libg++

COLLABORATORS

	TITLE : libg++			
ACTION	*NAME*	*DATE*	*SIGNATURE*	
WRITTEN BY		June 8, 2025		

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

libg++

1.1 libg++.guide

```
Introduction ************

This manual documents how to install and use the GNU C++ library.


Copying         GNU Library Public License says how you can copy
                and share the GNU C++ library.
Contributors    People who have contributed to GNU C++ library.
Installation    How to configure, compile and install GNU C++ library
Trouble         If you have trouble installing GNU C++ library.
General         Aims, objectives, and limitations of the GNU C++ library
Conventions     Stylistic conventions
OK              Support for representation invariants
Proto           Introduction to container class prototypes
Pix             Pseudo-indexes
Representations How variable-sized objects are represented
Expressions     Some guidance on programming expression-oriented classes
Headers         Header files and other support for interfacing C++ to C
Builtin         Utility functions for builtin types
New             Library dynamic allocation primitives
IOStream

                The input/output library (istreams and ostreams).
Stream          obsolete I/O library
Obstack         Obstacks and their uses.
AllocRing       A place to store objects for a while
String          String, SubString, and Regex classes.
Integer         Multiple precision Integer class.
Rational        Multiple precision Rational class
Complex         Complex number class
Fix             Fixed point proportion classes
Bit             BitSet and BitString classes
Random          Random number generators
Data            SampleStatistic and related classes for data collection
Curses          CursesWindow class
List            Lisp-like List prototype
LinkList        Singly and doubly linked list class prototypes
Vector          Vector prototypes
Plex            Plex (adjustable array) prototypes
```

```
Stack              Stack prototypes
Queue              Queue prototypes
Deque              Double ended queue prototypes
PQ                 Heap (priority queue) class prototypes
Set                Set class prototypes
Bag                Bag class prototypes
Map                Map (Associative array) prototypes
GetOpt             C++ class-based version of the GNU/UNIX getopt function
Projects           Things Still Left to do
```

1.2 libg++.guide/Copying

```
GNU LIBRARY GENERAL PUBLIC LICENSE
**********************************

                    Version 2, June 1991

    Copyright (C) 1991 Free Software Foundation, Inc.
    675 Mass Ave, Cambridge, MA 02139, USA
    Everyone is permitted to copy and distribute verbatim copies
    of this license document, but changing it is not allowed.

    [This is the first released version of the library GPL.  It is
     numbered 2 because it goes with version 2 of the ordinary GPL.]


Preamble
========

    The licenses for most software are designed to take away your
freedom to share and change it.  By contrast, the GNU General Public
Licenses are intended to guarantee your freedom to share and change
free software--to make sure the software is free for all its users.

    This license, the Library General Public License, applies to some
specially designated Free Software Foundation software, and to any
other libraries whose authors decide to use it.  You can use it for
your libraries, too.

    When we speak of free software, we are referring to freedom, not
price.  Our General Public Licenses are designed to make sure that you
have the freedom to distribute copies of free software (and charge for
this service if you wish), that you receive source code or can get it
if you want it, that you can change the software or use pieces of it in
new free programs; and that you know you can do these things.

    To protect your rights, we need to make restrictions that forbid
anyone to deny you these rights or to ask you to surrender the rights.
These restrictions translate to certain responsibilities for you if you
distribute copies of the library, or if you modify it.

    For example, if you distribute copies of the library, whether gratis
or for a fee, you must give the recipients all the rights that we gave
you.  You must make sure that they, too, receive or can get the source
code.  If you link a program with the library, you must provide
```

complete object files to the recipients so that they can relink them
with the library, after making changes to the library and recompiling
it. And you must show them these terms so they know their rights.

   Our method of protecting your rights has two steps: (1) copyright
the library, and (2) offer you this license which gives you legal
permission to copy, distribute and/or modify the library.

   Also, for each distributor's protection, we want to make certain
that everyone understands that there is no warranty for this free
library. If the library is modified by someone else and passed on, we
want its recipients to know that what they have is not the original
version, so that any problems introduced by others will not reflect on
the original authors' reputations.

   Finally, any free program is threatened constantly by software
patents. We wish to avoid the danger that companies distributing free
software will individually obtain patent licenses, thus in effect
transforming the program into proprietary software. To prevent this,
we have made it clear that any patent must be licensed for everyone's
free use or not licensed at all.

   Most GNU software, including some libraries, is covered by the
ordinary GNU General Public License, which was designed for utility
programs. This license, the GNU Library General Public License,
applies to certain designated libraries. This license is quite
different from the ordinary one; be sure to read it in full, and don't
assume that anything in it is the same as in the ordinary license.

   The reason we have a separate public license for some libraries is
that they blur the distinction we usually make between modifying or
adding to a program and simply using it. Linking a program with a
library, without changing the library, is in some sense simply using
the library, and is analogous to running a utility program or
application program. However, in a textual and legal sense, the linked
executable is a combined work, a derivative of the original library,
and the ordinary General Public License treats it as such.

   Because of this blurred distinction, using the ordinary General
Public License for libraries did not effectively promote software
sharing, because most developers did not use the libraries. We
concluded that weaker conditions might promote sharing better.

   However, unrestricted linking of non-free programs would deprive the
users of those programs of all benefit from the free status of the
libraries themselves. This Library General Public License is intended
to permit developers of non-free programs to use free libraries, while
preserving your freedom as a user of such programs to change the free
libraries that are incorporated in them. (We have not seen how to
achieve this as regards changes in header files, but we have achieved
it as regards changes in the actual functions of the Library.) The
hope is that this will lead to faster development of free libraries.

   The precise terms and conditions for copying, distribution and
modification follow. Pay close attention to the difference between a
"work based on the library" and a "work that uses the library". The
former contains code derived from the library, while the latter only

works together with the library.

   Note that it is possible for a library to be covered by the ordinary
General Public License rather than by this special one.

   TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

  0. This License Agreement applies to any software library which
     contains a notice placed by the copyright holder or other
     authorized party saying it may be distributed under the terms of
     this Library General Public License (also called "this License").
     Each licensee is addressed as "you".

     A "library" means a collection of software functions and/or data
     prepared so as to be conveniently linked with application programs
     (which use some of those functions and data) to form executables.

     The "Library", below, refers to any such software library or work
     which has been distributed under these terms. A "work based on the
     Library" means either the Library or any derivative work under
     copyright law: that is to say, a work containing the Library or a
     portion of it, either verbatim or with modifications and/or
     translated straightforwardly into another language. (Hereinafter,
     translation is included without limitation in the term
     "modification".)

     "Source code" for a work means the preferred form of the work for
     making modifications to it. For a library, complete source code
     means all the source code for all modules it contains, plus any
     associated interface definition files, plus the scripts used to
     control compilation and installation of the library.

     Activities other than copying, distribution and modification are
     not covered by this License; they are outside its scope. The act
     of running a program using the Library is not restricted, and
     output from such a program is covered only if its contents
     constitute a work based on the Library (independent of the use of
     the Library in a tool for writing it). Whether that is true
     depends on what the Library does and what the program that uses
     the Library does.

  1. You may copy and distribute verbatim copies of the Library's
     complete source code as you receive it, in any medium, provided
     that you conspicuously and appropriately publish on each copy an
     appropriate copyright notice and disclaimer of warranty; keep
     intact all the notices that refer to this License and to the
     absence of any warranty; and distribute a copy of this License
     along with the Library.

     You may charge a fee for the physical act of transferring a copy,
     and you may at your option offer warranty protection in exchange
     for a fee.

  2. You may modify your copy or copies of the Library or any portion
     of it, thus forming a work based on the Library, and copy and
     distribute such modifications or work under the terms of Section 1
     above, provided that you also meet all of these conditions:

   a. The modified work must itself be a software library.

   b. You must cause the files modified to carry prominent notices
      stating that you changed the files and the date of any change.

   c. You must cause the whole of the work to be licensed at no
      charge to all third parties under the terms of this License.

   d. If a facility in the modified Library refers to a function or
      a table of data to be supplied by an application program that
      uses the facility, other than as an argument passed when the
      facility is invoked, then you must make a good faith effort
      to ensure that, in the event an application does not supply
      such function or table, the facility still operates, and
      performs whatever part of its purpose remains meaningful.

      (For example, a function in a library to compute square roots
      has a purpose that is entirely well-defined independent of the
      application. Therefore, Subsection 2d requires that any
      application-supplied function or table used by this function
      must be optional: if the application does not supply it, the
      square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If
identifiable sections of that work are not derived from the
Library, and can be reasonably considered independent and separate
works in themselves, then this License, and its terms, do not
apply to those sections when you distribute them as separate
works. But when you distribute the same sections as part of a
whole which is a work based on the Library, the distribution of
the whole must be on the terms of this License, whose permissions
for other licensees extend to the entire whole, and thus to each
and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or
contest your rights to work written entirely by you; rather, the
intent is to exercise the right to control the distribution of
derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the
Library with the Library (or with a work based on the Library) on
a volume of a storage or distribution medium does not bring the
other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public
   License instead of this License to a given copy of the Library.
   To do this, you must alter all the notices that refer to this
   License, so that they refer to the ordinary GNU General Public
   License, version 2, instead of to this License. (If a newer
   version than version 2 of the ordinary GNU General Public License
   has appeared, then you can specify that version instead if you
   wish.) Do not make any other change in these notices.

   Once this change is made in a given copy, it is irreversible for
   that copy, so the ordinary GNU General Public License applies to
   all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

    If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

    However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

    When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

    If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

    Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also compile or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the
Library is used in it and that the Library and its use are covered
by this License. You must supply a copy of this License. If the
work during execution displays copyright notices, you must include
the copyright notice for the Library among them, as well as a
reference directing the user to the copy of this License. Also,
you must do one of these things:

   a. Accompany the work with the complete corresponding
      machine-readable source code for the Library including
      whatever changes were used in the work (which must be
      distributed under Sections 1 and 2 above); and, if the work
      is an executable linked with the Library, with the complete
      machine-readable "work that uses the Library", as object code
      and/or source code, so that the user can modify the Library
      and then relink to produce a modified executable containing
      the modified Library. (It is understood that the user who
      changes the contents of definitions files in the Library will
      not necessarily be able to recompile the application to use
      the modified definitions.)

   b. Accompany the work with a written offer, valid for at least
      three years, to give the same user the materials specified in
      Subsection 6a, above, for a charge no more than the cost of
      performing this distribution.

   c. If distribution of the work is made by offering access to copy
      from a designated place, offer equivalent access to copy the
      above specified materials from the same place.

   d. Verify that the user has already received a copy of these
      materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the
Library" must include any data and utility programs needed for
reproducing the executable from it. However, as a special
exception, the source code distributed need not include anything
that is normally distributed (in either source or binary form)
with the major components (compiler, kernel, and so on) of the
operating system on which the executable runs, unless that
component itself accompanies the executable.

It may happen that this requirement contradicts the license
restrictions of other proprietary libraries that do not normally
accompany the operating system. Such a contradiction means you
cannot use both them and the Library together in an executable
that you distribute.

7. You may place library facilities that are a work based on the
   Library side-by-side in a single library together with other
   library facilities not covered by this License, and distribute
   such a combined library, provided that the separate distribution
   of the work based on the Library and of the other library
   facilities is otherwise permitted, and provided that you do these
   two things:

     a. Accompany the combined library with a copy of the same work
       based on the Library, uncombined with any other library
       facilities. This must be distributed under the terms of the
       Sections above.

     b. Give prominent notice with the combined library of the fact
       that part of it is a work based on the Library, and explaining
       where to find the accompanying uncombined form of the same
       work.

8. You may not copy, modify, sublicense, link with, or distribute the
   Library except as expressly provided under this License. Any
   attempt otherwise to copy, modify, sublicense, link with, or
   distribute the Library is void, and will automatically terminate
   your rights under this License. However, parties who have
   received copies, or rights, from you under this License will not
   have their licenses terminated so long as such parties remain in
   full compliance.

9. You are not required to accept this License, since you have not
   signed it. However, nothing else grants you permission to modify
   or distribute the Library or its derivative works. These actions
   are prohibited by law if you do not accept this License.
   Therefore, by modifying or distributing the Library (or any work
   based on the Library), you indicate your acceptance of this
   License to do so, and all its terms and conditions for copying,
   distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the
    Library), the recipient automatically receives a license from the
    original licensor to copy, distribute, link with or modify the
    Library subject to these terms and conditions. You may not impose
    any further restrictions on the recipients' exercise of the rights
    granted herein. You are not responsible for enforcing compliance
    by third parties to this License.

11. If, as a consequence of a court judgment or allegation of patent
    infringement or for any other reason (not limited to patent
    issues), conditions are imposed on you (whether by court order,
    agreement or otherwise) that contradict the conditions of this
    License, they do not excuse you from the conditions of this
    License. If you cannot distribute so as to satisfy simultaneously
    your obligations under this License and any other pertinent
    obligations, then as a consequence you may not distribute the
    Library at all. For example, if a patent license would not permit
    royalty-free redistribution of the Library by all those who
    receive copies directly or indirectly through you, then the only
    way you could satisfy both it and this License would be to refrain
    entirely from distribution of the Library.

    If any portion of this section is held invalid or unenforceable
    under any particular circumstance, the balance of the section is
    intended to apply, and the section as a whole is intended to apply
    in other circumstances.

    It is not the purpose of this section to induce you to infringe any
    patents or other property right claims or to contest validity of

any such claims; this section has the sole purpose of protecting
the integrity of the free software distribution system which is
implemented by public license practices. Many people have made
generous contributions to the wide range of software distributed
through that system in reliance on consistent application of that
system; it is up to the author/donor to decide if he or she is
willing to distribute software through any other system and a
licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed
to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in
    certain countries either by patents or by copyrighted interfaces,
    the original copyright holder who places the Library under this
    License may add an explicit geographical distribution limitation
    excluding those countries, so that distribution is permitted only
    in or among countries not thus excluded. In such case, this
    License incorporates the limitation as if written in the body of
    this License.

13. The Free Software Foundation may publish revised and/or new
    versions of the Library General Public License from time to time.
    Such new versions will be similar in spirit to the present version,
    but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the
    Library specifies a version number of this License which applies
    to it and "any later version", you have the option of following
    the terms and conditions either of that version or of any later
    version published by the Free Software Foundation. If the Library
    does not specify a license version number, you may choose any
    version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free
    programs whose distribution conditions are incompatible with these,
    write to the author to ask for permission. For software which is
    copyrighted by the Free Software Foundation, write to the Free
    Software Foundation; we sometimes make exceptions for this. Our
    decision will be guided by the two goals of preserving the free
    status of all derivatives of our free software and of promoting
    the sharing and reuse of software generally.

                              NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO
    WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE
    LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT
    HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT
    WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT
    NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND
    FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE
    QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE
    LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY
    SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN

WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY
MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE
LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL,
INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR
INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF
DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU
OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY
OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN
ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

                        END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Libraries
==

   If you develop a new library, and you want it to be of the greatest
possible use to the public, we recommend making it free software that
everyone can redistribute and change. You can do so by permitting
redistribution under these terms (or, alternatively, under the terms of
the ordinary General Public License).

   To apply these terms, attach the following notices to the library.
It is safest to attach them to the start of each source file to most
effectively convey the exclusion of warranty; and each file should have
at least the "copyright" line and a pointer to where the full notice is
found.

    ONE LINE TO GIVE THE LIBRARY'S NAME AND AN IDEA OF WHAT IT DOES.
    Copyright (C) YEAR NAME OF AUTHOR

    This library is free software; you can redistribute it and/or
    modify it under the terms of the GNU Library General Public
    License as published by the Free Software Foundation; either
    version 2 of the License, or (at your option) any later version.

    This library is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
    Library General Public License for more details.

    You should have received a copy of the GNU Library General Public
    License along with this library; if not, write to the
    Free Software Foundation, Inc., 675 Mass Ave, Cambridge,
    MA 02139, USA.

   Also add information on how to contact you by electronic and paper
mail.

   You should also get your employer (if you work as a programmer) or
your school, if any, to sign a "copyright disclaimer" for the library,
if necessary. Here is a sample; alter the names:

    Yoyodyne, Inc., hereby disclaims all copyright interest in
    the library 'Frob' (a library for tweaking knobs) written
    by James Random Hacker.

    SIGNATURE OF TY COON, 1 April 1990

    Ty Coon, President of Vice

  That's all there is to it!

1.3 libg++.guide/Contributors

Contributors to GNU C++ library
****

    Aside from Michael Tiemann, who worked out the front end for GNU
C++, and Richard Stallman, who worked out the back end, the following
people (not including those who have made their contributions to GNU
CC) should not go unmentioned.

  * Doug Lea contributed most otherwise unattributed classes.

  * Per Bothner contributed the iostream I/O classes.

  * Dirk Grunwald contributed the Random number generation classes,
    and PairingHeaps.

  * Kurt Baudendistel contributed Fixed precision reals.

  * Doug Schmidt contributed ordered hash tables, a perfect hash
    function generator, and several other utilities.

  * Marc Shapiro contributed the ideas and preliminary code for Plexes.

  * Eric Newton contributed the curses window classes.

  * Some of the I/O code is derived from BSD 4.4, and was developed by
    the University of California, Berkeley.

  * The code for converting accurately between floating point numbers
    and their string representations was written by David M. Gay of
    AT&T.

1.4 libg++.guide/Installation

Installing GNU C++ library
****

  1. Read through the README file and the Makefile. Make sure that all
     paths, system-dependent compile switches, and program names are
     correct.

  2. Check that files 'values.h', 'stdio.h', and 'math.h' declare and
     define values appropriate for your system.

  3. Type 'make all' to compile the library, test, and install.

      Current details about contents of the tests and utilities are in
      the 'README' file.

1.5 libg++.guide/Trouble

```
Trouble in Installation
***********************
```

   Here are some of the things that have caused trouble for people
installing GNU C++ library.

  1. Make sure that your GNU C++ version number is at least as high as
     your libg++ version number. For example, libg++ 1.22.0 requires
     g++ 1.22.0 or later releases.

  2. Double-check system constants in the header files mentioned above.

1.6 libg++.guide/General

```
GNU C++ library aims, objectives, and limitations
*************************************************
```

   The GNU C++ library, libg++ is an attempt to provide a variety of C++
programming tools and other support to GNU C++ programmers.

   Differences in distribution policy are only part of the difference
between libg++.a and AT&T libC.a. libg++ is not intended to be an
exact clone of libC. For one, libg++ contains bits of code that depend
on special features of GNU g++ that are either different or lacking in
the AT&T version, including slightly different inlining and overloading
strategies, dynamic local arrays, etc. All of these differences are
minor. For example, while the AT&T and GNU stream classes are
implemented in very different ways, the vast majority of C++ programs
compile and run under either version with no visible difference.
Additionally, all g++-specific constructs are conditionally compiled;
The library is designed to be compatible with any 2.0 C++ compiler.

   libg++ has also contained workarounds for some limitations in g++:
both g++ and libg++ are still undergoing rapid development and
testing--a task that is helped tremendously by the feedback of active
users. This manual is also still under development; it has some
catching up to do to include all the facilities now in the library.

   libg++ is not the only freely available source of C++ class
libraries. Some notable alternative sources are Interviews and NIHCL.
(InterViews has been available on the X-windows X11 tapes and also from
interviews.stanford.edu. NIHCL is available by anonymous ftp from GNU
archives (such as the pub directory of prep.ai.mit.edu), although it is

not supported by the FSF - and needs some work before it will work with
g++.)

    As every C++ programmer knows, the design (moreso than the
implementation) of a C++ class library is something of a challenge.
Part of the reason is that C++ supports two, partially incompatible,
styles of object-oriented programming - The "forest" approach,
involving a collection of free-standing classes that can be mixed and
matched, versus the completely hierarchical (smalltalk style) approach,
in which all classes are derived from a common ancestor. Of course,
both styles have advantages and disadvantages. So far, libg++ has
adopted the "forest" approach. Keith Gorlen's OOPS library adopts the
hierarchical approach, and may be an attractive alternative for C++
programmers who prefer this style.

    Currently (and/or in the near future) libg++ provides support for a
few basic kinds of classes:

    The first kind of support provides an interface between C++ programs
and C libraries. This includes basic header files (like 'stdio.h') as
well as things like the File and stream classes. Other classes that
interface to other aspects of C libraries (like those that maintain
environmental information) are in various stages of development; all
will undergo implementation modifications when the forthcoming GNU libc
library is released.

    The second kind of support contains general-purpose basic classes
that transparently manage variable-sized objects on the freestore. This
includes Obstacks, multiple-precision Integers and Rationals, arbitrary
length Strings, BitSets, and BitStrings.

    Third, several classes and utilities of common interest (e.g.,
Complex numbers) are provided.

    Fourth, a set of pseudo-generic prototype files are available as a
mechanism for generating common container classes. These are described
in more detail in the introduction to container prototypes. Currently,
only a textual substitution mechanism is available for generic class
creation.

1.7 libg++.guide/Conventions

GNU C++ library stylistic conventions
****

    * C++ source files have file extension '.cc'. Both C-compatibility
      header files and class declaration files have extension '.h'.

    * C++ class names begin with capital letters, except for 'istream'
      and 'ostream', for AT&T C++ compatibility. Multi-word class names
      capitalize each word, with no underscore separation.

    * Include files that define C++ classes begin with capital letters
      (as do the names of the classes themselves). 'stream.h' is

uncapitalized for AT&T C++ compatibility.

* Include files that supply function prototypes for other C
  functions (system calls and libraries) are all lower case.

* All include files define a preprocessor variable _X_h, where X is
  the name of the file, and conditionally compile only if this has
  not been already defined. The '#pragma once' facility is also used
  to avoid re-inclusion.

* Structures and objects that must be publicly defined, but are not
  intended for public use have names beginning with an underscore.
  (for example, the '_Srep' struct, which is used only by the String
  and SubString classes.)

* The underscore is used to separate components of long function
  names,
  e.g., 'set_File_exception_handler()'.

* When a function could be usefully defined either as a member or a
  friend, it is generally a member if it modifies and/or returns
  itself, else it is a friend. There are cases where naturalness of
  expression wins out over this rule.

* Class declaration files are formatted so that it is easy to
  quickly check them to determine function names, parameters, and so
  on. Because of the different kinds of things that may appear in
  class declarations, there is no perfect way to do this. Any
  suggestions on developing a common class declaration formatting
  style are welcome.

* All classes use the same simple error (exception) handling
  strategy. Almost every class has a member function named
  'error(char* msg)' that invokes an associated error handler
  function via a pointer to that function, so that the error
  handling function may be reset by programmers. By default nearly
  all call '*lib_error_handler', which prints the message and then
  aborts execution. This system is subject to change. In general,
  errors are assumed to be non-recoverable: Library classes do not
  include code that allows graceful continuation after exceptions.

1.8 libg++.guide/OK

Support for representation invariants
****

   Most GNU C++ library classes possess a method named 'OK()', that is
useful in helping to verify correct performance of class operations.

   The 'OK()' operations checks the "representation invariant" of a
class object. This is a test to check whether the object is in a valid
state. In effect, it is a (sometimes partial) verification of the
library's promise that (1) class operations always leave objects in
valid states, and (2) the class protects itself so that client functions

cannot corrupt this state.

   While no simple validation technique can assure that all operations
perform correctly, calls to 'OK()' can at least verify that operations
do not corrupt representations. For example for 'String a, b, c; ... a
= b + c;', a call to 'a.OK();' will guarantee that 'a' is a valid
'String', but does not guarantee that it contains the concatenation of
'b + c'. However, given that 'a' is known to be valid, it is possible
to further verify its properties, for example via 'a.after(b) == c &&
a.before(c) == b'. In other words, 'OK()' generally checks only those
internal representation properties that are otherwise inaccessible to
users of the class. Other class operations are often useful for further
validation.

   Failed calls to 'OK()' call a class's 'error' method if one exists,
else directly call 'abort'. Failure indicates an implementation error
that should be reported.

   With only rare exceptions, the internal support functions for a class
never themselves call 'OK()' (although many of the test files in the
distribution call 'OK()' extensively).

   Verification of representational invariants can sometimes be very
time consuming for complicated data structures.

1.9   libg++.guide/Proto

Introduction to container class prototypes
****

   As a temporary mechanism enabling the support of generic classes,
the GNU C++ Library distribution contains a directory ('g++-include')
of files designed to serve as the basis for generating container
classes of specified elements. These files can be used to generate
'.h' and '.cc' files in the current directory via a supplied shell
script program that performs simple textual substitution to create
specific classes.

   While these classes are generated independently, and thus share no
code, it is possible to create versions that do share code among
subclasses. For example, using 'typedef void* ent', and then generating
a 'entList' class, other derived classes could be created using the
'void*' coercion method described in Stroustrup, pp204-210.

   This very simple class-generation facility is useful enough to serve
current purposes, but will be replaced with a more coherent mechanism
for handling C++ generics in a way that minimally disrupts current
usage. Without knowing exactly when or how parametric classes might be
added to the C++ language, provision of this simplest possible
mechanism, textual substitution, appears to be the safest strategy,
although it does require certain redundancies and awkward constructions.

   Specific classes may be generated via the 'genclass' shell script
program. This program has arguments specifying the kinds of base

types(s) to be used. Specifying base types requires two arguments. The
first is the name of the base type, which may be any named type, like
'int' or 'String'. Only named types are supported; things like 'int*'
are not accepted. However, pointers like this may be used by supplying
the appropriate typedefs (e.g., editing the resulting files to include
'typedef int* intp;'). The type name must be followed by one of the
words 'val' or 'ref', to indicate whether the base elements should be
passed to functions by-value or by-reference.

   You can specify basic container classes using 'genclass base
[val,ref] proto', where 'proto' is the name of the class being
generated. Container classes like dictionaries and maps that require
two types may be specified via 'genclass -2 keytype [val, ref],
basetype [val, ref] proto', where the key type is specified first and
the contents type second. The resulting classnames and filenames are
generated by prepending the specified type names to the prototype names,
and separating the filename parts with dots. For example, 'genclass
int val List' generates class 'intList' residing in files 'int.List.h'
and 'int.List.cc'. 'genclass -2 String ref int val VHMap' generates
(the awkward, but unavoidable) class name 'StringintVHMap'. Of course,
programmers may use 'typedef' or simple editing to create more
appropriate names. The existence of dot seperators in file names
allows the use of GNU make to help automate configuration and
recompilation. An example Makefile exploiting such capabilities may be
found in the 'libg++/proto-kit' directory.

   The 'genclass' utility operates via simple text substitution using
'sed'. All occurrences of the pseudo-types '<T>' and '<C>' (if there
are two types) are replaced with the indicated type, and occurrences of
'<T&>' and '<C&>' are replaced by just the types, if 'val' is
specified, or types followed by "&" if 'ref' is specified.

   Programmers will frequently need to edit the '.h' file in order to
insert additional '#include' directives or other modifications. A
simple utility, 'prepend-header' to prepend other '.h' files to
generated files is provided in the distribution.

   One dubious virtue of the prototyping mechanism is that, because
sources files, not archived library classes, are generated, it is
relatively simple for programmers to modify container classes in the
common case where slight variations of standard container classes are
required.

   It is often a good idea for programmers to archive (via 'ar')
generated classes into '.a' files so that only those class functions
actually used in a given application will be loaded. The test
subdirectory of the distribution shows an example of this.

   Because of '#pragma interface' directives, the '.cc' files should be
compiled with '-O' or '-DUSE_LIBGXX_INLINES' enabled.

   Many container classes require specifications over and above the base
class type. For example, classes that maintain some kind of ordering of
elements require specification of a comparison function upon which to
base the ordering. This is accomplished via a prototype file 'defs.hP'
that contains macros for these functions. While these macros default to
perform reasonable actions, they can and should be changed in

particular cases. Most prototypes require only one or a few of these.
No harm is done if unused macros are defined to perform nonsensical
actions. The macros are:

'DEFAULT_INITIAL_CAPACITY'
     The initial capacity for containers (e.g., hash tables) that
     require an initial capacity argument for constructors. Default:
     100

'<T>EQ(a, b)'
     return true if a is considered equal to b for the purposes of
     locating, etc., an element in a container. Default: (a == b)

'<T>LE(a, b)'
     return true if a is less than or equal to b Default: (a <= b)

'<T>CMP(a, b)'
     return an integer < 0 if a<b, 0 if a==b, or > 0 if a>b. Default:
     (a <= b)? (a==b)? 0 : -1 : 1

'<T>HASH(a)'
     return an unsigned integer representing the hash of a. Default:
     hash(a) ; where extern unsigned int hash(<T&>). (note: several
     useful hash functions are declared in builtin.h and defined in
     hash.cc)

   Nearly all prototypes container classes support container traversal
via 'Pix' pseudo indices, as described elsewhere.

   All object containers must perform either a 'X::X(X&)' (or 'X::X()'
followed by 'X::operator =(X&)') to copy objects into containers. (The
latter form is used for containers built from C++ arrays, like
'VHSets'). When containers are destroyed, they invoke 'X::~X()'. Any
objects used in containers must have well behaved constructors and
destructors. If you want to create containers that merely reference
(point to) objects that reside elsewhere, and are not copied or
destroyed inside the container, you must use containers of pointers,
not containers of objects.

   All prototypes are designed to generate *HOMOGENOUS* container
classes. There is no universally applicable method in C++ to support
heterogenous object collections with elements of various subclasses of
some specified base class. The only way to get heterogenous structures
is to use collections of pointers-to-objects, not collections of objects
(which also requires you to take responsibility for managing storage for
the objects pointed to yourself).

   For example, the following usage illustrates a commonly encountered
danger in trying to use container classes for heterogenous structures:

     class Base { int x; ...}
     class Derived : public Base { int y; ... }

     BaseVHSet s; // class BaseVHSet generated via something like
                  // 'genclass Base ref VHSet'

     void f()

```
    {
      Base b;
      s.add(b); // OK

      Derived d;
      s.add(d);  // (CHOP!)
    }
```

At the line flagged with '(CHOP!)', a 'Base::Base(Base&)' is called
inside 'Set::add(Base&)'--*not* 'Derived::Derived(Derived&)'.
Actually, in 'VHSet', a 'Base::operator =(Base&)', is used instead to
place the element in an array slot, but with the same effect. So only
the Base part is copied as a 'VHSet' element (a so-called
chopped-copy). In this case, it has an 'x' part, but no 'y' part; and a
Base, not Derived, vtable. Objects formed via chopped copies are rarely
sensible.

To avoid this, you must resort to pointers:

```
    typedef Base* BasePtr;

    BasePtrVHSet s; // class BaseVHSet generated via something like
                    // 'genclass BasePtr val VHSet'

    void f()
    {
      Base* bp = new Base;
      s.add(b);

      Base* dp = new Derived;
      s.add(d);  // works fine.

      // Don't forget to delete bp and dp sometime.
      // The VHSet won't do this for you.
    }
```

Example
=======

The prototypes can be difficult to use on first attempt. Here is an
example that may be helpful. The utilities in the 'proto-kit' simplify
much of the actions described, but are not used here.

Suppose you create a class 'Person', and want to make an Map that
links the social security numbers associated with each person. You start
off with a file 'Person.h'

```
    #include <String.h>

    class Person
    {
      String nm;
      String addr;
      //...
    public:
      const String& name() { return nm; }
```

```
        const String& address() { return addr; }
        void          print() { ... }
        //...
    }
```

And in file 'SSN.h',

```
    typedef unsigned int SSN;
```

Your first decision is what storage/usage strategy to use. There are
several reasonable alternatives here: You might create an "object
collection" of Persons, a "pointer collection" of pointers-to-Persons,
or even a simple String map, housing either copies of pointers to the
names of Persons, since other fields are unused for purposes of the
Map. In an object collection, instances of class Person "live" inside
the Map, while in a pointer collection, the instances live elsewhere.
Also, as above, if instances of subclasses of Person are to be used
inside the Map, you must use pointers. In a String Map, the same
difference holds, but now only for the name fields. Any of these
choices might make sense in particular applications.

The second choice is the Map implementation strategy. Either a tree
or a hash table might make sense. Suppose you want an AVL tree Map.
There are two things to now check. First, as an object collection, the
AVLMap requires that the elsement class contain an 'X(X&)' constructor.
In C++, if you don't specify such a constructor, one is constructed for
you, but it is a very good idea to always do this yourself, to avoid
surprises. In this example, you'd use something like
```
    class Person
    { ...;
        Person(const Person& p) :nm(p.nm), addr(p.addr) {}
    };
```

Also, an AVLMap requires a comparison function for elements in order
to maintain order. Rather than requiring you to write a particular
comparison function, a 'defs' file is consulted to determine how to
compare items. You must create and edit such a file.

Before creating 'Person.defs.h', you must first make one additional
decision. Should the Map member functions like 'm.contains(p)' take
arguments ('p') by reference (i.e., typed as 'int Map::contains(const
Person& p)' or by value (i.e., typed as 'int Map::contains(const Person
p)'. Generally, for user-defined classes, you want to pass by
reference, and for builtins and pointers, to pass by value. SO you
should pick by-reference.

You can now create 'Person.defs.h' via 'genclass Person ref defs'.
This creates a simple skeleton that you must edit. First, add '#include
"Person.h"' to the top. Second, edit the '<T>CMP(a,b)' macro to compare
on name, via

```
    #define <T>CMP(a, b) ( compare(a.name(), b.name()) )
```

which invokes the 'int compare(const String&, const String&)' function
from 'String.h'. Of course, you could define this in any other way as
well. In fact, the default versions in the skeleton turn out to be OK
(albeit inefficient) in this particular example.

You may also want to create file 'SSN.defs.h'. Here, choosing
call-by-value makes sense, and since no other capabilities (like
comparison functions) of the SSNs are used (and the defaults are OK
anyway), you'd type

```
genclass SSN val defs
```

and then edit to place '#include "SSN.h"' at the top.

Finally, you can generate the classes. First, generate the base
class for Maps via

```
genclass -2 Person ref SSN val Map
```

This generates only the abstract class, not the implementation, in file
'Person.SSN.Map.h' and 'Person.SSN.Map.cc'. To create the AVL
implementation, type

```
genclass -2 Person ref SSN val AVLMap
```

This creates the class 'PersonSSNAVLMap', in 'Person.SSN.AVLMap.h' and
'Person.SSN.AVLMap.cc'.

To use the AVL implementation, compile the two generated '.cc'
files, and specify '#include "Person.SSN.AVLMap.h"' in the application
program. All other files are included in the right ways automatically.

One last consideration, peculiar to Maps, is to pick a reasonable
default contents when declaring an AVLMap. Zero might be appropriate
here, so you might declare a Map,

```
PersonSSNAVLMap m((SSN)0);
```

Suppose you wanted a 'VHMap' instead of an 'AVLMap' Besides
generating different implementations, there are two differences in how
you should prepare the 'defs' file. First, because a VHMap uses a C++
array internally, and because C++ array slots are initialized
differently than single elements, you must ensure that class Person
contains (1) a no-argument constructor, and (2) an assignment operator.
You could arrange this via

```
class Person
{ ...;
    Person() {}
  void operator = (const Person& p) { nm = p.nm; addr = p.addr; }
};
```

(The lack of action in the constructor is OK here because 'Strings'
possess usable no-argument constructors.)

You also need to edit 'Person.defs.h' to indicate a usable hash
function and default capacity, via something like

```
#include <builtin.h>
#define <T>HASH(x)  (hashpjw(x.name().chars()))
#define DEFAULT_INITIAL_CAPACITY 1000
```

Since the `hashpjw` function from `builtin.h` is appropriate here.
Changing the default capacity to a value expected to exceed the actual
capacity helps to avoid "hidden" inefficiencies when a new VHMap is
created without overriding the default, which is all too easy to do.

Otherwise, everything is the same as above, substituting `VHMap` for
`AVLMap`.

1.10 libg++.guide/Representations

```
Variable-Sized Object Representation
*************************************
```

One of the first goals of the GNU C++ library is to enrich the kinds
of basic classes that may be considered as (nearly) "built into" C++. A
good deal of the inspiration for these efforts is derived from
considering features of other type-rich languages, particularly Common
Lisp and Scheme. The general characteristics of most class and friend
operators and functions supported by these classes has been heavily
influenced by such languages.

Four of these types, Strings, Integers, BitSets, and BitStrings (as
well as associated and/or derived classes) require representations
suitable for managing variable-sized objects on the free-store. The
basic technique used for all of these is the same, although various
details necessarily differ from class to class.

The general strategy for representing such objects is to create
chunks of memory that include both header information (e.g., the size
of the object), as well as the variable-size data (an array of some
sort) at the end of the chunk. Generally the maximum size of an object
is limited to something less than all of addressable memory, as a
safeguard. The minimum size is also limited so as not to waste
allocations expanding very small chunks. Internally, chunks are
allocated in blocks well-tuned to the performance of the `new` operator.

Class elements themselves are merely pointers to these chunks. Most
class operations are performed via inline "translation" functions that
perform the required operation on the corresponding representation.
However, constructors and assignments operate by copying entire
representations, not just pointers.

No attempt is made to control temporary creation in expressions and
functions involving these classes. Users of previous versions of the
classes will note the disappearance of both "Tmp" classes and reference
counting. These were dropped because, while they did improve
performance in some cases, they obscure class mechanics, lead
programmers into the false belief that they need not worry about such
things, and occasionally have paradoxical behavior.

These variable-sized object classes are integrated as well as
possible into C++. Most such classes possess converters that allow
automatic coercion both from and to builtin basic types. (e.g., char*

to and from String, long int to and from Integer, etc.). There are
pro's and con's to circular converters, since they can sometimes lead
to the conversion from a builtin type through to a class function and
back to a builtin type without any special attention on the part of the
programmer, both for better and worse.

   Most of these classes also provide special-case operators and
functions mixing basic with class types, as a way to avoid constructors
in cases where the operations do not rely on anything special about the
representations. For example, there is a special case concatenation
operator for a String concatenated with a char, since building the
result does not rely on anything about the String header. Again, there
are arguments both for and against this approach. Supporting these cases
adds a non-trivial degree of (mainly inline) function proliferation, but
results in more efficient operations. Efficiency wins out over parsimony
here, as part of the goal to produce classes that provide sufficient
functionality and efficiency so that programmers are not tempted to try
to manipulate or bypass the underlying representations.

1.11 libg++.guide/Expressions

Some guidelines for using expression-oriented classes
****

   The fact that C++ allows operators to be overloaded for user-defined
classes can make programming with library classes like 'Integer',
'String', and so on very convenient. However, it is worth becoming
familiar with some of the inherent limitations and problems associated
with such operators.

   Many operators are *constructive*, i.e., create a new object based
on some function of some arguments. Sometimes the creation of such
objects is wasteful. Most library classes supporting expressions
contain facilities that help you avoid such waste.

   For example, for 'Integer a, b, c; ...; c = a + b + a;', the plus
operator is called to sum a and b, creating a new temporary object as
its result. This temporary is then added with a, creating another
temporary, which is finally copied into c, and the temporaries are then
deleted. In other words, this code might have an effect similar to
'Integer a, b, c; ...; Integer t1(a); t1 += b; Integer t2(t1); t2 += a;
c = t2;'.

   For small objects, simple operators, and/or non-time/space critical
programs, creation of temporaries is not a big problem. However, often,
when fine-tuning a program, it may be a good idea to rewrite such code
in a less pleasant, but more efficient manner.

   For builtin types like ints, and floats, C and C++ compilers already
know how to optimize such expressions to reduce the need for
temporaries. Unfortunately, this is not true for C++ user defined
types, for the simple (but very annoying, in this context) reason that
nothing at all is guaranteed about the semantics of overloaded operators
and their interrelations. For example, if the above expression just

involved ints, not Integers, a compiler might internally convert the
statement into something like ' c = a; c += b; c+= a; ', or perhaps
something even more clever. But since C++ does not know that Integer
operator += has any relation to Integer operator +, A C++ compiler
cannot do this kind of expression optimization itself.

In many cases, you can avoid construction of temporaries simply by
using the assignment versions of operators whenever possible, since
these versions create no temporaries. However, for maximum flexibility,
most classes provide a set of "embedded assembly code" procedures that
you can use to fully control time, space, and evaluation strategies.
Most of these procedures are "three-address" procedures that take two
'const' source arguments, and a destination argument. The procedures
perform the appropriate actions, placing the results in the destination
(which is may involve overwriting old contents). These procedures are
designed to be fast and robust. In particular, aliasing is always
handled correctly, so that, for example 'add(x, x, x); ' is perfectly
OK. (The names of these procedures are listed along with the classes.)

For example, suppose you had an Integer expression ' a = (b - a) *
-(d / c); '

This would be compiled as if it were ' Integer t1=b-a; Integer
t2=d/c; Integer t3=-t2; Integer t4=t1*t3; a=t4;'

But, with some manual cleverness, you might yourself some up with '
sub(a, b, a); mul(a, d, a); div(a, c, a); '

A related phenomenon occurs when creating your own constructive
functions returning instances of such types. Suppose you wanted to
write function 'Integer f(const Integer& a) { Integer r = a; r += a;
return r; }'

This function, when called (as in ' a = f(a); ') demonstrates a
similar kind of wasted copy. The returned value r must be copied out of
the function before it can be used by the caller. In GNU C++, there is
an alternative via the use of named return values. Named return values
allow you to manipulate the returned object directly, rather than
requiring you to create a local inside a function and then copy it out
as the returned value. In this example, this can be done via 'Integer
f(const Integer& a) return r(a) { r += a; return; }'

A final guideline: The overloaded operators are very convenient, and
much clearer to use than procedural code. It is almost always a good
idea to make it right, *then* make it fast, by translating expression
code into procedural code after it is known to be correct.

1.12 libg++.guide/Pix

Pseudo-indexes
****

Many useful classes operate as containers of elements. Techniques for
accessing these elements from a container differ from class to class.

In the GNU C++ library, access methods have been partially standardized across different classes via the use of pseudo-indexes called 'Pixes'. A 'Pix' acts in some ways like an index, and in some ways like a pointer. (Their underlying representations are just 'void*' pointers). A 'Pix' is a kind of "key" that is translated into an element access by the class. In virtually all cases, 'Pixes' are pointers to some kind internal storage cells. The containers use these pointers to extract items.

   'Pixes' support traversal and inspection of elements in a collection using analogs of array indexing. However, they are pointer-like in that '0' is treated as an invalid 'Pix', and unsafe insofar as programmers can attempt to access nonexistent elements via dangling or otherwise invalid 'Pixes' without first checking for their validity.

   In general it is a very bad idea to perform traversals in the the midst of destructive modifications to containers.

   Typical applications might include code using the idiom
     for (Pix i = a.first(); i != 0; a.next(i)) use(a(i));
   for some container 'a' and function 'use'.

   Classes supporting the use of 'Pixes' always contain the following methods, assuming a container 'a' of element types of 'Base'.

'Pix i = a.first()'
    Set i to index the first element of a or 0 if a is empty.

'a.next(i)'
    advance i to the next element of a or 0 if there is no next
    element;

'Base x = a(i); a(i) = x;'
    a(i) returns a reference to the element indexed by i.

'int present = a.owns(i)'
    returns true if Pix i is a valid Pix in a. This is often a
    relatively slow operation, since the collection must usually
    traverse through elements to see if any correspond to the Pix.

   Some container classes also support backwards traversal via

'Pix i = a.last()'
    Set i to the last element of a or 0 if a is empty.

'a.prev(i)'
    sets i to the previous element in a, or 0 if there is none.

   Collections supporting elements with an equality operation possess

'Pix j = a.seek(x)'
    sets j to the index of the first occurrence of x, or 0 if x is not
    contained in a.

   Bag classes possess

'Pix j = a.seek(x, Pix from = 0)'

    sets j to the index of the next occurrence of x following i, or 0
    if x is not contained in a. If i == 0, the first occurrence is
    returned.

  Set, Bag, and PQ classes possess

'Pix j = a.add(x) (or a.enq(x) for priority queues)'
    add x to the collection, returning its Pix. The Pix of an item can
    change in collections where further additions and deletions
    involve the actual movement of elements (currently in OXPSet,
    OXPBag, XPPQ, VOHSet), but in all other cases, an item's Pix may
    be considered a permanent key to its location.

1.13 libg++.guide/Headers

Header files for interfacing C++ to C
****

    The following files are provided so that C++ programmers may invoke
common C library and system calls. The names and contents of these
files are subject to change in order to be compatible with the
forthcoming GNU C library. Other files, not listed here, are simply
C++-compatible interfaces to corresponding C library files.

'values.h'
    A collection of constants defining the numbers of bits in builtin
    types, minimum and maximum values, and the like. Most names are
    the same as those found in 'values.h' found on Sun systems.

'std.h'
    A collection of common system calls and 'libc.a' functions. Only
    those functions that can be declared without introducing new type
    definitions (socket structures, for example) are provided. Common
    'char*' functions (like 'strcmp') are among the declarations. All
    functions are declared along with their library names, so that
    they may be safely overloaded.

'string.h'
    This file merely includes '<std.h>', where string function
    prototypes are declared. This is a workaround for the fact that
    system 'string.h' and 'strings.h' files often differ in contents.

'osfcn.h'
    This file merely includes '<std.h>', where system function
    prototypes are declared.

'libc.h'
    This file merely includes '<std.h>', where C library function
    prototypes are declared.

'math.h'
    A collection of prototypes for functions usually found in libm.a,
    plus some '#define'd constants that appear to be consistent with
    those provided in the AT&T version. The value of 'HUGE' should be

checked before using. Declarations of all common math functions
are preceded with 'overload' declarations, since these are
commonly overloaded.

'stdio.h'
    Declaration of 'FILE' ('_iobuf'), common macros (like 'getc'), and
    function prototypes for 'libc.a' functions that operate on
    'FILE*''s. The value 'BUFSIZ' and the declaration of '_iobuf'
    should be checked before using.

'assert.h'
    C++ versions of assert macros.

'generic.h'
    String concatenation macros useful in creating generic classes.
    They are similar in function to the AT&T CC versions.

'new.h'
    Declarations of the default global operator new, the two-argument
    placement version, and associated error handlers.

1.14 libg++.guide/Builtin

Utility functions for built in types
****

   Files 'builtin.h' and corresponding '.cc' implementation files
contain various convenient inline and non-inline utility functions.
These include useful enumeration types, such as 'TRUE', 'FALSE' ,the
type definition for pointers to libg++ error handling functions, and
the following functions.

'long abs(long x); double abs(double x);'
    inline versions of abs. Note that the standard libc.a version,
    'int abs(int)' is *not* declared as inline.

'void clearbit(long& x, long b);'
    clears the b'th bit of x (inline).

'void setbit(long& x, long b);'
    sets the b'th bit of x (inline)

'int testbit(long x, long b);'
    returns the b'th bit of x (inline).

'int even(long y);'
    returns true if x is even (inline).

'int odd(long y);'
    returns true is x is odd (inline).

'int sign(long x); int sign(double x);'
    returns -1, 0, or 1, indicating whether x is less than, equal to,
    or greater than zero (inline).

`long gcd(long x, long y);'
     returns the greatest common divisor of x and y.

`long lcm(long x, long y);'
     returns the least common multiple of x and y.

`long lg(long x);'
     returns the floor of the base 2 log of x.

`long pow(long x, long y); double pow(double x, long y);'
     returns x to the integer power y using via the iterative O(log y)
     "Russian peasant" method.

`long sqr(long x); double sqr(double x);'
     returns x squared (inline).

`long sqrt(long y);'
     returns the floor of the square root of x.

`unsigned int hashpjw(const char* s);'
     a hash function for null-terminated char* strings using the method
     described in Aho, Sethi, & Ullman, p 436.

`unsigned int multiplicativehash(int x);'
     a hash function for integers that returns the lower bits of
     multiplying x by the golden ratio times pow(2, 32). See Knuth,
     Vol 3, p 508.

`unsigned int foldhash(double x);'
     a hash function for doubles that exclusive-or's the first and
     second words of x, returning the result as an integer.

`double start_timer()'
     Starts a process timer.

`double return_elapsed_time(double last_time)'
     Returns the process time since last_time. If last_time == 0
     returns the time since the last start_timer. Returns -1 if
     start_timer was not first called.

   File `Maxima.h' includes versions of `MAX, MIN' for builtin types.

   File `compare.h' includes versions of `compare(x, y)' for builtin
types. These return negative if the first argument is less than the
second, zero for equal, and positive for greater.

1.15 libg++.guide/New

Library dynamic allocation primitives
****

   Libg++ contains versions of `malloc, free, realloc' that were
designed to be well-tuned to C++ applications. The source file

'malloc.c' contains some design and implementation details. Here are
the major user-visible differences from most system malloc routines:

1. These routines *overwrite* storage of freed space. This means that
   it is never permissible to use a 'delete''d object in any way.
   Doing so will either result in trapped fatal errors or random
   aborts within malloc, free, or realloc.

2. The routines tend to perform well when a large number of objects
   of the same size are allocated and freed. You may find that it is
   not worth it to create your own special allocation schemes in such
   cases.

3. The library sets top-level 'operator new()' to call malloc and
   'operator delete()' to call free. Of course, you may override these
   definitions in C++ programs by creating your own operators that
   will take precedence over the library versions. However, if you do
   so, be sure to define *both* 'operator new()' and 'operator
   delete()'.

4. These routines do *not* support the odd convention, maintained by
   some versions of malloc, that you may call 'realloc' with a pointer
   that has been 'free''d.

5. The routines automatically perform simple checks on 'free''d
   pointers that can often determine whether users have accidentally
   written beyond the boundaries of allocated space, resulting in a
   fatal error.

6. The function 'malloc_usable_size(void* p)' returns the number of
   bytes actually allocated for 'p'. For a valid pointer (i.e., one
   that has been 'malloc''d or 'realloc''d but not yet 'free''d) this
   will return a number greater than or equal to the requested size,
   else it will normally return 0. Unfortunately, a non-zero return
   can not be an absolutely perfect indication of lack of error. If a
   chunk has been 'free''d but then re-allocated for a different
   purpose somewhere elsewhere, then 'malloc_usable_size' will return
   non-zero. Despite this, the function can be very valuable for
   performing run-time consistency checks.

7. 'malloc' requires 8 bytes of overhead per allocated chunk, plus a
   mmaximum alignment adjustment of 8 bytes. The number of bytes of
   usable space is exactly as requested, rounded to the nearest 8
   byte boundary.

8. The routines do *not* contain any synchronization support for
   multiprocessing. If you perform global allocation on a shared
   memory multiprocessor, you should disable compilation and use of
   libg++ malloc in the distribution 'Makefile' and use your system
   version of malloc.

1.16   libg++.guide/Stream

The old I/O library
****

   WARNING: This chapter describes classes that are *obsolete*. These
classes are normally not available when libg++ is installed normally.
The sources are currently included in the distribution, and you can
configure libg++ to use these classes instead of the new iostream
classes. This is only a temporary measure; you should convert your
code to use iostreams as soon as possible. The iostream classes
provide some compatibility support, but it is very incomplete (there is
no longer a 'File' class).

File-based classes
==================

   The 'File' class supports basic IO on Unix files. Operations are
based on common C stdio library functions.

   'File' serves as the base class for istreams, ostreams, and other
derived classes. It contains the interface between the Unix stdio file
library and these more structured classes. Most operations are
implemented as simple calls to stdio functions. 'File' class operations
are also fully compatible with raw system file reads and writes (like
the system 'read' and 'lseek' calls) when buffering is disabled (see
below). The 'FILE*' stdio file pointer is, however maintained as
protected. Classes derived from File may only use the IO operations
provided by File, which encompass essentially all stdio capabilities.

   The class contains four general kinds of functions: methods for
binding 'File's to physical Unix files, basic IO methods, file and
buffer control methods, and methods for maintaining logical and
physical file status.

   Binding and related tasks are accomplished via 'File' constructors
and destructors, and member functions 'open, close, remove, filedesc,
name, setname'.

   If a file name is provided in a constructor or open, it is
maintained as class variable 'nm' and is accessible via 'name'. If no
name is provided, then 'nm' remains null, except that 'Files' bound to
the default files stdin, stdout, and stderr are automatically given the
names '(stdin), (stdout), (stderr)' respectively. The function
'setname' may be used to change the internal name of the 'File'. This
does not change the name of the physical file bound to the File.

   The member function 'close' closes a file. The '~File' destructor
closes a file if it is open, except that stdin, stdout, and stderr are
flushed but left open for the system to close on program exit since
some systems may require this, and on others it does not matter.
'remove' closes the file, and then deletes it if possible by calling the
system function to delete the file with the name provided in the 'nm'
field.

Basic IO
========

   * `read' and `write' perform binary IO via stdio `fread' and
     `fwrite'.

   * `get' and `put' for chars invoke stdio `getc' and `putc' macros.

   * `put(const char* s)' outputs a null-terminated string via stdio
     `fputs'.

   * `unget' and `putback' are synonyms. Both call stdio `ungetc'.

File Control
============

   `flush', `seek', `tell', and `tell' call the corresponding stdio
functions.

   `flush(char)' and `fill()' call stdio `_flsbuf' and `_filbuf'
respectively.

   `setbuf' is mainly useful to turn off buffering in cases where
nonsequential binary IO is being performed. `raw' is a synonym for
`setbuf(_IONBF)'.  After a `f.raw()', using the stdio functions instead
of the system `read, write', etc., calls entails very little overhead.
Moreover, these become fully compatible with intermixed system calls
(e.g., `lseek(f.filedesc(), 0, 0)'). While intermixing `File' and
system IO calls is not at all recommended, this technique does allow
the `File' class to be used in conjunction with other functions and
libraries already set up to operate on file descriptors. `setbuf'
should be called at most once after a constructor or open, but before
any IO.

File Status
===========

   File status is maintained in several ways.

   A `File' may be checked for accessibility via `is_open()', which
returns true if the File is bound to a usable physical file,
`readable()', which returns true if the File can be read from (opened
for reading, and not in a _fail state), or `writable()', which returns
true if the File can be written to.

   `File' operations return their status via two means: failure and
success are represented via the logical state. Also, the return values
of invoked stdio and system functions that return useful numeric values
(not just failure/success flags) are held in a class variable
accessible via `iocount'. (This is useful, for example, in determining
the number of items actually read by the `read' function.)

   Like the AT&T i/o-stream classes, but unlike the description in the
Stroustrup book, p238, `rdstate()' returns the bitwise OR of `_eof',
`_fail' and `_bad', not necessarily distinct values. The functions
`eof()', `fail()', `bad()', and `good()' can be used to test for each of
these conditions independently.

   `_fail' becomes set for any input operation that could not read in
the desired data, and for other failed operations. As with all Unix IO,

'_eof' becomes true only when an input operations fails because of an
end of file. Therefore, '_eof' is not immediately true after the last
successful read of a file, but only after one final read attempt. Thus,
for input operations, '_fail' and '_eof' almost always become true at
the same time. 'bad' is set for unbound files, and may also be set by
applications in order to communicate input corruption. Conversely,
'_good' is defined as 0 and is returned by 'rdstate()' if all is well.

   The state may be modified via 'clear(flag)', which, despite its
name, sets the corresponding state_value flag. 'clear()' with no
arguments resets the state to '_good'. 'failif(int cond)' sets the
state to '_fail' only if 'cond' is true.

   Errors occuring during constructors and file opens also invoke the
function 'error'. 'error' in turn calls a resetable error handling
function pointed to by the non-member global variable
'File_error_handler' only if a system error has been generated. Since
'error' cannot tell if the current system error is actually responsible
for a failure, it may at times print out spurious messages. Three
error handlers are provided. The default, 'verbose_File_error_handler'
calls the system function 'perror' to print the corresponding error
message on standard error, and then returns to the caller.
'quiet_File_error_handler' does nothing, and simply returns.
'fatal_File_error_handler' prints the error and then aborts execution.
These three handlers, or any other user-defined error handlers can be
selected via the non-member function 'set_File_error_handler'.

   All read and write operations communicate either logical or physical
failure by setting the '_fail' flag. All further operations are
blocked if the state is in a '_fail' or'_bad' condition. Programmers
must explicitly use 'clear()' to reset the state in order to continue
IO processing after either a logical or physical failure. C
programmers who are unfamiliar with these conventions should note that,
unlike the stdio library, 'File' functions indicate IO success, status,
or failure solely through the state, not via return values of the
functions. The 'void*' operator or 'rdstate()' may be used to test
success. In particular, according to c++ conversion rules, the 'void*'
coercion is automatically applied whenever the 'File&' return value of
any 'File' function is tested in an 'if' or 'while'. Thus, for
example, an easy way to copy all of stdin to stdout until eof (at which
point 'get' fails) or some error is 'char c; while(cin.get(c) &&
cout.put(c));'.

   The current version of istreams and ostreams differs significantly
from previous versions in order to obtain compatibility with AT&T 1.2
streams. Most code using previous versions should still work. However,
the following features of 'File' are not incorporated in streams (they
are still present in 'File'): 'scan(const char* fmt...), remove(),
read(), write(), setbuf(), raw()'. Additionally, the feature of
previous streams that allowed free intermixing of stream and stdio
input and output is no longer guaranteed to always behave as desired.

1.17   libg++.guide/Obstack

The Obstack class
****

   The 'Obstack' class is a simple rewrite of the C obstack macros and
functions provided in the GNU CC compiler source distribution.

   Obstacks provide a simple method of creating and maintaining a string
table, optimized for the very frequent task of building strings
character-by-character, and sometimes keeping them, and sometimes not.
They seem especially useful in any parsing application. One of the test
files demonstrates usage.

   A brief summary:
'grow'
     places something on the obstack without committing to wrap it up
     as a single entity yet.

'finish'
     wraps up a constructed object as a single entity, and returns the
     pointer to its start address.

'copy'
     places things on the obstack, and *does* wrap them up. 'copy' is
     always equivalent to first grow, then finish.

'free'
     deletes something, and anything else put on the obstack since its
     creation.

   The other functions are less commonly needed:
'blank'
     is like grow, except it just grows the space by size units without
     placing anything into this space

'alloc'
     is like 'blank', but it wraps up the object and returns its
     starting address.

'chunk_size, base, next_free, alignment_mask, size, room'
     returns the appropriate class variables.

'grow_fast'
     places a character on the obstack without checking if there is
     enough room.

'blank_fast'
     like 'blank', but without checking if there is enough room.

'shrink(int n)'
     shrink the current chunk by n bytes.

'contains(void* addr)'
     returns true if the Obstack holds the address addr.

   Here is a lightly edited version of the original C documentation:

These functions operate a stack of objects. Each object starts life
small, and may grow to maturity. (Consider building a word syllable by
syllable.) An object can move while it is growing. Once it has been
"finished" it never changes address again. So the "top of the stack"
is typically an immature growing object, while the rest of the stack is
of mature, fixed size and fixed address objects.

These routines grab large chunks of memory, using the GNU C++ 'new'
operator. On occasion, they free chunks, via 'delete'.

Each independent stack is represented by a Obstack.

One motivation for this package is the problem of growing char
strings in symbol tables. Unless you are a "fascist pig with a
read-only mind" [Gosper's immortal quote from HAKMEM item 154, out of
context] you would not like to put any arbitrary upper limit on the
length of your symbols.

In practice this often means you will build many short symbols and a
few long symbols. At the time you are reading a symbol you don't know
how long it is. One traditional method is to read a symbol into a
buffer, 'realloc()'ating the buffer every time you try to read a symbol
that is longer than the buffer. This is beaut, but you still will want
to copy the symbol from the buffer to a more permanent symbol-table
entry say about half the time.

With obstacks, you can work differently. Use one obstack for all
symbol names. As you read a symbol, grow the name in the obstack
gradually. When the name is complete, finalize it. Then, if the
symbol exists already, free the newly read name.

The way we do this is to take a large chunk, allocating memory from
low addresses. When you want to build a symbol in the chunk you just
add chars above the current "high water mark" in the chunk. When you
have finished adding chars, because you got to the end of the symbol,
you know how long the chars are, and you can create a new object.
Mostly the chars will not burst over the highest address of the chunk,
because you would typically expect a chunk to be (say) 100 times as
long as an average object.

In case that isn't clear, when we have enough chars to make up the
object, *they are already contiguous in the chunk* (guaranteed) so we
just point to it where it lies. No moving of chars is needed and this
is the second win: potentially long strings need never be explicitly
shuffled. Once an object is formed, it does not change its address
during its lifetime.

When the chars burst over a chunk boundary, we allocate a larger
chunk, and then copy the partly formed object from the end of the old
chunk to the beginning of the new larger chunk. We then carry on
accreting characters to the end of the object as we normally would.

A special version of grow is provided to add a single char at a time
to a growing object.

Summary:

* We allocate large chunks.

* We carve out one object at a time from the current chunk.

* Once carved, an object never moves.

* We are free to append data of any size to the currently growing object.

* Exactly one object is growing in an obstack at any one time.

* You can run one obstack per control block.

* You may have as many control blocks as you dare.

* Because of the way we do it, you can 'unwind' a obstack back to a previous state. (You may remove objects much as you would with a stack.)

   The obstack data structure is used in many places in the GNU C++ compiler.

   Differences from the the GNU C version
1. The obvious differences stemming from the use of classes and inline functions instead of structs and macros. The C 'init' and 'begin' macros are replaced by constructors.

2. Overloaded function names are used for grow (and others), rather than the C 'grow', 'grow0', etc.

3. All dynamic allocation uses the the built-in 'new' operator. This restricts flexibility by a little, but maintains compatibility with usual C++ conventions.

4. There are now two versions of finish:

    1. finish() behaves like the C version.

    2. finish(char terminator) adds 'terminator', and then calls 'finish()'. This enables the normal invocation of 'finish(0)' to wrap up a string being grown character-by-character.

5. There are special versions of grow(const char* s) and copy(const char* s) that add the null-terminated string 's' after computing its length.

6. The shrink and contains functions are provided.

1.18 libg++.guide/AllocRing

The AllocRing class
****

   An AllocRing is a bounded ring (circular list), each of whose
elements contains a pointer to some space allocated via 'new
char[some_size]'. The entries are used cyclicly. The size, n, of the
ring is fixed at construction. After that, every nth use of the ring
will reuse (or reallocate) the same space. AllocRings are needed in
order to temporarily hold chunks of space that are needed transiently,
but across constructor-destructor scopes. They mainly useful for storing
strings containing formatted characters to print across various
functions and coercions. These strings are needed across routines, so
may not be deleted in any one of them, but should be recovered at some
point. In other words, an AllocRing is an extremely simple minded
garbage collection mechanism. The GNU C++ library used to use one
AllocRing for such formatting purposes, but it is being phased out, and
is now only used by obsolete functions. These days, AllocRings are
probably not very useful.

   Support includes:

'AllocRing a(int n)'
    constructs an Alloc ring with n entries, all null.

'void* mem = a.alloc(sz)'
    moves the ring pointer to the next entry, and reuses the space if
    their is enough, also allocates space via new char[sz].

'int present = a.contains(void* ptr)'
    returns true if ptr is held in one of the ring entries.

'a.clear()'
    deletes all space pointed to in any entry. This is called
    automatically upon destruction.

'a.free(void* ptr)'
    If ptr is one of the entries, calls delete of the pointer, and
    resets to entry pointer to null.

1.19 libg++.guide/String

The String class
****

   The 'String' class is designed to extend GNU C++ to support string
processing capabilities similar to those in languages like Awk. The
class provides facilities that ought to be convenient and efficient
enough to be useful replacements for 'char*' based processing via the C
string library (i.e., 'strcpy, strcmp,' etc.) in many applications.
Many details about String representations are described in the
Representation section.

   A separate 'SubString' class supports substring extraction and
modification operations. This is implemented in a way that user
programs never directly construct or represent substrings, which are
only used indirectly via String operations.

Another separate class, 'Regex' is also used indirectly via String
operations in support of regular expression searching, matching, and the
like. The Regex class is based entirely on the GNU Emacs regex
functions. See Syntax of Regular Expressions, for a full explanation
of regular expression syntax. (For implementation details, see the
internal documentation in files 'regex.h' and 'regex.c'.)

Constructors
============

    Strings are initialized and assigned as in the following examples:

'String x; String y = 0; String z = "";'
    Set x, y, and z to the nil string. Note that either 0 or "" may
    always be used to refer to the nil string.

'String x = "Hello"; String y("Hello");'
    Set x and y to a copy of the string "Hello".

'String x = 'A'; String y('A');'
    Set x and y to the string value "A"

'String u = x; String v(x);'
    Set u and v to the same string as String x

'String u = x.at(1,4); String v(x.at(1,4));'
    Set u and v to the length 4 substring of x starting at position 1
    (counting indexes from 0).

'String x("abc", 2);'
    Sets x to "ab", i.e., the first 2 characters of "abc".

'String x = dec(20);'
    Sets x to "20". As here, Strings may be initialized or assigned
    the results of any 'char*' function.

    There are no directly accessible forms for declaring SubString
variables.

    The declaration 'Regex r("[a-zA-Z_][a-zA-Z0-9_]*");' creates a
compiled regular expression suitable for use in String operations
described below. (In this case, one that matches any C++ identifier).
The first argument may also be a String. Be careful in distinguishing
the role of backslashes in quoted GNU C++ char* constants versus those
in Regexes. For example, a Regex that matches either one or more tabs
or all strings beginning with "ba" and ending with any number of
occurrences of "na" could be declared as 'Regex r =
"\(\t+\)\|\(ba\(na\)*\)"' Note that only one backslash is needed
to signify the tab, but two are needed for the parenthesization and
virgule, since the GNU C++ lexical analyzer decodes and strips
backslashes before they are seen by Regex.

    There are three additional optional arguments to the Regex
constructor that are less commonly useful:

'fast (default 0)'

     'fast' may be set to true (1) if the Regex should be
     "fast-compiled". This causes an additional compilation step that
     is generally worthwhile if the Regex will be used many times.

'bufsize (default max(40, length of the string))'
     This is an estimate of the size of the internal compiled
     expression. Set it to a larger value if you know that the
     expression will require a lot of space. If you do not know, do not
     worry: realloc is used if necessary.

'transtable (default none == 0)'
     The address of a byte translation table (a char[256]) that
     translates each character before matching.

   As a convenience, several Regexes are predefined and usable in any
program. Here are their declarations from 'String.h'.

```
     extern Regex RXwhite;       // = "[ \n\t]+"
     extern Regex RXint;         // = "-?[0-9]+"
     extern Regex RXdouble;      // = "-?\(\([0-9]+\.[0-9]*\)\|
                                 //      \([0-9]+\)\|
                                 //      \(\.[0-9]+\)\)
                                 //      \([eE][---+]?[0-9]+\)?"
     extern Regex RXalpha;       // = "[A-Za-z]+"
     extern Regex RXlowercase;   // = "[a-z]+"
     extern Regex RXuppercase;   // = "[A-Z]+"
     extern Regex RXalphanum;    // = "[0-9A-Za-z]+"
     extern Regex RXidentifier;  // = "[A-Za-z_][A-Za-z0-9_]*"
```

Examples
========

   Most 'String' class capabilities are best shown via example. The
examples below use the following declarations.

```
        String x = "Hello";
        String y = "world";
        String n = "123";
        String z;
        char*  s = ",";
        String lft, mid, rgt;
        Regex  r = "e[a-z]*o";
        Regex  r2("/[a-z]*/");
        char   c;
        int    i, pos, len;
        double f;
        String words[10];
        words[0] = "a";
        words[1] = "b";
        words[2] = "c";
```

Comparing, Searching and Matching
=================================

   The usual lexicographic relational operators ('==, !=, <, <=, >, >=')
are defined. A functional form 'compare(String, String)' is also
provided, as is 'fcompare(String, String)', which compares Strings

without regard for upper vs. lower case.

   All other matching and searching operations are based on some form
of the (non-public) 'match' and 'search' functions. 'match' and
'search' differ in that 'match' attempts to match only at the given
starting position, while 'search' starts at the position, and then
proceeds left or right looking for a match. As seen in the following
examples, the second optional 'startpos' argument to functions using
'match' and 'search' specifies the starting position of the search: If
non-negative, it results in a left-to-right search starting at position
'startpos', and if negative, a right-to-left search starting at
position 'x.length() + startpos'. In all cases, the index returned is
that of the beginning of the match, or -1 if there is no match.

   Three String functions serve as front ends to 'search' and 'match'.
'index' performs a search, returning the index, 'matches' performs a
match, returning nonzero (actually, the length of the match) on success,
and 'contains' is a boolean function performing either a search or
match, depending on whether an index argument is provided:

'x.index("lo")'
     returns the zero-based index of the leftmost occurrence of
     substring "lo" (3, in this case). The argument may be a String,
     SubString, char, char*, or Regex.

'x.index("l", 2)'
     returns the index of the first of the leftmost occurrence of "l"
     found starting the search at position x[2], or 2 in this case.

'x.index("l", -1)'
     returns the index of the rightmost occurrence of "l", or 3 here.

'x.index("l", -3)'
     returns the index of the rightmost occurrence of "l" found by
     starting the search at the 3rd to the last position of x,
     returning 2 in this case.

'pos = r.search("leo", 3, len, 0)'
     returns the index of r in the 'char*' string of length 3, starting
     at position 0, also placing the length of the match in reference
     parameter len.

'x.contains("He")'
     returns nonzero if the String x contains the substring "He". The
     argument may be a String, SubString, char, char*, or Regex.

'x.contains("el", 1)'
     returns nonzero if x contains the substring "el" at position 1.
     As in this example, the second argument to 'contains', if present,
     means to match the substring only at that position, and not to
     search elsewhere in the string.

'x.contains(RXwhite);'
     returns nonzero if x contains any whitespace (space, tab, or
     newline). Recall that 'RXwhite' is a global whitespace Regex.

'x.matches("lo", 3)'

```
     returns nonzero if x starting at position 3 exactly matches "lo",
     with no trailing characters (as it does in this example).
```

`x.matches(r)'
     returns nonzero if String x as a whole matches Regex r.

`int f = x.freq("l")'
     returns the number of distinct, nonoverlapping matches to the
     argument (2 in this case).

Substring extraction
====================

   Substrings may be extracted via the `at', `before', `through',
`from', and `after' functions. These behave as either lvalues or
rvalues.

`z = x.at(2, 3)'
     sets String z to be equal to the length 3 substring of String x
     starting at zero-based position 2, setting z to "llo" in this
     case. A nil String is returned if the arguments don't make sense.

`x.at(2, 2) = "r"'
     Sets what was in positions 2 to 3 of x to "r", setting x to "Hero"
     in this case. As indicated here, SubString assignments may be of
     different lengths.

`x.at("He") = "je";'
     x("He") is the substring of x that matches the first occurrence of
     it's argument. The substitution sets x to "jello". If "He" did not
     occur, the substring would be nil, and the assignment would have
     no effect.

`x.at("l", -1) = "i";'
     replaces the rightmost occurrence of "l" with "i", setting x to
     "Helio".

`z = x.at(r)'
     sets String z to the first match in x of Regex r, or "ello" in this
     case. A nil String is returned if there is no match.

`z = x.before("o")'
     sets z to the part of x to the left of the first occurrence of
     "o", or "Hell" in this case. The argument may also be a String,
     SubString, or Regex. (If there is no match, z is set to "".)

`x.before("ll") = "Bri";'
     sets the part of x to the left of "ll" to "Bri", setting x to
     "Brillo".

`z = x.before(2)'
     sets z to the part of x to the left of x[2], or "He" in this case.

`z = x.after("Hel")'
     sets z to the part of x to the right of "Hel", or "lo" in this
     case.

`z = x.through("el")`
    sets z to the part of x up and including "el", or "Hel" in this
    case.

`z = x.from("el")`
    sets z to the part of x from "el" to the end, or "ello" in this
    case.

`x.after("Hel") = "p";`
    sets x to "Help";

`z = x.after(3)`
    sets z to the part of x to the right of x[3] or "o" in this case.

`z = "  ab c"; z = z.after(RXwhite)`
    sets z to the part of its old string to the right of the first
    group of whitespace, setting z to "ab c"; Use gsub(below) to strip
    out multiple occurrences of whitespace or any pattern.

`x[0] = 'J';`
    sets the first element of x to 'J'. x[i] returns a reference to
    the ith element of x, or triggers an error if i is out of range.

`common_prefix(x, "Help")`
    returns the String containing the common prefix of the two Strings
    or "Hel" in this case.

`common_suffix(x, "to")`
    returns the String containing the common suffix of the two Strings
    or "o" in this case.

```
Concatenation
=============
```

`z = x + s + ' ' + y.at("w") + y.after("w") + ".";`
    sets z to "Hello, world."

`x += y;`
    sets x to "Helloworld"

`cat(x, y, z)`
    A faster way to say z = x + y.

`cat(z, y, x, x)`
    Double concatenation; A faster way to say x = z + y + x.

`y.prepend(x);`
    A faster way to say y = x + y.

`z = replicate(x, 3);`
    sets z to "HelloHelloHello".

`z = join(words, 3, "/")`
    sets z to the concatenation of the first 3 Strings in String array
    words, each separated by "/", setting z to "a/b/c" in this case.
    The last argument may be "" or 0, indicating no separation.

Other manipulations
===================

`z = "this string has five words"; i = split(z, words, 10, RXwhite);'
    sets up to 10 elements of String array words to the parts of z
    separated by whitespace, and returns the number of parts actually
    encountered (5 in this case). Here, words[0] = "this", words[1] =
    "string", etc. The last argument may be any of the usual. If
    there is no match, all of z ends up in words[0]. The words array
    is *not* dynamically created by split.

`int nmatches x.gsub("l","ll")'
    substitutes all original occurrences of "l" with "ll", setting x
    to "Hellllo". The first argument may be any of the usual,
    including Regex. If the second argument is "" or 0, all
    occurrences are deleted. gsub returns the number of matches that
    were replaced.

`z = x + y; z.del("loworl");'
    deletes the leftmost occurrence of "loworl" in z, setting z to
    "Held".

`z = reverse(x)'
    sets z to the reverse of x, or "olleH".

`z = upcase(x)'
    sets z to x, with all letters set to uppercase, setting z to
    "HELLO"

`z = downcase(x)'
    sets z to x, with all letters set to lowercase, setting z to
    "hello"

`z = capitalize(x)'
    sets z to x, with the first letter of each word set to uppercase,
    and all others to lowercase, setting z to "Hello"

`x.reverse(), x.upcase(), x.downcase(), x.capitalize()'
    in-place, self-modifying versions of the above.

Reading, Writing and Conversion
==============================

`cout << x'
    writes out x.

`cout << x.at(2, 3)'
    writes out the substring "llo".

`cin >> x'
    reads a whitespace-bounded string into x.

`x.length()'
    returns the length of String x (5, in this case).

`s = (const char*)x'
    can be used to extract the `char*' char array. This coercion is

useful for sending a String as an argument to any function
expecting a 'const char*' argument (like 'atoi', and
'File::open'). This operator must be used with care, since the
conversion returns a pointer to 'String' internals without copying
the characters: The resulting '(char*)' is only valid until the
next String operation, and you must not modify it. (The
conversion is defined to return a const value so that GNU C++ will
produce warning and/or error messages if changes are attempted.)

1.20 libg++.guide/Integer

The Integer class.
****

   The 'Integer' class provides multiple precision integer arithmetic
facilities. Some representation details are discussed in the
Representation section.

   'Integers' may be up to 'b * ((1 << b) - 1)' bits long, where 'b' is
the number of bits per short (typically 1048560 bits when 'b = 16').
The implementation assumes that a 'long' is at least twice as long as a
'short'. This assumption hides beneath almost all primitive operations,
and would be very difficult to change. It also relies on correct
behavior of *unsigned* arithmetic operations.

   Some of the arithmetic algorithms are very loosely based on those
provided in the MIT Scheme 'bignum.c' release, which is Copyright (c)
1987 Massachusetts Institute of Technology. Their use here falls within
the provisions described in the Scheme release.

   Integers may be constructed in the following ways:
'Integer x;'
     Declares an uninitialized Integer.

'Integer x = 2; Integer y(2);'
     Set x and y to the Integer value 2;

'Integer u(x); Integer v = x;'
     Set u and v to the same value as x.

 - Method: long Integer::as_long() const
     Used to coerce an 'Integer' back into longs via the 'long'
     coercion operator. If the Integer cannot fit into a long, this
     returns MINLONG or MAXLONG (depending on the sign) where MINLONG
     is the most negative, and MAXLONG is the most positive
     representable long.

 - Method: int Integer::fits_in_long() const
     Returns true iff the 'Integer' is '< MAXLONG' and '> MINLONG'.

 - Method: double Integer::as_double() const
     Coerce the 'Integer' to a 'double', with potential loss of
     precision. '+/-HUGE' is returned if the Integer cannot fit into a
     double.

- Method: int Integer::fits_in_double() const
    Returns true iff the 'Integer' can fit into a double.

   All of the usual arithmetic operators are provided ('+, -, *, /, %,
+=, ++, -=, --, *=, /=, %=, ==, !=, <, <=, >, >='). All operators
support special versions for mixed arguments of Integers and regular
C++ longs in order to avoid useless coercions, as well as to allow
automatic promotion of shorts and ints to longs, so that they may be
applied without additional Integer coercion operators. The only
operators that behave differently than the corresponding int or long
operators are '++' and '--'. Because C++ does not distinguish prefix
from postfix application, these are declared as 'void' operators, so
that no confusion can result from applying them as postfix. Thus, for
Integers x and y, ' ++x; y = x; ' is correct, but ' y = ++x; ' and ' y
= x++; ' are not.

   Bitwise operators ('~', '&', '|', '^', '<<', '>>', '&=', '|=', '^=',
'<<=', '>>=') are also provided. However, these operate on
sign-magnitude, rather than two's complement representations. The sign
of the result is arbitrarily taken as the sign of the first argument.
For example, 'Integer(-3) & Integer(5)' returns 'Integer(-1)', not -3,
as it would using two's complement. Also, '~', the complement operator,
complements only those bits needed for the representation. Bit
operators are also provided in the BitSet and BitString classes. One of
these classes should be used instead of Integers when the results of
bit manipulations are not interpreted numerically.

   The following utility functions are also provided. (All arguments
are Integers unless otherwise noted).

 - Function: void divide(const Integer& X, const Integer& Y, Integer&
        Q, Integer& R)
    Sets Q to the quotient and R to the remainder of X and Y. (Q and
    R are returned by reference).

 - Function: Integer pow(const Integer& X, const Integer& P)
    Returns X raised to the power P.

 - Function: Integer Ipow(long X, long P)
    Returns X raised to the power P.

 - Function: Integer gcd(const Integer& X, const Integer& P)
    Returns the greatest common divisor of X and Y.

 - Function: Integer lcm(const Integer& X, const Integer& P)
    Returns the least common multiple of X and Y.

 - Function: Integer abs(const Integer& X
    Returns the absolute value of X.

 - Method: void Integer::negate()
    Negates 'this' in place.

'Integer sqr(x)'
    returns x * x;

`Integer sqrt(x)'
     returns the floor of the square root of x.

`long lg(x);'
     returns the floor of the base 2 logarithm of abs(x)

`int sign(x)'
     returns -1 if x is negative, 0 if zero, else +1. Using `if
     (sign(x) == 0)' is a generally faster method of testing for zero
     than using relational operators.

`int even(x)'
     returns true if x is an even number

`int odd(x)'
     returns true if x is an odd number.

`void setbit(Integer& x, long b)'
     sets the b'th bit (counting right-to-left from zero) of x to 1.

`void clearbit(Integer& x, long b)'
     sets the b'th bit of x to 0.

`int testbit(Integer x, long b)'
     returns true if the b'th bit of x is 1.

`Integer atoI(char* asciinumber, int base = 10);'
     converts the base base char* string into its Integer form.

`void Integer::printon(ostream& s, int base = 10, int width = 0);'
     prints the ascii string value of `(*this)' as a base `base'
     number, in field width at least `width'.

`ostream << x;'
     prints x in base ten format.

`istream >> x;'
     reads x as a base ten number.

`int compare(Integer x, Integer y)'
     returns a negative number if x<y, zero if x==y, or positive if x>y.

`int ucompare(Integer x, Integer y)'
     like compare, but performs unsigned comparison.

`add(x, y, z)'
     A faster way to say z = x + y.

`sub(x, y, z)'
     A faster way to say z = x - y.

`mul(x, y, z)'
     A faster way to say z = x * y.

`div(x, y, z)'
     A faster way to say z = x / y.

```
'mod(x, y, z)'
     A faster way to say z = x % y.

'and(x, y, z)'
     A faster way to say z = x & y.

'or(x, y, z)'
     A faster way to say z = x | y.

'xor(x, y, z)'
     A faster way to say z = x ^ y.

'lshift(x, y, z)'
     A faster way to say z = x << y.

'rshift(x, y, z)'
     A faster way to say z = x >> y.

'pow(x, y, z)'
     A faster way to say z = pow(x, y).

'complement(x, z)'
     A faster way to say z = ~x.

'negate(x, z)'
     A faster way to say z = -x.
```

1.21 libg++.guide/Rational

```
The Rational Class
******************
```

   Class 'Rational' provides multiple precision rational number arithmetic. All rationals are maintained in simplest form (i.e., with the numerator and denominator relatively prime, and with the denominator strictly positive). Rational arithmetic and relational operators are provided ('+, -, *, /, +=, -=, *=, /=, ==, !=, <, <=, >, >='). Operations resulting in a rational number with zero denominator trigger an exception.

   Rationals may be constructed and used in the following ways:

'Rational x;'
     Declares an uninitialized Rational.

'Rational x = 2; Rational y(2);'
     Set x and y to the Rational value 2/1;

'Rational x(2, 3);'
     Sets x to the Rational value 2/3;

'Rational x = 1.2;'
     Sets x to a Rational value close to 1.2. Any double precision value
     may be used to construct a Rational. The Rational will possess

exactly as much precision as the double. Double values that do not
have precise floating point equivalents (like 1.2) produce
similarly imprecise rational values.

`Rational x(Integer(123), Integer(4567));'
     Sets x to the Rational value 123/4567.

`Rational u(x); Rational v = x;'
     Set u and v to the same value as x.

`double(Rational x)'
     A Rational may be coerced to a double with potential loss of
     precision. +/-HUGE is returned if it will not fit.

`Rational abs(x)'
     returns the absolute value of x.

`void x.negate()'
     negates x.

`void x.invert()'
     sets x to 1/x.

`int sign(x)'
     returns 0 if x is zero, 1 if positive, and -1 if negative.

`Rational sqr(x)'
     returns x * x.

`Rational pow(x, Integer y)'
     returns x to the y power.

`Integer x.numerator()'
     returns the numerator.

`Integer x.denominator()'
     returns the denominator.

`Integer floor(x)'
     returns the greatest Integer less than x.

`Integer ceil(x)'
     returns the least Integer greater than x.

`Integer trunc(x)'
     returns the Integer part of x.

`Integer round(x)'
     returns the nearest Integer to x.

`int compare(x, y)'
     returns a negative, zero, or positive number signifying whether x
     is less than, equal to, or greater than y.

`ostream << x;'
     prints x in the form num/den, or just num if the denominator is
     one.

`istream >> x;'
     reads x in the form num/den, or just num in which case the
     denominator is set to one.

`add(x, y, z)'
     A faster way to say z = x + y.

`sub(x, y, z)'
     A faster way to say z = x - y.

`mul(x, y, z)'
     A faster way to say z = x * y.

`div(x, y, z)'
     A faster way to say z = x / y.

`pow(x, y, z)'
     A faster way to say z = pow(x, y).

`negate(x, z)'
     A faster way to say z = -x.

1.22 libg++.guide/Complex

The Complex class.
****

   Class `Complex' is implemented in a way similar to that described by
Stroustrup. In keeping with libg++ conventions, the class is named
`Complex', not `complex'. Complex arithmetic and relational operators
are provided (`+, -, *, /, +=, -=, *=, /=, ==, !=''). Attempted
division by (0, 0) triggers an exception.

   Complex numbers may be constructed and used in the following ways:

`Complex x;'
     Declares an uninitialized Complex.

`Complex x = 2; Complex y(2.0);'
     Set x and y to the Complex value (2.0, 0.0);

`Complex x(2, 3);'
     Sets x to the Complex value (2, 3);

`Complex u(x); Complex v = x;'
     Set u and v to the same value as x.

`double real(Complex& x);'
     returns the real part of x.

`double imag(Complex& x);'
     returns the imaginary part of x.

`double abs(Complex& x);'
     returns the magnitude of x.

`double norm(Complex& x);'
     returns the square of the magnitude of x.

`double arg(Complex& x);'
     returns the argument (amplitude) of x.

`Complex polar(double r, double t = 0.0);'
     returns a Complex with abs of r and arg of t.

`Complex conj(Complex& x);'
     returns the complex conjugate of x.

`Complex cos(Complex& x);'
     returns the complex cosine of x.

`Complex sin(Complex& x);'
     returns the complex sine of x.

`Complex cosh(Complex& x);'
     returns the complex hyperbolic cosine of x.

`Complex sinh(Complex& x);'
     returns the complex hyperbolic sine of x.

`Complex exp(Complex& x);'
     returns the exponential of x.

`Complex log(Complex& x);'
     returns the natural log of x.

`Complex pow(Complex& x, long p);'
     returns x raised to the p power.

`Complex pow(Complex& x, Complex& p);'
     returns x raised to the p power.

`Complex sqrt(Complex& x);'
     returns the square root of x.

`ostream << x;'
     prints x in the form (re, im).

`istream >> x;'
     reads x in the form (re, im), or just (re) or re in which case the
     imaginary part is set to zero.

1.23   libg++.guide/Fix

```
Fixed precision numbers
***********************
```

Classes 'Fix16', 'Fix24', 'Fix32', and 'Fix48' support operations on
16, 24, 32, or 48 bit quantities that are considered as real numbers in
the range [-1, +1). Such numbers are often encountered in digital
signal processing applications. The classes may be be used in isolation
or together. Class 'Fix32' operations are entirely self-contained.
Class 'Fix16' operations are self-contained except that the
multiplication operation 'Fix16 * Fix16' returns a 'Fix32'. 'Fix24' and
'Fix48' are similarly related.

   The standard arithmetic and relational operations are supported
('=', '+', '-', '*', '/', '<<', '>>', '+=', '-=', '*=', '/=', '<<=',
'>>=', '==', '!=', '<', '<=', '>', '>='). All operations include
provisions for special handling in cases where the result exceeds +/-
1.0. There are two cases that may be handled separately: "overflow"
where the results of addition and subtraction operations go out of
range, and all other "range errors" in which resulting values go
off-scale (as with division operations, and assignment or
initialization with off-scale values). In signal processing
applications, it is often useful to handle these two cases differently.
Handlers take one argument, a reference to the integer mantissa of the
offending value, which may then be manipulated. In cases of overflow,
this value is the result of the (integer) arithmetic computation on the
mantissa; in others it is a fully saturated (i.e., most positive or
most negative) value. Handling may be reset to any of several provided
functions or any other user-defined function via 'set_overflow_handler'
and 'set_range_error_handler'. The provided functions for 'Fix16' are
as follows (corresponding functions are also supported for the others).

'Fix16_overflow_saturate'
     The default overflow handler. Results are "saturated": positive
     results are set to the largest representable value (binary
     0.111111...), and negative values to -1.0.

'Fix16_ignore'
     Performs no action. For overflow, this will allow addition and
     subtraction operations to "wrap around" in the same manner as
     integer arithmetic, and for saturation, will leave values
     saturated.

'Fix16_overflow_warning_saturate'
     Prints a warning message on standard error, then saturates the
     results.

'Fix16_warning'
     The default range_error handler. Prints a warning message on
     standard error; otherwise leaving the argument unmodified.

'Fix16_abort'
     prints an error message on standard error, then aborts execution.

   In addition to arithmetic operations, the following are provided:

'Fix16 a = 0.5;'
     Constructs fixed precision objects from double precision values.
     Attempting to initialize to a value outside the range invokes the
     range_error handler, except, as a convenience, initialization to
     1.0 sets the variable to the most positive representable value

(binary 0.1111111...) without invoking the handler.

'short& mantissa(a); long& mantissa(b);'
     return a * pow(2, 15) or b * pow(2, 31) as an integer. These are
     returned by reference, to enable "manual" data manipulation.

'double value(a); double value(b);'
     return a or b as floating point numbers.

1.24 libg++.guide/Bit

Classes for Bit manipulation
****

   libg++ provides several different classes supporting the use and
manipulation of collections of bits in different ways.

   * Class 'Integer' provides "integer" semantics. It supports
     manipulation of bits in ways that are often useful when treating
     bit arrays as numerical (integer) quantities. This class is
     described elsewhere.

   * Class 'BitSet' provides "set" semantics. It supports operations
     useful when treating collections of bits as representing
     potentially infinite sets of integers.

   * Class 'BitSet32' supports fixed-length BitSets holding exactly 32
     bits.

   * Class 'BitSet256' supports fixed-length BitSets holding exactly
     256 bits.

   * Class 'BitString' provides "string" (or "vector") semantics. It
     supports operations useful when treating collections of bits as
     strings of zeros and ones.

   These classes also differ in the following ways:

   * BitSets are logically infinite. Their space is dynamically altered
     to adjust to the smallest number of consecutive bits actually
     required to represent the sets. Integers also have this property.
     BitStrings are logically finite, but their sizes are internally
     dynamically managed to maintain proper length. This means that,
     for example, BitStrings are concatenatable while BitSets and
     Integers are not.

   * BitSet32 and BitSet256 have precisely the same properties as
     BitSets, except that they use constant fixed length bit vectors.

   * While all classes support basic unary and binary operations '~, &,
     |, ^, -', the semantics differ. BitSets perform bit operations that
     precisely mirror those for infinite sets. For example,
     complementing an empty BitSet returns one representing an infinite
     number of set bits. Operations on BitStrings and Integers operate

only on those bits actually present in the representation. For
BitStrings and Integers, the the '&' operation returns a BitString
with a length equal to the minimum length of the operands, and '|,
^' return one with length of the maximum.

* Only BitStrings support substring extraction and bit pattern
  matching.

BitSet
======

   BitSets are objects that contain logically infinite sets of
nonnegative integers. Representational details are discussed in the
Representation chapter. Because they are logically infinite, all
BitSets possess a trailing, infinitely replicated 0 or 1 bit, called
the "virtual bit", and indicated via 0* or 1*.

   BitSet32 and BitSet256 have they same properties, except they are of
fixed length, and thus have no virtual bit.

   BitSets may be constructed as follows:

'BitSet a;'
     declares an empty BitSet.

'BitSet a = atoBitSet("001000");'
     sets a to the BitSet 0010*, reading left-to-right. The "0*"
     indicates that the set ends with an infinite number of zero
     (clear) bits.

'BitSet a = atoBitSet("00101*");'
     sets a to the BitSet 00101*, where "1*" means that the set ends
     with an infinite number of one (set) bits.

'BitSet a = longtoBitSet((long)23);'
     sets a to the BitSet 111010*, the binary representation of decimal
     23.

'BitSet a = utoBitSet((unsigned)23);'
     sets a to the BitSet 111010*, the binary representation of decimal
     23.

   The following functions and operators are provided (Assume the
declaration of BitSets a = 0011010*, b = 101101*, throughout, as
examples).

'~a'
     returns the complement of a, or 1100101* in this case.

'a.complement()'
     sets a to ~a.

'a & b; a &= b;'
     returns a intersected with b, or 0011010*.

'a | b; a |= b;'
     returns a unioned with b, or 1011111*.

`'a - b; a -= b;'`
     returns the set difference of a and b, or 000010*.

`'a ^ b; a ^= b;'`
     returns the symmetric difference of a and b, or 1000101*.

`'a.empty()'`
     returns true if a is an empty set.

`'a == b;'`
     returns true if a and b contain the same set.

`'a <= b;'`
     returns true if a is a subset of b.

`'a < b;'`
     returns true if a is a proper subset of b;

`'a != b; a >= b; a > b;'`
     are the converses of the above.

`'a.set(7)'`
     sets the 7th (counting from 0) bit of a, setting a to 001111010*

`'a.clear(2)'`
     clears the 2nd bit bit of a, setting a to 00011110*

`'a.clear()'`
     clears all bits of a;

`'a.set()'`
     sets all bits of a;

`'a.invert(0)'`
     complements the 0th bit of a, setting a to 10011110*

`'a.set(0,1)'`
     sets the 0th through 1st bits of a, setting a to 110111110* The
     two-argument versions of clear and invert are similar.

`'a.test(3)'`
     returns true if the 3rd bit of a is set.

`'a.test(3, 5)'`
     returns true if any of bits 3 through 5 are set.

`'int i = a[3]; a[3] = 0;'`
     The subscript operator allows bits to be inspected and changed via
     standard subscript semantics, using a friend class BitSetBit. The
     use of the subscript operator a[i] rather than a.test(i) requires
     somewhat greater overhead.

`'a.first(1) or a.first()'`
     returns the index of the first set bit of a (2 in this case), or
     -1 if no bits are set.

`a.first(0)'
     returns the index of the first clear bit of a (0 in this case), or
     -1 if no bits are clear.

`a.next(2, 1) or a.next(2)'
     returns the index of the next bit after position 2 that is set (3
     in this case) or -1. `first' and `next' may be used as iterators,
     as in `for (int i = a.first(); i >= 0; i = a.next(i))...'.

`a.last(1)'
     returns the index of the rightmost set bit, or -1 if there or no
     set bits or all set bits.

`a.prev(3, 0)'
     returns the index of the previous clear bit before position 3.

`a.count(1)'
     returns the number of set bits in a, or -1 if there are an
     infinite number.

`a.virtual_bit()'
     returns the trailing (infinitely replicated) bit of a.

`a = atoBitSet("ababX", 'a', 'b', 'X');'
     converts the char* string into a bitset, with 'a' denoting false,
     'b' denoting true, and 'X' denoting infinite replication.

`a.printon(cout, '-', '.', 0)'
     prints `a' to `cout' represented with ``-'' for falses, ``.'' for
     trues, and no replication marker.

`cout << a'
     prints `a' to `cout' (representing lases by ``f'', trues by ``t'',
     and using ``*'' as the replication marker).

`diff(x, y, z)'
     A faster way to say z = x - y.

`and(x, y, z)'
     A faster way to say z = x & y.

`or(x, y, z)'
     A faster way to say z = x | y.

`xor(x, y, z)'
     A faster way to say z = x ^ y.

`complement(x, z)'
     A faster way to say z = ~x.

BitString
=========

   BitStrings are objects that contain arbitrary-length strings of
zeroes and ones. BitStrings possess some features that make them behave
like sets, and others that behave as strings. They are useful in
applications (such as signature-based algorithms) where both

capabilities are needed. Representational details are discussed in the Representation chapter. Most capabilities are exact analogs of those supported in the BitSet and String classes. A BitSubString is used with substring operations along the same lines as the String SubString class. A BitPattern class is used for masked bit pattern searching.

Only a default constructor is supported. The declaration 'BitString a;' initializes a to be an empty BitString. BitStrings may often be initialized via 'atoBitString' and 'longtoBitString'.

Set operations (' ~, complement, &, &=, |, |=, -, ^, ^=') behave just as the BitSet versions, except that there is no "virtual bit": complementing complements only those bits in the BitString, and all binary operations across unequal length BitStrings assume a virtual bit of zero. The '&' operation returns a BitString with a length equal to the minimum length of the operands, and '|, ^' return one with length of the maximum.

Set-based relational operations ('==, !=, <=, <, >=, >') follow the same rules. A string-like lexicographic comparison function, 'lcompare', tests the lexicographic relation between two BitStrings. For example, lcompare(1100, 0101) returns 1, since the first BitString starts with 1 and the second with 0.

Individual bit setting, testing, and iterator operations ('set, clear, invert, test, first, next, last, prev') are also like those for BitSets. BitStrings are automatically expanded when setting bits at positions greater than their current length.

The string-based capabilities are just as those for class String. BitStrings may be concatenated ('+, +='), searched ('index, contains, matches'), and extracted into BitSubStrings ('before, at, after') which may be assigned and otherwise manipulated. Other string-based utility functions ('reverse, common_prefix, common_suffix') are also provided. These have the same capabilities and descriptions as those for Strings.

String-oriented operations can also be performed with a mask via class BitPattern. BitPatterns consist of two BitStrings, a pattern and a mask. On searching and matching, bits in the pattern that correspond to 0 bits in the mask are ignored. (The mask may be shorter than the pattern, in which case trailing mask bits are assumed to be 0). The pattern and mask are both public variables, and may be individually subjected to other bit operations.

Converting to char* and printing ('(atoBitString, atoBitPattern, printon, ostream <<)') are also as in BitSets, except that no virtual bit is used, and an 'X' in a BitPattern means that the pattern bit is masked out.

The following features are unique to BitStrings.

Assume declarations of BitString a = atoBitString("01010110") and b = atoBitSTring("1101").

'a = b + c;'
    Sets a to the concatenation of b and c;

`a = b + 0; a = b + 1;'
    sets a to b, appended with a zero (one).

`a += b;'
    appends b to a;

`a += 0; a += 1;'
    appends a zero (one) to a.

`a << 2; a <<= 2'
    return a with 2 zeros prepended, setting a to 0001010110. (Note
    the necessary confusion of << and >> operators. For consistency
    with the integer versions, << shifts low bits to high, even though
    they are printed low bits first.)

`a >> 3; a >>= 3'
    return a with the first 3 bits deleted, setting a to 10110.

`a.left_trim(0)'
    deletes all 0 bits on the left of a, setting a to 1010110.

`a.right_trim(0)'
    deletes all trailing 0 bits of a, setting a to 0101011.

`cat(x, y, z)'
    A faster way to say z = x + y.

`diff(x, y, z)'
    A faster way to say z = x - y.

`and(x, y, z)'
    A faster way to say z = x & y.

`or(x, y, z)'
    A faster way to say z = x | y.

`xor(x, y, z)'
    A faster way to say z = x ^ y.

`lshift(x, y, z)'
    A faster way to say z = x << y.

`rshift(x, y, z)'
    A faster way to say z = x >> y.

`complement(x, z)'
    A faster way to say z = ~x.

1.25 libg++.guide/Random

Random Number Generators and related classes
****

    The two classes `RNG' and `Random' are used together to generate a

variety of random number distributions. A distinction must be made
between *random number generators*, implemented by class 'RNG', and
random number distributions. A random number generator produces a
series of randomly ordered bits. These bits can be used directly, or
cast to other representations, such as a floating point value. A
random number generator should produce a *uniform* distribution. A
random number distribution, on the other hand, uses the randomly
generated bits of a generator to produce numbers from a distribution
with specific properties. Each instance of 'Random' uses an instance
of class 'RNG' to provide the raw, uniform distribution used to produce
the specific distribution. Several instances of 'Random' classes can
share the same instance of 'RNG', or each instance can use its own copy.

RNG
===

   Random distributions are constructed from members of class 'RNG',
the actual random number generators. The 'RNG' class contains no data;
it only serves to define the interface to random number generators.
The 'RNG::asLong' member returns an unsigned long (typically 32 bits)
of random bits. Applications that require a number of random bits can
use this directly. More often, these random bits are transformed to a
uniform random number:

        //
        // Return random bits converted to either a float or a double
        //
        float asFloat();
        double asDouble();
    };

using either 'asFloat' or 'asDouble'. It is intended that 'asFloat'
and 'asDouble' return differing precisions; typically, 'asDouble' will
draw two random longwords and transform them into a legal 'double',
while 'asFloat' will draw a single longword and transform it into a
legal 'float'. These members are used by subclasses of the 'Random'
class to implement a variety of random number distributions.

ACG
===

   Class 'ACG' is a variant of a Linear Congruential Generator
(Algorithm M) described in Knuth, *Art of Computer Programming, Vol
III*. This result is permuted with a Fibonacci Additive Congruential
Generator to get good independence between samples. This is a very high
quality random number generator, although it requires a fair amount of
memory for each instance of the generator.

   The 'ACG::ACG' constructor takes two parameters: the seed and the
size. The seed is any number to be used as an initial seed. The
performance of the generator depends on having a distribution of bits
through the seed. If you choose a number in the range of 0 to 31, a
seed with more bits is chosen. Other values are deterministically
modified to give a better distribution of bits. This provides a good
random number generator while still allowing a sequence to be repeated
given the same initial seed.

   The 'size' parameter determines the size of two tables used in the generator. The first table is used in the Additive Generator; see the algorithm in Knuth for more information. In general, this table is 'size' longwords long. The default value, used in the algorithm in Knuth, gives a table of 220 bytes. The table size affects the period of the generators; smaller values give shorter periods and larger tables give longer periods. The smallest table size is 7 longwords, and the longest is 98 longwords. The 'size' parameter also determines the size of the table used for the Linear Congruential Generator. This value is chosen implicitly based on the size of the Additive Congruential Generator table. It is two powers of two larger than the power of two that is larger than 'size'. For example, if 'size' is 7, the ACG table is 7 longwords and the LCG table is 128 longwords. Thus, the default size (55) requires 55 + 256 longwords, or 1244 bytes. The largest table requires 2440 bytes and the smallest table requires 100 bytes. Applications that require a large number of generators or applications that aren't so fussy about the quality of the generator may elect to use the 'MLCG' generator.

MLCG
====

   The 'MLCG' class implements a *Multiplicative Linear Congruential Generator*. In particular, it is an implementation of the double MLCG described in *"Efficient and Portable Combined Random Number Generators"* by Pierre L'Ecuyer, appearing in *Communications of the ACM, Vol. 31. No. 6*. This generator has a fairly long period, and has been statistically analyzed to show that it gives good inter-sample independence.

   The 'MLCG::MLCG' constructor has two parameters, both of which are seeds for the generator. As in the 'MLCG' generator, both seeds are modified to give a "better" distribution of seed digits. Thus, you can safely use values such as '0' or '1' for the seeds. The 'MLCG' generator used much less state than the 'ACG' generator; only two longwords (8 bytes) are needed for each generator.

Random
======

   A random number generator may be declared by first declaring a 'RNG' and then a 'Random'. For example, 'ACG gen(10, 20); NegativeExpntl rnd (1.0, &gen);' declares an additive congruential generator with seed 10 and table size 20, that is used to generate exponentially distributed values with mean of 1.0.

   The virtual member 'Random::operator()' is the common way of extracting a random number from a particular distribution. The base class, 'Random' does not implement 'operator()'. This is performed by each of the subclasses. Thus, given the above declaration of 'rnd', new random values may be obtained via, for example, 'double next_exp_rand = rnd();' Currently, the following subclasses are provided.

Binomial
========

   The binomial distribution models successfully drawing items from a

pool. The first parameter to the constructor, 'n', is the number of
items in the pool, and the second parameter, 'u', is the probability of
each item being successfully drawn. The member 'asDouble' returns the
number of samples drawn from the pool. Although it is not checked, it
is assumed that 'n>0' and '0 <= u <= 1'. The remaining members allow
you to read and set the parameters.

Erlang
======

   The 'Erlang' class implements an Erlang distribution with mean
'mean' and variance 'variance'.

Geometric
=========

   The 'Geometric' class implements a discrete geometric distribution.
The first parameter to the constructor, 'mean', is the mean of the
distribution. Although it is not checked, it is assumed that '0 <=
mean <= 1'. 'Geometric()' returns the number of uniform random samples
that were drawn before the sample was larger than 'mean'. This
quantity is always greater than zero.

HyperGeometric
==============

   The 'HyperGeometric' class implements the hypergeometric
distribution. The first parameter to the constructor, 'mean', is the
mean and the second, 'variance', is the variance. The remaining
members allow you to inspect and change the mean and variance.

NegativeExpntl
==============

   The 'NegativeExpntl' class implements the negative exponential
distribution. The first parameter to the constructor is the mean. The
remaining members allow you to inspect and change the mean.

Normal
======

   The 'Normal'class implements the normal distribution. The first
parameter to the constructor, 'mean', is the mean and the second,
'variance', is the variance. The remaining members allow you to
inspect and change the mean and variance. The 'LogNormal' class is a
subclass of 'Normal'.

LogNormal
=========

   The 'LogNormal'class implements the logarithmic normal distribution.
The first parameter to the constructor, 'mean', is the mean and the
second, 'variance', is the variance. The remaining members allow you
to inspect and change the mean and variance. The 'LogNormal' class is
a subclass of 'Normal'.

Poisson

```
=======
```

   The 'Poisson' class implements the poisson distribution. The first
parameter to the constructor is the mean. The remaining members allow
you to inspect and change the mean.

```
DiscreteUniform
===============
```

   The 'DiscreteUniform' class implements a uniform random variable over
the closed interval ranging from '[low..high]'. The first parameter to
the constructor is 'low', and the second is 'high', although the order
of these may be reversed. The remaining members allow you to inspect
and change 'low' and 'high'.

```
Uniform
=======
```

   The 'Uniform' class implements a uniform random variable over the
open interval ranging from '[low..high)'. The first parameter to the
constructor is 'low', and the second is 'high', although the order of
these may be reversed. The remaining members allow you to inspect and
change 'low' and 'high'.

```
Weibull
=======
```

   The 'Weibull' class implements a weibull distribution with
parameters 'alpha' and 'beta'. The first parameter to the class
constructor is 'alpha', and the second parameter is 'beta'. The
remaining members allow you to inspect and change 'alpha' and 'beta'.

```
RandomInteger
=============
```

   The 'RandomInteger' class is *not* a subclass of Random, but a
stand-alone integer-oriented class that is dependent on the RNG
classes. RandomInteger returns random integers uniformly from the
closed interval '[low..high]'. The first parameter to the constructor
is 'low', and the second is 'high', although both are optional. The
last argument is always a generator. Additional members allow you to
inspect and change 'low' and 'high'. Random integers are generated
using 'asInt()' or 'asLong()'. Operator syntax ('()') is also
available as a shorthand for 'asLong()'. Because 'RandomInteger' is
often used in simulations for which uniform random integers are desired
over a variety of ranges, 'asLong()' and 'asInt' have 'high' as an
optional argument. Using this optional argument produces a single
value from the new range, but does not change the default range.

1.26   libg++.guide/Data

```
Data Collection
***************
```

   Libg++ currently provides two classes for *data collection* and
analysis of the collected data.

```
SampleStatistic
===============
```

   Class `SampleStatistic' provides a means of accumulating samples of
`double' values and providing common sample statistics.

   Assume declaration of `double x'.

`SampleStatistic a;'
     declares and initializes a.

`a.reset();'
     re-initializes a.

`a += x;'
     adds sample x.

`int n = a.samples();'
     returns the number of samples.

`x = a.mean;'
     returns the means of the samples.

`x = a.var()'
     returns the sample variance of the samples.

`x = a.stdDev()'
     returns the sample standard deviation of the samples.

`x = a.min()'
     returns the minimum encountered sample.

`x = a.max()'
     returns the maximum encountered sample.

`x = a.confidence(int p)'
     returns the p-percent (0 <= p < 100) confidence interval.

`x = a.confidence(double p)'
     returns the p-probability (0 <= p < 1) confidence interval.

```
SampleHistogram
===============
```

   Class `SampleHistogram' is a derived class of `SampleStatistic' that
supports collection and display of samples in bucketed intervals. It
supports the following in addition to `SampleStatisic' operations.

`SampleHistogram h(double lo, double hi, double width);'
     declares and initializes h to have buckets of size width from lo
     to hi. If the optional argument width is not specified, 10
     buckets are created. The first bucket and also holds samples less
     than lo, and the last one holds samples greater than hi.

```
'int n = h.similarSamples(x)'
     returns the number of samples in the same bucket as x.

'int n = h.inBucket(int i)'
     returns the number of samples in  bucket i.

'int b = h.buckets()'
     returns the number of buckets.

'h.printBuckets(ostream s)'
     prints bucket counts on ostream s.

'double bound = h.bucketThreshold(int i)'
     returns the upper bound of bucket i.
```

1.27 libg++.guide/Curses

```
Curses-based classes
********************
```

The 'CursesWindow' class is a repackaging of standard curses library
features into a class. It relies on 'curses.h'.

The supplied 'curses.h' is a fairly conservative declaration of
curses library features, and does not include features like "screen" or
X-window support. It is, for the most part, an adaptation, rather than
an improvement of C-based 'curses.h' files. The only substantive
changes are the declarations of many functions as inline functions
rather than macros, which was done solely to allow overloading.

The 'CursesWindow' class encapsulates curses window functions within
a class. Only those functions that control windows are included:
Terminal control functions and macros like 'cbreak' are not part of the
class. All 'CursesWindows' member functions have names identical to
the corresponding curses library functions, except that the "w" prefix
is generally dropped. Descriptions of these functions may be found in
your local curses library documentation.

   A 'CursesWindow' may be declared via

```
'CursesWindow w(WINDOW* win)'
     attaches w to the existing WINDOW* win. This is constructor is
     normally used only in the following special case.

'CursesWindow w(stdscr)'
     attaches w to the default curses library standard screen window.

'CursesWindow w(int lines, int cols, int begin_y, int begin_x)'
     attaches to an allocated curses window with the indicated size and
     screen position.

'CursesWindow sub(CursesWindow& w,int l,int c,int by,int bx,char ar='a')'
     attaches to a subwindow of w created via the curses 'subwin'
     command.  If ar is sent as 'r', the origin (by, bx) is relative to
```

the parent window, else it is absolute.

The class maintains a static counter that is used in order to
automatically call the curses library 'initscr' and 'endscr' functions
at the proper times. These need not, and should not be called
"manually".

'CursesWindow's maintain a tree of their subwindows. Upon
destruction of a 'CursesWindow', all of their subwindows are also
invalidated if they had not previously been destroyed.

It is possible to traverse trees of subwindows via the following
member functions

'CursesWindow* w.parent()'
    returns a pointer to the parent of the subwindow, or 0 if there is
    none.

'CursesWindow* w.child()'
    returns the first child subwindow of the window, or 0 if there is
    none.

'CursesWindow* w.sibling()'
    returns the next sibling of the subwindow, or 0 if there is none.

For example, to call some function 'visit' for all subwindows of a
window, you could write

```
void traverse(CursesWindow& w)
{
  visit(w);
  if (w.child() != 0) traverse(*w.child);
  if (w.sibling() != 0) traverse(*w.sibling);
}
```

1.28 libg++.guide/List

List classes
****

The files 'g++-include/List.hP' and 'g++-include/List.ccP' provide
pseudo-generic Lisp-type List classes. These lists are homogeneous
lists, more similar to lists in statically typed functional languages
like ML than Lisp, but support operations very similar to those found
in Lisp. Any particular kind of list class may be generated via the
'genclass' shell command. However, the implementation assumes that the
base class supports an equality operator '=='. All equality tests use
the '==' operator, and are thus equivalent to the use of 'equal', not
'eq' in Lisp.

All list nodes are created dynamically, and managed via reference
counts. 'List' variables are actually pointers to these list nodes.
Lists may also be traversed via Pixes, as described in the section

describing Pixes. See Pix

    Supported operations are mirrored closely after those in Lisp.
Generally, operations with functional forms are constructive,
functional operations, while member forms (often with the same name)
are sometimes procedural, possibly destructive operations.

    As with Lisp, destructive operations are supported. Programmers are
allowed to change head and tail fields in any fashion, creating
circular structures and the like. However, again as with Lisp, some
operations implicitly assume that they are operating on pure lists, and
may enter infinite loops when presented with improper lists. Also, the
reference-counting storage management facility may fail to reclaim
unused circularly-linked nodes.

    Several Lisp-like higher order functions are supported (e.g., 'map').
Typedef declarations for the required functional forms are provided int
the '.h' file.

    For purposes of illustration, assume the specification of class
'intList'. Common Lisp versions of supported operations are shown in
brackets for comparison purposes.

Constructors and assignment
===========================

'intList a;   [(setq a nil)]'
    Declares a to be a nil intList.

'intList b(2);   [(setq b (cons 2 nil))]'
    Declares b to be an intList with a head value of 2, and a nil tail.

'intList c(3, b);   [(setq c (cons 3 b))]'
    Declares c to be an intList with a head value of 3, and b as its
    tail.

'b = a;   [(setq b a)]'
    Sets b to be the same list as a.

   Assume the declarations of intLists a, b, and c in the following.
See Pix.

List status
===========

'a.null(); OR !a;   [(null a)]'
    returns true if a is null.

'a.valid();   [(listp a)]'
    returns true if a is non-null. Inside a conditional test, the
    'void*' coercion may also be used as in 'if (a) ...'.

'intList();   [nil]'
    intList() may be used to null terminate a list, as in 'intList
    f(int x) {if (x == 0) return intList(); ... } '.

'a.length();   [(length a)]'

        returns the length of a.

`a.list_length();   [(list-length a)]'
        returns the length of a, or -1 if a is circular.

heads and tails
===============

`a.get(); OR a.head() [(car a)]'
        returns a reference to the head field.

`a[2];   [(elt a 2)]'
        returns a reference to the second (counting from zero) head field.

`a.tail();   [(cdr a)]'
        returns the intList that is the tail of a.

`a.last();   [(last a)]'
        returns the intList that is the last node of a.

`a.nth(2);   [(nth a 2)]'
        returns the intList that is the nth node of a.

`a.set_tail(b);   [(rplacd a b)]'
        sets a's tail to b.

`a.push(2);   [(push 2 a)]'
        equivalent to a = intList(2, a);

`int x = a.pop()   [(setq x (car a)) (pop a)]'
        returns the head of a, also setting a to its tail.

Constructive operations
=======================

`b = copy(a);   [(setq b (copy-seq a))]'
        sets b to a copy of a.

`b = reverse(a);   [(setq b (reverse a))]'
        Sets b to a reversed copy of a.

`c = concat(a, b);   [(setq c (concat a b))]'
        Sets c to a concatenated copy of a and b.

`c = append(a, b);   [(setq c (append a b))]'
        Sets c to a concatenated copy of a and b. All nodes of a are
        copied, with the last node pointing to b.

`b = map(f, a);   [(setq b (mapcar f a))]'
        Sets b to a new list created by applying function f to each node
        of a.

`c = combine(f, a, b);'
        Sets c to a new list created by applying function f to successive
        pairs of a and b. The resulting list has length the shorter of a
        and b.

```
`b = remove(x, a);        [ (setq b (remove x a)) ]'
     Sets b to a copy of a, omitting all occurrences of x.

`b = remove(f, a);        [ (setq b (remove-if f a)) ]'
     Sets b to a copy of a, omitting values causing function f to
     return true.

`b = select(f, a);        [ (setq b (remove-if-not f a)) ]'
     Sets b to a copy of a, omitting values causing function f to
     return false.

`c = merge(a, b, f);    [ (setq c (merge a b f)) ]'
     Sets c to a list containing the ordered elements (using the
     comparison function f) of the sorted lists a and b.

Destructive operations
======================

`a.append(b);            [ (rplacd (last a) b) ]'
     appends b to the end of a. No new nodes are constructed.

`a.prepend(b);           [ (setq a (append b a)) ]'
     prepends b to the beginning of a.

`a.del(x);               [ (delete x a) ]'
     deletes all nodes with value x from a.

`a.del(f);               [ (delete-if f a) ]'
     deletes all nodes causing function f to return true.

`a.select(f);            [ (delete-if-not f a) ]'
     deletes all nodes causing function f to return false.

`a.reverse();            [ (nreverse a) ]'
     reverses a in-place.

`a.sort(f);              [ (sort a f) ]'
     sorts a in-place using ordering (comparison) function f.

`a.apply(f);             [ (mapc f a) ]'
     Applies void function f (int x) to each element of a.

`a.subst(int old, int repl); [ (nsubst repl old a) ]'
     substitutes repl for each occurrence of old in a. Note the
     different argument order than the Lisp version.

Other operations
================

`a.find(int x);          [ (find x a) ]'
     returns the intList at the first occurrence of x.

`a.find(b);              [ (find b a) ]'
     returns the intList at the first occurrence of sublist b.

`a.contains(int x);      [ (member x a) ]'
     returns true if a contains x.
```

```
'a.contains(b);          [ (member b a) ]'
     returns true if a contains sublist b.

'a.position(int x);      [ (position x a) ]'
     returns the zero-based index of x in a, or -1 if x does not occur.

'int x = a.reduce(f, int base); [ (reduce f a :initial-value base) ]'
     Accumulates the result of applying int function f(int, int) to
     successive elements of a, starting with base.
```

1.29 libg++.guide/LinkList

```
Linked Lists
************
```

   SLLists provide pseudo-generic singly linked lists. DLLists provide
doubly linked lists. The lists are designed for the simple maintenance
of elements in a linked structure, and do not provide the more extensive
operations (or node-sharing) of class 'List'. They behave similarly to
the 'slist' and similar classes described by Stroustrup.

   All list nodes are created dynamically. Assignment is performed via
copying.

   Class 'DLList' supports all 'SLList' operations, plus additional
operations described below.

   For purposes of illustration, assume the specification of class
'intSLList'. In addition to the operations listed here, SLLists support
traversal via Pixes. See Pix

```
'intSLList a;'
     Declares a to be an empty list.

'intSLList b = a;'
     Sets b to an element-by-element copy of a.

'a.empty()'
     returns true if a contains no elements

'a.length();'
     returns the number of elements in a.

'a.prepend(x);'
     places x at the front of the list.

'a.append(x);'
     places x at the end of the list.

'a.join(b)'
     places all nodes from b to the end of a, simultaneously destroying
     b.
```

`'x = a.front()'`
     returns a reference to the item stored at the head of the list, or
     triggers an error if the list is empty.

`'a.rear()'`
     returns a reference to the rear of the list, or triggers an error
     if the list is empty.

`'x = a.remove_front()'`
     deletes and returns the item stored at the head of the list.

`'a.del_front()'`
     deletes the first element, without returning it.

`'a.clear()'`
     deletes all items from the list.

`'a.ins_after(Pix i, item);'`
     inserts item after position i. If i is null, insertion is at the
     front.

`'a.del_after(Pix i);'`
     deletes the element following i. If i is 0, the first item is
     deleted.

```
Doubly linked lists
===================
```

   Class `'DLList'` supports the following additional operations, as well
as backward traversal via Pixes.

`'x = a.remove_rear();'`
     deletes and returns the item stored at the rear of the list.

`'a.del_rear();'`
     deletes the last element, without returning it.

`'a.ins_before(Pix i, x)'`
     inserts x before the i.

`'a.del(Pix& iint dir = 1)'`
     deletes the item at the current position, then advances forward if
     dir is positive, else backward.

1.30   libg++.guide/Vector

```
Vector classes
**************
```

   The files `'g++-include/Vec.ccP'` and `'g++-include/AVec.ccP'` provide
pseudo-generic standard array-based vector operations. The
corresponding header files are `'g++-include/Vec.hP'` and
`'g++-include/AVec.hP'`. Class `'Vec'` provides operations suitable for
any base class that includes an equality operator. Subclass `'AVec'`

provides additional arithmetic operations suitable for base classes
that include the full complement of arithmetic operators.

    'Vecs' are constructed and assigned by copying. Thus, they should
normally be passed by reference in applications programs.

    Several mapping functions are provided that allow programmers to
specify operations on vectors as a whole.

    For illustrative purposes assume that classes 'intVec' and 'intAVec'
have been generated via 'genclass'.

Constructors and assignment
===========================

'intVec a;'
     declares a to be an empty vector. Its size may be changed via
     resize.

'intVec a(10);'
     declares a to be an uninitialized vector of ten elements (numbered
     0-9).

'intVec b(6, 0);'
     declares b to be a vector of six elements, all initialized to
     zero. Any value can be used as the initial fill argument.

'a = b;'
     Copies b to a. a is resized to be the same as b.

'a = b.at(2, 4)'
     constructs a from the 4 elements of b starting at b[2].

    Assume declarations of 'intVec a, b, c' and 'int i, x' in the
following.

Status and access
=================

'a.capacity();'
     returns the number of elements that can be held in a.

'a.resize(20);'
     sets a's length to 20. All elements are unchanged, except that if
     the new size is smaller than the original, than trailing elements
     are deleted, and if greater, trailing elements are uninitialized.

'a[i];'
     returns a reference to the i'th element of a, or produces an error
     if i is out of range.

'a.elem(i)'
     returns a reference to the i'th element of a. Unlike the '[]'
     operator, i is not checked to ensure that it is within range.

'a == b;'
     returns true if a and b contain the same elements in the same

    order.

`a != b;'
    is the converse of a == b.

Constructive operations
=======================

`c = concat(a, b);'
    sets c to the new vector constructed from all of the elements of a
    followed by all of b.

`c = map(f, a);'
    sets c to the new vector constructed by applying int function
    f(int) to each element of a.

`c = merge(a, b, f);'
    sets c to the new vector constructed by merging the elements of
    ordered vectors a and b using ordering (comparison) function f.

`c = combine(f, a, b);'
    sets c to the new vector constructed by applying int function
    f(int, int) to successive pairs of a and b. The result has length
    the shorter of a and b.

`c = reverse(a)'
    sets c to a, with elements in reverse order.

Destructive operations
======================

`a.reverse();'
    reverses a in-place.

`a.sort(f)'
    sorts a in-place using comparison function f. The sorting method
    is a variation of the quicksort functions supplied with GNU emacs.

`a.fill(0, 4, 2)'
    fills the 2 elements starting at a[4] with zero.

Other operations
================

`a.apply(f)'
    applies function f to each element in a.

`x = a.reduce(f, base)'
    accumulates the results of applying function f to successive
    elements of a starting with base.

`a.index(int targ);'
    returns the index of the leftmost occurrence of the target, or -1,
    if it does not occur.

`a.error(char* msg)'
    invokes the error handler. The default version prints the error

message, then aborts.

AVec operations.
================

   AVecs provide additional arithmetic operations. All vector-by-vector
operators generate an error if the vectors are not the same length. The
following operations are provided, for 'AVecs a, b' and base element
(scalar) 's'.

'a = b;'
     Copies b to a. a and b must be the same size.

'a = s;'
     fills all elements of a with the value s. a is not resized.

'a + s; a - s; a * s; a / s'
     adds, subtracts, multiplies, or divides each element of a with the
     scalar.

'a += s; a -= s; a *= s; a /= s;'
     adds, subtracts, multiplies, or divides the scalar into a.

'a + b; a - b; product(a, b), quotient(a, b)'
     adds, subtracts, multiplies, or divides corresponding elements of
     a and b.

'a += b; a -= b; a.product(b); a.quotient(b);'
     adds, subtracts, multiplies, or divides corresponding elements of
     b into a.

's = a * b;'
     returns the inner (dot) product of a and b.

'x = a.sum();'
     returns the sum of elements of a.

'x = a.sumsq();'
     returns the sum of squared elements of a.

'x = a.min();'
     returns the minimum element of a.

'x = a.max();'
     returns the maximum element of a.

'i = a.min_index();'
     returns the index of the minimum element of a.

'i = a.max_index();'
     returns the index of the maximum element of a.

     Note that it is possible to apply vector versions other arithmetic
     operators via the mapping functions. For example, to set vector b
     to the cosines of doubleVec a, use 'b = map(cos, a);'. This is
     often more efficient than performing the operations in an
     element-by-element fashion.

1.31 libg++.guide/Plex

Plex classes
****

   A "Plex" is a kind of array with the following properties:

   * Plexes may have arbitrary upper and lower index bounds. For example
     a Plex may be declared to run from indices -10 .. 10.

   * Plexes may be dynamically expanded at both the lower and upper
     bounds of the array in steps of one element.

   * Only elements that have been specifically initialized or added may
     be accessed.

   * Elements may be accessed via indices. Indices are always checked
     for validity at run time. Plexes may be traversed via simple
     variations of standard array indexing loops.

   * Plex elements may be accessed and traversed via Pixes.

   * Plex-to-Plex assignment and related operations on entire Plexes
     are supported.

   * Plex classes contain methods to help programmers check the validity
     of indexing and pointer operations.

   * Plexes form "natural" base classes for many restricted-access data
     structures relying on logically contiguous indices, such as
     array-based stacks and queues.

   * Plexes are implemented as pseudo-generic classes, and must be
     generated via the 'genclass' utility.

   Four subclasses of Plexes are supported: A 'FPlex' is a Plex that
may only grow or shrink within declared bounds; an 'XPlex' may
dynamically grow or shrink without bounds; an 'RPlex' is the same as an
'XPlex' but better supports indexing with poor locality of reference; a
'MPlex' may grow or shrink, and additionally allows the logical
deletion and restoration of elements. Because these classes are virtual
subclasses of the "abstract" class 'Plex', it is possible to write user
code such as 'void f(Plex& a) ...' that operates on any kind of Plex.
However, as with nearly any virtual class, specifying the particular
Plex class being used results in more efficient code.

   Plexes are implemented as a linked list of 'IChunks'. Each chunk
contains a part of the array. Chunk sizes may be specified within Plex
constructors. Default versions also exist, that use a '#define'd'
default. Plexes grow by filling unused space in existing chunks, if
possible, else, except for FPlexes, by adding another chunk. Whenever
Plexes grow by a new chunk, the default element constructors (i.e.,
those which take no arguments) for all chunk elements are called at

once. When Plexes shrink, destructors for the elements are not called
until an entire chunk is freed. For this reason, Plexes (like C++
arrays) should only be used for elements with default constructors and
destructors that have no side effects.

Plexes may be indexed and used like arrays, although traversal
syntax is slightly different. Even though Plexes maintain elements in
lists of chunks, they are implemented so that iteration and other
constructs that maintain locality of reference require very little
overhead over that for simple array traversal Pix-based traversal is
also supported. For example, for a plex, p, of ints, the following
traversal methods could be used.

```
for (int i = p.low(); i < p.fence(); p.next(i)) use(p[i]);
for (int i = p.high(); i > p.ecnef(); p.prev(i)) use(p[i]);
for (Pix t = p.first(); t != 0; p.next(t)) use(p(i));
for (Pix t = p.last(); t != 0; p.prev(t)) use(p(i));
```

Except for MPlexes, simply using '++i' and '--i' works just as well
as 'p.next(i)' and 'p.prev(i)' when traversing by index. Index-based
traversal is generally a bit faster than Pix-based traversal.

'XPlexes' and 'MPlexes' are less than optimal for applications in
which widely scattered elements are indexed, as might occur when using
Plexes as hash tables or "manually" allocated linked lists. In such
applications, 'RPlexes' are often preferable. 'RPlexes' use a secondary
chunk index table that requires slightly greater, but entirely uniform
overhead per index operation.

Even though they may grow in either direction, Plexes are normally
constructed so that their "natural" growth direction is upwards, in
that default chunk construction leaves free space, if present, at the
end of the plex. However, if the chunksize arguments to constructors
are negative, they leave space at the beginning.

All versions of Plexes support the following basic capabilities.
(letting 'Plex' stand for the type name constructed via the genclass
utility (e.g., 'intPlex', 'doublePlex')). Assume declarations of 'Plex
p, q', 'int i, j', base element 'x', and Pix 'pix'.

'Plex p;'
    Declares p to be an initially zero-sized Plex with low index of
    zero, and the default chunk size. For FPlexes, chunk sizes
    represent maximum sizes.

'Plex p(int size);'
    Declares p to be an initially zero-sized Plex with low index of
    zero, and the indicated chunk size. If size is negative, then the
    Plex is created with free space at the beginning of the Plex,
    allowing more efficient add_low() operations. Otherwise, it leaves
    space at the end.

'Plex p(int low, int size);'
    Declares p to be an initially zero-sized Plex with low index of
    low, and the indicated chunk size.

'Plex p(int low, int high, Base initval, int size = 0);'

     Declares p to be a Plex with indices from low to high, initially
     filled with initval, and the indicated chunk size if specified,
     else the default or (high - low + 1), whichever is greater.

`Plex q(p);'
     Declares q to be a copy of p.

`p = q;'
     Copies Plex q into p, deleting its previous contents.

`p.length()'
     Returns the number of elements in the Plex.

`p.empty()'
     Returns true if Plex p contains no elements.

`p.full()'
     Returns true if Plex p cannot be expanded. This always returns
     false for XPlexes and MPlexes.

`p[i]'
     Returns a reference to the i'th element of p. An exception (error)
     occurs if i is not a valid index.

`p.valid(i)'
     Returns true if i is a valid index into Plex p.

`p.low(); p.high();'
     Return the minimum (maximum) valid index of the Plex, or the high
     (low) fence if the plex is empty.

`p.ecnef(); p.fence();'
     Return the index one position past the minimum (maximum) valid
     index.

`p.next(i); i = p.prev(i);'
     Set i to the next (previous) index. This index may not be within
     bounds.

`p(pix)'
     returns a reference to the item at Pix pix.

`pix = p.first(); pix = p.last();'
     Return the minimum (maximum) valid Pix of the Plex, or 0 if the
     plex is empty.

`p.next(pix); p.prev(pix);'
     set pix to the next (previous) Pix, or 0 if there is none.

`p.owns(pix)'
     Returns true if the Plex contains the element associated with pix.

`p.Pix_to_index(pix)'
     If pix is a valid Pix to an element of the Plex, returns its
     corresponding index, else raises an exception.

`ptr = p.index_to_Pix(i)'

if i is a valid index, returns a the corresponding Pix.

`p.low_element(); p.high_element();`
     Return a reference to the element at the minimum (maximum) valid
     index. An exception occurs if the Plex is empty.

`p.can_add_low();  p.can_add_high();`
     Returns true if the plex can be extended one element downward
     (upward). These always return true for XPlex and MPlex.

`j = p.add_low(x); j = p.add_high(x);`
     Extend the Plex by one element downward (upward). The new minimum
     (maximum) index is returned.

`j = p.del_low(); j = p.del_high()`
     Shrink the Plex by one element on the low (high) end. The new
     minimum (maximum) element is returned. An exception occurs if the
     Plex is empty.

`p.append(q);`
     Append all of Plex q to the high side of p.

`p.prepend(q);`
     Prepend all of q to the low side of p.

`p.clear()`
     Delete all elements, resetting p to a zero-sized Plex.

`p.reset_low(i);`
     Resets p to be indexed starting at low() = i. For example. if p
     were initially declared via `Plex p(0, 10, 0)`, and then
     re-indexed via `p.reset_low(5)`, it could then be indexed from
     indices 5 .. 14.

`p.fill(x)`
     sets all p[i] to x.

`p.fill(x, lo, hi)`
     sets all of p[i] from lo to hi, inclusive, to x.

`p.reverse()`
     reverses p in-place.

`p.chunk_size()`
     returns the chunk size used for the plex.

`p.error(const char * msg)`
     calls the resettable error handler.

   MPlexes are plexes with bitmaps that allow items to be logically
deleted and restored. They behave like other plexes, but also support
the following additional and modified capabilities:

`p.del_index(i); p.del_Pix(pix)`
     logically deletes p[i] (p(pix)). After deletion, attempts to
     access p[i] generate a error. Indexing via low(), high(), prev(),
     and next() skip the element. Deleting an element never changes the

        logical bounds of the plex.

`p.undel_index(i); p.undel_Pix(pix)'
        logically undeletes p[i] (p(pix)).

`p.del_low(); p.del_high()'
        Delete the lowest (highest) undeleted element, resetting the
        logical bounds of the plex to the next lowest (highest) undeleted
        index. Thus, MPlex del_low() and del_high() may shrink the bounds
        of the plex by more than one index.

`p.adjust_bounds()'
        Resets the low and high bounds of the Plex to the indexes of the
        lowest and highest actual undeleted elements.

`int i = p.add(x)'
        Adds x in an unused index, if possible, else performs add_high.

`p.count()'
        returns the number of valid (undeleted) elements.

`p.available()'
        returns the number of available (deleted) indices.

`int i = p.unused_index()'
        returns the index of some deleted element, if one exists, else
        triggers an error. An unused element may be reused via undel.

`pix = p.unused_Pix()'
        returns the pix of some deleted element, if one exists, else 0.
        An unused element may be reused via undel.

1.32   libg++.guide/Stack

Stacks
******

   Stacks are declared as an "abstract" class. They are currently
implemented in any of three ways.

`VStack'
        implement fixed sized stacks via arrays.

`XPStack'
        implement dynamically-sized stacks via XPlexes.

`SLStack'
        implement dynamically-size stacks via linked lists.

   All possess the same capabilities. They differ only in constructors.
VStack constructors require a fixed maximum capacity argument. XPStack
constructors optionally take a chunk size argument. SLStack
constructors take no argument.

    Assume the declaration of a base element 'x'.

'Stack s; or Stack s(int capacity)'
    declares a Stack.

's.empty()'
    returns true if stack s is empty.

's.full()'
    returns true if stack s is full. XPStacks and SLStacks never
    become full.

's.length()'
    returns the current number of elements in the stack.

's.push(x)'
    pushes x on stack s.

'x = s.pop()'
    pops and returns the top of stack

's.top()'
    returns a reference to the top of stack.

's.del_top()'
    pops, but does not return the top of stack. When large items are
    held on the stack it is often a good idea to use 'top()' to
    inspect and use the top of stack, followed by a 'del_top()'

's.clear()'
    removes all elements from the stack.

1.33 libg++.guide/Queue

Queues
******

   Queues are declared as an "abstract" class. They are currently
implemented in any of three ways.

'VQueue'
    implement fixed sized Queues via arrays.

'XPQueue'
    implement dynamically-sized Queues via XPlexes.

'SLQueue'
    implement dynamically-size Queues via linked lists.

   All possess the same capabilities; they differ only in constructors.
'VQueue' constructors require a fixed maximum capacity argument.
'XPQueue' constructors optionally take a chunk size argument.
'SLQueue' constructors take no argument.

Assume the declaration of a base element 'x'.

'Queue q; or Queue q(int capacity);'
    declares a queue.

'q.empty()'
    returns true if queue q is empty.

'q.full()'
    returns true if queue q is full. XPQueues and SLQueues are never
    full.

'q.length()'
    returns the current number of elements in the queue.

'q.enq(x)'
    enqueues x on queue q.

'x = q.deq()'
    dequeues and returns the front of queue

'q.front()'
    returns a reference to the front of queue.

'q.del_front()'
    dequeues, but does not return the front of queue

'q.clear()'
    removes all elements from the queue.

1.34 libg++.guide/Deque

```
Double ended Queues
*******************
```

   Deques are declared as an "abstract" class. They are currently
implemented in two ways.

'XPDeque'
    implement dynamically-sized Deques via XPlexes.

'DLDeque'
    implement dynamically-size Deques via linked lists.

   All possess the same capabilities. They differ only in constructors.
XPDeque constructors optionally take a chunk size argument. DLDeque
constructors take no argument.

   Double-ended queues support both stack-like and queue-like
capabilities:

   Assume the declaration of a base element 'x'.

'Deque d; or Deque d(int initial_capacity)'

```
     declares a deque.
```

`d.empty()'
     returns true if deque d is empty.

`d.full()'
     returns true if deque d is full. Always returns false in current
     implementations.

`d.length()'
     returns the current number of elements in the deque.

`d.enq(x)'
     inserts x at the rear of deque d.

`d.push(x)'
     inserts x at the front of deque d.

`x = d.deq()'
     dequeues and returns the front of deque

`d.front()'
     returns a reference to the front of deque.

`d.rear()'
     returns a reference to the rear of the deque.

`d.del_front()'
     deletes, but does not return the front of deque

`d.del_rear()'
     deletes, but does not return the rear of the deque.

`d.clear()'
     removes all elements from the deque.

1.35   libg++.guide/PQ

```
Priority Queue class prototypes.
********************************
```

   Priority queues maintain collections of objects arranged for fast
access to the least element.

   Several prototype implementations of priority queues are supported.

`XPPQs'
     implement 2-ary heaps via XPlexes.

`SplayPQs'
     implement PQs via Sleator and Tarjan's (JACM 1985) splay trees.
     The algorithms use a version of "simple top-down splaying"
     (described on page 669 of the article). The simple-splay
     mechanism for priority queue functions is loosely based on the one

used by D. Jones in the C splay tree functions available from
volume 14 of the uunet.uu.net archives.

'PHPQs'
implement pairing heaps (as described by Fredman and Sedgewick in
'Algorithmica', Vol 1, p111-129). Storage for heap elements is
managed via an internal freelist technique. The constructor allows
an initial capacity estimate for freelist space. The storage is
automatically expanded if necessary to hold new items. The
deletion technique is a fast "lazy deletion" strategy that marks
items as deleted, without reclaiming space until the items come to
the top of the heap.

All PQ classes support the following operations, for some PQ class
'Heap', instance 'h', 'Pix ind', and base class variable 'x'.

'h.empty()'
returns true if there are no elements in the PQ.

'h.length()'
returns the number of elements in h.

'ind = h.enq(x)'
Places x in the PQ, and returns its index.

'x = h.deq()'
Dequeues the minimum element of the PQ into x, or generates an
error if the PQ is empty.

'h.front()'
returns a reference to the minimum element.

'h.del_front()'
deletes the minimum element.

'h.clear();'
deletes all elements from h;

'h.contains(x)'
returns true if x is in h.

'h(ind)'
returns a reference to the item indexed by ind.

'ind = h.first()'
returns the Pix of first item in the PQ or 0 if empty. This need
not be the Pix of the least element.

'h.next(ind)'
advances ind to the Pix of next element, or 0 if there are no more.

'ind = h.seek(x)'
Sets ind to the Pix of x, or 0 if x is not in h.

'h.del(ind)'
deletes the item with Pix ind.

1.36 libg++.guide/Set

```
Set class prototypes
********************
```

   Set classes maintain unbounded collections of items containing no
duplicate elements.

   These are currently implemented in several ways, differing in
representation strategy, algorithmic efficiency, and appropriateness for
various tasks. (Listed next to each are average (followed by worst-case,
if different) time complexities for [a] adding, [f] finding (via seek,
contains), [d] deleting, elements, and [c] comparing (via ==, <=) and
[m] merging (via |=, -=, &=) sets).

'XPSets'
     implement unordered sets via XPlexes. ([a O(n)], [f O(n)], [d
     O(n)], [c O(n^2)] [m O(n^2)]).

'OXPSets'
     implement ordered sets via XPlexes. ([a O(n)], [f O(log n)], [d
     O(n)], [c O(n)] [m O(n)]).

'SLSets'
     implement unordered sets via linked lists ([a O(n)], [f O(n)], [d
     O(n)], [c O(n^2)] [m O(n^2)]).

'OSLSets'
     implement ordered sets via linked lists ([a O(n)], [f O(n)], [d
     O(n)], [c O(n)] [m O(n)]).

'AVLSets'
     implement ordered sets via threaded AVL trees ([a O(log n)], [f
     O(log n)], [d O(log n)], [c O(n)] [m O(n)]).

'BSTSets'
     implement ordered sets via binary search trees. The trees may be
     manually rebalanced via the O(n) 'balance()' member function. ([a
     O(log n)/O(n)], [f O(log n)/O(n)], [d O(log n)/O(n)], [c O(n)] [m
     O(n)]).

'SplaySets'
     implement ordered sets via Sleator and Tarjan's (JACM 1985) splay
     trees. The algorithms use a version of "simple top-down splaying"
     (described on page 669 of the article). (Amortized: [a O(log n)],
     [f O(log n)], [d O(log n)], [c O(n)] [m O(n log n)]).

'VHSets'
     implement unordered sets via hash tables. The tables are
     automatically resized when their capacity is exhausted. ([a
     O(1)/O(n)], [f O(1)/O(n)], [d O(1)/O(n)], [c O(n)/O(n^2)] [m
     O(n)/O(n^2)]).

'VOHSets'
     implement unordered sets via ordered hash tables The tables are
     automatically resized when their capacity is exhausted. ([a
     O(1)/O(n)], [f O(1)/O(n)], [d O(1)/O(n)], [c O(n)/O(n^2)] [m
     O(n)/O(n^2)]).

'CHSets'
     implement unordered sets via chained hash tables. ([a O(1)/O(n)],
     [f O(1)/O(n)], [d O(1)/O(n)], [c O(n)/O(n^2)] [m O(n)/O(n^2)]).

   The different implementations differ in whether their constructors
require an argument specifying their initial capacity. Initial
capacities are required for plex and hash table based Sets. If none is
given 'DEFAULT_INITIAL_CAPACITY' (from '<T>defs.h') is used.

   Sets support the following operations, for some class 'Set',
instances 'a' and 'b', 'Pix ind', and base element 'x'. Since all
implementations are virtual derived classes of the '<T>Set' class, it
is possible to mix and match operations across different
implementations, although, as usual, operations are generally faster
when the particular classes are specified in functions operating on
Sets.

   Pix-based operations are more fully described in the section on
Pixes. See Pix

'Set a; or Set a(int initial_size);'
     Declares a to be an empty Set. The second version is allowed in
     set classes that require initial capacity or sizing specifications.

'a.empty()'
     returns true if a is empty.

'a.length()'
     returns the number of elements in a.

'Pix ind = a.add(x)'
     inserts x into a, returning its index.

'a.del(x)'
     deletes x from a.

'a.clear()'
     deletes all elements from a;

'a.contains(x)'
     returns true if x is in a.

'a(ind)'
     returns a reference to the item indexed by ind.

'ind = a.first()'
     returns the Pix of first item in the set or 0 if the Set is empty.
     For ordered Sets, this is the Pix of the least element.

'a.next(ind)'
     advances ind to the Pix of next element, or 0 if there are no more.

`ind = a.seek(x)'
     Sets ind to the Pix of x, or 0 if x is not in a.

`a == b'
     returns true if a and b contain all the same elements.

`a != b'
     returns true if a and b do not contain all the same elements.

`a <= b'
     returns true if a is a subset of b.

`a |= b'
     Adds all elements of b to a.

`a -= b'
     Deletes all elements of b from a.

`a &= b'
     Deletes all elements of a not occurring in b.

1.37 libg++.guide/Bag

```
Bag class prototypes
********************
```

   Bag classes maintain unbounded collections of items potentially
containing duplicate elements.

   These are currently implemented in several ways, differing in
representation strategy, algorithmic efficiency, and appropriateness for
various tasks. (Listed next to each are average (followed by worst-case,
if different) time complexities for [a] adding, [f] finding (via seek,
contains), [d] deleting elements).

`XPBags'
     implement unordered Bags via XPlexes. ([a O(1)], [f O(n)], [d
     O(n)]).

`OXPBags'
     implement ordered Bags via XPlexes. ([a O(n)], [f O(log n)], [d
     O(n)]).

`SLBags'
     implement unordered Bags via linked lists ([a O(1)], [f O(n)], [d
     O(n)]).

`OSLBags'
     implement ordered Bags via linked lists ([a O(n)], [f O(n)], [d
     O(n)]).

`SplayBags'
     implement ordered Bags via Sleator and Tarjan's (JACM 1985) splay

trees. The algorithms use a version of "simple top-down splaying"
(described on page 669 of the article). (Amortized: [a O(log n)],
[f O(log n)], [d O(log n)]).

'VHBags'
    implement unordered Bags via hash tables. The tables are
    automatically resized when their capacity is exhausted. ([a
    O(1)/O(n)], [f O(1)/O(n)], [d O(1)/O(n)]).

'CHBags'
    implement unordered Bags via chained hash tables. ([a O(1)/O(n)],
    [f O(1)/O(n)], [d O(1)/O(n)]).

   The implementations differ in whether their constructors require an
argument to specify their initial capacity. Initial capacities are
required for plex and hash table based Bags. If none is given
'DEFAULT_INITIAL_CAPACITY' (from '<T>defs.h') is used.

   Bags support the following operations, for some class 'Bag',
instances 'a' and 'b', 'Pix ind', and base element 'x'. Since all
implementations are virtual derived classes of the '<T>Bag' class, it
is possible to mix and match operations across different
implementations, although, as usual, operations are generally faster
when the particular classes are specified in functions operating on
Bags.

   Pix-based operations are more fully described in the section on
Pixes. See Pix

'Bag a; or Bag a(int initial_size)'
    Declares a to be an empty Bag. The second version is allowed in
    Bag classes that require initial capacity or sizing specifications.

'a.empty()'
    returns true if a is empty.

'a.length()'
    returns the number of elements in a.

'ind = a.add(x)'
    inserts x into a, returning its index.

'a.del(x)'
    deletes one occurrence of x from a.

'a.remove(x)'
    deletes all occurrences of x from a.

'a.clear()'
    deletes all elements from a;

'a.contains(x)'
    returns true if x is in a.

'a.nof(x)'
    returns the number of occurrences of x in a.

`a(ind)'
     returns a reference to the item indexed by ind.

`int = a.first()'
     returns the Pix of first item in the Bag or 0 if the Bag is empty.
     For ordered Bags, this is the Pix of the least element.

`a.next(ind)'
     advances ind to the Pix of next element, or 0 if there are no more.

`ind = a.seek(x, Pix from = 0)'
     Sets ind to the Pix of the next occurrence x, or 0 if there are
     none. If from is 0, the first occurrence is returned, else the
     following from.

1.38 libg++.guide/Map

Map Class Prototypes
****

   Maps support associative array operations (insertion, deletion, and
membership of records based on an associated key). They require the
specification of two types, the key type and the contents type.

   These are currently implemented in several ways, differing in
representation strategy, algorithmic efficiency, and appropriateness for
various tasks. (Listed next to each are average (followed by worst-case,
if different) time complexities for [a] accessing (via op [],
contains), [d] deleting elements).

`AVLMaps'
     implement ordered Maps via threaded AVL trees ([a O(log n)], [d
     O(log n)]).

`RAVLMaps'
     Similar, but also maintain ranking information, used via
     `ranktoPix(int r)', that returns the `Pix' of the item at rank r,
     and `rank(key)' that returns the rank of the corresponding item.
     ([a O(log n)], [d O(log n)]).

`SplayMaps'
     implement ordered Maps via Sleator and Tarjan's (JACM 1985) splay
     trees. The algorithms use a version of "simple top-down splaying"
     (described on page 669 of the article). (Amortized: [a O(log n)],
     [d O(log n)]).

`VHMaps'
     implement unordered Maps via hash tables. The tables are
     automatically resized when their capacity is exhausted. ([a
     O(1)/O(n)], [d O(1)/O(n)]).

`CHMaps'
     implement unordered Maps via chained hash tables. ([a O(1)/O(n)],
     [d O(1)/O(n)]).

   The different implementations differ in whether their constructors
require an argument specifying their initial capacity. Initial
capacities are required for hash table based Maps. If none is given
'DEFAULT_INITIAL_CAPACITY' (from '<T>defs.h') is used.

   All Map classes share the following operations (for some Map class,
'Map' instance 'd', 'Pix ind' and key variable 'k', and contents
variable 'x').

   Pix-based operations are more fully described in the section on
Pixes. See Pix

'Map d(x); Map d(x, int initial_capacity)'
     Declare d to be an empty Map. The required argument, x, specifies
     the default contents, i.e., the contents of an otherwise
     uninitialized location. The second version, specifying initial
     capacity is allowed for Maps with an initial capacity argument.

'd.empty()'
     returns true if d contains no items.

'd.length()'
     returns the number of items in d.

'd[k]'
     returns a reference to the contents of item with key k. If no such
     item exists, it is installed with the default contents. Thus d[k]
     = x installs x, and x = d[k] retrieves it.

'd.contains(k)'
     returns true if an item with key field k exists in d.

'd.del(k)'
     deletes the item with key k.

'd.clear()'
     deletes all items from the table.

'x = d.dflt()'
     returns the default contents.

'k = d.key(ind)'
     returns a reference to the key at Pix ind.

'x = d.contents(ind)'
     returns a reference to the contents at Pix ind.

'ind = d.first()'
     returns the Pix of the first element in d, or 0 if d is empty.

'd.next(ind)'
     advances ind to the next element, or 0 if there are no more.

'ind = d.seek(k)'
     returns the Pix of element with key k, or 0 if k is not in d.

1.39 libg++.guide/GetOpt

C++ version of the GNU getopt function
****

   The GetOpt class provides an efficient and structured mechanism for
processing command-line options from an application program. The sample
program fragment below illustrates a typical use of the GetOpt class
for some hypothetical application program:

```
    #include <stdio.h>
    #include <GetOpt.h>
    //...
    int debug_flag, compile_flag, size_in_bytes;

    int
    main (int argc, char **argv)
    {
      // Invokes ctor 'GetOpt (int argc, char **argv,
      //                       char *optstring);'
      GetOpt getopt (argc, argv, "dcs:");
      int option_char;

      // Invokes member function 'int operator ()(void);'
      while ((option_char = getopt ()) != EOF)
        switch (option_char)
          {
             case 'd': debug_flag = 1; break;
             case 'c': compile_flag = 1; break;
             case 's': size_in_bytes = atoi (getopt.optarg); break;
             case '?': fprintf (stderr,
                                "usage: %s [dcs<size>]\n", argv[0]);
          }
    }
```

   Unlike the C library version, the libg++ GetOpt class uses its
constructor to initialize class data members containing the argument
count, argument vector, and the option string. This simplifies the
interface for each subsequent call to member function 'int operator
()(void)'.

   The C version, on the other hand, uses hidden static variables to
retain the option string and argument list values between calls to
'getopt'. This complicates the 'getopt' interface since the argument
count, argument vector, and option string must be passed as parameters
for each invocation. For the C version, the loop in the previous
example becomes:

```
    while ((option_char = getopt (argc, argv, "dcs:")) != EOF)
      // ...
```

   which requires extra overhead to pass the parameters for every call.

Along with the GetOpt constructor and 'int operator ()(void)', the
other relevant elements of class GetOpt are:

'char *optarg'
    Used for communication from 'operator ()(void)' to the caller.
    When 'operator ()(void)' finds an option that takes an argument,
    the argument value is stored here.

'int optind'
    Index in 'argv' of the next element to be scanned. This is used
    for communication to and from the caller and for communication
    between successive calls to 'operator ()(void)'.

    When 'operator ()(void)' returns EOF, this is the index of the
    first of the non-option elements that the caller should itself
    scan.

    Otherwise, 'optind' communicates from one call to the next how much
    of 'argv' has been scanned so far.

    The libg++ version of GetOpt acts like standard UNIX 'getopt' for
the calling routine, but it behaves differently for the user, since it
allows the user to intersperse the options with the other arguments.

    As GetOpt works, it permutes the elements of 'argv' so that, when it
is done, all the options precede everything else. Thus all application
programs are extended to handle flexible argument order.

    Setting the environment variable _POSIX_OPTION_ORDER disables
permutation. Then the behavior is completely standard.

1.40 libg++.guide/Projects

Projects and other things left to do
****

Coming Attractions
==================

    Some things that will probably be available in libg++ in the near
future:

    * Revamped C-compatibility header files that will be compatible with
      the forthcoming (ANSI-based) GNU libc.a

    * A revision of the File-based classes that will use the GNU stdio
      library, and also be 100% compatible (even at the streambuf level)
      with the AT&T 2.0 stream classes.

    * Additional container class prototypes.

    * generic Matrix class prototypes.

    * A task package probably based on Dirk Grunwald's threads package.

Wish List
=========

   Some things that people have mentioned that they would like to see
in libg++, but for which there have not been any offers:

   * A method to automatically convert or incorporate libg++ classes so
     they can be used directly in Gorlen's OOPS environment.

   * A class browser.

   * A better general exception-handling strategy.

   * Better documentation.

How to contribute
=================

   Programmers who have written C++ classes that they believe to be of
general interest are encourage to write to dl at rocky.oswego.edu.
Contributing code is not difficult. Here are some general guidelines:

   * FSF must maintain the right to accept or reject potential
     contributions. Generally, the only reasons for rejecting
     contributions are cases where they duplicate existing or
     nearly-released code, contain unremovable specific machine
     dependencies, or are somehow incompatible with the rest of the
     library.

   * Acceptance of contributions means that the code is accepted for
     adaptation into libg++. FSF must reserve the right to make
     various editorial changes in code. Very often, this merely entails
     formatting, maintenance of various conventions, etc. Contributors
     are always given authorship credit and shown the final version for
     approval.

   * Contributors must assign their copyright to FSF via a form sent out
     upon acceptance. Assigning copyright to FSF ensures that the code
     may be freely distributed.

   * Assistance in providing documentation, test files, and debugging
     support is strongly encouraged.

   Extensions, comments, and suggested modifications of existing libg++
features are also very welcome.