forgotten to free pi, then it could never be deallocated after the routine returned and so would be a memory leak.

3.2 Memory management in C++

Because it is essentially a superset of C, C++ can compile and execute the above code. C++ style, however, replaces malloc by new, and free by delete. Thus one would rewrite the above code as:

```cpp
rout() {
    int* pi=new int;
    *pi=foo();
    delete pi;
}
```

Until recently this was perfectly acceptable code. If, however, the function foo() throws an exception, the call to delete pi; will never be reached and executed. C++ does guarantee to call the destructors of all objects in the stack frames that an exception jumps over, but C pointers, such as pi, don’t have destructors. You might think a program is safe if it never explicitly throws its own exceptions. That is not so, because the C++ environment can now throw its own exceptions. For example, if new is called and cannot allocate the needed memory, it now throws an exception. Thus it is no longer safe to use ordinary C pointers on the stack to point to newly created memory (Meyers, 1996).

3.3 The library memory management discipline

Our memory management discipline is quite simply this: space allocated on the heap is owned by one and only one object, and that object alone may maintain a pointer to that space (except for short-term purposes). The destructor for the object is responsible for deallocating the heap space via delete. Except for short-lived objects like iterators (cf. section 4.2), objects must not maintain pointers to heap memory owned by another object, and functions must not pass or return such pointers.

C++ supports this style with features that help to prevent memory leaks and dangling pointers. Declaring a local variable with a class type, e.g.,

```cpp
Foo bar;
```

causes the class’s constructor routine to be called when the variable enters the scope and the class’s destructor routine to be called when it leaves the scope. In our style, any heap space used by the object will be allocated in the constructor and deleted in the destructor.
An unfortunate consequence of these automatic constructor/destructor calls is that when C++ passes objects as arguments, returns them from functions, or assigns them to locations in data structures, it must copy them, since the original object will cease to exist when it goes out of scope. To avoid the copy on function calls, C++ supports call by reference with the syntax `bar(Foo& x)`. Unfortunately, references introduce the risk of dangling references, which are analogous to dangling pointers. For small objects allocated on the stack, programmers typically accept the cost of copying. For larger heap-allocated objects such as vectors, matrices, images, hash tables, etc., the cost of copying the whole structure on every assignment would be prohibitive.

Most existing libraries solve this problem by referring to heap-allocated objects using pointers. Unfortunately, this defeats C++'s automatic memory management facilities. When a pointer goes out of scope, there is no way to know whether the object it points to should be deallocated. For example, there might be another pointer to it on the stack that is still in scope or that is itself allocated on the heap. Therefore, such libraries usually implement a form of reference counting to keep track of the number of distinct references to an object. The reference counting mechanism requires extra heap space which must be separately allocated and whose contents must be incremented or decremented on every assignment. The usual organization also requires an extra level of indirection in all accesses to the underlying object. In addition to extra performance costs, these mechanisms introduce new levels of complexity for the programmer and more sources for error.

Our memory management discipline, on the other hand, allows us to avoid incurring any extra space or time overhead when using heap-allocated space, simply by virtue of the fact that we do not pass around pointers to allocated space, hence we do not have to devote space or time to keeping track of such pointers.

The only extra burden is the addition of just one function to the interface of each class that uses heap storage. This function, called `take`, permits transferring the unique ownership of heap storage between two objects, without any copying or reference counting.

In our implementations, the typical object consists of a number of fields which hold one of the built-in types (bool, char, int, float, or double) or a pointer to heap-allocated space. Such an object is the owner of its heap space and the object's destructor is written so that it will deallocate that heap space when the object is destroyed. No object should point at the heap space of any other object (except for temporary purposes such as iterating through the object).

Each object supports a default constructor which initializes any pointer field to be the null pointer. This means that declaring a local variable does not by itself cause any heap allocation. Similarly, allocating an array of objects does not allocate any further storage when each object in the array is constructed; only the space required to store the array itself is allocated. Objects which point to C arrays will usually have a field named `asz` (for 'allocated size') that specifies how much space is allocated. When `asz` is 0, the pointer should be
3.3.1 Avoiding copying with take

Rather than pass pointers around, the way we avoid copying heap space when moving objects around is to require every class that owns heap storage to support the operation

friend void take(Tk dest, Tk src).

This operation transfers ownership of heap storage from src to dest. More precisely, it first deletes any heap storage pointed to by dest, sets dest to point to the storage formerly pointed to by src and sets each and every heap pointer in src to null. Thus, it effectively moves any heap storage pointed to by src over to dest without copying it, while maintaining the invariant that only one object may own a given block of heap store. Note that both src and dest are modified in this operation.

For example, the relevant portion of the vector class Vec looks like:

```cpp
class Vec
{
  int sz;       // Dimension of the vector.
  Db* p;        // Ptr to the contents, null iff sz==0.

  friend void take(Vec & v1, Vec & v2) {
    // Make 'v1' be 'v2' and 'v2' be null.
    delete[] v1.p;
    v1.sz = v2.sz;
    v1.p = v2.p;
    v2.sz = 0;
    v2.p = 0;}
};
```

take can be used to move objects from one place to another without heap allocation or deallocation. The container classes define additional operations based on take (e.g., the stack class Stk<T> defines push_take and pop_take). This allows one to define a stack of vectors and to push and pop them without copying the vector on each operation. The user must, however, be aware that these operations zero out the source.

Container template classes such as Stk<T> rely on the existence of take for any class T they might be used to contain, including built-in types as well as user-defined types that do not allocate heap storage. For example, one might want to define a Stk of Ints. To relieve the programmer from defining take for each and every class, the library defines a default templated version of take in All.h which simply does the assignment dest = src. The programmer overrides this default version by defining a take with explicit (i.e., non-templated) arguments.
Since = defaults to memberwise assignment in the absence of an explicit
definition, the default take will generally not move any heap-allocated space,
but will simply copy the pointer to it. This violates the memory management
discipline and could result in dangling pointers. Therefore, any class which
allocates storage must define its own take. If we had not defined the default
take routine, this kind of error would always be caught by the compiler. This
kind of error arises when extending the library with new classes rather than
when using existing classes. New classes are extensively tested and tend to be re-used repeatedly. We felt the increased risk of forgetting to define take is
more than offset by the resulting ease of defining simple classes.

As with all C++ classes, each type should also define a copy constructor
and an assignment operation which copy the heap portions of the object. These
operations are used much less frequently than in other libraries, however. If
a class's objects do not own heap storage, one may omit defining the copy
constructor and assignment operator, in which case the compiler will generate
memberwise copy. But failure to do so if heap storage is owned will result in a
bug analogous to the one described above.

Also in common with most other designs, most functions should take arguments which are references to objects. Care should be taken to not delete an
object which has outstanding references to it.

3.3.2 Side effects and return values

Our solution to the problem of copying on function return is to (mostly) avoid
returning objects! Usually we have designed our routines to modify either this
or a reference argument, rather than return an object as the result. For example,
the Vec class supports

```cpp
void add(const Vec& v); // Make 'this' be its sum with 'v'.
```

rather than the usual + operation which would return a new vector that is the
sum of two existing vectors. While code based on our approach is slightly less
natural looking, it is substantially faster.

Names starting with to_ are typically used for functions which modify this
in a way that is independent of its current value. For example, Vec defines:

```cpp
void to_zero() {to_const(0);} // Make 'this' be the origin.
```

(Meyers, 1996) describes in detail the consequences of return by value. It
may require several copies by the time the returned object is ready to be used
by the caller. Another approach is to return either references or pointers to
objects. As described above, this approach typically requires reference counting
to avoid memory leaks.

In a functional programming discipline, a natural approach is to create an
object in that routine which knows what the object should be. For example, if
we wanted a vector with each component equal to 1, we can imagine invoking a
function Vec& const_vec(Int d, Dbl val), which would allocate a vector of
dimension $d$, with components all equal to $val$, and then return a reference to the allocated Vec. There is no way to do this and still take advantage of C++'s automatic destruction feature, because the result must be allocated inside the routine const_vec. If we allocate the return object (not its heap storage) on the stack with a simple declaration, it will go out of scope when const_vec returns, and we will be handed a dangling pointer. The only way to avoid that is to allocate the return object on the heap with new, in which case it will not be destroyed on exit, preventing a dangling reference, but by the same token also defeating the automatic destruction of stack variables provided by C++.

Instead, what we need is the scope of the returned object to be in the caller, where it is being used, not in the routine where it is being built for the caller's consumption. Then, when it is done being used, it goes out of scope, and is deallocated automatically.

The solution we have adopted is to use function definitions such as

```cpp
void to_const(Dbl s); // Make each component of 'this' be 's'.
```

This is used like this:

```cpp
Vec v(d); // Constructor to make a Vec of dim d.
v.to_const(1.1); // Set each component of 'v' to 1.1.
// 'v' is now ready for further use.
```

$v$ is declared and created in the caller, so its storage is freed automatically when it goes out of scope in the caller. to_const merely sees to it that each component is set to the desired value. No copying of heap objects is involved.

Similarly, we sum an array of Vecs using

```cpp
void sum_in(const Arr<Vec>& va, Vec& sum); // Make 'sum' be the sum of the vectors in 'va'.
```

This might be used as

```cpp
Arr<Vec> va(10);
for (Int i=0; i<10; i++) {
    va[i].to_dim(5); // storage for dim 5 Vec
    va[i].to_const(1.1); // set each component to 1.1
VeC sum;
sum_in(va, sum); // make 'sum' be the sum over 'va'
// sum is now Vec(11 11 11 11 11)
```

Again, the argument sum is declared in the caller, so its storage — which is reallocated inside the function sum_in — is freed automatically when it goes out of scope in the caller.

The reference in the argument list that gets the return value is to an object in the caller. It therefore cannot become a dangling reference when the routine returns. It is tempting, therefore, to also return the same reference, which
would allow a more functional style of programming. We have avoided doing this since it would likely cause more confusion rather than less. The problem is that one can't look at a C++ function call and see that an argument is being passed by reference — it's explicit only in the declaration of the function. If the function returns a value, it is all too easy to forget that the argument is being changed by side effect. To avoid this pitfall, we have adopted the discipline that functions that operate by modifying their arguments should be of type void, i.e., not return anything. This guarantees that it is syntactically obvious from the function call that the function is acting by side-effect.

A nonstandard consequence of this discipline is that = returns void in the library classes. This causes constructs like `Vec u,v,w; ... u=(v=w);` to generate compile-time errors. This has the beneficial result that erroneously using `v=w` when one means `v==w` leads to a compile-time error. It also renders illegal the confusing construct `u=v=w`. This is confusing because of the ambiguity over whether this means `(u=v)=w` or `u=(v=w)`.

We have not redefined = for built-in types such as Int, Db1, Bool\(^1\) (and indeed C++ does not allow such a redefinition). Also, the compiler generates a default =, which does memberwise assignment, and returns a reference to the result. We did not feel that requiring the programmer to redefine the default = for classes without heap storage solely to provide a different return type was worth the cost.

### 3.4 Inheritance

"Object-oriented programming" is nearly synonymous with "inheritance." Yet the only place in the library where we use inheritance is in the random number generator class, where `Rnd` contains a variable which is a pointer to a random number generator object, that must inherit from the abstract random number generator class `ARndGen`. We generally avoid inheritance because in C++ it almost always causes more trouble than it is worth.

In C++, using inheritance requires that pointers to objects be passed around, since it is only a pointer to a parent type that can safely be used to "contain" a child type. Unfortunately, the passing around of pointers is the root of most of the memory management problems suffered by other C++ programming disciplines, and is the practice we are trying to avoid in ours.

We have also found that liberal use of inheritance results in inscrutable programs because the information needed to understand what a class even consists of, much less what it is doing, is scattered over the transitive closure of all the places that it inherits from. While this saves some programming effort in the initial construction phase, it makes maintaining and using the class more difficult and error-prone in the future. Multiple inheritance compounds this problem, as does the fact that C++ has complicated rules governing inheritance, and in some cases, such as initialization in multiple modules, results can even be undefined.

\(^1\)Actually, the types we list are typedefed to built-in types.
4 Recurring programming constructs

4.1 Amortized doubling

Many languages (e.g., Lisp) and libraries use linked lists as the fundamental container data structure. The cache structure of modern machines, however, can dramatically favor array based structures because of their better locality and more efficient memory usage. The problem with arrays is that their size must be known in advance. To overcome this limitation, we use the technique of *amortized doubling* in several places in the library. When the size limit of an array-based structure is reached, its size is doubled and the elements are moved (using `take`) to the new expanded storage space. Though this copying can be expensive, it happens rarely. If \( n \) elements have been added to a structure, then at most \( \log n \) expansions will occur. The total storage allocated is of order \( 1 + 2 + 4 + 8 + \ldots + n \approx 2n \), as is the total number of moves of all objects. The amortized cost of constructing such structures is thus only a small constant times the number of elements contained.

We did some simple experiments to compare the performance of array-based methods using amortized doubling to methods based on linked lists. We implemented stacks of integers using each of the two techniques\(^2\) and measured the time to push, pop, and search for varying numbers of integers. We ran the experiments on a machine based on an Intel Pentium Pro 200MHz processor under Windows NT 3.51, with compilation set to maximize all optimizations under Microsoft Visual C++ 4.2.\(^3\)

The results are plotted in figures 1, 2, and 3. Array-based stacks are about 3–5 times faster than linked lists to push a specified number of elements, despite the overhead of amortized doubling. This is because of the repeated calls to `new` that the linked list implementation must make. Array-based stacks are about 7–14 times as fast on repeated popping, because of the repeated calls to `delete` that linked lists must make. The relative performance for pushing and popping appears quite constant over a range of sizes. The behavior for searching is slightly more interesting. As shown in the graph they are comparable for small lists, but for large lists the array version is about 5 times as fast as the linked list version. This is probably due to caching issues which show up for large lists.

4.2 Iterators

Iteration through data structures is an important and common operation. It is therefore desirable to encapsulate iteration in classes (Murer et al., 1993). Many designs have been proposed for C++ iterators. Because iteration is quite literally “in the inner loop” of the most costly code, efficiency is of paramount

\(^2\)Linked lists were implemented using a random permutation of items to prevent contiguous items in the list from always also being contiguous in memory, to simulate what would typically happen in an application. However, this did not have a pronounced effect on performance.

\(^3\)These measurements were made using a timer which can include cycles used by other processes, and so should be regarded as a rough indication of relative performance, rather than a precise benchmark.
Figure 1: The time to push a specified number of integers onto a stack.
Figure 2: The time to pop a specified number of integers off a stack.
Figure 3: The time to search for an integer which is not on the stack.
importance. We require that iterators be lightweight and preferably stack allocated. We would also like them to be easy to use and read, both syntactically and semantically.

Our approach to iterators also avoids returning objects. Iterator classes have names which end in It. They define a constructor which takes the container that is to be iterated through as an argument. Most iterators also define the infix operator >> to assign successive elements of the iterator on the LHS\(^4\) to the variable on the RHS. Except for the StrIt class, >> also returns a Bool which is true if there was an entry to assign, and false if the iterator has finished. For example, the class IntSet defines sets of integers. Associated with IntSet is the iterator class IntSetIt which iterates through the current set. It defines:

\[
\text{Bool operator>>(Int& i);} \\
// If there is another element of the set, assign it to 'i' and \\
// return 'true', otherwise return 'false'.
\]

This allows the terse, efficient, and fairly clear usage:

\[
\text{IntSet s; IntSetIt it(s); Int i;} \\
\text{while(it>>i) some_function(i);}
\]

which causes some_function to be evaluated on each element of the set. For containers of complex elements, the iterators usually assign a pointer into the container to avoid copying and to make explicit that the container is the owner of the object.

Iterators over the string class Str are special in a number of ways and are described in a later section (7.5).

### 4.3 Assertion checking

The library makes extensive use of the assert() macro. In debug mode — indicated during compilation by the macro NDEBUG not being defined — an assert() expands into a test whether its boolean argument is true. If the test fails, assert prints an error message giving the source code location of the failure. In release mode, i.e. not debug mode, the macro NDEBUG is defined (usually by a compiler switch), in which case the assert macro expands into whitespace and is therefore a no-op. In particular, its argument is not evaluated and takes up no code space.

Our most common use of assert is to test preconditions that must hold for proper behavior of a routine.

Notice that it is better to use multiple assert rather than an equivalent single test which is a conjunction. E.g.,

\[
\text{assert(a); assert(b);} \\
\]

\(^4\text{LHS means "left hand side." RHS is "right hand side."}\)
is preferred over

```
assert(a && b);
```

The reason is that in the first case, if one of the separate asserts fails, the error message printed by that `assert` will identify it. But if the `assert` of the conjunction fails, there will be no indication which of the conjuncts `a` or `b` failed. The loss of efficiency is immaterial because `assert()` is only compiled into the code in debug mode, and has no effect on the production version of code.

### 4.4 Object null state

Most library classes support the notion of a **null** state which is the state of an object when it is produced by the default constructor. For example, when a vector is created by `Vee v;` it has dimension 0 and no heap space. The member routine `Bool is_null()` tests whether an object is in its null state, whereas `void to_null()` transforms the object to the null state, and deletes any of its existing heap-allocated state.

Some classes, such as `Str` and `Stk` also define operations `to_empty` and `is_empty`. `to_empty` makes an object contain no elements, but does not delete any storage it may own. This is provided purely for efficiency reasons, except for classes like `IntSet`, where the term 'empty' has a traditional meaning. When `is_empty` is provided, it is synonymous with `is_null`; in particular, this means that `is_null` will return `true` on an object that owns storage, as long as the object currently has no elements. The reasoning behind this behavior is that as far as the interface, and the abstract data type of the object, is concerned, it should not matter whether an object has storage which is currently not used, or whether it does not even currently have storage; all that matters is that it contains no elements. An empty object should be able to be used exactly the same as a null object.

### 5 Stylistic conventions

#### 5.1 Class layout

Each routine definition begins on a new line. The header file (but not the `.cpp` file) should have a comment for each routine describing its function. For readability, all declarations should be formatted in the same way. The routine type signature should appear first. The descriptive comment should appear next starting on the same line if there is room. If the implementation is provided in the header, this should appear next. Some examples from `Vec`:

```
Vec(Int d);          // 'd'-dim Vec initialized with 0's.
```
Vector

To:

II

Default constructor. A null vector.

sz(0) {}

bool is_null() const {
  // 'True' if 'this' is null.
  return sz==0;
}

5.2 Comments

We use only the C++ style comments preceded by //. The older style C comments demarked by paired "/* and */ are harder to visually parse and in any case are rendered redundant by the new // convention.

Every function prototype in a header file should have a comment describing its use and function (see section 5.1 above).

5.3 Naming conventions

A coherent naming convention can make a library more readable, easier to write, and less susceptible to bugs. If all classes adhere to the same conventions, users are able to transfer their experience among them. Routines are less likely to be misused if their names reflect their intended use in a well-defined and consistent way. Code is easier to write if there are rules specifying how things should be named. The rest of this section describes the conventions for class names, routine names, and other names.

5.3.1 Global constants

Names of global constants are entirely upper case, with words separated by underscores. For example:

const static int HASH_PRIME=516595003;

5.3.2 Class names

Each class name appears many times in typical C++ code. One reason for this is that the definition of each member function in a .cpp (or .C or .cc) file is prefaced by the class name, e.g., Foo::bar() {...}. The class name also appears in every declaration of a variable of that type. In addition, C++ uses class names to name constructor routines, so the class name appears at every explicit constructor call. Our naming style often creates new class names by extending existing names; similarly, declaring an instance of a templated class requires both the template name and at least one parametrized class, e.g., Stk<Db1> for a stack of doubles.

All of these factors provide motivation for keeping class names short. The library will probably have only a few hundred class names (as opposed to many thousands of routines) and so it is relatively easy for a user to keep track of the names even if they are abbreviated. The most common classes are therefore given one-word names of three or four letters (e.g., Str, Vec). To help distinguish
class names from routine names, all class names are capitalized (first letter in upper case, other letters in lower case). Consistent with other libraries, when a name consists of multiple words, each word is capitalized and no spaces appear between them (e.g., StrIt, IntSet).

To make all our code consistent with this convention, we provide capitalized names for all the built-in C++ types used by the libraries, which we have implemented by typedefs. These typedefs appear in the header file All.h which is included in every library source file. They are:

- **Bool** represents the C++ boolean type bool on compilers that support it, char on those which don't.
- **Char** represents the C++ character type char.
- **Uchar** represents the C++ unsigned char type (this is needed, e.g., in classes which manipulate image pixels).
- **Int** represents the C++ integer type int.
- **Uint** represents the C++ unsigned int type.
- **Dbl** represents the C++ double precision floating point type double.
- **Flt** represents the C++ single precision floating point type float.
- **Short** represents the C++ short type.
- **Ushort** represents the C++ unsigned short type.
- **Ulong** represents the C++ unsigned long type.

Use of Dbl is preferred over Flt, and Uint, Short, Ushort, and Ulong are primarily for use in interfacing to system functions that require these types. Some of the other short names include:

- **Ball** for multi-dimensional balls,
- **Box** for multi-dimensional hyper-rectangular boxes,
- **Cpx** for complex numbers,
- **Flat** for multi-dimensional affine subspaces,
- **File** for the file access class,
- **Gph** for directed graphs,
- **Gsn** for multidimensional Gaussian probability distributions,
- **Ivl** for real intervals,
- **Mat** for matrices of doubles,
Mnl for multinomial probability distributions,
Rnd for the random number generation class,
Str for the string class,
Time for the time measurement class,
Vec for vectors of doubles.

Some less commonly used classes with longer names include:

Gnuplot for the interface class to the gnuplot plotting package,
GsnMix for Gaussian mixtures, and
Histogram for the histogramming class.

There are several forms of suffix that can be appended to class names to yield multi-word names for related classes:

-Set to name a set of items of the given type, e.g., IntSet.
-Mix to name a mixture of probability distributions or mappings, e.g., AffMix, GsnMix.
-It to name iterator objects which step through a given container type, e.g., StrIt, IntSetIt.
-1, -2, or -3 for special lower-dimensional variants of a type, e.g., Vec2, Vec3, Gsn1.

As far as possible, we define one class per file, with the base of the file name the same as the class name. But in some cases, there are closely coupled classes which belong together. The source files containing class definitions are then given the name of the most basic underlying class. Thus the definition of Vec appears in the header file Vec.h and the body file Vec.cpp. We try to name related classes in the same file by starting them with the name of the base class, ensuring that the file name is a prefix of every class name defined in the file (though there are a few exceptions). This makes it easy when given a class name, to find the file where it is defined.

Abstract classes have a name starting with A; for example, ARndGen. However, we generally avoid inheritance, and therefore also the use of abstract classes.
5.4 Routine names

Routine names appear only when they are defined or called, so there is less pressure to keep them short. We make routine names entirely lower case to help visually distinguish them from class names (except for constructor routines, whose name C++ requires to be identical to the class name and hence is capitalized). When a name consists of more than one word, the words are separated by underscores (e.g., `reflect_through_zero`).

We use abbreviations much less frequently in routine names than in class names since here expressiveness and readability are more important than compactness. A few standard abbreviations that do appear include max for ‘maximum’, min for ‘minimum’, rnd for ‘random’, const for ‘constant’, dim for ‘dimension’, sqrt for ‘square root’, and elt for ‘element’.

Names are often designed to read as if part of grammatical sentences. To this end, there are a few prefixes and suffixes that are used with standardized meanings:

- **to_** is a prefix that says that this is to be modified to have the property which follows. For example, in the Vec class, to_const makes all the elements of a vector be a specified constant, to_normal_rnd makes them be random selections from a normal distribution, to_interpolate_between makes the vector interpolate between two specified vectors. These routines modify this in a way that ignores its present value.

- **from_** is a prefix for routines that change this to come from a different type. e.g., in the pixel class `Pxlc`, the routine from_vec3 modifies a pixel to have color components derived from a specified 3-vector. These routines modify this in a way that ignores its present value.

- **is_** is a prefix for a Bool routine which evaluates a predicate and returns true or false. E.g., is_null tests whether an object is in the null state:

  ```
  if (v.is_null()) do_something_about_it();
  ```

- **_in** is a suffix that indicates that a routine places one or more results into argument objects (typically reference arguments). For example, `Vec` defines `mean_variance_in` which computes the mean and variance of the components of the vector and places them into two specified arguments.

6 Typical class features

This section describes features in typical library classes. We use routines from the vector class `Vec` as illustrations. A `Vec` object consists of an integer `Int sz` which stores the size (dimension) of the vector and a pointer `p` which points to the array of doubles on the heap that represents the components. The class looks like:
class Vec
{    // Vector of doubles.
protected:
    Int sz;    // Dimension of the vector.
    Dbl* p;    // Ptr to the contents, null iff sz==0.
    ...
}

The null state of a Vec is defined as sz==0 and a null pointer p, i.e., p==0.

6.1 Default constructor
Every class has a default constructor which creates an object in the null state. This is important because when an array of objects is created, each object is initialized using its default constructor. The default constructor for Vec is:

Vec() : sz(0), p(0) {}

This just ensures that sz is set to zero, and that p is null. (If the constructor had omitted the initializations of the members to 0, these values would contain garbage, since built-ins are not initialized by C++.)

6.2 Copy constructor
The copy constructor creates a new object, recursively copying any members. This requires allocating new heap storage for any heap-allocated objects, and copying their contents.

The copy constructor for Vec is:

Vec::Vec(const Vec& v) : sz(v.sz), p(new Dbl[v.sz]) {
    Int one=1;
    dcopy_(&sz,v.p,&one,p,&one);
}

This first copies the sz field from v. Then p is set to point to an array of doubles of size sz, created by the call to new. The array values which comprise the components of the Vec v are copied by the Blas routine dcopy_. This is a simple copy of an array of doubles, but using Blas has the potential for efficiency gain if the Blas routines have been hand-coded and optimized for a particular platform, as is the case for many platforms.

6.3 Other constructors
Classes typically supply other constructors with appropriate arguments. Constructors which take a single argument of another type define conversion operators which C++ may implicitly use for tacit conversions of function arguments. Our experience has been that implicit conversions lead to very annoying bugs,
typically when some unforeseen conversion takes place with a constructor whose purpose was unrelated. The C++ proposed draft standard (ANSI Accredited Standards Committee X3, 1995) introduces the keyword explicit to modify the declaration of such a constructor to tell the compiler not to use that constructor for implicit conversions. MS Visual C++ 4.2 did not implement the explicit modifier. Consequently, we avoided defining constructors of one argument of a different type whenever possible. MS Visual C++ 5.0 does implement the explicit declaration, so we have used this declarator in the few cases where we found such constructors useful.

For example, Vec.h declares:

```cpp
explicit Vec(Int d); // 'd'-dim Vec initialized with 0's.
```

which creates a d-dimensional vector at the origin.

6.4 Assignment operator: =

Unlike most other class libraries, and C++ itself, we adopt the convention that assignment does not return the assigned object (nor a reference to it). Instead it is declared to return void. The standard C operation of `a=b=c` is ambiguous, requiring the programmer and anyone reading the code to remember the associativity (right or left) of `. When = has side effects it is especially confusing to do more than one assignment in a single statement. The alternative `b=c; a=b;` is much more readable and not much larger. This choice also saves the final return instruction in the definition of = and consequently sidesteps issues revolving around whether the assignment returns a reference or returns an object.

6.5 String form generator: write

As part of our program of having a machine-readable print form for every non-trivial class (see section 8), any such class must define write, which creates a Str representation of its objects. A template in Str.h automatically uses this write to define an append operator `<<` for this class. Vec's write is:

```cpp
friend void write(Str& s, const Vec& v);
   // A string version of 'this' of the form "Vec(1.1 2.2 3.3)".

void write(Str& s, const Vec& v) {
   s << "Vec";
   for(Int i=0; i<v.sz; i++) {
      if(i!=0) s << ', ';
      s << v.p[i];
   } s << ');
```

This code creates a Str that looks like 'Vec(2 3 5 6)'.

29
6.6 String reader: read

Reading the string form requires another function, read. Another template in Str.h uses the class-defined read to define the StrIt iterator. Here's how Vec does it:

```cpp
friend void read(Vec& v, StrIt& it);
// If 'it' points to a substring of the form:
// "Vec(1.0 2.0 3.0)"
// then move it to the following character and construct its value.
// Otherwise, throw a StrItEx exception describing the error.

void read(Vec& v, StrIt& it) {
  v.to_null();
  if (it.is_done()) it.err("Vec: nothing to read.");
  if (!it.check_str("Vec(")) it.err("Vec: wrong name.");
  Dbl d; Stk<Dbl> dbuf;
  while(1) {
    it.skip_comments();
    if (!it.check_str("")') break;
    if (!it.check_dbl_in(d)) it.err("Vec: missing closing paren.");
    dbuf.push(d);
  }
  v.make_dim(dbuf.size());
  for(Int i=0; i<v.dim(); i++) v.p[i]=dbuf[i];
}
```

Note that read and write must be declared friend in the function prototype.

6.7 Storage transfer: take

Central to our memory management discipline is the ability to change which object is the unique owner of the heap storage that contains its data. take is like assignment; in fact, data members other than pointers it simply copies. But when it comes to a pointer to heap storage that is owned by the object, the storage is not copied, but rather its ownership is transferred to the target object by assigning and zeroing pointers.

Here is how Vec defines its take:

```cpp
friend void take(Vec& v1, Vec& v2) { // Make 'v1' be 'v2' and 'v2' be null.
  if (&v1 == &v2) return; // nothing to do; avoid clobbering 'v2'
  delete[] v1.p;
  v1.sz = v2.sz;
  v1.p = v2.p;
  v2.sz = 0;
  v2.p = 0;
}
```
6.8 Null state: is_null and to_null

Objects with storage have a null state, which is their state when they are born from a default constructor, and which does not yet have any allocated storage. This assures that the default constructor is cheap.

Vec objects use heap storage, so Vec defines:

```cpp
Bool is_null() const { // 'True' if 'this' is null.
    return sz==0;
}

void to_null() { // Set 'this' to null.
    sz=0; delete[] p; p=0;
}
```

6.9 Default destructor

Our memory management discipline requires that the destructor of each class deallocate any storage that the object being destroyed may own.

```cpp
~Vec() { // Destructor.
    sz=0; delete[] p; p=0;
}
```

6.10 What properly written classes contain

We summarize what should appear in every class definition.

- Default constructor.
- String form writer routine write.
- String form reader routine read.

If the class contains heap storage, it should also define:

- Assignment operator =.
- Storage transfer routine take.
- Default destructor.
- Copy constructor.
- Null test and nullifying routines is_null and to_null.

If the class is meant to be usable as a Sequence:

- Indexing operator [].

31
• Size reporting routine size.
• Size setting routine to_size.
• Element type typedef elt_type.

And if a Sequence type is to be sortable:
• Order comparison <= or <.

7 The string class: Str

The design of the string class is fundamental to the whole library because every class must be able to read and write string representations of the data structures it defines. We therefore discuss the Str class in detail here rather than with the other library classes in section 9.

We would like strings to be both flexible and efficient in space and time. Because string representations are typically built up a component at a time, it is important that strings efficiently support incremental construction.

One choice would be to simply use the standard strings from C of type char*. In this representation a string is simply a pointer to an array of characters terminated by the null character '\0'. The primary advantages of this are compatibility with existing C routines and space efficiency. Disadvantages include lack of support for incremental construction and time inefficiency due to having to search for the terminating '\0' in any operation which needs the length of the string.

A C++ version with the same advantages and disadvantages would look like:

```cpp
class Str
{
protected:
    char* p;       // Pointer to the character string.
    ...
}
```

A less space efficient version which avoids the time overhead of searching for the terminating null, would include a size attribute:

```cpp
class Str
{
protected:
    int sz;          // Actual string size, without the '\0'.
    char* p;        // Pointer to the character string,
    ...
}
```
Because of the importance of incremental construction of strings, our design uses the amortized doubling technique. Our representation stores both the size of the allocated space and the size of the string itself:

```cpp
class Str
{
protected:
    int asz;           // Allocated size.
    int sz;            // Actual string size, without the '\0'.
    char* p;           // Pointer to the character string,
    ...
}
```

The terminating '\0' is not strictly necessary for the class itself, but we require it for ease and efficiency in interfacing with C functions.

### 7.1 Empty strings

As with all our objects, we would like to avoid allocating heap space for empty strings. Str objects therefore obey the constraint that the string pointer p is null if and only if the allocated size asz==0.

It may happen, however, due to a read of an empty C string, or a call to `to_empty`, that a Str has storage, but the storage is not being used. In such a case the storage contains an empty C string, i.e., the first Char of the storage is '\0', and the size of the Str, sz, is 0. `is_null` called on such a Str will return true.

### 7.2 Outputting Str's on streams

Strings should naturally interface with standard C++ libraries for printing on output streams. To support outputting a Str to an ostream, we therefore define:

```cpp
inline ostream& operator<<(ostream& o, const Str& s) {
    // This allows anything that generates a Str to be put on cout, cerr, clog.
    if(!s.is_null()) o << s.c_str();
    return o;
}
```

This definition can be found in the header file `IostreamRCL.h`, rather than in `Str.h`, which might seem more natural. The reason for this is that the standard C++ library requires reading considerable header information for this definition to work, and since `Str.h` is read by essentially every module that uses RCL, this would impose an excessive compilation time penalty. Instead, everything that pertains to `iostreams` is localized in `IostreamRCL.h`.

Any module that uses `iostreams` should include `IostreamRCL.h`. `IostreamRCL.h` in turn, includes the appropriate `iostream` header file. In order to assure consistency in linking the right C/C++ runtime, you should not directly include
<iostream.h> or <iostream>, or other component headers of iostreams. (For more details about this see the appendix section B.1.

7.3 Appending objects to strings

We intend that strings will be primarily constructed by repeated appending, since this is more efficient than repeated copying. The standard C++ syntax for appending objects to a stream (as defined in the <ostream.h> library (Stroustrup, 1991), p. 325) utilizes the « operator. Following this usage, we overload this operator to append to Strs as well.

The string class Str therefore directly supports the following overloads of operator« for appending to a Str either another Str or one of the "string-like" built-in types Char* and Char:

```c
inline Str& operator<(Str& s, const Str& sl) 
inline Str& operator<(Str& s, const Char* sl) 
inline Str& operator<(Str& s, Char* const & sl) 
inline Str& operator<(Str& s, Char c) 
```

Defining Str print forms for arbitrary objects allows extending « to allow appending the Str print form of any object to a Str using the same syntax. This is described in more detail in section 8.2.

7.4 Hashing strings

There are many important uses for hash tables indexed by strings. Symbol tables, for example, are a typical application.

Str provides the routine Int hash() const to compute hash functions on strings. The hash function is taken from (Knuth, 1993), p. 300. He claims that simpler hash functions produced noticeably poorer results.

After t is initialized to be a pointer to a C string we want to hash on, the inner loop is:

```c
static const Int HASH_MULT=314159; // Random multiplier
static const Int HASH_PRIME=516595003; // The 27182818th prime. < 2^29.
for(Int h=0; *t; t++) {
    h+=(h~(h>>1))+HASH_MULT * (Uchar)*t;
    while (h>=HASH_PRIME) h-=HASH_PRIME;
}
return h;
```

The result is a positive integer bounded by HASH_PRIME. To cover a smaller range fairly uniformly, the result should be taken modulo the desired range.

7.5 String iterators: StrIt

The class StrIt defines iterators over strings.

The typical class Foo defines a function of the form:
friend void read(Foo& v, StrIt& it);

which reads an object of that class from a string (via a StrIt). This routine takes a string iterator which should be pointing at a substring of the form that describes the object, e.g., “Foo(1 2 3)”.\(^5\) If successful, read moves the iterator past the object description and fills in f with the object. If unsuccessful, read throws a StrItEx exception by calling the StrIt member function err, with an explanatory message. The err function does the necessary bookkeeping to help the user isolate where in the scan the error occurred.

For example, here is the definition of the read routine from the vector class Vec:

```cpp
void read(Vec& v, StrIt& it) {
    v.to_null();
    if (it.is_done()) it.err("Vec: nothing to read.");
    if (!it.check_str("Vec(")) it.err("Vec: wrong name.");
    Dbl d; Stk<Dbl> dbuf;
    while(1) {
        it.skip_comments();
        if (!it.check_str(")") break;
        if (!it.check_dbl_in(d)) it.err("Vec: missing closing paren.");
        dbuf.push(d);
        v.make_dim(dbuf.size());
        for(Int i=0; i<v.dim(); i++) v.p[i]=dbuf[i];}
```

It is sometimes useful to check for the existence or correct formation of a built-in type such as Int or Dbl without throwing an exception. This allows handling such eventualities in a calling routine without recourse to the exception handling system, which is reserved for truly exceptional occurrences (and which is therefore slow). For this purpose, Str.h provides check_int_in and check_dbl_in.

The is_null function is not defined for the StrIt type. The reason is that there is no clean semantics in this case. E.g., is a null StrIt one which points to a null string, or which points to nothing? We do not allow a default constructor for StrIt, so a null StrIt in the usual sense should not occur. We felt it was better to force the user to test for the specific sense of 'null' that he has in mind, rather than to arbitrarily decide what this might mean for StrIt.

8 Print forms, readers, and string iterators

As in many dialects of Lisp, we have adopted the idea that as far as practical, every data object should have an external representation, known as a print form (or string form, to avoid the inference that it is exclusively related to printing). Similarly, every print form should be able to be read back into the

\(^5\)The substring is what is within the quotes, and does not include the quotes themselves, which rather are part of our text.
internal representation, resulting in a semantically identical object. Print forms are most naturally represented as strings. Nearly all of the library classes provide a string form for their objects. Some of the uses for string forms are:

- **Program output.**
  For example, a vector can be output directly using its string form rather than designing a special output form for each application. The string class provides routines for "pretty-printing" such string representations.

- **Manual input.**
  The object description syntax is simple and intuitive; interactive programs can use it directly for user input.

- **Debugging.**
  The string form makes it easy to display data structures during debugging.

- **Checkpointing.**
  During long computations, the internal state of the computation can be periodically stored in a file. If a machine goes down, a problem occurs with a later stage of a computation, or if parameter changes must be made in a later stage of a computation, the computation can be restarted from the checkpointed state rather than from the beginning.

- **Interprogram communication.**
  The string form can be used to communicate structures between different programs, via the file system or pipes. Simple text files can be used as intermediaries and can be readily examined by humans.

- **Cross-architecture communication.**
  The string representation of an object is the same on all architectures, regardless of their integer precision, their floating point format, whether they are big or little endian, etc.

These uses require that the string representation should be easy for both machines and people to generate, read, and modify. Each class defines the specific format for the string form of its objects by defining a routine `write` to generate the string form, and a routine `read` to parse it. We impose a uniform structure on this representation, however, so that the formatter and other tools can easily manipulate the strings.

### 8.1 Print form anatomy

Since objects of one class may contain members which are objects of some other class, the general string form representation is defined recursively.

In general, the string form of an object is of the form

```
Class(member member ... member)
```
where *Class* is the name of the class of the object, and each *member* is the value of the corresponding component of the object in question.

For example, a *Vec* can be represented as

\[ \text{Vec}(1 \ 2 \ 13 \ 42 \ 5 \ 6 \ 7) \]

Note that there are no commas separating the parenthesized list of components.

Objects of type *Str, Char*, and *Char* are treated specially, since their representations use the standard C quote syntax.

For example:

"Hello, world" // Str print form
"Hello, world" // Char* print form
'H' // Char print form

Here is a more detailed recursive description of the string representation for objects:

The print form of a *Str* or *Char* is a character sequence delimited by double quotes ", with the same conventions as C strings. After the opening double quote, a *Str* print form is terminated by the first following double quote which is not preceded by a backslash \. Within a *Str* or *Char* print form, the sequence \n represents the newline character, \ \ represents a backslash, \" represents a double quote, and a backslash and an immediately following newline, or carriage return-newline pair, are ignored (to allow continuation on the following line). Additionally, when a *Str* print form is read, \t is converted to a tab, \r is converted to a carriage return, and \b is converted to a backspace, in conformance with the standard C syntax. Note that the C character escapes \a, \f, \v, \?, and \0, as well as all numeric escapes, are *not* permitted.

The print form of a *Char* is a character delimited by single quotes '. The same escape (backslash) sequences as for *Str* are applicable, except that the \<newline> and \<carriage return><newline> escapes are not permitted.

The print form of an object *other than Str or Char* consists of the name of the type, a left parenthesis (, a sequence of zero or more items, and a right parenthesis ). Whitespace is not allowed between the type name and the opening parenthesis but may appear arbitrarily on either side of other tokens. Each class defines the sequence of items which it allows and they may be separated by arbitrary whitespace and comments. Strings should be represented as described above and integers and floating point values should be represented using the standard C syntax. Integer and floating point values may have leading whitespace. Attempting to read a string with an incorrect format will throw a *StrItEx* exception.

A comment within a print form consists of a pair of slashes // outside the print form of any *Str, Char*, or *Char*, and all characters following until the next newline. Comments are ignored when they appear where whitespace is allowed.

For example, a *Vec* can be represented as either of:
8.2 Generating the string form: write and <<

Every class Foo specifies its print form by defining a function

```cpp
friend void write(Str& s, const Foo& f)
```

which appends the print form of its second argument to the first argument s, in conformance to the specifications above.

The class-defined write must be declared friend in order to override the universal template routine write, which merely sets its Str arg to the string Unprintable_Type, and is provided to simplify the programming of lightweight classes that do not require external representations.

Once a class has defined write, the template function

```cpp
operator<<(Str& s, const T& o)
```

defined in Str.h, automatically defines << to permit appending to its left argument the print form generated by write.

Notice that this template only allows appending to a Str, but not to an ostream object, such as cout. Str.h overloads << to append a Str to an ostream; thus to append the print form of an arbitrary object foo to an ostream, one could say

```cpp
cout << (Str() << foo); // NOT recommended;
```

The rightmost << is the print-form-to-Str appender, while the leftmost << is the Str-to-ostream appender. However, this will generate a compiler warning since it involves changing a temporary, and anyway it is clumsy — and confusing — since

```cpp
cout << Str() << foo; // ERROR!
```

will cause a compiler error, since the rightmost << would then be interpreted as the overload which appends to ostream, and there is no such overload defined which takes a right hand argument of the type of foo.6

So, in order to make it more convenient to append an object’s string form to an ostream, Str.h defines the template function inline Str str(const T& o), which returns (by copying) the print form of an arbitrary object, by encapsulating the operation (Str() << foo) above (but without modifying a temporary!). This enables the preferred form for output

---

6Except of course if the type foo is a built-in, in which case the overload has been defined by ostream, or if it is Str, which is the one overload we have defined.
cout << str(foo);

Note that str is *not* capitalized. Str(foo) would be a constructor for Str taking a single argument of the type of foo. If such a constructor existed, it would not be necessary to explicitly invoke it, since the compiler would do so automatically as a conversion of foo to type Str. This would admittedly lead to the even more streamlined form cout << foo, since << for ostream can append Strs, and foo would be implicitly converted to Str. However, we have not provided constructor/converters to Str for two reasons. First, if we adhered to that discipline, every class would have yet another function it would have to define: operator Str(). We want to minimize the overhead needed to write new classes. Second, we found that defining implicit conversions frequently led to undesired and unexpected conversions. Requiring one in essentially every class would likely result in annoying complications as a result of the constraints it would impose to avoid undesired conversions.⁷

8.3 Reading a string form: read and »

Since the print form of an object can recursively contain print forms of other objects, reading a print form is most naturally done using calls to the readers for sub-objects. This requires some way of maintaining the current state of the read operation. Our approach to this is to use an object of class StrIt, which maintains the state of iterating through a Str. Class StrIt is described in more detail in section 7.5. For the moment, what is important is that a StrIt knows what string it is iterating through, and keeps an index of where it currently is in that iteration.

In order to support reading from a string (Str or Char*), a class Foo must provide a friend function read with the signature

friend void read(Foo& f, StrIt& it);

Repeated applications of read to a StrIt object advance the StrIt index further along the Str which is being iterated through, thus reading additional objects from the Str. (And thus both arguments must be modifiable.) So, for example

Str s; s = "Vec(1 2 3)";    // make 's' be "Vec(1 2 3)" excluding quotes
StrIt si(s);               // make a StrIt 'si' pointing to 's'
Vec v; read(v, si);       // read the Vec 'v' from 'si'

⁷We also decided against another route that would have afforded a similar functionality: defining a template function with a left argument of type ostream. This can be done, but we discovered that because of the rules for instantiating templates, as well as the lack of standardisation among compilers, this would have required a large number of obscure definitions of templates for various ostream objects. We felt that the minimal extra convenience was not worth the considerable cluttering of the interface, and we also considered it likely that as the Microsoft C++ template facility evolved to conform to the evolving C++ ANSI draft standard, it would become necessary to keep altering our code to keep it working. Put more tersely, some aspects of templates are still a somewhat dark corner of the language and compiler, and so their limits are best not tested.
Or, illustrating the maintaining of \texttt{StrIt} state,

\begin{verbatim}
Str s;
s = "Vec(1 2 3)Vec(4 5 6)"; // make 's' be "Vec(1 2 3)Vec(4 5 6)"
StrIt si(s); // make a StrIt 'si' pointing to 's'
Vec v1; read(v1, si); // now v1 == Vec(1 2 3)
Vec v2; read(v2, si); // now v2 == Vec(4 5 6)
\end{verbatim}

If \texttt{read} encounters an error, it throws a \texttt{StrItEx} exception. Prematurely reaching the end of the string, or trying to read from a \texttt{StrIt} whose pointer is already at the end is considered an error, and throws such an exception.

Once a class has defined a \texttt{read} function, the \texttt{StrIt} iterator \texttt{operator>>} is automatically defined via a template function in \texttt{Str.h}. Then the above operation can equally well be done as

\begin{verbatim}
Str s;
s = "Vec(1 2 3)Vec(4 5 6)"; // make 's' be "Vec(1 2 3)Vec(4 5 6)"
StrIt si(s); // make a StrIt 'si' pointing to 's'
Vec v1; si >> v1; // now v1 == Vec(1 2 3)
Vec v2; si >> v2; // now v2 == Vec(4 5 6)
\end{verbatim}

Using the class's \texttt{StrIt} version of \texttt{read}, \texttt{Str.h} also defines template versions of \texttt{read} for \texttt{Char*} and \texttt{const Str8*} arguments, so that one can read directly from \texttt{Str}s and \texttt{Char*s}, rather than only from a \texttt{StrIt}. For example, instead of saying

\begin{verbatim}
Str s; s = "Vec(1 2 3)"; // make 's' be "Vec(1 2 3)" excluding quotes
StrIt si(s); // make a StrIt 'si' pointing to 's'
Vec v; read(v, si); // read the Vec 'v' from 'si'
\end{verbatim}

one can more simply say

\begin{verbatim}
Str s; s = "Vec(1 2 3)"; // make 's' be "Vec(1 2 3)" excluding quotes
Vec v; read(v, s); // read the Vec 'v' from 's'
\end{verbatim}

Note, though, that this does not permit maintaining state, so that if we tried the example of more than one print form in a single string, we would now have

\begin{verbatim}
Str s;
s = "Vec(1 2 3)Vec(4 5 6)"; // make 's' be "Vec(1 2 3)Vec(4 5 6)"
Vec v1; read(v1, s); // now v1 == Vec(1 2 3)
Vec v2; read(v2, s); // now v2 == Vec(1 2 3) *NOT* Vec(4 5 6)
\end{verbatim}

I.e., both \texttt{reads} from the \texttt{Str s} yielded the same \texttt{Vec v} that is the initial print form in \texttt{s}, since the \texttt{Str s} has no way of maintaining state — a temporary \texttt{StrIt} is created anew each time we read this way from a \texttt{Str}.

\texttt{Str.h} defines yet another template as well, which defines \texttt{operator>>} to \texttt{read} from \texttt{Str} and \texttt{Char*} based on the class definition for \texttt{read} from \texttt{StrIt}. This then allows us to replace the following
Str s; s = "Vec(1 2 3)";  // make 's' be "Vec(1 2 3)" excluding quotes
Vec v; read(v, s);     // read the Vec 'v' from 's'

with

Str s; s = "Vec(1 2 3)";  // make 's' be "Vec(1 2 3)" excluding quotes
Vec v; s >> v;            // read the Vec 'v' from 's'

And, of course the same caveat applies for the lack of state memory as above.

One of the benefits of having a string form reader for a class is that it makes it convenient to set variables to "literal" values. With the templates above, instead of (as above)

Str s; s = "Vec(1 2 3)";  // make 's' be "Vec(1 2 3)" excluding quotes
Vec v; s >> v;            // read the Vec 'v' from 's'

to set a Vec we can now even more simply write:

Vec v; "Vec(1 2 3)" >> v;  // read the Vec 'v'

This type of construction is largely the motivation for the templates we described above.8

8.3.1 StrItEx exceptions

As mentioned above, if read attempts to parse a portion of a string that is not properly formed for the type of object it is trying to read into, a StrItEx exception is thrown. The StrItEx object has members for the line number, character number within the line, a context string for printing the location of the error, and an error string describing the error. To throw a StrItEx object, the reader routine calls err() on the StrIt with an argument string that describes the problem. For example, Vec defines:

```cpp
void read(Vec& v, StrIt& it) {
    v.to_null();
    if (it.is_done()) it.err("Vec: nothing to read.");
    if (!it.check_str("Vec(")) it.err("Vec: wrong name.");
```

8We would have preferred to be able to use the syntax

Vec v = "Vec(1 2 3)";

and in fact, earlier versions of the library supported this syntax. However, in a declaration with initialization using =, the = is not an assignment operator, but rather indicates that its right hand side is to be used as the argument to a constructor which creates the left hand side. As far as C++ is concerned, the right hand side is a Char* literal (or, more accurately a char[]), so the only way to allow such a construction would be to define a constructor Vec(Char*). This had two drawbacks. First, it represents yet another function that every class would have to define, since there is no way to templatize a constructor. Second, since we could not declare this constructor explicit — indeed the construction uses it implicitly — we were subject to many problems arising from unwanted implicit conversions and ambiguous conversions.

41
Dbl d; Stk<Dbl> dbuf;
while(1) {
   it.skip_comments();
   if (it.check_str("")]) break;
   if (!it.check_dbl_in(d)) it.err("Vec: missing closing paren.");
   dbuf.push(d);
   v.make_dim(dbuf.size());
   for(Int i=0; i<v.dim(); i++) v.p[i]=dbuf[i];
}

9 Library classes

As we mentioned in the introduction, RCL's classes fall roughly into the following categories:

- Fundamental data structures
- Linear algebra
- Random numbers and probabilistic models
- Image processing and vision
- Geometric structures
- User interface
- Operating system interface

In this section, we discuss in varying degrees of detail the classes that are part of the library.

We group the classes into the above categories as follows:

- **Fundamental data structures**
  - **Arr** Arrays
  - **Stk** Stacks with array indexing
  - **Str** Strings
  - **StrIt** String iterators
  - **Vec1** Vectors with integer components
  - **Vec12** 2-dimensional vectors with integer components
  - **Sequence** Sequences — templates for Arr, Stk, Vec, Str, etc.
  - **Gph** Directed and undirected graphs
  - **IntSet** Sets of integers
  - **IntSetIt** Iterators through sets of integers
  - **IntMap** Maps from integers (tables)
  - **IntMapIt** Iterators through maps on integers
  - **Cpx** Complex numbers

---

9 Although Vec1 and Vec12 are in the category of fundamental data structures, they are discussed in the section about Vec in the linear algebra category, since the underlying representation is identical except for data type.
• Linear algebra
  Vec  Vectors
  Mat  Arbitrary matrices
  MatLU LU Matrices
  MatQR QR Matrices
  Flat n-dimensional affine subspaces
  MapAff Affine maps between vector spaces
  MatSym Symmetric matrices
  Vec2 2-dimensional vectors
  Vec3 3-dimensional vectors

• Random numbers and probabilistic models
  Rnd  Random number generators
  RndFun Special functions
  Histogram Histograms
  Mn1  Multinomials — random variables with \( n \) discrete values
  Dirichlet Dirichlet probability distributions
  Gsn  Gaussian distributions of several random variables
  Gsn1 Gaussian distributions of 1 random variable
  GsnMix Mixtures of Gaussians of several random variables
  GsnCnd Conditional multivariate Gaussian distributions
  GsnMixCnd Conditional Gaussian mixtures

• Image processing and vision
  Img  Monochrome images with 8 bit integer pixels
  ImgC  Color images with 24 bit pixels of 8 bits per color
  PxIC  24 bit color pixels of 8 bits per color

• Geometric structures
  Iv1  Interval arithmetic
  Ivli Intervals of integers
  Ball \( n \)-dimensional balls
  Box \( n \)-dimensional products of intervals

• User interface
  Tst  Testing tools
  Gui  Interface to Windows graphical user interface
  Gnuplot Interface to gnuplot plotting package
  Gnuplot3d Interface to gnuplot plotting package
  Easyplot Interface to gnuplot plotting package

• Operating system interface
  File General file I/O
  Timer Timing and time-of-day and date functions
  Sys Synchronous and asynchronous system commands

Each of the class header files contains documentation on the use of specific functions. We don’t, therefore, make an attempt here to exhaustively document
all the classes and all the functions they contain. Rather, we concentrate on key features and aspects that may not be obvious from reading the library files.

The Str class is probably the most basic, in the sense that all other classes make use of it. It was discussed above in section 7.

9.1 Fundamental data structures

9.1.1 The container classes Stk and Arr

Stacks and arrays are implemented by the classes Stk and Arr. They are template classes, so that one can construct a Stk or Arr of any kind of object, including built-in types such as Int, DbI, Char, and pointers, and use them with a uniform interface. For, example,

\[
\text{Stk<Int> si;}
\]

declares si to be a stack of integers. It allocates space on the program stack for the two Int's asz and sz, and for a pointer which points to the contents of the Stk si (and which is initially set to the null pointer).

\[
\text{Stk<Str> ss;}
\]

on the other hand, declares ss to be a stack of Str's.

Stk's are implemented with amortized doubling, so the programmer is relieved of the need to be concerned with allocating additional space as the stack grows. An optional constructor taking a single integer argument is provided, however, which specifies an initial allocation size for the new Stk. This is occasionally useful when one knows in advance the approximate size to which a stack will grow, and the overhead of amortized doubling is not acceptable.

Stk supports the routines push, pop, is_empty, and top with their standard meanings. top returns a reference to the top of the stack without modifying the stack, in contrast to pop, which returns a copy of the object at the top of the stack, removing it from the stack in the process.

In addition, some other operations that Stk<T> supports include:

\[
\text{void push_take(T& e) // 'take' the element 'e' and push it onto the} \n\text{top of the stack (this will null 'e').}
\]

\[
\text{void pop_take_in(T& e) // A version of 'pop' which avoids copying.} \n\text{\'e' takes the current top of the stack and the} \n\text{size of the stack is decremented.}
\]

\[
\text{T& operator[] (int i) // Index into the Stk, modifiable.}
\]

\[
\text{void remove_ind(Int ind) // Remove the element with index 'ind' and} \n\text{shift all later elements down.}
\]
void remove_val(const T& v)  // Remove all elements with the value 'v' and  
   // shift other elements down.

Bool push_if_new(const T& e)  // If 'e' isn't already on 'this', then push  
   // it and return true. Otherwise return false.

Bool push_take_if_new(T& e)   // If 'e' isn't already on 'this', then push  
   // it and return true. Otherwise return false.

Int remove_duplicates()       // Remove any duplicate elements.  
   // Return number removed.

push_take and pop_take_in provide versions of the push and pop opera­
    tions that avoid copying. Although this means that push_takeing an object onto the stack makes the original object null, while reincarnating it on the stack, it avoids any copy overhead, which can be prohibitive. Although pop_take_in also has the side-effect that the original object on the stack is destroyed, the same is true of pop; the difference is that by using an _in reference argument, pop_take_in is able to avoid copying, at the cost of losing the functional pro­
    gramming style afforded by the copied return argument of pop.

A feature of our use of C arrays for the implementation of stacks is the ability to index into the stack at constant cost, and this is frequently extremely useful. The [] operator is the indexing operator, and can be used as an lvalue, i.e., on the left hand side of an assignment (or in a modifiable argument) to change what is stored at that position in the Stk.

We also provide some additional functionality that is convenient for main­
    taining lightweight sets as stacks.

The class Arr defines arrays of arbitrary objects via a templated class im­
    plementation, similarly to Stk.

The default constructor for Arr, like all our default constructors, creates a null Arr, with no heap storage. Unlike Stks, Arrs cannot grow; but the member function void to_size(Int s) reallocates an Arr<T> to be of size s, i.e., allocates storage for s elements of type T. If T is a class, as opposed to a built-in type such as Int, to_size also guarantees that all of those elements are null, even if the Arr was already the right size and contained any number of non-null elements. In the case of built-in types, there is no canonical notion of null — just as there is no canonical initial state guaranteed by the compiler for their initialization — and consequently when T is a built-in type, after the operation to_size(s), the values of the Arr<T> elements are undefined and arbitrary.

Arr also defines a constructor of a single integer to specify the size of the new array. When this constructor is called, storage is allocated for the specified number of objects of the templated type T. It’s important to note, though, that this does not proceed recursively, i.e., even if the class T can also own storage, that storage is not allocated — only the Arr storage to hold n null objects of type T is allocated. We have found that it is a common source of bugs to tacitly
but erroneously assume that space has been allocated within these objects and
to attempt to assign to or reference it, typically when the class T supports an
indexing operator.

In addition to standard operations such as assignment, equality, etc., Arr
supports the fundamental array operation [ ]. This allows one to either extract
or set an element at a given array index. There are no special provisions for
multi-dimensional arrays, so there are no array types that take multiple indexes.
To define a multidimensional Arr, one does essentially what is done in standard
C:

\[
\text{Arr< Arr<T> > foo;} \\
\]

(Note, incidentally, that the space in >u> is essential: if this space is omitted,
the C++ compiler will parse >> as a right shift and issue an error.) The above
declares foo to be a 2-dimensional array of objects of type T, or rather an array
of arrays of objects of type T which we interpret as a 2-dimensional array. As
in C, one accesses an element of foo with the construct

\[
\text{T bar = foo[y][x];} \\
\]

Note that the x index represents the inner array.

9.1.2 Sequences: Sequence.h

The classes Arr<T>, Stk<T>, Str, Vec, and Veci are all examples of Sequences.
The file Sequence.h defines a set of template functions that can be applied to
any Sequence.

Beside supporting the indexing operator [], there are a few more things a
class must do to support the Sequence operations:

• It must define the member function size() which returns the number of
indexable elements.

• The indices must run consecutively from 0 through size()-1.

• To support sorting and certain mapping, it must include a typedef of the
form typedef T elt_type indicating the type of the indexed elements.
E.g., Vec defines

\[
\text{typedef Dbl elt_type; // Used by template routines, e.g. sort.} \\
\]

This is for the convenience of the template functions in Sequence.h in
determining the element type. Such a typedef later allows the syntax
S::elt_type x, where S is the class name, to declare x to be of the type
T that it was typedef'ed to.
The last point merits a fuller explanation. Certain of the Sequence template functions need to know the type of an element of the sequence acted upon, e.g., to declare a temporary. This could have been accomplished with additional arguments — which would lead to clutter and be unwieldy — or by using RTTI (run-time type information) constructs that theoretically should be evaluable at compile time. However, including RTTI can impose additional overhead for the whole program. Instead we opted for requiring another definition in any class desiring to be a Sequence.

Since this takes advantage of a somewhat obscure corner of the C++ language, here is how it works in more detail. In the specification of class Foo, which is a container of elements of type Bar, and which aspires to be a Sequence, one must write:

```cpp
class Foo
{
    ...
    // other stuff

typedef Bar elt_type;

    ...
    // other stuff
};
```

Then, in another place, e.g., in the templates in Sequence.h, one can say

```cpp
Foo::elt_type baz;
```

This is then exactly equivalent to saying

```cpp
Bar baz;
```

namely, it declares baz to be of type Foo::elt_type, which in class Foo was defined to be type Bar. The key is that elt_type, by virtue of its typedef within the class Foo, acquires a namespace modifier from the class. This works even when the type Foo is a template parameter T inside a template function.

The Sequence templates provide three main kinds of functionality:

- sorting
- searching
- mapping

There are also a few other routines which do not fit into these categories.

For sorting a Sequence, we provide insertion sort and quicksort. There is also the plain sort function, which simply calls quicksort. Quicksort uses insertion sort for sequences of size less than 10. A Sequence class's element type must define the operator <= or < in order for the Sequence to be sortable. (<= is what is explicitly used in the sort routines, but providing < generates an automatic definition of <= via templates defined in All.h.)
There is no provision for reverse-sorting; however, Sequence.h does provide the function reverse, which reverses a sequence.

We also provide the same sort functions supplemented with the ability to permute a second sequence in parallel to the sort of the first, where the order comparisons are applied only to the first sequence. This is similar to a key sort. One can use this directly to sort a second sequence that should maintain a fixed relationship to the key sequence. Or, viewing a sort as a permutation of a sequence, one can record what that permutation is. The Veci class supports permutation operations, so if the second Sequence is a Veci representing the identity permutation (created, e.g., by calling the Veci member function to_identity_perm()), the sort-permute functions will rearrange it to represent the permutation performed by the sort.

Sequence.h also contains the permute function, which can apply a Veci permutation, gotten from a sort-permute or elsewhere, to an arbitrary Sequence.

Sequence searching is quite rudimentary: we simply perform a sequential search. search requires that \(==\) be defined by the element class of the sequence to be searched. If there are duplicates, search will always find the first matching element, since subsequent searches restart from the beginning. However, we also provide the function has_duplicates. Stk defines remove-duplicates, but there is no satisfactory canonical way for all Sequences to support this operation, so it is not provided in Sequence.h.

The Sequence mapping functions distribute a function over the elements of a sequence; this is “mapping” in the Lisp sense of the word, à la mapcar. One could, for example, take the square root of every element of an Arr of Dbls by mapping the function sqrt. There are versions of these for member as well as non-member or static functions, and they can be used with both function pointers and function objects (Musser and Saini, 1996).

Sequence.h also defines

```
rnd_perm(s)       // Randomly permute the entries in 's'
take_elts(s1, s2) // 's1' takes all the elts of 's2'
take_append(s1, s2) // 'take' the elts of 's2' and append them to 's1'
                   // (no copying is needed)
to_const(s, const t) // Set each element of 's' to be 't'.
ensure_size(s, z)  // Ensure that 's' is of size 'z'.
                   // 's' is unchanged if it is already of size 'z',
                   // else it is reallocated.
```

### 9.1.3 Directed graphs: Gph

The class Gph implements algorithms for directed graphs. The implementation uses an adjacency list representation, where a graph is a Stk of edge lists, and each edge list is in turn a Stk of Ints. These Ints simply represent the nodes of the graph.

Because Stks allow random access with [], and are based on C arrays, indexing into an edge list is cheap and straightforward.
Gph provides routines for adding and deleting vertices and edges, as well as checking whether the graph can be viewed as undirected.

Additionally, topological sort, transitive closure, and clique creation are supported.

Finally, there are several random graph operations. These include creating a random directed graph with a fixed number of vertices and constant out degree, and a random graph with a fixed number of vertices and a given probability of containing each possible edge. There are similar routines for random undirected graphs and DAGs (directed acyclic graphs).

9.1.4 Sets of integers: IntSet and IntSetIt

IntSet.h defines the classes IntSet and IntSetIt, which provide arbitrary size sets of integers, and a way to iterate through the members of such a set, resp. IntSets are implemented using hash tables based on arrays with amortized doubling. IntSet operations are quite fast, which is largely a consequence of the simple hash function that we use: we simply use the low-order bits as the hash value, which of course can be computed in a single AND instruction. This works quite well in practice. We have found that the amortized cost of handling collisions is less than the penalty for using an expensive hash function on every access.

IntSet supports the standard set operations, and allows adding and removing elements. Efficient support for deletion is another strong argument for using a simple approach to collision resolution.

IntSetIt allows iterating through the elements of an IntSet by using the overloaded operator ». An IntSetIt iterator is created by construction from an IntSet, e.g.:

```
IntSet is;
...               // operations on 'is'
IntSetIt isi(is);  // create iterator for 'is'
```

Then to use the iterator, calls to the operator » produce successive elements of the IntSet, finally returning false when all elements have been produced. E.g.:

```
Int i;
while(isi » i) cout << i << \n;
```

will print all the elements of the IntSet is. » produces the elements in an arbitrary order that depends on the state of the IntSet's internal data structure.

9.1.5 Tables; maps from integer sets: IntMap<T>, IntMapIt<T>

The IntSet technology is easily extended to include maps from integer sets to objects of an arbitrary class. This is accomplished with the template class IntMap<T>. The type T specifies the class of the range; for a given IntMap<T> the range consists of objects of only the single class T.
IntMaps provide functions to record and delete (key, value) pairs, as well as functions to return a value given a key, i.e., to evaluate the integer map. One of these is simply operator (), which takes an integer as an argument and returns a reference to the object that integer is mapped to by the IntMap.

IntMapIts operate analogously to IntSetIts, except that they return a (key, value) pair, putting it into an IntMapEntry<T> object.

9.1.6 Complex numbers: Cpx

The class Cpx represents complex numbers. It supports simple operations on complex numbers, and also computes trigonometric functions, logarithms, exponentials, hyperbolic functions, and the like in the complex domain.

9.2 Linear algebra

9.2.1 Vectors: Vec, Veci, Vec3, Vec2, Veci2

The basic real vector class is Vec. Veci is a vector of integers. The classes Vec3, Vec2, and Veci2 are lower dimensional special cases of Vec and Veci, resp., which exist primarily to provide efficiency gains for these common special cases. They also permit some savings in programmer effort.

Vec, Vec3, Vec2

A Vec is based on a C array of doubles, and is therefore essentially the same as Arr<Db1>. It has a special name because vectors are so important and because there are many special functions which are applicable to them that should be defined as member functions. There is no reasonable way to add such routines to a templated class such as Arr<T>. In applications, it is more natural to think in terms of vectors rather than arrays of doubles.

Vecs can be of arbitrary dimension, and can be reset to a different dimension, but they do not grow or shrink like Stks: they own heap storage that is fixed once it is allocated, though the storage can be released and new fixed storage allocated of a different size.

Following are the vector-specific operations supported by Vec:

```c
void to_dim(Int d);               // Make 'this' be the origin in dim 'd'.
Bool ensure_dim(Int d)            // Make sure 'this' has dim 'd'.
void to_subvector(const Vec& v, Int st, Int num) // Make 'this' be the subvector of 'v' of
                                                    // length 'num' starting at 'st'.
void set_subvector(Int st, const Vec& v)   // Set the subvector of 'this' starting at
                                          // 'st' to 'v'.
Int    dim()                       // Dimension of 'this'.
void add(const Vec& v);            // Make 'this' be its sum with 'v'.
void subtract(const Vec& v);       // Subtract 'v' from 'this'.
```
Dbl dot(const Vec& v) const; // The dot product of 'this' and 'v'.
void scale(Dbl s); // Scale 'this' by 's'.
void scaled_add(Dbl s, const Vec& v); // Add 's' times 'v' to 'this'.
void elt_times(const Vec& v); // Multiply each element of 'this' by the corresponding element of 'v'.
void to_const(Dbl s); // Make each component of 'this' be 's'.
void to_zero(); // Make 'this' be the origin.
void to_ones(); // Set each component of 'this' to 1.0.
void to_uniform_rnd(); // Make 'this' be a uniform sample from the cube defined by each coordinate in [0,1).
void to_uniform_rnd(Int n); // Make 'this' be a uniform sample from the n-dimensional cube with each coordinate in [0,1).
void to_ball_rnd(); // Make 'this' be a uniform sample from the ball of radius 1 centered at the origin.
void to_unit_rnd(); // Make 'this' be a unit vector sampled uniformly from the sphere of radius 1 centered at the origin.
void to_normal_rnd(); // Make 'this' be a vector sampled from a spherically symmetric Gaussian with standard deviation 1 and centered at the origin.
void to_simplex_rnd(); // Make 'this' be a vector sampled uniformly from the basic simplex defined by the origin and each of the positive unit vectors along the axes.
void to_simplex_rnd(const Arr<Vec>& va); // Make 'this' be a vector sampled uniformly from the simplex defined by the vectors in 'va', which must be in general position.
void to_cantor_rnd(); // Make 'this' be a sample from the Cantor distribution in the cube defined by each coordinate in [0,1).
Dbl elt_sum() const; // The sum of the elements of 'this'.
Dbl elt_product() const; // The product of the elements of 'this'.
Bool reciprocal(); // Set each element of 'this' to its reciprocal. 'true' if successful, 'false' if some element vanished.
void square(); // Set each element of 'this' to its square.
Bool sqrt(); // Set each element of 'this' to its square root. 'true' if successful, 'false' if some element was negative.
Dbl length2() const; // The square of the Euclidean length of 'this'.
Dbl length() const; // The Euclidean length of 'this'.

51
Dbl l1_norm() const; // The L1 norm of 'this'.
void normalize(); // Scale 'this' to be unit length.
Bool is_normalized() const // True if 'this' is approximately normalized.
Dbl distance2(const Vec& v) const;
   // The square of the Euclidean distance from 'this' to 'v'.
Dbl distance(const Vec& v) const
   // The Euclidean distance from 'this' to 'v'.
Dbl bounded_distance2(const Vec& v, Dbl sbnd) const;
   // The Euclidean distance from 'this' to 'v' if its square is <= 'sbnd'.
   // -1.0 if it is greater than it. Can often avoid some operations if used in a
   // bounding test.
void midpoint(const Vec& v); // Make 'this' be the midpoint between its
   // current location and 'v'.
void away_from(const Vec& v, Dbl d);
   // Move 'this' a distance 'd' in the direction away from 'v'.
void toward(const Vec& v, Dbl d) // Move 'this' a distance 'd' toward 'v'.
void to_interpolate_between(const Vec& v0, const Vec& v1, Dbl t);
   // Set 'this' to linearly interpolate between 'v0' and 'v1' such that when
   // 't==0.0' it is 'v0' and when 't==1.0'
   // it is 'v1'.
Int max_index() const; // The index of the maximum component of 'this'.
Dbl max_value() const; // The value of the maximum component of 'this'.
Int min_index() const; // The index of the minimum component of 'this'.
Dbl min_value() const; // The value of the minimum component of 'this'.
void truncate_min(Dbl m); // Set any element of 'this' which is
   // less than 'm' to 'm'.
void truncate_max(Dbl m); // Set any element of 'this' which is
   // greater than 'm' to 'm'.
Dbl angle(const Vec& v) const; // The angle between 'this' and 'v' in radians.
void make_orthogonal_to_unit(const Vec& v);
   // Subtract off the projection of 'this' onto the unit vector 'v'.
void reflect_through_zero(); // Reflect 'this' through the origin.
Bool is_approx(const Vec& v, Dbl tol=DBL_EPS) const;
   // True if the components of 'this' are within 'tol' of the components of 'v'.
52
Db1 sup_norm() const; // The sup norm of 'this'
    // (i.e. max absolute value of the components.)
Db1 p_norm(Db1 pv) const; // The p-norm of 'this'.

void update_mean_in(Db1 &sz, Vec &mv) const;
    // Assuming 'sz' holds the number of vectors
    // seen so far, and 'mv' holds their current
    // mean, update these to reflect the new
    // vector 'this'.

Db1 beta(); // The multi-dimensional generalization of
    // the Beta function applied to the
    // components of 'this'.

Db1 mean() const; // The mean of the elements in 'this'.

void mean_variance_in(Db1 &m, Db1 &v) const;
    // Fill 'm' and 'v' with the mean and variance
    // of the scalar samples in 'this'.
    // The variance is computed with the "N-1"
    // denominator appropriate for Gaussian
    // estimation.

void ml_mean_variance_in(Db1 &m, Db1 &v) const;
    // Fill 'm' and 'v' with the mean and variance
    // of the scalar samples in 'this'.
    // The variance is the maximum likelihood
    // variance computed with the "N" denominator.

void weighted_ml_mean_variance_in(const Vec &wa, Db1 &w, Db1 &m, Db1 &v) const;
    // Fill 'w' with the total weight in 'wa',
    // 'm' with the weighted mean of the samples
    // in 'this' and 'v' with the weighted maximum
    // likelihood variance of the samples in
    // 'this'.

void moments12_in(Db1 &m1, Db1 &m2) const;
    // Treating 'this' as a set of real samples,
    // put the first two moments in 'm1' and 'm2'.
    // 'm1' is the mean <s> and 'm2' is the
    // second moment <s^2>.

void moments123_in(Db1 &m1, Db1 &m2, Db1 &m3) const;
    // Treating 'this' as a set of real samples,
    // put the first three moments in
    // 'm1', 'm2', and 'm3'. 'm1' is the mean <s>,
    // 'm2' is the second moment <s^2>, and 'm3'
    // is <s^3>.

void moments1234_in(Db1 &m1, Db1 &m2, Db1 &m3, Db1 &m4) const;
    // Treating 'this' as a set of real samples,
    // put the first four moments in
    // 'm1', 'm2', 'm3', and 'm4'. 'm1' is the
    // mean <s>, 'm2' is the second moment <s^2>,
    // 'm3' is <s^3>, and 'm4' is <s^4>.
Vec also defines several non-member functions which apply to arrays of vectors. These are:

```cpp
void to_size(Arr<Vec>& va, Int s, Int d);
   // Make 'va' be of size 's' and fill it with zero vectors of dimension 'd'.

void to_vec(Arr<Vec>& va, const Vec& v);
   // Set each element of 'this' to a copy of 'v'.

void to_zero(Arr<Vec>& va);  // Set each element of 'this' to the zero vector.

void to_uniform_rnd(Arr<Vec>& va);
   // Make each element of 'this' be a random vector.

void to_uniform_rnd(Arr<Vec>& va, Int s, Int d);
   // Make 'this' be of size 's' and fill it with uniform random vectors of
   // dimension 'd'.

void sum_in(const Arr<Vec>& va, Vec& v);
   // Make 'v' be the sum of the vectors in 'va'.

void mean_in(const Arr<Vec>& va, Vec& v);
   // Make 'v' be the mean of the vectors in 'va'.

void weighted_mean_in(const Arr<Vec>& va, const Vec& w, Vec& v);
   // Make 'v' be the mean of the vectors in 'va' weighted by 'w'.

void mean_covariance_in(const Arr<Vec>& va, Vec& mn, Mat& cov);
   // Fill 'mn' and 'cov' with the mean and covariance matrix of the
   // samples in 'this'. The covariance matrix is computed with the
   // "N-1" denominator appropriate for Gaussian estimation. Choices a
   // covariance of 1 if there are 0 or 1 samples, chooses the mean to
   // be the origin if there are 0 samples.

void ml_mean_covariance_in(const Arr<Vec>& va, Vec& mn, Mat& cov);
   // Fill 'mn' and 'cov' with the mean and covariance matrix of the
   // samples in 'this'. The covariance matrix is computed with the
   // "N" denominator appropriate for maximum likelihood estimation.
   // Chooses a covariance of 1 if there are 0 or 1 samples, chooses
   // the mean to be the origin if there are 0 samples.

void weighted_ml_mean_covariance_in(const Arr<Vec>& va,
                                            const Vec& w, Vec& mn, Mat& cov);
   // Fill 'mn' with the weighted mean of the samples in 'this' and
   // 'cov' with the weighted maximum likelihood covariance of the samples
   // in 'this'. Chooses a covariance of 1 if there are 0 or 1 samples,
```
// chooses the mean to be the origin if there are 0 samples.

void add(Arr<Vec>& va1, const Arr<Vec>& va2);
// Add each vector in 'va2' into the corresponding vector in 'va1'.

void add_vec(Arr<Vec>& va, const Vec& v);
// Add 'v' to each vector in 'va'.

void scale(Arr<Vec>& va, Dbl s);
// Scale each vector in 'va' by 's'.

Int closest_index(const Arr<Vec>& va, const Vec& v);
// Return the index of the first closest vector in 'va' to 'v'.

Dbl mean_nearest_distance2(const Arr<Vec>& va);
// The mean square distance from each vector in 'va' to its nearest
// neighbor.

Int kmeans_in(const Arr<Vec>& va, Arr<Vec>& vc, Int iters);
// Perform "k-means" clustering of the points in 'va' on the cluster
// centers in 'vc'. Perform a maximum of 'iters' iterations [default 50].
// Return the number of iterations til convergence. -1 if didn’t
// converge.

Int kmeans_in(const Arr<Vec>& va, Arr<Vec>& vc, Veci& cl, Int iters);
// Perform "k-means" clustering of the points in 'va' on the cluster
// centers in 'vc'. Perform a maximum of 'iters' iterations [default 50].
// Set 'cl' to the cluster labels of the vectors in 'va'.
// Return the number of iterations til convergence. -1 if didn’t
// converge.

The routine to_size(va, s, d) provides the convenience of populating an
Arr<Vec> with Vecs which already have storage allocated for their components.
Vec2 and Vec3 also define a few operations that are specific to their dimension. For example, Vec2 defines a rotate-by-90-degrees operation, while Vec3
defines cross-product.

Veci and Veci2

The Veci class defines arbitrary dimension vectors with integer components. These support the bulk of the Vec operations that make sense for
integer vectors, such as max_index(), as well as some special ones such as
to_rnd_uniform_up_to(Int m), which sets the Veci this to n integers sampled
uniformly (with replacement) from the interval [0, m] (where n is the size() of
the Veci).

Veci also supports several operations related to permutations. A Veci can
be considered to represent a permutation if its components consist of exactly
the integers 0, ..., size() - 1, in arbitrary order. Veci provides is_perm() to test whether a Veci belongs to that subclass. Also, the following functions are available:

```cpp
void rnd_perm(); // Randomly permute the entries in 'this'.
Bool is_perm() const; // True if 'this' is a permutation
void to_identity_perm(); // Make 'this' be the identity permutation
void to_rnd_perm(); // Make 'this' be a random permutation
void to_product_perm(); // Make 'this' be the product of two
                       // permutation args.
void to_inverse_perm(); // Make 'this' be the inverse of the
                       // permutation arg.
Int rank_of_perm(); // Rank in the Johnson-Trotter ordering.
void to_rank_perm(); // Make 'this' be of arg rank.
Bool is_even_perm(); // True if 'this' is an even permutation.
```

Of these, only rank_of_perm() and is_even_perm() actually require that this already be a permutation. For the rank computations, the Johnson-Trotter ordering is described in (Stanton and White, 1986).

Veci2 is a minimal class that provides operations on integer points in the plane. It is convenient for operations on discrete grids, such as pixel arrays.

### 9.2.2 Matrices: Mat, MatLU, MatQR, MatSym

Mat.h defines the classes Mat, MatLU, and MatQR. MatSym, which is specialized for efficiency with symmetric matrices, is defined separately in MatSym.h.

Mat defines a considerable number of matrix operations, ranging from elementary ones such as addition, subtraction, scalar product, product with matrix, and action on a vector, through more complex ones such as setting the diagonal, extracting a row or column, extracting a submatrix, taking outer products of vectors, computing the Frobenius norm, checking for symmetry, anti-symmetry, and orthogonality, and exchanging rows or columns, to finding determinants, solving linear equations, least squares, computing eigenvalues and eigenvectors, singular value decompositions, LU and QR decompositions, and generating random matrices.

MatSym implements outer product operations using the special BLAS routines for symmetric matrices. Taking outer products is a major part of the computational load when computing covariances. In computing covariance, one takes the outer product of a sample vector with itself to yield a symmetric update to a symmetric (unnormalized) covariance matrix. Taking advantage of the special properties of symmetric matrices — not the least of which is simply that about half the entries can be ignored — yields considerable speedup.

### 9.2.3 Affine subspaces: Flat

The Flat class represents affine subspaces of arbitrary (finite-dimensional) vector spaces, sometimes known as flats. Flat can compute the distance of a flat
from a point, and can also fit a flat to a set of points.

9.2.4 Affine maps: MapAff

The class MapAff implements affine maps between arbitrary dimensional vector spaces, i.e., maps consisting of a linear map and a translation. This includes routines that fit an affine map to a set of argument-value pairs, as well as ones to generate random maps.

9.3 Random numbers and probabilistic models

9.3.1 Random numbers: Rnd

The Rnd class, defined in Rnd.h, is the basis for the generation of samples from all probability distributions in the library.

Random number generation is a very tricky business with many perils and pitfalls (Knuth, 1973; Park and Miller, 1988). The library’s primary generator comes from (Knuth, 1993) and is based on the recurrence

\[ a_n = (a_{n-24} - a_{n-55}) \mod 2^{31}. \]

It has a period of at least \(2^{55} - 1\), and conjectured to be \(2^{85} - 2^{30}\) for all but one seed value.

Rnd.h defines the classes ARndGen, RndGenMS, RndGenGB, Rnd, and RndFun. The most common user-level classes are Rnd and RndFun, so we describe those first.

Rnd is an objectless class, i.e., it exists to provide a namespace so that calls to its routines take the form Rnd::routine_name(). By default, Rnd uses Knuth’s Stanford Graph Base generator mentioned above, with a standard seed. Facilities are provided however, to instead use the “minimal standard” generator (Lewis et al., 1969; Park and Miller, 1988), and to seed either generator with a seed derived from the current clock time (expressed as the number of seconds since 00:00:00 GMT, Jan. 1, 1970).

Rnd contains routines for computing the following:

- uniform DBl samples from \([0, 1)\)
- uniform DBl samples from \([l, u)\), \(l, u \in \mathbb{R}\)
- uniform DBl samples from \([0, u)\), \(u \in \mathbb{R}\)
- uniform Int samples from \([l, u)\), \(l, u \in \mathbb{Z}\)
- uniform Int samples from \([0, u)\), \(u \in \mathbb{Z}\)
- bool samples with a given probability of true
- samples from a normal distribution with mean 0 and variance 1

\[ p(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \]
• samples from a normal distribution with given mean and variance
\[ p(x) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \]

• samples from the exponential distribution
\[ p(x) = \frac{1}{\lambda} e^{-\frac{x}{\lambda}} \]

• samples from the gamma distribution
\[ p(x) = \frac{x^{\alpha-1} e^{-x}}{\Gamma(\alpha)} \]

• samples from the beta distribution
\[ p(x) = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1}(1-x)^{\beta-1} \]

• samples from the chi-square distribution
\[ p(x) = e^{-\frac{x}{2}} \frac{x^{\frac{\nu}{2}-1}}{2^{\frac{\nu}{2}}\Gamma\left(\frac{\nu}{2}\right)} \]

• samples from the F distribution with given degrees of freedom
\[ p(x) = \frac{\Gamma\left(\frac{\nu_1 + \nu_2}{2}\right)\nu_1^{\frac{\nu_1}{2}}\nu_2^{\frac{\nu_2}{2}}}{\Gamma\left(\frac{\nu_1}{2}\right)\Gamma\left(\frac{\nu_2}{2}\right)} x^{\frac{\nu_1}{2}-1}(\nu_2 + \nu_1x)^{-\frac{\nu_1 + \nu_2}{2}} \]

• samples from the t distribution with given degrees of freedom
\[ p(x) = \frac{1}{\sqrt{\pi\nu}} \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\Gamma\left(\frac{\nu}{2}\right)} \left(1 + \frac{x^2}{\nu}\right)^{-\frac{\nu+1}{2}} \]

• samples from the geometric distribution
\[ P(n) = p(1-p)^{n-1} \]

• samples from the binomial distribution
\[ P(t) = \binom{n}{t} p^t (1-p)^{n-t} \]

• samples from the Poisson distribution
\[ p(j) = \frac{x^j e^{-x}}{j!} \]
- samples from the Cauchy (Lorentzian) distribution
\[ p(x) = \frac{1}{\pi(1 + x^2)} \]
- samples drawn uniformly from a Cantor set.

Also, Rnd provides a computation of the numerical value of each of the above densities at a given point.

The Dirichlet distribution is represented in a separate class, Dirichlet, described in section 9.3.4.

RndFun provides the following functions:
- gamma function
- natural log of the gamma function
- factorial
- natural log of the factorial
- binomial coefficient
\[
\binom{n}{k} = \frac{n!}{k!(n-k)!}
\]
- beta function
\[
\frac{\Gamma(z)\Gamma(w)}{\Gamma(z+w)}
\]

Using other random number generators: ARndGen, RndGenMS, RndGenGB

The class Rnd maintains a state which specifies which random number generator object to use to generate the next sample. That generator object maintains its own generator state which it updates to compute the next sample. All random number generator objects descend from the abstract class ARndGen.

The state of Rnd is implemented by maintaining a protected class static variable Rnd::gen of type ARndGen* which holds a pointer to the random number generator to use to produce the next sample, i.e., the basic uniform generator that provides the uniform sample which all the other distribution samplers use. By default, Rnd::gen is set to point to a generator object of class RndGenGB, which is a Graph Base generator, as described above.

Rnd::gen can be set to point to any other random generator object that descends from ARndGen, by using the function Rnd::swap_gen(ARndGen*& ngen). E.g., the class RndGenMS implements the “minimal standard” generator. If one wished to use this generator instead of the Graph Base generator, one could say

ARndGen* rg = new RndGenMS;
Rnd::swap_gen(&rg); // swap rg <-> Rnd::gen
This allocates a "minimal standard" generator \(rg\) (of class RndGenMS) on the heap and then swaps \(rg\) with Rnd::gen. After the swap operation, Rnd will use the RndGenMS object that was created in the first statement. Since there is generally no way to know what type of generator was previously being used, or what state it was in, when code is finished using a new generator it should swap back the old generator and delete the one it has allocated:

\[
\text{Rnd::swap_gen(&rg); // swap rg ---> Rnd::gen delete rg;}
\]

One can define one's own generator to be used with Rnd by inheriting from the abstract random number generator class ARndGen.

### 9.3.2 Histograms

The Histogram class represents histograms. A Histogram lets you successively hand it real (Db1) samples; it keeps track of how many it has seen, and counts how many fell into each of a set of specified bins. These equally subdivide any real interval. Histogram will also compute interpolated densities with error bars estimated from the bin values, and put its counts into arrays suitable for plotting with the Gnuplot interface.

### 9.3.3 Multinomials: Mnl

The class Mnl represents multinomial probability distributions, i.e., distributions over a finite discrete set of outcomes, such as the rolls of dice.

The representation uses a Vec of dimension \(n\) to represent the probabilities of each of the \(n\) possible outcomes of a discrete trial. Like other distributions, Mnl can be set to a random multinomial on \(n\) outcomes, can generate samples, one at a time or in a group, and will return the probability of a particular sample, or the joint probability of a set of samples. There are also statistical measures related to counts of each sample type, as well as computation of entropy, Kullback-Leibler distance (relative entropy), and Hellinger distance. Mnls are also useful for representing mixtures of distributions. (Devroye, 1986) was a source of some of the algorithms used in this class.

### 9.3.4 Dirichlet probability distributions: Dirichlet

The Dirichlet class represents Dirichlet probability distributions, given by

\[
p(x) = \frac{\Gamma \left( \sum_{i=0}^{k} \alpha_i \right)}{\prod_{i=0}^{k} \Gamma(\alpha_i)} \prod_{i=0}^{k} x_i^{\alpha_i - 1}
\]

Dirichlet contains routines that make it convenient to use Dirichlet distributions for Bayesian analysis, such as updating a prior distribution based on a sample or set of samples to get a posterior distribution.
9.3.5 Gaussians: Gsn, Gsn1, and GsnCnd

The class Gsn defines objects which represent multivariate Gaussian probability distributions. Gsn1 deals with the special case of Gaussians of a single random variable in the class Gsn1.

Gsn implements a number of standard statistical operations (Anderson, 1984; Cover and Thomas, 1991; Hand, 1981; Jolliffe, 1986; Kay, 1993); however, the underlying representation is unique to this library.

We represent a Gaussian with the following 4 objects:

```cpp
Vec mn; // mean
Mat dir; // first k principal axes of error ellipsoid
Vec sd; // standard deviations of first k principal axes
Dbl ssd; // spherical standard deviation for orthogonal subspace
```

The most notable feature of this representation is that we allow an arbitrary number of dimensions to be approximated by a spherical Gaussian. This can lead to enormous performance gains in many situations, for example when a significant number of dimensions have a very small variance. Our experience has been that this happens quite often in applications, where all the action tends to occur on a lower-dimensional submanifold of a high-dimensional space. When the embedding dimension is in the hundreds, this can be crucial.

The columns of the matrix dir represent the principal axes of the Gaussian in the subspace which is not being represented by a spherical Gaussian, and sd represents the standard deviations in the principal directions. ssd represents the single standard deviation of the spherical Gaussian on the spherical subspace — which is the orthogonal complement to the span of the columns of dir.

Gsn provides routines to fit Gaussians to given covariances, as well as to given sample sets, including weighted samples. It can also compute the

- product of two Gaussians
- convolution of two Gaussians
- image of a Gaussian under an affine change of coordinates
- entropy of a Gaussian
- Kullback-Leibler distance between two Gaussians
- conditionalization of a Gaussian on any subset of coordinates
- marginal of a Gaussian on any subset of coordinates

The class GsnCnd, defined in GsnCnd.h, provides a computationally efficient mechanism for repeated conditionalizations of the same Gaussian. This class exploits the fact that the covariance of a conditionalized Gaussian does not depend on the value of the conditioning variable, only on the coordinates conditionalized over. (The mean does, however, depend on that value.) If many conditional
distributions must be computed based on the same Gaussian, this can provide considerable economy. GsnCnd essentially implements a Gaussian which takes a parameter representing the value at which it has been conditionalized.

9.3.6 Gaussian mixtures: GsnMix and GsnMixCnd

GsnMix.h defines the interface for the classes GsnMix and GsnMixCnd, which implement routines for arbitrary dimensional Gaussian mixture distributions.

A mixture of \( n \) distributions is a distribution where each sample has a fixed probability of being drawn from one of the \( n \) distributions. The probability density of the mixture is then just a linear combination of the probability densities of the component distributions, with coefficients summing to \( 1 \). This can be thought of as consisting of two parts: a multinomial distribution on \( n \) outcomes, which specifies the probability of each of the \( n \) component distributions, and the ordered \( n \)-tuple of the \( n \) distributions themselves.

A Gaussian mixture is simply a mixture where the \( n \) distributions are all Gaussians, and GsnMix in fact implements the mixture as a Mnl and an Arr of Gsns.

GsnMix provides routines for generating samples from Gaussian mixtures, computing the probability of a sample under a given Gaussian mixture, computing conditional and marginal Gaussian mixtures, and fitting a Gaussian mixture to a set of samples. Fitting a Gaussian mixture is a nontrivial process. We implement it by performing an EM (Expectation-Maximization) (Dempster et al., 1977) optimization on the parameters of the mixture. In order to start the EM process in a good place, GsnMix first uses a k-means clustering step for an initial fit.

GsnMix also provides some miscellaneous functions, such as computing the entropy of a mixture by a montecarlo method.

GsnMixCnd provides computational improvements for dealing with conditionalized Gaussian mixtures, analogously to GsnCnd.

9.4 Image processing and vision

9.4.1 Images: Img, Imgc, and Pxlc

Img provides monochrome images with Uchar pixels; Imgc provides color images with pixels of class Pxlc, which consist of 3 Uchar RGB values. These image and pixel types are defined separately in header files of the same name.

These Uchar-based images are primarily for the purpose of containing information that comes from or is headed for some other source, such as a file or a display. The pixels have the range \([0, 255]\) representing the intensities from black to white (or black to full intensity red, green or blue). The classes provide routines to convert between Img or Imgc patches and Vecs which map between the Uchar range \([0, 255]\) and the Db1 range \([0.0, 1.0]\).

The Img* classes provide basic image functionality, such as indexing into images by \((x, y)\) coordinates, both for reading and writing, extracting height
and width of images, setting a subimage to some image, setting an entire image to a constant pixel value, etc. Additionally, images can be read and written to files as DIBs (device-independent bitmaps), also known as .bmp files, which are the Windows native image format. The library also provides classes for reading and writing JPEG images, using the Independent JPEG Group's LIBJPEG library, which is redistributed with RCL. The freely available PBM (portable bitmap) package provides routines for converting among a large number of image formats, including .bmp and TIFF. The Image classes also can write an image to an encapsulated postscript file.

The image classes provide some basic statistical functions, such as generating random images, and computing pixel means and covariances over an image. A few basic graphics functions that draw lines, circles, or dots directly into images are also provided. Image types can be converted among each other.

The file PxlColornames.h provides 752 predefined Pxl colors corresponding to the standard X-windows color names.

We also provide some functions for creating a tiling of other images into a single Image or Img object.

Displaying of images is handled by the Gui class, described in section 9.6.2.

9.5 Geometric structures

9.5.1 Intervals and interval arithmetic: Iv1, Iv1i

The class Iv1 implements real intervals and real interval arithmetic based on Dbls. An Iv1 represents the closed segment \([l, u]\), where \(-\infty \leq l \leq u \leq \infty\). We support not only the standard set-theoretic and arithmetic operations on intervals, but also interval arithmetic (Neumaier, 1990; Hammer et al., 1993) which permits provably true computations using finite-precision computations.

Iv1 supports the interval arithmetic functions + (sum), - (subtract), neg (negate), \(*\) (scalar multiply or Iv1 multiply), abs (absolute value), sqrt, square, cube, exp (e to the power of an Iv1), log, reciprocal, powk (Iv1 raised to an integral power), pow (an Iv1 raised to an Iv1 power).

The class Iv1i represents intervals of integers. Since interval arithmetic has no nontrivial analog for integers, this class is concerned simply with sets of integers of the form \([l, u]\), where \(u\) and \(l\) are Ints.

9.5.2 Hyperrectangles and hyperballs: Box and Ball

The class Box represents arbitrary dimension hyperrectangles that are aligned with the coordinate axes. A Box can be thought of as the Cartesian product of \(n\) intervals, and in fact it is internally represented as an array of Iv1s.

Boxes are useful as bounding boxes of sets of points, in addition to being the building blocks of representations of higher dimensional structures, such as boxtrees.

Balls are similar constructs, defined as the closed set of points within a given radius of a center point, i.e., a filled sphere.
9.6 User interface

9.6.1 Testing: Tst.h

Every library class is tested in a corresponding test module. The test module provides at least one test for each routine. We maintain these tests in a test directory and rerun them whenever changes are made, thus guarding against breaking code that previously worked.

C++ does not offer testing convenience like that of Lisp systems where one can submit a function invocation to the interpreter from an editor; rather one must follow the usual edit-compile-test-edit cycle. The tedium of this process unfortunately can discourage good testing hygiene, and many bugs slip by for want of testing at the proper stage.

Both of these processes are simplified with the TST macro which is defined in Tst.h. TST takes two arguments: the expression you want to check, and the value it should have. TST is silent as long as the test passes, but if it fails, a message is printed on standard output indicating the line number of the test, what the expression’s value was, and what it should have been.

TST permits either of its arguments to be the print form of any object; this is then tested against an expression that produces an object of that type. This is convenient for testing against fixed literal values, as is typical in tests. E.g.,

Vec v;
TST(v, "Vec()");

results in a passed test.

TST works by converting both of its arguments to Strs, and then comparing whether the Strs are the same. For non-string arguments, it does this by generating the print form of the object(s) and comparing the print forms as Strs. Occasionally, this leads to a situation which requires some care. For example, in certain classes, such as IntSet, the order of members in the Str representation is unspecified. Thus,

IntSet is; "IntSet(4 1 3 7)" >> is;
TST(is, "IntSet(1 4 3 7)");

might fail the TST, although it might conceivably pass

TST(is, "IntSet(4 1 3 7)");

even though both print forms represent the same IntSet. The result would depend on what the print form generator for IntSet produces for the print form of is, the first argument to TST. However, since this is a deterministic process, after all, once one determines what print form will be generated for is — and ascertains it is correct — one can place that as the literal in the test and be assured that the print form for is will not change if the IntSet class is not changed.

Note, however, that the following test always succeeds:
IntSet $i_1$; "IntSet(4 1 3 7)" » $i_1$
IntSet $i_2$; "IntSet(1 4 3 7)" » $i_2$
TST($i_1$==$i_2$, true);

However, this produces a less informative error message should it fail (presumably due to a bug in IntSet, which is under test).\(^\text{10}\)

9.6.2 Interaction with the user: Gui

The file Gui.h defines a multi-purpose interface to the Windows user interface. It insulates the library user from Windows API calls and relieves him of the need to interact with the operating system or the window system to design a simple user interface.

Gui allows the programmer to create a dialog containing any number of buttons, sliders, or text boxes, and to display any number of images in their own windows, with automatic zooming that maintains the aspect ratio, tied to window size via corner dragging.

Such a dialog is created by a single call to Gui::create_dialog which specifies the desired numbers of buttons, sliders, and text boxes. Functions are provided to set labels, ranges, values, callbacks, etc. Gui constructs a dialog window having the requested components in a fixed configuration.

Gui is predicated on a callback-driven programming mode; i.e., buttons and sliders generate calls to routines which the programmer has registered. This requires that Gui run a loop under its control as long as the user interface is to be active. This loop is started with the single call Gui::win_loop(); all handling of Windows “messages” is done by Gui, generating calls to routines the programmer has registered with such calls to Gui as

\[ \text{Gui::set_button_fun}(2, \text{get_file}); \]

which specifies that when button \#2 is pressed, the function get_file, previously defined in the current program — or anywhere else that can be linked in — should be called.

9.6.3 Plotting: Gnuplot, Gnuplot3d, and Easyplot

Gnuplot.h defines routines for generating plots with gnuplot, a freely available plotting package.\(^\text{11}\) We support gnuplot 3.5; a 3.6 version is currently in beta test. Locally at NECI, the gnuplot distribution is accessible in /packages/gnuplot/3.5, including a postscript manual and tutorial.

Gnuplot supports 2-dimensional and 3-dimensional plots in a vast profusion of styles. As distributed, it uses a tedious dialog-driven file-based interactive

\(^{10}\)Why not just test object equality instead of print form equality in TST, and avoid such anomalies? There are several reasons. We would have to assume that the objects being tested have operator \(==\) defined, which may not always be the case. Likewise for read. And in addition, either tests or fancy footwork would be required to decide whether to do a read operation on either TST argument, depending on whether it was a string literal.

\(^{11}\)WWW home page http://www.cs.dartmouth.edu/gnuplot.info.html.
interface which is poorly suited to programmatic use. We have provided a C++
interface which simplifies generating plots from programs. One can generate
both on-screen plots and encapsulated postscript files of plots for printing or
inclusion in other documents.

Gnuplot.h provides 3 independent interfaces:

- Based on plot objects to plot curves and surfaces.
- Quick and dirty access to object-based plotting.
- Based on single function calls of many arguments.

The interface based on plot objects (of classes Gnuplot or Gnuplot3d) offers
the most complete access to gnuplot features. This includes multiple curves with
multiple properties on a single plot, for example. The price for access to a large
number of features is increased complexity in setting up plotting operations.

The Easyplot interface, on the other hand, can't do very much, but is very
simple to use, and is appropriate for quick plots that need not be of publishable
quality. (For example, there is no custom labelling of axes or titling of the
plot.) Easyplot contains just the overloaded function plot, and the function
scatter_plot. If you hand plot a Vec as its sole argument, it plots a curve
representing the values as a function of the indices of the Vec. If you hand it an
Arr<Vec> instead, it plots a surface plot of the values as a function of the two
sets of indices.

The multi-argument function call interface lies somewhere between the other
two. It provides plotting only of curves, and different types of plots are generated
by different calls.

Other programs needed for the Gnuplot class
Gnuplot relies on the external programs gnuplot and gsview. gnuplot gener-
ates the plots on the screen, as well as postscript files of the plots. gsview, an
equivalent to ghostview on Unix, is a postscript viewer based on ghostscript.
gnuplot and gsview must be findable in the current path, under those names,
for these features of Gnuplot to work. More details about using gnuplot and
gsview can be found in appendix sections B.5 (p. 77) and B.4 (p. 76), resp.

9.7 Operating system interface
9.7.1 File I/O: File
The File class contains routines related to the file system. The File class only
exists to provide a namespace for file-related routines; it has no data members
or member functions. There is no File object; the File class deals with files
specified entirely by Strs containing the file names.

These routines fall into the categories:

- Reading and writing files
• Changing, reading, creating, and deleting directories

• Parsing and manipulating path strings

The library provides a `safe_write()` function, which never clobbers an existing file. Rather, it maintains versions, using the Emacs (Stallman, 1994) version numbering convention of the form `filename.ext.-3`. When a pending collision is detected in a write operation, the existing file is first renamed with the lowest unused version number and only then is the new file written. We also provide `safe_version_name_in()`, which does not write a file, but merely fills a string with the next version name that is available. Emacs provides commands for doing housekeeping on files with this kind of versioning, such as keeping only a certain number of old versions and deleting the rest.

These routines are not based on file-locking mechanisms. There is therefore a remote chance that another thread or process might create a file just before the routine writes or renames to the same name, leading to an unhandled collision.

The library routines read files exclusively into `Strs` and write them exclusively from `Strs`. This is in keeping with the philosophy that every persistent type should have a `Str` print form. Files are not left open after any call to a `File` routine. This greatly simplifies resource management by eliminating it — no files can be unintentionally left open, and no crashes can occur due to inadequate system file handles or buffers, etc. This is analogous to the memory management discipline: the resource is localized where used, and deallocated where allocated.

9.7.2 Timing and telling time: Timer, UserTimer, and Time

`Timer.h` defines routines both for timing computations and for getting wall clock time. Timing is done by `Timer` and `UserTimer`, while wall clock functions are handled by `Time`.

**Timing code execution: Timer and UserTimer**

Many computer science research projects involve timing experiments. The class `Timer` is used to time the execution of code. It encapsulates the interface to the system timing mechanisms. Typical usage looks like:

```c
Timer t; Dbl x=32.3; Dbl y=655.7; Dbl z;
for(Int i=0; i<3000000; i++) z=x*y;
cout << "Time for 3 million multiplies: " << t.time() << "\n";
```

`Timer::time()` returns a `Dbl` to ensure a sufficiently large range.

Intel Pentium and Pentium Pro ("p5" and "p6", resp.) processors contain a register that counts processor clock ticks. If the processor is a 150 MHz Pentium, this register is incremented 150 million times per second. The Windows Win32 API provides a way to access this counter, as well as a way to determine its frequency, thus allowing very precise time measurements.
However, it is difficult to guarantee that the counter only operates while the user program is executing, and not while system tasks and other user programs run. It is therefore advisable when using this timer to measure fairly short operations that are likely to run to completion in a single time slice for the thread in which they occur. One can repeat this measurement many times, discount outliers as including other processor activity, and average the results for the remaining reliable samples. Minimizing the number of other processes running when a timing test is performed will also minimize the likelihood that other threads may intervene in a count.

As an alternate way to measure time intervals, we also provide the UserTimer class, which measures only user time. In Windows NT 3.51, however, our experiments indicate that the accuracy of this timer is limited to about 1/60 (.0167) of a second, despite documentation which indicates a 200 nsec resolution. Meaningful measurements therefore should include computations which take a large number of these 60Hz ticks. UserTimer does, however, use a system call which is documented as only measuring user time, and not time taken by other threads or processes. We have not, however, independently confirmed that this documented behavior actually obtains.

Wall clock time: Time
The class Time contains routines to access the absolute time and date. The routine:

```cpp
Db1 Time::absolute_time();
// Time since 00:00:00 GMT, Jan. 1, 1970 in seconds.
```

is used for initializing random number generators to a different seed on every run.

9.7.3 Submitting a command to the shell: Sys
Sys.h contains the objectless class Sys. Sys defines the functions system and system_async, which submit a Str as a command to a shell. system submits the command synchronously, i.e., the calling program blocks until the command is complete and its process finishes. system_async, on the other hand, submits the command asynchronously, i.e., it returns immediately, not waiting for the new process to complete.

9.8 Classless header files
There are also a few header files that do not correspond to any class. We created these because they contain definitions used in many places, but which are not associated with a single class, or looking in a class header file might not be an obvious place to find them.
9.8.1 **All.h: a potpourri included by all class definitions**

`All.h` contains the definitions that are used throughout the library. For example, it contains the `typedef`s that we use to provide capitalized names for the built-in types such as `double`.

`All.h` also sees to it that `Str.h` is included, at the appropriate point. `Str.h` must be included by every class that wants to have a reader or a print form, or that wants to put any type other than a built-in type onto `cout` or `cerr`. Since that is essentially every class, it’s included automatically by `All.h`.

Templates for the default `take` and for `swap` are in `All.h`, as are ones for `min` and `max`, some global `Int` constants, and templates for automatically defining `<=`, `>=`, `!`, and `!=` in terms of `<` and `==`.

9.8.2 **Db1 routines and constants**

`Db1.h` defines such routines as `square`, `cube`, `log2`, and `approx_eq` for `Db1s`, as well as constants such as `pi`, positive and negative infinity, the “epsilon” for approximate equality, `e` (base of natural logs), and the minimum and maximum representable `Db1s`. `Db1.cpp` currently exists only to initialize the value for positive infinity.

9.8.3 **Win32api.h: the Microsoft Windows API interface**

*Almost always, users of our library will not need to make any calls directly to the Windows API.* However, for RCL’s own use in interfacing to that API, and for those rare occasions when it is needed, `Win32api.h` should be included. It does very little, but can save much grief. Mainly what it does is to include the Windows header file `<windows.h>`. But, it guards it so that it cannot be included more than once, which would cause compilation errors, and additionally it includes `Warn.h`, described below, which turns off compilation warnings that the Windows header files themselves cause, as well as defining `STRICT`, which the Windows header files respect by adhering to some ANSI standards about pointer types, which Windows uses extensively for things like handles of all kinds. This can catch pointer assignment bugs.

9.8.4 **Warn.h: controlling compiler warnings**

We recommend running the Microsoft compilers at the highest warning level, which is 4 (the default for Microsoft Visual C++ 4.2 and 5.0 is 3). Level 4 is the strictest check for ANSI C++. It turns out that Microsoft’s own Windows API header files do not compile without generating some compiler warnings under level 4. These are all either benign, or come from dubious compiler diagnoses. `Warn.h` turns off these compiler warnings. These are mainly drawn from (Richter and Locke, 1995).
10 Acknowledgments

We would like to thank Warren D. Smith for providing two random number generators, and, along with Igor Durdanovic and Andrew Wright, for helpful comments about this document. Talal Shamoon alerted us to the existence of the Independent JPEG Group's freely available LibJPEG software. Henry Cejtin made some helpful suggestions regarding timing and other issues.

A Appendix. Numerical linear algebra: Blas.h and Lapack.h

A.1 What are BLAS and LAPACK

Many of our classes depend on linear algebra operations such as finding eigenvalues or the products of matrices. Over the years an immense effort in scientific computing has been expended to optimize the computation of standard linear algebra. This has culminated in the standard packages BLAS (Lawson et al., 1979; Dongarra et al., 1988; Dongarra et al., 1990) and LAPACK (Anderson et al., 1994), which the library uses for its linear algebra. ('BLAS' is an acronym for Basic Linear Algebra Subprograms, while 'LAPACK' is an acronym-contraction of Linear Algebra PACKage.) LAPACK is a descendent of LINPACK and EISPACK. It uses block-structured versions of linear algebra algorithms which are especially efficient on pipelined machines and on machines with caches. LAPACK is designed to make use of the BLAS whenever possible, as part of a strategy where the BLAS routines are carefully optimized on a per platform basis. Such optimized versions are available for the SGI/IRIX 5.3 and 6.1 platforms, but we were unable to locate any freely available version for the x86/Windows platforms. The standard distribution of LAPACK includes a generic version of the BLAS, which we use for the Windows implementation. We use a version of LAPACK that consists of translations into C of the original FORTRAN code, known as CLAPACK. LAPACK is available by ftp from netlib at http://www.netlib.org/lapack.

We have compiled the LAPACK code used by our classes into a library that can be linked to, and created header files Blas.h and Lapack.h with prototypes for the BLAS and LAPACK routines that we use. Since BLAS and LAPACK comprise a very large number of routines, we have been selective in what we included. Primarily, this was driven by our own needs. Thus, for example, we do not include complex versions of routines, or single precision versions. It is a straightforward matter to add further routines as the need arises.

A.2 Who needs Blas.h and Lapack.h

A user of the library generally should not need to use the BLAS or LAPACK interface directly. Rather, he should make use of the abstractions provided in the
Vec, Mat, and MatSym classes. For those situations where BLAS and LAPACK may be needed, though, we provide the following documentation.

### A.3 What is in BLAS and LAPACK

Following is the complete matrix of all BLAS routines, from the online BLAS man page. Note that we only provide an interface to a small fraction of these. To read this matrix, one discerns, for example, the name of the double precision dot product of two vectors (or matrices) by reading the first line and the second column to obtain ddot. The initial letter of the name indicates ‘single’, ‘double’, ‘complex’, or ‘double complex’ (z). Subsequent letters for level 2 and 3 functions indicate special types of matrices. For example, for the ‘D’ series these special types are: general, general band, symmetric-packed storage, symmetric, symmetric band, triangular, triangular band, triangular-packed storage.

We use only the general and symmetric real matrix routines at this time. Using special routines for symmetric matrices affords a significant cost savings in computations involving multivariate correlations and covariances, since these are represented by symmetric matrices.

#### BLAS Level 1:

<table>
<thead>
<tr>
<th>Function</th>
<th>Prefix, Suffix</th>
<th>Rootname</th>
</tr>
</thead>
<tbody>
<tr>
<td>dot product</td>
<td>s- d- c-u c-c z-u z-c</td>
<td>-dot-</td>
</tr>
<tr>
<td>y = a*x + y</td>
<td>s- d-</td>
<td>-axpy</td>
</tr>
<tr>
<td>setup Givens rotation</td>
<td>s- d-</td>
<td>-rotg</td>
</tr>
<tr>
<td>apply Givens rotation</td>
<td>s- d- cs-</td>
<td>-rot</td>
</tr>
<tr>
<td>copy x into y</td>
<td>s- d- c-</td>
<td>-copy</td>
</tr>
<tr>
<td>swap x and y</td>
<td>s- d- c-</td>
<td>-swap</td>
</tr>
<tr>
<td>Euclidean norm</td>
<td>s- d- sc-</td>
<td>-nrm2</td>
</tr>
<tr>
<td>sum of absolute values</td>
<td>s- d- sc-</td>
<td>-asum</td>
</tr>
<tr>
<td>x = a*x</td>
<td>s- d- cs- c- zd- z-</td>
<td>-scal</td>
</tr>
<tr>
<td>index of max abs value</td>
<td>is- id- ic- iz-</td>
<td>-amax</td>
</tr>
</tbody>
</table>
BLAS Level 2:

- **MV** Matrix vector multiply
- **R** Rank one update to a matrix
- **R2** Rank two update to a matrix
- **SV** Solving certain triangular matrix problems.

### Single Precision Level 2 BLAS

<table>
<thead>
<tr>
<th>MV</th>
<th>R</th>
<th>R2</th>
<th>SV</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGE</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>SGB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SSP</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>SSY</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>SSB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STR</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>STB</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STP</td>
<td>x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Double Precision Level 2 BLAS

<table>
<thead>
<tr>
<th>MV</th>
<th>R</th>
<th>R2</th>
<th>SV</th>
</tr>
</thead>
<tbody>
<tr>
<td>DGE</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>DGB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSP</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>DSY</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>DSB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DTR</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DTB</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DTP</td>
<td>x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Complex Level 2 BLAS

<table>
<thead>
<tr>
<th>MV</th>
<th>R</th>
<th>RC</th>
<th>RU</th>
<th>R2</th>
<th>SV</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGE</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>CGB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHE</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHP</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CTR</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CTB</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CTP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Double Precision Complex Level 2 BLAS

<table>
<thead>
<tr>
<th>MV</th>
<th>R</th>
<th>RC</th>
<th>RU</th>
<th>R2</th>
<th>SV</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZGE</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>ZGB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZHE</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZHP</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZHB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZTR</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZTB</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZTP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
BLAS Level 3:
- MM Matrix matrix multiply
- RK Rank-k update to a matrix
- R2K Rank-2k update to a matrix
- SM Solving triangular matrix with many right-hand-sides.

<table>
<thead>
<tr>
<th>single precision Level 3 BLAS</th>
<th>Double precision Level 3 BLAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>MM</td>
<td>RK</td>
</tr>
<tr>
<td>SGE</td>
<td>x</td>
</tr>
<tr>
<td>SY</td>
<td>x</td>
</tr>
<tr>
<td>STR</td>
<td>x</td>
</tr>
</tbody>
</table>

complex Level 3 BLAS

<table>
<thead>
<tr>
<th>complex Level 3 BLAS</th>
<th>Double precision complex Level 3 BLAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>MM</td>
<td>RK</td>
</tr>
<tr>
<td>CGE</td>
<td>x</td>
</tr>
<tr>
<td>CSY</td>
<td>x</td>
</tr>
<tr>
<td>CHE</td>
<td>x</td>
</tr>
<tr>
<td>CTR</td>
<td>x</td>
</tr>
</tbody>
</table>

A.4 BLAS and LAPACK routines provided by RCL

Of the above, we currently provide an interface to the following BLAS routines:
Level 1 BLAS:
- dasum
- daxpy
- dcopy
- ddot
- dnrm2
- dscal
- dswap
- idamax

Level 2 BLAS:
- dgemv
- dger
- dsyr

Level 3 BLAS:
- dgemm

This is evidently a small fraction of all the available BLAS routines; nevertheless it has been ample for our applications.

We also provide an interface to the following LAPACK routines:
A.5 BLAS and LAPACK routine naming conventions

Note that our prototypes for BLAS and LAPACK routines all have a trailing underscore (_) appended to the BLAS/LAPACK name. This is a feature of the distributed CLAPACK code, which decorates the FORTRAN names so as to maintain callability from FORTRAN (by keeping separate the C and FORTRAN namespaces). This means that to call the BLAS routine dgemm, one actually writes dgemm_( ... ). Also, in conformance with FORTRAN calling conventions, all arguments to BLAS and LAPACK routines are passed by passing pointers to the data.

B Appendix. Compiling, linking, testing, and environment issues.

In this appendix, we discuss a number of points regarding practical details about compiling and linking programs with RCL, as well as the compile time and run time environment needed to assure that everything works correctly.

For detailed information about using Microsoft Visual C++ to compile and link programs using RCL, please see the separate technical note “How to compile and link using the library,” distributed with the library in RCL/doc\howto.

B.1 Using the right C/C++ runtime

Starting with MSVC 4.2, Microsoft distributes two incompatible versions of the C and C++ runtimes intermixed in the same include directory. This was done in order to support the Standard C++ library, which contains iostream functions which are incompatible with the previous versions.
By default, RCL uses what Microsoft refers to as the "old" runtime library. To use the "new" runtime library, you should

```cpp
#define NEW_STD_LIB
```

before you

```cpp
#include "IostreamRCL.h"
```

Because the "old" and "new" libraries are incompatible, you should be careful not to mix the two. In some cases, doing so can fail to generate compile time errors, but will lead to strange runtime behavior. E.g., if you use both libraries, each will maintain its own runtime stack. Both sets of header files do checks and cause the runtime headers that they think you really want to be loaded, and they embed linker directives into the object files to ensure that these are linked.

The new runtime library is necessary if you will be using STL.

The reason we have made the old library the default is this. It turns out that the new library headers are constructed in such a way that they would generate large numbers of level 3 and level 4 compiler warnings in MSVC 4.2 and 5.0. In order to minimize the annoyance of all these warnings, the MSVC library header files contain pragmas to turn them on and off as appropriate. Unfortunately, this creates some unpleasant side effects. First, since many of these warnings are triggered only at the time that a template defined in a header is instantiated in user code, the header designers found it necessary to turn off the warnings in such a way that they would remain off when compiling user code in addition to just when compiling the headers. This means that user code does not get compiled at the warning level that one thinks one has set. On the other hand, there are many level 4 compiler warnings which are triggered by code within the headers. However, if the user attempts to suppress these by wrapping the include of the headers in pragmas to turn the warnings off and on, the attempt will fail to have the desired effect. This is because the headers themselves set the warnings back on by virtue of resetting them to "default" behavior after temporarily turning them off themselves. Later headers that are recursively included then trigger the warnings. We felt that this crippling or rendering unusable of many of the warnings provided by the compiler warranted avoiding this header set in the absence of a compelling reason to use it, such as the need to use STL. Hopefully, future releases of MSVC++, beyond 5.0, will improve this situation. An additional benefit of using the old header set is that it compiles much faster: the new one uses about 40 header files containing intricate template gymnastics.

### B.2 What to #include

We have adopted the view that the user need know only which header defines the class he needs, and not what other headers are in the transitive closure of
classes required by the first header.\footnote{12} This is a corollary of the idea that the implementation of a class should not be the user's concern.

Accordingly, as far as possible, users of a class need specify only a minimal number of \#includes. The header files use guard macros to ensure that they are scanned only once, so there is no harm in specifying redundant \#includes (except for the cost of file access). If any additional headers are needed, they are included by the first header. Occasionally, however, there are exceptions to this rule, when there would be excessive overhead in reading a large header and it is not needed for most applications or is not directly related to the class defined in the first header.

We have also provided a project who_includes_who in RCL_projects, which analyzes all the .cpp and .h files in a directory to generate a listing of which files each such file \#includes. The results of running this on the RCL src files is included in the RCL\doc directory in the file includes.txt.

A related project includes_transitive_closure recursively analyzes the includes in the current directory from a given file.

\section{Testing RCL}

RCL is distributed with a preinstalled workspace called Tests, which contains all the test modules for testing the components of RCL, one to a project. Also, it contains a master project AllTests which builds and runs all the tests.

Most of the test modules simply use the TST macro, and generate no output if they are passed in full. However, a few modules — those that make extensive use of graphics or random functions — generate graphic and/or text output that requires some interpretation and user interaction. You will have to examine the source code in the test subdirectory to interpret the results of the tests for the latter subset of test modules.

The CLAPACK and JPEG libraries include test suites of their own. These are not run as part of the RCL Tests modules, since we do not contemplate modifying these libraries. These tests were, however, run and passed when we built those libraries.

\section{Using the postscript viewer GSview with RCL}

The Gnuplot classes contain functions that start up a postscript viewer. By default, the name that is used is GSview, which is contained in the static Str psviewer, defined in Gnuplot.h. GSview is the name of a free postscript viewer written by Russell Lang, which is redistributed with RCL in the directory RCL-distribution\RCL-helpers\ghostscript\GSview. GSview is analogous to the Unix program ghostview.

Gnuplot calls GSview by submitting a command of the form GSview file.ps to the command processor. For this to work, an executable invoked as GSview

\begin{footnote}{12}I.e., for the purpose of \#includes. Unfortunately, for linking, there is no way to avoid specifying all object modules that are needed.\end{footnote}
must be in the path.\textsuperscript{13}

To make it more convenient to accomplish that, we have added two items to the \texttt{GSview} directory. The original executable, \texttt{gsview32.exe}, has been copied to \texttt{gsview.exe}. Also, a file \texttt{gsview.bat} has been added.

If one chooses to add the \texttt{GSView} directory to the path, the \texttt{Gnuplot} call will execute \texttt{gsview32.exe}. Alternately, if you do not like a proliferation of directories in your path, you can copy the \texttt{GSview.bat} file to a repository of executables that is in your path. You may have to edit \texttt{GSview.bat} if you do not have the RCL-distribution directory at top level in your C: drive, to contain the correct location of \texttt{gsview32.exe}.

\texttt{GSview} relies on the presence of a ghostscript interpreter in the directory \texttt{RCL-distribution\RCL-helpers\ghostscript\gs4.01}. (The version number will change as newer releases become available.) This directory contains the Aladdin ghostscript interpreter, which is free software.\textsuperscript{14}

\textbf{B.5 Using the gnuplot plotting package with RCL}

The \texttt{Gnuplot} classes perform plotting by writing control and data files, and then invoking the \texttt{gnuplot}\textsuperscript{15} plotter on those files. This is done by submitting commands of the form \texttt{gnuplot filename} to the command processor.

To be sure that such a call runs \texttt{gnuplot} requires the same steps as above for \texttt{GSview} (section B.4). The distributed \texttt{gnuplot} binary is called \texttt{wgnupl32.exe}. We have copied this to \texttt{gnuplot.exe}, and also provided \texttt{gnuplot.bat}, to be used as explained above for \texttt{GSview}.

RCL currently uses version 3.5 of \texttt{gnuplot}, although a beta version 3.6 is available.\textsuperscript{16}

\textbf{B.6 Viewing and scrolling console windows}

The Windows NT default console window has only 25 lines, and cannot be scrolled.\textsuperscript{17} This is not very convenient if you might ever produce more than 25 lines of output to the console. Fortunately, in anticipation of even such an unlikely occurrence, Windows NT provides a mechanism to increase the number of visible and scrollable lines. To change this setting, select \texttt{<upper left corner of console window> Properties > Layout > Screen Buffer Size > Height > <numlines>}, where \texttt{<numlines>} is

\textsuperscript{13}When you submit the command \texttt{GSview} to the command processor, it searches your path for any of the executable files \texttt{GSview.exe}, \texttt{GSview.com}, \texttt{GSview.pif}, \texttt{GSview.bat}. A \texttt{bat} file is a script file that the command processor interprets.

\textsuperscript{14}Both \texttt{GSview} and Aladdin ghostscript are governed by licenses contained in their respective directories. These licenses essentially grant the right to free use and redistribution if not for commercial gain. See the licenses for exact conditions.

\textsuperscript{15}\texttt{gnuplot} is also free software, governed by a very lenient license. It is \textit{not} distributed under the so-called GNU General Public License, used by the Free Software Foundation. Although \texttt{gnuplot} is distributed through the FSF with GNU programs, the etymology of 'gnuplot' is independent of that of the GNU project, and it is not a part of or a fruit of the GNU project.

\textsuperscript{16}Version 3.6beta has some undesirable interactions with process control.

\textsuperscript{17}Probably an homage to MS-DOS. 
an integer no greater than 9999. When you exit this dialog, you are given a choice whether to set these properties for every new incarnation of the same console; we suggest you choose to set it for future consoles, too. This only applies to consoles of the same name, which by default is the name of the executable, and is stored in the registry. Note that the memory for all these lines is allocated as soon as the console is created, not as needed.

RCL also provides the function gui::set_console_buffer_height in the gui class to set this from a program. If you use the gui graphic interface, this is automatically initialized to a reasonably large number.

### B.7 Spurious carriage returns (ctrl-M)

Microsoft operating systems use the 2 character sequence `<CR>`<LF> (carriage return, linefeed) as the newline separator, while unix systems use just `<LF>`. The ASCII representation of carriage return is control-M, while linefeed is control-J. This means that applications native to Windows NT put `<CR>`s into files which they write. When these files are viewed using a unix application, they appear to have a spurious control-M at the end of each line.

This is often annoying. It is, however, possible to avoid this problem to some extent. First, although MSVC writes `<CR>`s whenever it writes a file, e.g., using the integrated editor, it is not sensitive to the presence or absence of `<CR>`s when it reads a file. This means that source files that use only `<LF>` for newline work perfectly well with MSVC and the compiler.

Second, if you use emacs under Windows NT, you can turn off the default behavior of writing `<CR>`s by evaluating `(using-unixfilesystems t)`. Emacs will then act like unix applications do when they write text files, using only `<LF>` as newline.

A related feature of the Win32 programming environment is that file operations can be done in either “text” or “binary” mode. What these terms mean is simply that in “text” mode, all reads from a file convert every `<CR>`<LF> into `<LF>`, and all writes do the inverse. This feature leads to an astounding frequency of bugs, e.g., byte counts in memory are different than on disk, the operation is not invertible, and of course binary files do not take kindly to such conversions.

The Win32 API provides several routines to read and write files. Some of these have optional arguments to specify text or binary mode, while some choose the mode based on global settings, which default to text mode. The interface that RCL provides to file I/O always uses binary mode to avoid all the problems engendered by text mode. If you need to use the Win32 routines directly, however, beware of this trap.

---

18 I.e., a file containing an isolated `<LF>` that is read then written will then contain a `<CR>`<LF> sequence.

19 Application Programmer Interface.
References


79


---

80


