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.