A Quick Overview of the B-Tree Framework

Any Container Type Is Based on a B-Tree Structure

- the b-tree framework provides associative and linear container types with an interface equivalent to existing STL implementations
- also, this framework allows for polymorphic accesses by providing abstract container types, as oppose to any other STL implementation
- furthermore, since any other container type (e.g. array, link list, ...) can develop poor performance when it comes to large amounts of data, b-trees have been chosen as the basis for this framework

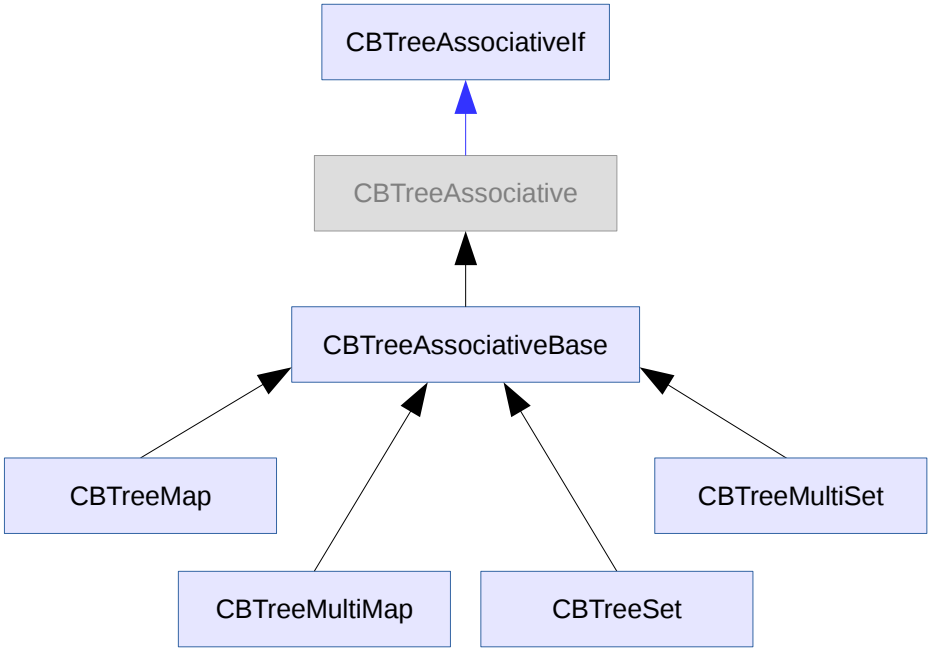
- arrays can be slow to insert data at the beginning
- link lists need to walk through a potentially high number of nodes to perform random accesses
- binary trees can end up very unbalanced if the same or similar keys are in use
- red-black trees can result in a high number of hops on access

Any Container Can Be Abstracted

- any container type provided by this framework is based on abstract types
- this can be exploited to use polymorphic accesses on different container types
- also this allows applications to create their own class hierarchies
- to some extend it is possible to create relational data base models

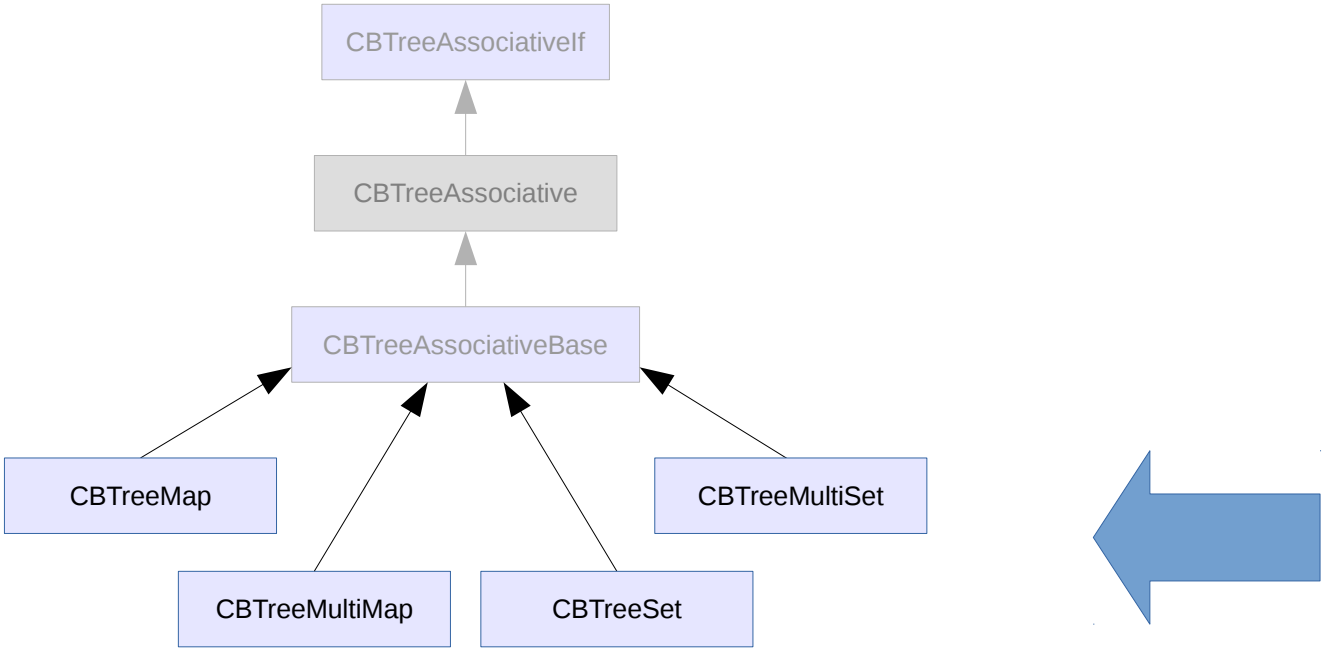
How Is Polymorphism Within This Framework Possible?

- the framework itself has a class hierarchy
- part of which can be seen below



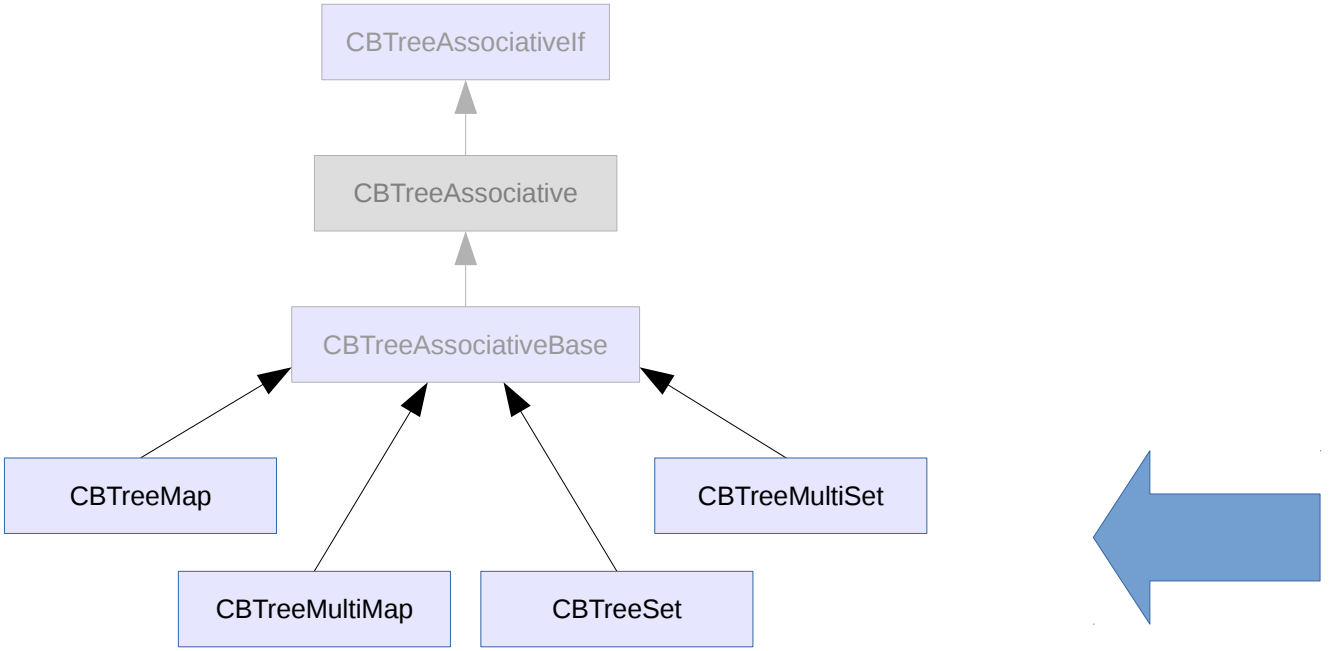
How Is Polymorphism Within This Framework Possible?

- the specific data classes provide us with a starting point for your application class hierarchy



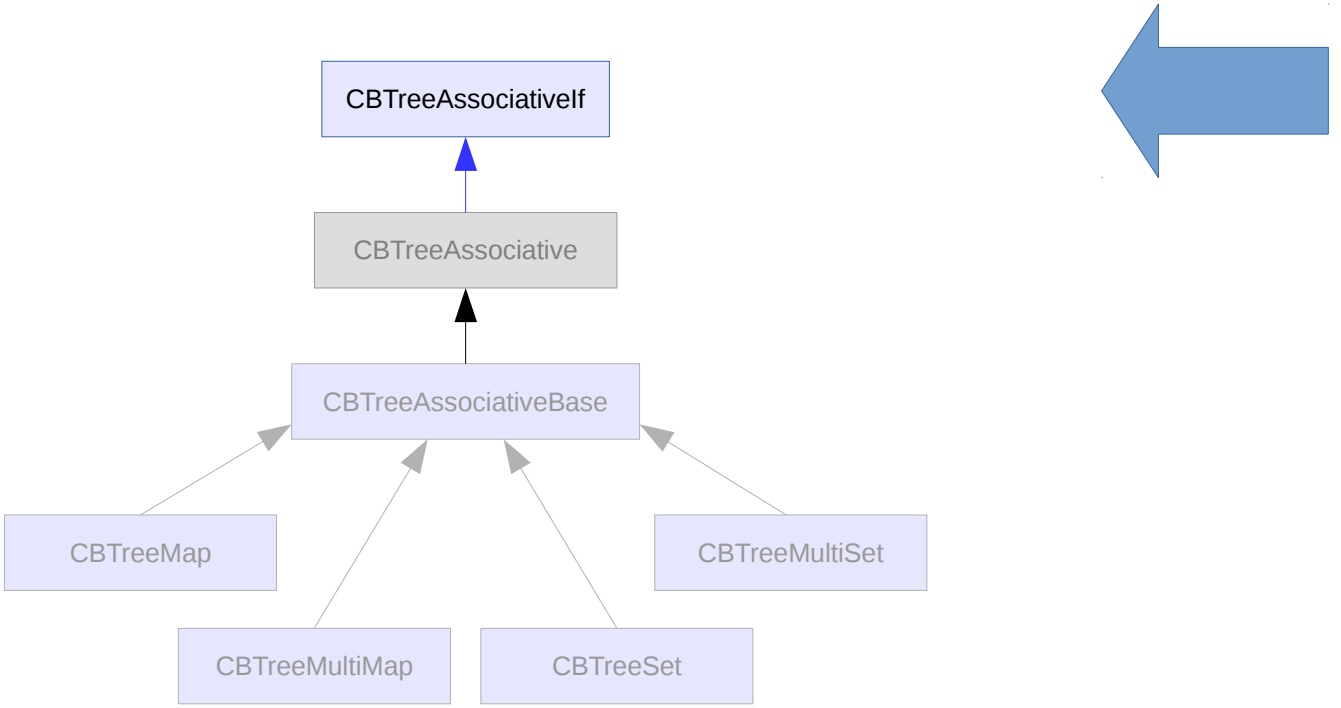
How Is Polymorphism Within This Framework Possible?

- also the specific data classes may be used directly by the application as a container type



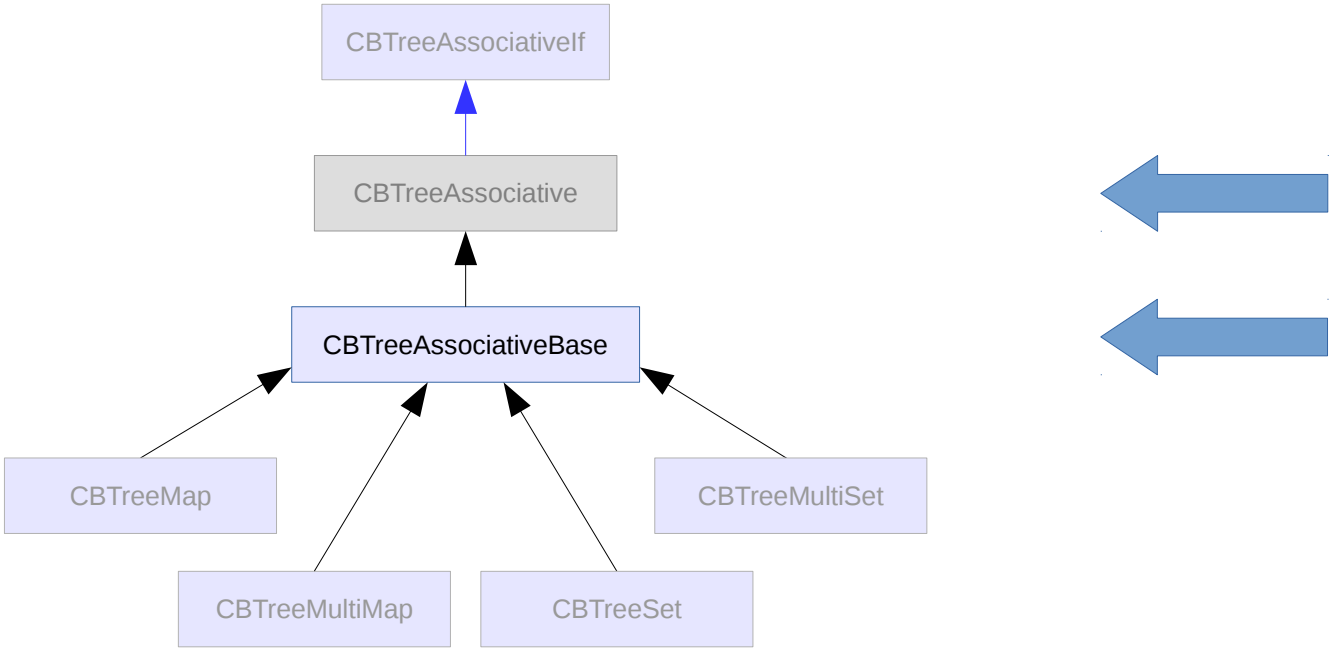
How Is Polymorphism Within This Framework Possible?

- the abstract interface class contains virtual methods acting as an interface for the application



How Is Polymorphism Within This Framework Possible?

- the abstract data classes provide the functionality for the abstract interface class



What Can I Do With It?

- it is possible to support polymorphic calls
- example 1* shows how it inserts data into a map and a multi map container via an abstract class

```
    for (i = 0; i < 16; i++)
    {
         uint32_t  j;

         // this will create data items with equal keys
         // the multi map container will accept any data
         // while the map container will only allow for data to stored that has a unique
key value
         for (j = 0; j < (sizeof (apPolymorphicMaps) / sizeof (*apPolymorphicMaps)); j++)
         {
              value_type          sValue (i / 2, i);

              apPolymorphicMaps[j]->insert (sValue);
         }
    }
```

* - ./src/btreetest/testbench/examples/example_1_polymorphic_calls.cpp

What Can I Do With It?

- one could say example 1 can also be solved using a template function
- that would work unless you face a situation like the one below

```
class A
{
public:

     template<class _t_container>
     virtual   void test_fn (_t_container *pContainer);
};
```

What Can I Do With It?

- virtual methods involving template parameters are not possible, since templates are resolved during the compilation and virtual calls at run time

```
class A
{
public:

     template<class _t_container>
     virtual   void test_fn (_t_container *pContainer);
};
```

What Can I Do With It?

- the b-tree framework solves that by pushing the abstraction problem to a different layer

```
class A
{
public:

     typedef CBTreeIOpropertiesRAM<>                  ram_data_layer_property_type;

     typedef CBTreeMap<uint32_t, uint32_t, ram_data_layer_property_type>
                                                      btree_map_type;

     typedef btree_map_type::CBTreeAssociativeIf_t
                                                      btree_polymorphic_type;

     virtual   void test_fn (btree_uniform_type *pContainer);
};
```

What Can I Do With It?

- it is also possible to create very simple temporary data bases
- example 2* shows code of a specific application class, that orders data using different criteria

```
template<class _t_keytype, int _t_nSelectKey, class _t_datalayerproperties>
class car_holders : public CBTreeMultiSet<_t_keytype, _t_datalayerproperties>
{
     // ...


};

     typedef car_holders<car_holders_t, 0, ram_data_layer_property_type>
                                                car_holders_sort_by_name;
     typedef car_holders<car_holders_t, 1, ram_data_layer_property_type>
                                                car_holders_sort_by_license_plate;
     typedef car_holders<car_holders_t, 2, ram_data_layer_property_type>
                                                car_holders_sort_by_phone_number;
```

* - ./src/btreetest/testbench/examples/example_2_simple_database.cpp

What Can I Do With It?

- the different sort criteria types are then instantiated and put into an abstract type list
- this will allow us to talk to any sort containers the same way

```
car_holders_sort_by_name           *psCarHoldersSortByName =
               new car_holders_sort_by_name (sRAMproperties, 16);
car_holders_sort_by_license_plate  *psCarHoldersSortByLicensePlate =
               new car_holders_sort_by_license_plate (sRAMproperties, 16);
car_holders_sort_by_phone_number        *psCarHoldersSortByPhoneNumer =
               new car_holders_sort_by_phone_number (sRAMproperties, 16);

polymorphic_sort_type              *apSortCarHolders[3];

apSortCarHolders[0] = dynamic_cast<polymorphic_sort_type *> (psCarHoldersSortByName);
apSortCarHolders[1] = dynamic_cast<polymorphic_sort_type *> (psCarHoldersSortByLicensePlate);
apSortCarHolders[2] = dynamic_cast<polymorphic_sort_type *> (psCarHoldersSortByPhoneNumer);
```

* - ./src/btreetest/testbench/examples/example_2_simple_database.cpp

What Can I Do With It?

- all b-tree framework containers use the same iterator type

```
for (i = 0; i < (sizeof (apSortCarHolders) / sizeof (*apSortCarHolders)); i++)
{
     auto sCIter = psCarHolders->cbegin ();
     auto sCIterEnd = psCarHolders->cend ();

     // feed data entries to have those sorted
     apSortCarHolders[i]->insert (sCIter, sCIterEnd);

     // Yes, you can trust your eyes!
     // Any container type within this framework uses the same iterator types...
     sCIter = apSortCarHolders[i]->cbegin ();
     sCIterEnd = apSortCarHolders[i]->cend ();
```

* - ./src/btreetest/testbench/examples/example_2_simple_database.cpp

What Can I Do With It?

- this is useful to employ iterators to insert data via a polymorphic container interface, by using a specialised call

```
for (i = 0; i < (sizeof (apSortCarHolders) / sizeof (*apSortCarHolders)); i++)
{
     auto sCIter = psCarHolders->cbegin ();
     auto sCIterEnd = psCarHolders->cend ();

     // feed data entries to have those sorted
     apSortCarHolders[i]->insert (sCIter, sCIterEnd);

     // Yes, you can trust your eyes!
     // Any container type within this framework uses the same iterator types...
     sCIter = apSortCarHolders[i]->cbegin ();
     sCIterEnd = apSortCarHolders[i]->cend ();
```

* - ./src/btreetest/testbench/examples/example_2_simple_database.cpp

What Can I Do With It?

- to accept external iterators displaying the input, a virtual template method would be required, which is not possible
- here our abstract iterators are good enough

```
for (i = 0; i < (sizeof (apSortCarHolders) / sizeof (*apSortCarHolders)); i++)
{
     auto sCIter = psCarHolders->cbegin ();
     auto sCIterEnd = psCarHolders->cend ();

     // feed data entries to have those sorted
     apSortCarHolders[i]->insert (sCIter, sCIterEnd);

     // Yes, you can trust your eyes!
     // Any container type within this framework uses the same iterator types...
     sCIter = apSortCarHolders[i]->cbegin ();
     sCIterEnd = apSortCarHolders[i]->cend ();
```

* - ./src/btreetest/testbench/examples/example_2_simple_database.cpp

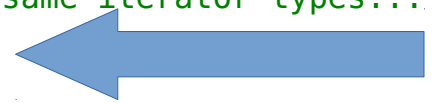
What Can I Do With It?

- this also does allow for an iterator to be re-assigned by an associative type, even if that iterator was generated by a linear type
  - which is a so called “scary assignment”

```
for (i = 0; i < (sizeof (apSortCarHolders) / sizeof (*apSortCarHolders)); i++)
{
     auto sCIter = psCarHolders->cbegin ();
     auto sCIterEnd = psCarHolders->cend ();

     // feed data entries to have those sorted
     apSortCarHolders[i]->insert (sCIter, sCIterEnd);

     // Yes, you can trust your eyes!
     // Any container type within this framework uses the same iterator types...
     sCIter = apSortCarHolders[i]->cbegin ();
     sCIterEnd = apSortCarHolders[i]->cend ();
```



* - ./src/btreetest/testbench/examples/example_2_simple_database.cpp

Can I change my address range?

- the size_type for all b-tree containers can be modified and as a result the programmer is not bound to what size_t is defined as

```
// I need a large address space, since I am no longer writing 32 bit applications
typedef CBTreeIOpropertiesRAM<uint64_t>     ram_data_layer_properties_large_type;

// I want a smaller type, since it creates me slightly faster code, as a 32 bit
// variable has a lesser chance to create CPU cache conflicts compared to a 64 bit type
typedef CBTreeIOpropertiesRAM<uint32_t>     ram_data_layer_properties_small_type;

// I still want to use size_t, since it makes sure I don’t see build warnings of mismatching
// types, when using STL containers in combination with b-tree framework iterators
typedef CBTreeIOpropertiesRAM<size_t>           ram_data_layer_properties_size_t_type;
```

Conclusion

- as oppose to any existing STL implementation, polymorphic containers are possible
- therewith iterator abstraction is possible
  - allowing for so called “scary assignments”
- also applications can create their own abstraction hierarchies, allowing for even better abstract code