User's Guide to the GNU C++ Library

last updated April 29, 1992 for version 2.0 by Doug Lea (dl@g.oswego.edu)

Note: The GNU C++ library is still in test release. You will be performing a valuable service if you report any bugs you encounter.

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.

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 ªmake allº to compile the library, test, and install. Current details about contents of the tests and utilities are in the README file.

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.

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, wrappers, 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. The most notable alternative sources are Interviews and OOPS. (A g++-compatible version of OOPS is currently available on prep.ai.mit.edu. InterViews has been available on the X-windows X11 tapes and also from interviews.stanford.edu.)

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 programmingÐThe "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.

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.

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.

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***)**	Returns 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***)**	Returns true if *a* is less than or equal to *b*. Default: (*a* <= *b*)
<T>**CMP(***a*, *b***)**	Returns 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***)**	Returns an unsigned integer representing the hash of *a*. Default: **extern unsigned int hash(<T&>** *a***)**. (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 ª(CHOP!)º, a **Base::Base(Base&)** is called inside **Set::add(Base&)**Ð*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**.

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.

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.

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 **Pix**es. 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, **Pix**es 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 **Pix**es 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 **Pix**es always contain the following methods, assuming a container **a** of element types of **Base**.

Pix i = a.first()	Sets **i** to index the first element of **a** or 0 if **a** is empty.
a.next(i)	Advances **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() Sets **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)
Pix j = a.enq(x) /* for priority queues */ Adds **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.

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 builtintypes, 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.

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.

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.