Using

Prerequisites

The library contains only header files, and does not require any other libraries except the standard C++ library . All classes are defined in namespace `__gnu_pbds`. The library internally uses macros beginning with `PB_DS`, but `#undef`s anything it `#define`s (except for header guards). Compiling the library in an environment where macros beginning in `PB_DS` are defined, may yield unpredictable results in compilation, execution, or both.

Further dependencies are necessary to create the visual output for the performance tests. To create these graphs, an additional package is needed: **pychart**.

Organization

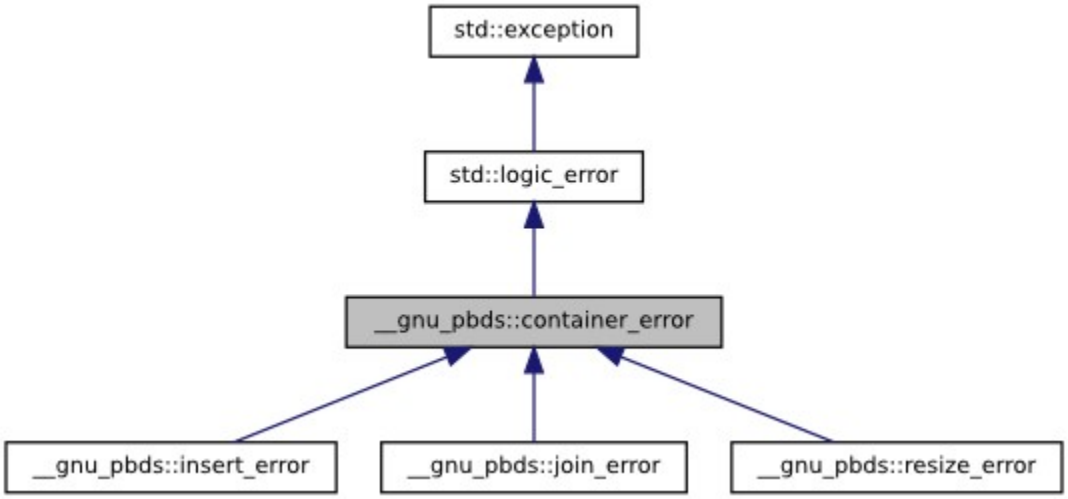
The various data structures are organized as follows.

- Branch-Based
  - `basic_branch` is an abstract base class for branched-based associative-containers
  - `tree` is a concrete base class for tree-based associative-containers
  - `trie` is a concrete base class trie-based associative-containers
- Hash-Based
  - `basic_hash_table` is an abstract base class for hash-based associative-containers
  - `cc_hash_table` is a concrete collision-chaining hash-based associative-containers
  - `gp_hash_table` is a concrete (general) probing hash-based associative-containers
- List-Based
  - `list_update` list-based update-policy associative container
- Heap-Based
  - `priority_queue` A priority queue.

The hierarchy is composed naturally so that commonality is captured by base classes. Thus `operator[]` is defined at the base of any hierarchy, since all derived containers support it. Conversely `split` is defined in `basic_branch`, since only tree-like containers support it.

In addition, there are the following diagnostics classes, used to report errors specific to this library's data structures.

Figure 22.7. Exception Hierarchy



Tutorial

Basic Use

For the most part, the policy-based containers containers in namespace `__gnu_pbds` have the same interface as the equivalent containers in the standard C++ library, except for the names used for the container classes themselves. For example, this shows basic operations on a collision-chaining hash-based container:

```
#include <ext/pb_ds/assoc_container.h>

int main()
{
__gnu_pbds::cc_hash_table<int, char> c;
c[2] = 'b';
assert(c.find(1) == c.end());
};
```

The container is called `__gnu_pbds::cc_hash_table` instead of `std::unordered_map`, since “unordered map” does not necessarily mean a hash-based map as implied by the C++ library (C++11 or TR1). For example, list-based associative containers, which are very useful for the construction of "multimaps," are also unordered.

This snippet shows a red-black tree based container:

```
#include <ext/pb_ds/assoc_container.h>

int main()
{
__gnu_pbds::tree<int, char> c;
c[2] = 'b';
assert(c.find(2) != c.end());
};
```

The container is called `tree` instead of `map` since the underlying data structures are being named with specificity.

The member function naming convention is to strive to be the same as the equivalent member functions in other C++ standard library containers. The familiar methods are unchanged: `begin`, `end`, `size`, `empty`, and `clear`.

This isn't to say that things are exactly as one would expect, given the container requirments and interfaces in the C++ standard.

The names of containers' policies and policy accessors are different then the usual. For example, if hash_type is some type of hash-based container, then

```
hash_type::hash_fn
```

gives the type of its hash functor, and if `obj` is some hash-based container object, then

```
obj.get_hash_fn()
```

will return a reference to its hash-functor object.

Similarly, if tree_type is some type of tree-based container, then

```
tree_type::cmp_fn
```

gives the type of its comparison functor, and if `obj` is some tree-based container object, then

```
obj.get_cmp_fn()
```

will return a reference to its comparison-functor object.

It would be nice to give names consistent with those in the existing C++ standard (inclusive of TR1). Unfortunately, these standard containers don't consistently name types and methods. For example, `std::tr1::unordered_map` uses hasher for the hash functor, but `std::map` uses key_compare for the comparison functor. Also, we could not find an accessor for `std::tr1::unordered_map`'s hash functor, but `std::map` uses `compare` for accessing the comparison functor.

Instead, `__gnu_pbds` attempts to be internally consistent, and uses standard-derived terminology if possible.

Another source of difference is in scope: `__gnu_pbds` contains more types of associative containers than the standard C++ library, and more opportunities to configure these new containers, since different types of associative containers are useful in different settings.

Namespace `__gnu_pbds` contains different classes for hash-based containers, tree-based containers, trie-based containers, and list-based containers.

Since associative containers share parts of their interface, they are organized as a class hierarchy.

Each type or method is defined in the most-common ancestor in which it makes sense.

For example, all associative containers support iteration expressed in the following form:

```
const_iterator
begin() const;
```

```
iterator
begin();

const_iterator
end() const;

iterator
end();
```

But not all containers contain or use hash functors. Yet, both collision-chaining and (general) probing hash-based associative containers have a hash functor, so `basic_hash_table` contains the interface:

```
const hash_fn&
get_hash_fn() const;

hash_fn&
get_hash_fn();
```

so all hash-based associative containers inherit the same hash-functor accessor methods.

Configuring via Template Parameters

In general, each of this library's containers is parametrized by more policies than those of the standard library. For example, the standard hash-based container is parametrized as follows:

```
template<typename Key, typename Mapped, typename Hash,
typename Pred, typename Allocator, bool Cache_Hashe_Code>
class unordered_map;
```

and so can be configured by key type, mapped type, a functor that translates keys to unsigned integral types, an equivalence predicate, an allocator, and an indicator whether to store hash values with each entry. this library's collision-chaining hash-based container is parametrized as

```
template<typename Key, typename Mapped, typename Hash_Fn,
typename Eq_Fn, typename Comb_Hash_Fn,
typename Resize_Policy, bool Store_Hash
typename Allocator>
class cc_hash_table;
```

and so can be configured by the first four types of `std::tr1::unordered_map`, then a policy for translating the key-hash result into a position within the table, then a policy by which the table resizes, an indicator whether to store hash values with each entry, and an allocator (which is typically the last template parameter in standard containers).

Nearly all policy parameters have default values, so this need not be considered for casual use. It is important to note, however, that hash-based containers' policies can dramatically alter their performance in different settings, and that tree-based containers' policies can make them useful for other purposes than just look-up.

As opposed to associative containers, priority queues have relatively few configuration options. The priority queue is parametrized as follows:

```
template<typename Value_Type, typename Cmp_Fn,typename Tag,
typename Allocator>
class priority_queue;
```

The `Value_Type`, `Cmp_Fn`, and `Allocator` parameters are the container's value type, comparison-functor type, and allocator type, respectively; these are very similar to the standard's priority queue. The `Tag` parameter is different: there are a number of pre-defined tag types corresponding to binary heaps, binomial heaps, etc., and `Tag` should be instantiated by one of them.

Note that as opposed to the `std::priority_queue`, `__gnu_pbds::priority_queue` is not a sequence-adapter; it is a regular container.

Querying Container Attributes

A containers underlying data structure affect their performance; Unfortunately, they can also affect their interface. When manipulating generically associative containers, it is often useful to be able to statically determine what they can support and what the cannot.

Happily, the standard provides a good solution to a similar problem - that of the different behavior of iterators. If `It` is an iterator, then

```
typename std::iterator_traits<It>::iterator_category
```

is one of a small number of pre-defined tag classes, and

```
typename std::iterator_traits<It>::value_type
```

is the value type to which the iterator "points".

Similarly, in this library, if C is a container, then `container_traits` is a trait class that stores information about the kind of container that is implemented.

```
typename container_traits<C>::container_category
```

is one of a small number of predefined tag structures that uniquely identifies the type of underlying data structure.

In most cases, however, the exact underlying data structure is not really important, but what is important is one of its other attributes: whether it guarantees storing elements by key order, for example. For this one can use

```
typename container_traits<C>::order_preserving
```

Also,

```
typename container_traits<C>::invalidation_guarantee
```

is the container's invalidation guarantee. Invalidation guarantees are especially important regarding priority queues, since in this library's design, iterators are practically the only way to manipulate them.

Point and Range Iteration

This library differentiates between two types of methods and iterators: point-type, and range-type. For example, `find` and `insert` are point-type methods, since they each deal with a specific element; their returned iterators are point-type iterators. `begin` and `end` are range-type methods, since they are not used to find a specific element, but rather to go over all elements in a container object; their returned iterators are range-type iterators.

Most containers store elements in an order that is determined by their interface. Correspondingly, it is fine that their point-type iterators are synonymous with their range-type iterators. For example, in the following snippet

```
std::for_each(c.find(1), c.find(5), foo);
```

two point-type iterators (returned by `find`) are used for a range-type purpose - going over all elements whose key is between 1 and 5.

Conversely, the above snippet makes no sense for self-organizing containers - ones that order (and reorder) their elements by implementation. It would be nice to have a uniform iterator system that would allow the above snippet to compile only if it made sense.

This could trivially be done by specializing `std::for_each` for the case of iterators returned by `std::tr1::unordered_map`, but this would only solve the problem for one algorithm and one container. Fundamentally, the problem is that one can loop using a self-organizing container's point-type iterators.

This library's containers define two families of iterators: point_const_iterator and point_iterator are the iterator types returned by point-type methods; const_iterator and iterator are the iterator types returned by range-type methods.

```
class <- some container ->
{
public:
...

typedef <- something -> const_iterator;

typedef <- something -> iterator;

typedef <- something -> point_const_iterator;

typedef <- something -> point_iterator;

...

public:
...

const_iterator begin () const;

iterator begin();

point_const_iterator find(...) const;

point_iterator find(...);
};
```

For containers whose interface defines sequence order , it is very simple: point-type and range-type iterators are exactly the same, which means that the above snippet will compile if it is used for an order-preserving

associative container.

For self-organizing containers, however, (hash-based containers as a special example), the preceding snippet will not compile, because their point-type iterators do not support `operator++`.

In any case, both for order-preserving and self-organizing containers, the following snippet will compile:

```
typename Cntnr::point_iterator it = c.find(2);
```

because a range-type iterator can always be converted to a point-type iterator.

Distingushing between iterator types also raises the point that a container's iterators might have different invalidation rules concerning their de-referencing abilities and movement abilities. This now corresponds exactly to the question of whether point-type and range-type iterators are valid. As explained above, `container_traits` allows querying a container for its data structure attributes. The iterator-invalidation guarantees are certainly a property of the underlying data structure, and so

```
container_traits<C>::invalidation_guarantee
```

gives one of three pre-determined types that answer this query.

Examples

Additional code examples are provided in the source distribution, as part of the regression and performance testsuite.

Intermediate Use

- Basic use of maps: `basic_map.cc`
- Basic use of sets: `basic_set.cc`
- Conditionally erasing values from an associative container object: `erase_if.cc`
- Basic use of multimaps: `basic_multimap.cc`
- Basic use of multisets: `basic_multiset.cc`
- Basic use of priority queues: `basic_priority_queue.cc`
- Splitting and joining priority queues: `priority_queue_split_join.cc`
- Conditionally erasing values from a priority queue: `priority_queue_erase_if.cc`

Querying with `container_traits`

- Using `container_traits` to query about underlying data structure behavior: `assoc_container_traits.cc`
- A non-compiling example showing wrong use of finding keys in hash-based containers: `hash_find_neg.cc`

- Using `container_traits` to query about underlying data structure behavior: `priority_queue_container_traits.cc`

By Container Method

Hash-Based

size Related

- Setting the initial size of a hash-based container object: `hash_initial_size.cc`
- A non-compiling example showing how not to resize a hash-based container object: `hash_resize_neg.cc`
- Resizing the size of a hash-based container object: `hash_resize.cc`
- Showing an illegal resize of a hash-based container object: `hash_illegal_resize.cc`
- Changing the load factors of a hash-based container object: `hash_load_set_change.cc`

Hashing Function Related

- Using a modulo range-hashing function for the case of an unknown skewed key distribution: `hash_mod.cc`
- Writing a range-hashing functor for the case of a known skewed key distribution: `shift_mask.cc`
- Storing the hash value along with each key: `store_hash.cc`
- Writing a ranged-hash functor: `ranged_hash.cc`

Branch-Based

split or join Related

- Joining two tree-based container objects: `tree_join.cc`
- Splitting a PATRICIA trie container object: `trie_split.cc`
- Order statistics while joining two tree-based container objects: `tree_order_statistics_join.cc`

Node Invariants

- Using trees for order statistics: `tree_order_statistics.cc`
- Augmenting trees to support operations on line intervals: `tree_intervals.cc`

trie

- Using a PATRICIA trie for DNA strings: `trie_dna.cc`

- Using a PATRICIA trie for finding all entries whose key matches a given prefix: `trie_prefix_search.cc`

Priority Queues

- Cross referencing an associative container and a priority queue: `priority_queue_xref.cc`
- Cross referencing a vector and a priority queue using a very simple version of Dijkstra's shortest path algorithm: `priority_queue_dijkstra.cc`

Prev | Up | Next
Chapter 22. Policy-Based Data Structures | Home | Design