Type- Erased Container Iteration

Author: Stephen Kelly <stephen.kelly@kdab.com>

Abstract
Disclosed is a method of processing the elements of a container in a strongly typed language without compile-time knowledge of the type of the container or of the element type within the container.

Keywords: Type-erasure, container, iteration, rtti, runtime type information, dynamic type, dynamic typing, variable type, any type

Introduction
A strong type system provides readability, safety of data interpretation and can be used for compatibility between different code from different vendors. The type of all code components is determined at time of compilation of source code and is immutable. The consumer of an instance of data must know the type of the data in order to process it. Intermediate code may transport references to data, without communicating the type of the data, by using the void pointer type. Such a void pointer may be coupled with an runtime integral value which corresponds to the strict, static type. The integral value may be used at runtime to determine the viability of interpretation of the data as a particular static type. This is already common practice.

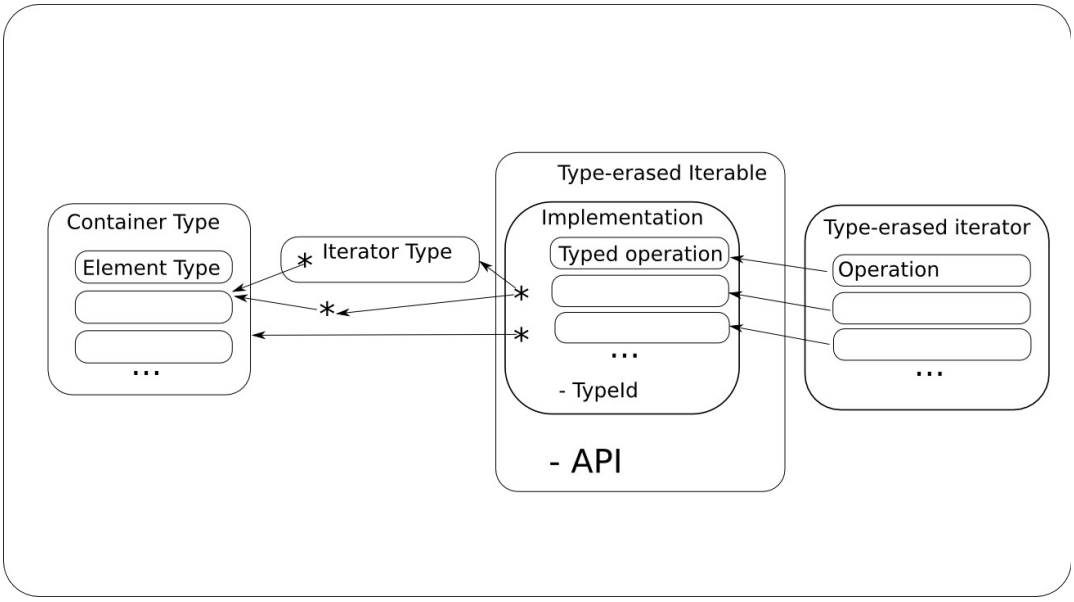
Containers are a category of type which refer to a collection of instances of a particular element type. Containers differ in implementation details such as whether elements are stored contiguously in memory, or as a linked list for example. Containers have associated types which may be used to iterate over and process the elements in the array. Such iterator types are strongly coupled to the element type in the container. Thus, to iterate over and process the elements in a container, the type of the container must be statically (ie, at compile time) known and the type of the elements in the container must be statically known.

Description
A runtime registration system is required to assign an integral value to each unique type which may be contained, and each container. This may use``rtti`` (run-time type information) supplied by the compiler, or it may use external scaffolding.

What is disclosed is a method of ``type erasure`` whereby the elements of a container may be iterated over, without static knowledge of the type of the container or the type of the elements of the container.

This method is applicable to a language which:
1) Is statically typed
2) Has a way to represent data whose type is not known (eg a void pointer)
3) Has a way to cast/interpret such non-typed data as a particular static type, where the type is defined at compile-time.
4) Has a common way to represent operations on containers (eg iterators, algorithms) with "static polymorphism" - that is, the API is the same regardless of the precise type of the container or the type of the element in the container.
5) Has no convenient built-in to provide container abstraction features.

That includes C++, but may include other languages.



In the diagram, the strongly typed container is on the left side 'Container Type', and it has an associated 'Iterator Type'.

1) A static ``Variant`` type is defined as an abstraction containing a void pointer to data and a run-time type identifier. Such a type may be created with statically typed data only. When such a type is created with data which is a container, a registry is populated to relate the run-time type identifier to a container abstraction implementation.

2) The container abstraction implementation ('Implementation' in the diagram) stores a type-erased, immutable void-pointer to the container, a type-erased, mutable void pointer representing an iterator, and various function pointers for the operations ('Typed Operation' in the diagram) which need to be performed on an iterator to the container. Such operations include but are not limited to increment, decrement, dereference, advance, distance, comparison and assignment.

3) The static type used to create the ``Variant`` instance is also used to construct the container abstraction implementation. The function pointers in the container abstraction implementation are generated with a factory template-dependent method. The API of the pointed-to functions are type-erased.

4) The implementation of the pointed-to functions operate on iterators and are implemented according to standards and library features for iterators, including but not limited to increment, decrement dereference, advance, distance, comparison and assignment.

5) There are generally two different forms of container iterators. One is a pointer to contained data, which is suitable for containers which store elements contiguously in memory. The other is a standalone type which supports the standard API required of an iterator type. The two forms are handled differently. In the first case, the type-erased mutable void pointer representing an iterator stored in the container abstraction implementation may point to the actual data in the container. In the second case, the type-erased mutable void pointer representing an iterator stored in the container abstraction implementation may point to a heap-assigned copy of the standalone type which supports the standard API required of an iterator type. The heap-assigned copy is deleted when no longer required.

6) An abstraction is defined which encapsulates the operations on the container iterator objects. The abstraction must handle at least assignment (which may include heap construction), deletion (in the case of heap construction only), advancing the iterator, dereferencing and comparison. The abstraction is implemented in terms of an iterator as a-pointer-to-contained data and as a standalone type. In all cases, the result of mutation is stored in the type-erased, mutable void pointer representing an iterator in the container abstraction implementation.

7) The container abstraction is defined in terms of the container abstraction implementation. The container abstraction ('Type-erased Iterable' in the diagram) contains standard methods to allow creation of type-erased iterator implementations pointing to the beginning and end of a container, as well as iterators pointing to particular seeked elements. This allows the use of standard library algorithms and language features as they apply to the containers. The Type-erased Iterator is implemented in terms of the container abstraction implementation.

8) Dereferencing a type-erased iterator yields an instance of a ``Variant``, which contains a type-erased void pointer to data in the container, and an integral value representing the type. The integral value may be examined to interpret the void pointer as a static type which can be used with other code.

Listing 1: Basic Test

```
/*
  This file is part of an example implementation of type-erased container
  iteration

  Copyright (C) 2013,2014 Klarälvdalens Datakonsult AB, a KDAB Group company,
  info@kdab.com
  Author: Stephen Kelly <stephen.kelly@kdab.com>

  All rights reserved.

  Redistribution and use in source and binary forms, with or without
  modification, are permitted provided that the following conditions are met:

  1. Redistributions of source code must retain the above copyright notice,
  this list of conditions and the following disclaimer.

  2. Redistributions in binary form must reproduce the above copyright notice,
  this list of conditions and the following disclaimer in the documentation
  and/or other materials provided with the distribution.

  3. Neither the name of the copyright holder nor the names of its contributors
  may be used to endorse or promote products derived from this software without
  specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
  LIABLE FOR   ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES   (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE.
*/

#include "types.h"

#include <iostream>
#include <vector>
#include <list>
#include <deque>
#include <forward_list>

void print(const TypeErasure::Variant &var)
{
   std::cout << "Item: ";
   if (var.type() == typeid(int).hash_code())
      std::cout << var.as<int>();
   else if (var.type() == typeid(std::string).hash_code())
      std::cout << var.as<std::string>();
   else if (var.type() == typeid(double).hash_code())
      std::cout << var.as<double>();
    // Delibrately omitted to show that the limitation in the testcase is of
    // printing elements only.  The type-erased container abstraction can
    // iterate over the elements, can determine the size of the container etc.
//    else if (var.type() == typeid(bool).hash_code())
//       std::cout << std::boolalpha << var.as<bool>();
```

```
    else
        std::cout << "<Unknown>";
    std::cout << std::endl;
}

int main(int argc, char **argv)
{
    {
    std::vector<int> vec;
    vec.push_back(4);
    vec.push_back(7);
    vec.push_back(4);
    vec.push_back(1);

    TypeErasure::Variant var(vec);

    TypeErasure::SequentialIterable iter = var.as<TypeErasure::SequentialIterable>();

    // Demonstrate stl-style iteration.
    auto it = std::begin(iter);
    const auto endIt = std::end(iter);

    for( ; it != endIt; ++it)
        {
        print(*it);
        }
    }

    {
    std::vector<std::string> vec2;
    vec2.push_back("fee");
    vec2.push_back("fih");
    vec2.push_back("foh");
    vec2.push_back("fum");

    TypeErasure::Variant var(vec2);

    TypeErasure::SequentialIterable iter = var.as<TypeErasure::SequentialIterable>();

    // Demonstrate c++11-style range-for iteration.
    for (TypeErasure::Variant v : iter)
        {
        print(v);
        }
    }

    {
    std::list<int> li;
    li.push_back(42);
    li.push_back(57);
    li.push_back(47);
    li.push_back(15);

    TypeErasure::Variant var(li);

    TypeErasure::SequentialIterable iter = var.as<TypeErasure::SequentialIterable>();

    std::cout << "List size: " << iter.size();

    // Demonstrate the runtime determination of whether it is possible to
    // iterate backwards over a container.
```

```
    std::cout << " (Can " << (iter.canReverseIterate() ? "" : "not ") << "reverse iterate)" << std::endl;

    // Demonstrate c++11-style range-for iteration.
    for (auto v : iter)
        {
        print(v);
        }

    std::cout << "Reverse:" << std::endl;
    const auto beginIt = std::begin(iter);
    auto it = std::end(iter);

    do
        {
        --it;
        print(*it);
        } while (it != beginIt);
    }

    {
    std::deque<bool> de;
    de.push_back(true);
    de.push_back(false);
    de.push_back(true);

    TypeErasure::Variant var(de);

    TypeErasure::SequentialIterable iter = var.as<TypeErasure::SequentialIterable>();

    std::cout << "Deque size: " << iter.size() << std::endl;

    for (auto v : iter)
        {
        print(v);
        }
    }

    {
    std::forward_list<double> fl;
    fl.push_front(3.14);
    fl.push_front(9.8);

    TypeErasure::Variant var(fl);

    TypeErasure::SequentialIterable iter = var.as<TypeErasure::SequentialIterable>();

    std::cout << "Forward list size: " << iter.size();
    std::cout << " (Can " << (iter.canReverseIterate() ? "" : "not ") << "reverse iterate)" << std::endl;

    for (auto v : iter)
        {
        print(v);
        }
    }

    return 0;
}
```

Listing 2: Implementation

```
/*
  This file is part of an example implementation of type-erased container
  iteration

  Copyright (C) 2013,2014 Klarälvdalens Datakonsult AB, a KDAB Group company,
  info@kdab.com
  Author: Stephen Kelly <stephen.kelly@kdab.com>

  All rights reserved.

  Redistribution and use in source and binary forms, with or without
  modification, are permitted provided that the following conditions are met:

  1. Redistributions of source code must retain the above copyright notice,
  this list of conditions and the following disclaimer.

  2. Redistributions in binary form must reproduce the above copyright notice,
  this list of conditions and the following disclaimer in the documentation
  and/or other materials provided with the distribution.

  3. Neither the name of the copyright holder nor the names of its contributors
  may be used to endorse or promote products derived from this software without
  specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
  LIABLE FOR   ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES   (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE.
*/

#include <typeinfo>
#include <map>
#include <atomic>

#include <assert.h>

namespace TypeErasure
{

/**
  @brief Structure of void pointer and runtime type id.

  This is the data-implementation of the ``Variant``, but must be separate
  because the ``Variant`` has API which depends on the
  SequentialIterableImplementation and must be declared after it.  This
  ``VariantData`` can be declared early and used in the
  SequentialIterableImplementation code.

  As it is the data implementation of the ``Variant`` type, this relates to
  [Disclosure 1].
 */
struct VariantData
```

```
{
   VariantData(const std::size_t metaTypeId_,
               const void *data_)
     : metaTypeId(metaTypeId_)
     , data(data_)
   {
   }
   const std::size_t metaTypeId;
   const void *data;
};

/**
 @brief Iterator operation abstraction

 An implementation of IteratorAPI for containers whose const_iterator is a
 standalone class, not simply a pointer to a contained element.

 It is therefore necessary to copy-construct const_iterator types on the heap
 and manage the memory by deleting appropriately.

 Accessing the actual data in getData() may require de-referencing the pointer to
 the iterator, then de-referencing the actual iterator and retrieving the
 address of the relevant data.

 This is the implementation of [Disclosure 6] and relates to [Disclosure 5].
 These methods implement [Disclosure 4] through the use of algorithms such as
 ``std::advance`` and other operations on iterators.
*/
template<typename const_iterator>
struct IteratorAPI
{
   static void assign(void **ptr, const_iterator iterator)
   {
      *ptr = new const_iterator(iterator);
   }
   static void assign(void **ptr, void * const * src)
   {
      *ptr = new const_iterator(*static_cast<const_iterator*>(*src));
   }

   static void advance(void **iterator, int step)
   {
      const_iterator &it = *static_cast<const_iterator*>(*iterator);
      std::advance(it, step);
   }

   static void destroy(void **ptr)
   {
      delete static_cast<const_iterator*>(*ptr);
   }

   static const void *getData(void * const *iterator)
   {
      return &**static_cast<const_iterator*>(*iterator);
   }

   static const void *getData(const_iterator it)
   {
      return &*it;
   }
```

```
    static bool equal(void * const *it, void * const *other)
    {
        return *static_cast<const_iterator*>(*it) == *static_cast<const_iterator*>(*other);
    }
};
/**
  @brief Iterator operation abstraction

  An implementation of IteratorAPI for containers whose const_iterator is
  simply a pointer to a contained element, not a standalone class.

  It is not necessary (or necessarily possible) to copy-construct such elements
  on the heap as in the alternative implementation above.  Assignment is
  therefore more simple, and 'deletion' is a no-op.

  Accessing the actual data in getData() may require only de-referencing the
  pointer to the iterator. The iterator is already the address of the relevant
  data.

  This is the implementation of [Disclosure 6] and relates to [Disclosure 5].
  These methods implement [Disclosure 4] through the use of algorithms such as
  ``std::advance`` and other operations on iterators.
 */
template<typename value_type>
struct IteratorAPI<const value_type*>
{
    static void assign(void **ptr, const value_type *iterator )
    {
        *ptr = const_cast<value_type*>(iterator);
    }
    static void assign(void **ptr, void * const * src)
    {
        *ptr = static_cast<value_type*>(*src);
    }

    static void advance(void **iterator, int step)
    {
        value_type *it = static_cast<value_type*>(*iterator);
        std::advance(it, step);
        *iterator = it;
    }

    static void destroy(void **)
    {
    }

    static const void *getData(void * const *iterator)
    {
        return *iterator;
    }

    static const void *getData(const value_type *it)
    {
        return it;
    }

    static bool equal(void * const *it, void * const *other)
    {
        return static_cast<value_type*>(*it) == static_cast<value_type*>(*other);
    }
};
```

```
/**
  Capabilities which map to some standard concepts for recording and use at
  runtime.  This allows runtime determination of whether it is possible to
  perform backwards or random iteration with the container.

  As it relates to API for use by downstreams, this relates to [Disclosure 7]
 */
enum IteratorCapability
{
    ForwardCapability = 1,
    BiDirectionalCapability = 2,
    RandomAccessCapability = 4
};

template<typename T, typename Category = typename std::iterator_traits<typename T::const_iterator>::iterator_category>
struct CapabilitiesImpl;

template<typename T>
struct CapabilitiesImpl<T, std::forward_iterator_tag>
{ enum { IteratorCapabilities = ForwardCapability }; };
template<typename T>
struct CapabilitiesImpl<T, std::bidirectional_iterator_tag>
{ enum { IteratorCapabilities = BiDirectionalCapability | ForwardCapability }; };
template<typename T>
struct CapabilitiesImpl<T, std::random_access_iterator_tag>
{ enum { IteratorCapabilities = RandomAccessCapability | BiDirectionalCapability | ForwardCapability }; };

template<typename T>
struct ContainerAPI : CapabilitiesImpl<T>
{
    /**
      This method implements [Disclosure 4] through the use of the
      ``std::distance`` algorithm.  The ContainerAPI template may be
      specialized for particular containers which may implement a more-efficient
      way to determine the size.
    */
    static int size(const T *t) { return std::distance(t->begin(), t->end()); }
};

/**
  @brief Structure of reference to a container data and operations to perform on it.

  Store a type-erased immutable reference to the container, the runtime-id of the type of
  the elements in the container, the capabilities so that usable API may be
  determined at runtime, function pointers for relevant operations, and a
  mutable location to store a type-erased iterator while it is in use.

  The function pointers for iterator operations have type-erased API - they
  have parameters and return types which are void pointers or basic types.
  Although the API of the functions are type-erased, particular typed
  implementations of these operations are generated in the contructor of
  SequentialIterableImplementation ([Disclosure 3]).

  This class implements [Disclosure 2].
  The methods in this class implement [Disclosure 4].
 */
class SequentialIterableImplementation
{
public:
    const void * _iterable;
```

```
void *_iterator;
std::size_t _metaType_id;
unsigned _iteratorCapabilities;
typedef int(*sizeFunc)(const void *p);
typedef const void * (*atFunc)(const void *p, int);
typedef void (*moveIteratorFunc)(const void *p, void **);
typedef void (*advanceFunc)(void **p, int);
typedef VariantData (*getFunc)( void * const *p, std::size_t metaTypeId);
typedef void (*destroyIterFunc)(void **p);
typedef bool (*equalIterFunc)(void * const *p, void * const *other);
typedef void (*copyIterFunc)(void **, void * const *);

sizeFunc _size;
atFunc _at;
moveIteratorFunc _moveToBegin;
moveIteratorFunc _moveToEnd;
advanceFunc _advance;
getFunc _get;
destroyIterFunc _destroyIter;
equalIterFunc _equalIter;
copyIterFunc _copyIter;

template<class T>
static int sizeImpl(const void *p)
{ return ContainerAPI<T>::size(static_cast<const T*>(p)); }

template<class T>
static const void* atImpl(const void *p, int idx)
{
   typename T::const_iterator i = static_cast<const T*>(p)->begin();
   std::advance(i, idx);
   return IteratorAPI<typename T::const_iterator>::getData(i);
}

template<class T>
static void advanceImpl(void **p, int step)
{ IteratorAPI<typename T::const_iterator>::advance(p, step); }

template<class T>
static void moveToBeginImpl(const void *container, void **iterator)
{ IteratorAPI<typename T::const_iterator>::assign(iterator, static_cast<const T*>(container)->begin()); }

template<class T>
static void moveToEndImpl(const void *container, void **iterator)
{ IteratorAPI<typename T::const_iterator>::assign(iterator, static_cast<const T*>(container)->end()); }

template<class T>
static void destroyIterImpl(void **iterator)
{ IteratorAPI<typename T::const_iterator>::destroy(iterator); }

template<class T>
static bool equalIterImpl(void * const *iterator, void * const *other)
{ return IteratorAPI<typename T::const_iterator>::equal(iterator, other); }

template<class T>
static VariantData getImpl(void * const *iterator, std::size_t metaTypeId)
{ return VariantData(metaTypeId, IteratorAPI<typename T::const_iterator>::getData(iterator)); }

template<class T>
static void copyIterImpl(void **dest, void * const * src)
{ IteratorAPI<typename T::const_iterator>::assign(dest, src); }
```

```
public:
    /**
      @brief Constructor taking a pointer to a strongly-typed container.

      Although the reference to the strongly-typed container is stored as a
      type-erased void pointer, the strong type is necessary here in order
      to create typed function pointers (with type-erased API) for each
      relevant operation.

      This method implements [Disclosure 3].
      The actual implementations of the created functions relate to
      [Disclosure 5].
     */
    template<class T> SequentialIterableImplementation(const T*p)
      : _iterable(p)
      , _iterator(0)
      , _metaType_id(typeid(typename T::value_type).hash_code())
      , _iteratorCapabilities(ContainerAPI<T>::IteratorCapabilities)
      , _size(sizeImpl<T>)
      , _at(atImpl<T>)
      , _moveToBegin(moveToBeginImpl<T>)
      , _moveToEnd(moveToEndImpl<T>)
      , _advance(advanceImpl<T>)
      , _get(getImpl<T>)
      , _destroyIter(destroyIterImpl<T>)
      , _equalIter(equalIterImpl<T>)
      , _copyIter(copyIterImpl<T>)
    {
    }

    /**
      @brief Default constructor

      Initialize all data to null.
     */
    SequentialIterableImplementation()
      : _iterable(0)
      , _iterator(0)
      , _metaType_id(typeid(void).hash_code())
      , _iteratorCapabilities(0)
      , _size(0)
      , _at(0)
      , _moveToBegin(0)
      , _moveToEnd(0)
      , _advance(0)
      , _get(0)
      , _destroyIter(0)
      , _equalIter(0)
      , _copyIter(0)
    {
    }

    inline void moveToBegin() { _moveToBegin(_iterable, &_iterator); }
    inline void moveToEnd() { _moveToEnd(_iterable, &_iterator); }
    inline bool equal(const SequentialIterableImplementation &other) const { return _equalIter(&_iterator,
&other._iterator); }
    inline SequentialIterableImplementation &advance(int i) {
      assert(i > 0 || _iteratorCapabilities & BiDirectionalCapability);
      _advance(&_iterator, i);
      return *this;
```

```
    }

    inline VariantData getCurrent() const {
      return _get(&_iterator, _metaType_id);
    }

    VariantData at(int idx) const
    { return VariantData(_metaType_id, _at(_iterable, idx)); }

    int size() const { assert(_iterable); return _size(_iterable); }

    inline void destroyIter() { _destroyIter(&_iterator); }

    void copy(const SequentialIterableImplementation &other)
    {
      *this = other;
      _copyIter(&_iterator, &other._iterator);
    }
};

/**
  This container is populated with mappings from runtime type identifier to
  implementation of type-erased operations [Disclosure 1].
 */
std::map<std::size_t, SequentialIterableImplementation> converterRegistry;

/**
  @brief User-facing API for using the type-erased container implementation.

  This class implements [Disclosure 7].
 */
class SequentialIterable
{
    SequentialIterableImplementation m_impl;
public:
    struct const_iterator;

    friend struct const_iterator;

    explicit SequentialIterable(SequentialIterableImplementation impl);

    const_iterator begin() const;
    const_iterator end() const;

    int size() const;

    bool canReverseIterate() const;
};

int SequentialIterable::size() const
{
    return m_impl.size();
}

bool SequentialIterable::canReverseIterate() const
{
    return m_impl._iteratorCapabilities & BiDirectionalCapability;
}
```

```
/**
  @brief User-facing API for handling type-erased data which may be a container.

  This class implements [Disclosure 1].
 */
struct Variant
{
  VariantData data;

  /**
    @brief Contructor accepting a strongly-typed container.

    Populate the converterRegistry ([Disclosure 1]). Forward the strong type
    to SequentialIterableImplementation to implement [Disclosure 3].
   */
  template<typename T>
  Variant(const T& t)
    : data(typeid(T).hash_code(), &t)
  {
     converterRegistry.insert(std::make_pair(typeid(T).hash_code(), SequentialIterableImplementation(&t)));
  }
  Variant(const VariantData data_)
    : data(data_)
  {
  }

  std::size_t type() const
  {
     return data.metaTypeId;
  }

  template<typename T>
  T as() const
  {
     return *reinterpret_cast<const T*>(data.data);
  }
};

template<>
SequentialIterable Variant::as<SequentialIterable>() const
{
   assert(converterRegistry.find(data.metaTypeId) != converterRegistry.end());

   auto impl = converterRegistry[data.metaTypeId];

   return SequentialIterable{ impl };
}

/**
  @brief User-facing API for using the type-erased container implementation.

  The implementation of these methods forward to the SequentialIterableImplementation.

  This class implements [Disclosure 7]
 */
struct SequentialIterable::const_iterator
{
private:
   SequentialIterableImplementation m_impl;
   std::atomic<int>* ref;
   friend class SequentialIterable;
```

```
    explicit const_iterator(const SequentialIterable &iter, std::atomic<int>* ref_);

    explicit const_iterator(const SequentialIterableImplementation &impl, std::atomic<int>* ref_);

    void begin();
    void end();
public:
    ~const_iterator();

    const_iterator(const const_iterator &other);

    const_iterator& operator=(const const_iterator &other);

    const Variant operator*() const;
    bool operator==(const const_iterator &o) const;
    bool operator!=(const const_iterator &o) const;
    const_iterator &operator++();
    const_iterator operator++(int);
    const_iterator &operator--();
    const_iterator operator--(int);
    const_iterator &operator+=(int j);
    const_iterator &operator-=(int j);
    const_iterator operator+(int j) const;
    const_iterator operator-(int j) const;
};

SequentialIterable::SequentialIterable(SequentialIterableImplementation impl)
  : m_impl(impl)
{
}

SequentialIterable::const_iterator::const_iterator(const SequentialIterable &iter, std::atomic<int>* ref_)
  : m_impl(iter.m_impl), ref(ref_)
{
    ref->fetch_add(1);
}

SequentialIterable::const_iterator::const_iterator(const SequentialIterableImplementation &impl, std::atomic<int>* ref_)
  : m_impl(impl), ref(ref_)
{
    ref->fetch_add(1);

}

void SequentialIterable::const_iterator::begin()
{
    m_impl.moveToBegin();
}

void SequentialIterable::const_iterator::end()
{
    m_impl.moveToEnd();
}

SequentialIterable::const_iterator SequentialIterable::begin() const
{
    const_iterator it(*this, new std::atomic<int>{});
    it.begin();
    return it;
}
```

```
SequentialIterable::const_iterator SequentialIterable::end() const
{
    const_iterator it(*this, new std::atomic<int>{});
    it.end();
    return it;
}

SequentialIterable::const_iterator::~const_iterator() {
    if (!ref->fetch_sub(1)) {
        m_impl.destroyIter();
        delete ref;
    }
}

SequentialIterable::const_iterator::const_iterator(const const_iterator &other)
  : m_impl(other.m_impl), ref(other.ref)
{
    ref->fetch_add(1);
}

SequentialIterable::const_iterator&
SequentialIterable::const_iterator::operator=(const const_iterator &other)
{
    if (!m_impl.equal(other.m_impl)) {
        m_impl = other.m_impl;
        ref = other.ref;
    }
    ref->fetch_add(1);
    return *this;
}

/**
  Dereference operator implements [Disclosure 8]
 */
const Variant SequentialIterable::const_iterator::operator*() const
{
    const VariantData d = m_impl.getCurrent();
    if (d.metaTypeId == typeid(Variant).hash_code())
        return *reinterpret_cast<const Variant*>(d.data);
    Variant v = { d };
    return v;
}

bool SequentialIterable::const_iterator::operator==(const const_iterator &other) const
{
    return m_impl.equal(other.m_impl);
}

bool SequentialIterable::const_iterator::operator!=(const const_iterator &other) const
{
    return !m_impl.equal(other.m_impl);
}

SequentialIterable::const_iterator &SequentialIterable::const_iterator::operator++()
{
    m_impl.advance(1);
    return *this;
}

SequentialIterable::const_iterator SequentialIterable::const_iterator::operator++(int)
{
```

```
    SequentialIterableImplementation impl;
    impl.copy(m_impl);
    m_impl.advance(1);
    return const_iterator(impl, new std::atomic<int>{});
}

SequentialIterable::const_iterator &SequentialIterable::const_iterator::operator--()
{
    m_impl.advance(-1);
    return *this;
}

SequentialIterable::const_iterator SequentialIterable::const_iterator::operator--(int)
{
    SequentialIterableImplementation impl;
    impl.copy(m_impl);
    m_impl.advance(-1);
    return const_iterator(impl, new std::atomic<int>{});
}

SequentialIterable::const_iterator &SequentialIterable::const_iterator::operator+=(int j)
{
    m_impl.advance(j);
    return *this;
}

SequentialIterable::const_iterator &SequentialIterable::const_iterator::operator-=(int j)
{
    m_impl.advance(-j);
    return *this;
}

SequentialIterable::const_iterator SequentialIterable::const_iterator::operator+(int j) const
{
    SequentialIterableImplementation impl;
    impl.copy(m_impl);
    impl.advance(j);
    return const_iterator(impl, new std::atomic<int>{});
}

SequentialIterable::const_iterator SequentialIterable::const_iterator::operator-(int j) const
{
    SequentialIterableImplementation impl;
    impl.copy(m_impl);
    impl.advance(-j);
    return const_iterator(impl, new std::atomic<int>{});
}

}
```