

# Geometric Tools Library: Utility

David Eberly

April 6, 2022

# Contents

<b>1</b>	<b>Exceptions</b>	<b>3</b>
<b>2</b>	<b>StringUtility</b>	<b>5</b>
<b>3</b>	<b>AtomicMinMax</b>	<b>7</b>
<b>4</b>	<b>RangeIteration</b>	<b>9</b>
<b>5</b>	<b>Pointer Comparison</b>	<b>13</b>
5.1	Shared Pointer Comparison . . . . .	13
5.2	Weak Pointer Comparison . . . . .	15
<b>6</b>	<b>HashCombine</b>	<b>19</b>
<b>7</b>	<b>ContainerAdapter</b>	<b>21</b>
<b>8</b>	<b>Multiarray</b>	<b>27</b>
8.1	Lattices . . . . .	27
8.2	Multidimensional Arrays . . . . .	31
8.3	Multidimensional Array Adapters . . . . .	34
<b>9</b>	<b>MinHeap</b>	<b>37</b>
<b>10</b>	<b>Timer</b>	<b>47</b>
	<b>Bibliography</b>	<b>47</b>

This is the reference manual for the utility code of the Geometric Tools Library (GTL). The source code is in C++ and requires at least version 14 of the language. Throughout the documentation, I will refer to C++ with the implication that the version is at least 14. No backport of GTL is available for previous versions of C++.

The utility library consists only of header files that can be used in applications as-is. These files contain some basic features that I find to be useful in the GTL. These include

- Macros to support throwing C++ standard exceptions.
- Simple string conversion and parsing utilities.
- Atomic minimum and maximum operations.
- Range-based iteration using reversed order of traversal of containers.
- Comparisons between objects referenced by smart pointers or weak pointers.
- Support for creating hash values for a list of types, each such type `T` having a valid `std::hash<T>()` function.
- Container adapters, one for wrapping an array of numbers of type `T` with number of elements known at compile time (mimics `std::array<T, N>`) and one for wrapping a vector of numbers of type `T` with number of elements known only at runtime (mimics `std::vector<T>`).
- Support for multidimensional arrays (number of elements known at compile time) and for multidimensional vectors (number of elements known only at runtime). This extends the concepts of `std::array` and `std::vector` to dimensions 2 and larger. Also provide are adapters for mapping 1-dimensional data to multiple dimensions in lexicographical order. The index mapping is provided by a lattice class.
- A min-heap data structure that has the ability to modify interior tree nodes and with tree updating in logarithmic time.
- A simple 64-bit timer for performance measurements.



# Chapter 1

## Exceptions

[GTL/Utility/Exceptions.h](#)

The GTL can throw exceptions for unexpected conditions at runtime. These are wrapped with macros to allow classification of the type of exception. The caller is responsible for catching the exceptions and handling them as desired.

The macros are shown in Listing 1.

---

**Listing 1.** The general-purpose macros and wrappers for runtime or invalid-argument assertions and errors are shown here. Other examples can be found in the `Exceptions.h` file.

```
// The generic assertion allows you to specify any exception type you
// prefer, including user-defined exception types.
#define GTL_ASSERT(condition, exception, message) \
{ \
    std::string sLine = std::to_string(__LINE__); \
    std::string sFile(__FILE__); \
    std::string sFunc(__FUNCTION__); \
    std::string report = sFile+"("+sFunc+", "+sLine+"): "+message+"\n"; \
    throw exception(report); \
}

// The generic error allows you to specify any exception type you
// prefer, including user-defined exception types.
#define GTL_ERROR(exception, message) \
{ \
    std::string sLine = std::to_string(__LINE__); \
    std::string sFile(__FILE__); \
    std::string sFunc(__FUNCTION__); \
    std::string report = sFile+"("+sFunc+", "+sLine+"): "+message+"\n"; \
    throw exception(report); \
}

// Runtime errors occur when invalid conditions occur during program
// execution.
#define GTL_RUNTIME_ASSERT(condition, message) \
GTL_ASSERT(condition, std::runtime_error, message)

#define GTL_RUNTIME_ERROR(message) \
GTL_ERROR(std::runtime_error, message)
```

```

#define GTLARGUMENT_ASSERT(condition, message) \
GTL_ASSERT(condition, std::invalid_argument, message)

#define GTLARGUMENT_ERROR(message) \
GTL_ERROR(std::invalid_argument, message)

```

---

The `GTL_ASSERT` macro has a conditional expression as its first argument. If that argument is `false`, the exception is thrown of the type of the second macro argument. The exception message is the last argument. The `GTL_ERROR` macro is used when external code logic leads to a code block that should not be reached under normal conditions. An exception is thrown similar to that in `GTL_ASSERT`.

Listing 2 shows a simple example that uses the exception macros.

---

**Listing 2.** An example for using the exception macros. Assume this code is in a file named `MyDivisionFile.cpp`.

```

1  #include <GTL/Mathematics/Utility/Exceptions.h>
2  #include <cmath>
3
4  float DoDivision(float numerator, float denominator)
5  {
6      // An exception is thrown when the denominator is zero.
7      GTLARGUMENT_ASSERT(denominator != 0.0f, "Divison by zero.");
8      return numerator / denominator;
9  }
10
11 float AnotherDoDivision(float numerator, float denominator)
12 {
13     float ratio = numerator / denominator;
14     if (denominator != 0.0f)
15     {
16         return numerator / denominator;
17     }
18
19     // An exception is thrown when the denominator is zero.
20     GTLARGUMENT_ERROR("Divison by zero.");
21 }

```

---

For `DoDivision`, the exception string reported by the `what()` member of the exception class is

“MyDivisionFile.cpp(DoDivision,7): Division by zero.”

For `AnotherDoDivision`, the exception string is

“MyDivisionFile.cpp(AnotherDoDivision,20): Division by zero.”

## Chapter 2

# StringUtility

[GTL/Utility/StringUtility.h](#)

The string utilities consist of several convenient functions that are used through the GTL. The interfaces are shown in Listing 3.

---

**Listing 3.** Convenient utilities for manipulating strings. The functions are defined in the `gtl` namespace.

```
std::wstring ConvertNarrowToWide(std::string const& input);
std::string ConvertWideToNarrow(std::wstring const& input);
std::string ToLower(std::string const& input);
std::string ToUpper(std::string const& input);
void GetTokens(std::string const& input, std::string const& whiteSpace, std::vector<std::string>& tokens);
void GetTextTokens(std::string const& input, std::vector<std::string>& tokens);
void GetAdvancedTextTokens(std::string const& input, std::vector<std::string>& tokens);
```

---

The functions `ConvertNarrowToWide`, `ConvertWideToNarrow`, `ToLower` and `ToUpper` use the `<algorithm>` functions `std::transform` and `std::back_inserter`. Previously, the conversion between single-byte and multibyte string used `<codecvt>`, but this approach is now deprecated in C++.

The function `GetTokens` parses the input strings for tokens which depend on a definition for white-space characters and non-white-space characters. Each returned token consists of a maximum-length string of consecutive non-white-space characters. In the default locale for C++ strings, the white-space characters are space (0x20, ' '), form feed (0x0C, '\f'), line feed (0x0A, '\n'), carriage return (0x0D, '\r'), horizontal tab (0x09, '\t') and vertical tab (0x0B, '\v').

The function `GetTextTokens` calls `GetTokens` with white-space characters chosen to be ASCII values 0x00–0x20 and 0x7F–0xFF. The function `GetAdvancedTextTokens` calls `GetTokens` with white-space characters chosen to be the ASCII values 0x00–0x20 and 0x7F. The characters with codes 0x80–0xFF are retained as non-white-space characters.

Listing 4 illustrates how to use the functions.

---

**Listing 4.** Examples that use the string utilities of the GTL.

```
std::string nstr, tolower, toupper;
std::wstring wstr;
std::vector<std::string> tokens;

nstr = "abc DEF_123\t?\n";
wstr = ConvertNarrowToWide(nstr); // L"abc DEF_123\t?\n"
nstr = ConvertWideToNarrow(wstr); // "abc DEF_123\t?\n"
tolower = ToLower(nstr); // "abc def_123\t?\n"
toupper = ToUpper(nstr); // "ABC DEF_123\t?\n"

GetTokens(nstr, "aE?", tokens); // tokens = { "bc D", "F_123\t", "\n" }
GetTextTokens(nstr, tokens); // tokens = { "abc", "DEF_123", "?" }

nstr[4] = '\0'; // ASCII 0xF8
GetTextTokens(nstr, tokens); // tokens = { "abc", "EF_123", "?" }
GetAdvancedTextTokens(nstr, tokens); // tokens = { "abc", "\0EF_123", "?" }
```

---



## Chapter 3

# AtomicMinMax

[GTL/Utility/AtomicMinMax.h](#)

Computing the minimum or maximum of two numbers as an atomic operation might seem like a simple thing to do. However, accomplishing this depends on the architecture of the memory system. C++ has atomic operations, including compare-and-exchange, that support computing the minimum or maximum of numbers. I provide an implementation that uses a weak form of compare-and-exchange, which on some platforms supposedly performs better than using a strong form of compare-and-exchange. Listing 5 contains the source code in the file `AtomicMinMax.h`.

---

**Listing 5.** Computing the minimum or maximum of two numbers as an atomic operation. The functions are defined in the `gtl` namespace.

```
template <typename T>
void AtomicMin(std::atomic<T>& v0, T const& v1)
{
    T vInitial, vMin;
    do
    {
        vInitial = v0;
        vMin = std::min(vInitial, v1);
    }
    while (!std::atomic_compare_exchange_weak(&v0, &vInitial, vMin));

    // On return, v0 = min(v0, v1) and vInitial is the original value of
    // v0 that was passed to the function.
    return vInitial;
}

template <typename T>
T AtomicMax(std::atomic<T>& v0, T const& v1)
{
    T vInitial, vMax;
    do
    {
        vInitial = v0;
        vMax = std::max(vInitial, v1);
    }
    while (!std::atomic_compare_exchange_weak(&v0, &vInitial, vMax));

    // On return, v0 = max(v0, v1) and vInitial is the original value of
```

```
    // v0 that was passed to the function.  
    return vInitial;  
}
```

---

The function `std::atomic_compare_exchange_weak(x,y,z)` compares the values `*x` and `*y`. If the values are the same, then `*x` is replaced by `z` atomically and the function returns `true`. If the values are different, then `*y` is replaced by `z` and the function returns `false`. The assignment is not explicitly atomic, but in Listing 5, `vInitial` is on the stack for the thread calling `AtomicMin` or `AtomicMax`, so no other thread can interfere with the assignment.

The comparison is bitwise in the sense of `std::memcmp`. It is possible that the numbers differ bitwise but are equal in the sense of `operator==`. For example, if the type `T` is `float`, the bit pattern for `+0.0f` is `0x00000000` and the bit pattern for `-0.0f` is `0x80000000`. The bitwise comparison says the two numbers are different, but the comparison as floating-point numbers says they are the same. Whether or not this is a problem in your application is for you to decide.

The weak version of compare-and-exchange can have a spurious failure, where the function returns `false` even though `*x` and `*y` are the same value. This is an issue of the hardware architecture for the memory system, which you can explore via Internet searches on how to compute atomic minimum or maximum. In the code for `AtomicMin` and `AtomicMax`, regardless of whether a comparison or a spurious failure leads to `std::atomic_compare_exchange_weak` returning `false`, the loop ensures another attempt will be made to compare and exchange.

On return from `AtomicMin` or `AtomicMax`, the returned value `vInitial` is the original value contained by `v0`.

## Chapter 4

# RangeIteration

[GTL/Utility/RangeIteration.h](#)

C++ supports range-based for-loops to iterate over a range of elements, typically those in a standard container. In previous versions of the language, the iteration over a range is performed using standard iterators. Listing 6 provides an example of iterating over a `std::vector` container. It also shows the iteration using range-based for-loops.

---

**Listing 6.** Read-only looping over a `std::vector` container.

```
std::vector<int> numbers = { 0, 1, 2, 3 };

// Forward iteration using standard iterators.
for (std::vector<int>::const_iterator i = numbers.begin(); i != numbers.end(); ++i)
{
    std::cout << *i << ' ';
}
// output: 0 1 2 3

// Forward iteration using a range-based for-loop.
for (auto number : numbers)
{
    std::cout << number << ' ';
}
// output: 0 1 2 3
```

---

The number variable in the loop obtains a copy of each element in the `numbers` container. To avoid a copy, use `auto const& number` instead of `auto number`.

The sample code uses a `const_iterator` because the container elements are to be read only. If you need to write to the container, you can use `iterator`. Listing 7 provides an example.

---

**Listing 7.** Read-write looping over a `std::vector` container.

```
std::vector<int> numbers(4);
```

---

```

// Forward iteration using standard iterators.
int value = 0;
for (std::vector<int>::iterator i = numbers.begin(); i != numbers.end(); ++i)
{
    *i = value++;
}
// numbers = \{ 0, 1, 2, 3 \}

// Forward iteration using a range-based for-loop.
int value = 0;
for (auto& number : numbers)
{
    number = 2 * value + 1;
    ++value;
}
// numbers = \{ 1, 3, 5, 7 \}

```

---

The main advantage of range-based for-loops is readability compared to the verbose nature of iterator loops. The concept applies to user-defined containers as long as they implement iterators including the function calls `begin()` and `end()`.

One thing that appears to be missing from the range-based for-loops is the ability to reverse iterate over a container. As it turns out, it is possible to do so but requires some template programming. Listing 8 provides an example.

---

**Listing 8.** Read-only looping over a `std::vector` container using reverse iteration.

```

std::vector<int> numbers = { 0, 1, 2, 3 };

// Reverse iteration using standard iterators.
for (std::vector<int>::const_reverse_iterator i = numbers.rbegin(); i != numbers.rend(); ++i)
{
    std::cout << *i << ' ';
}
// output: 3 2 1 0

// Reverse iteration using a range-based for-loop.
for (auto const& number : gtl::reverse(numbers))
{
    std::cout << number << ' ';
}
// output: 3 2 1 0

```

---

The problem is to design a template function `reverse` whose input is a container and has support for reverse iteration. Listing 9 shows an implementation.

---

**Listing 9.** Support for reverse iteration using range-based for-loops. Because of the inclusion in the `gtl` namespace, the range-based for-loop uses `gtl::reverse(container)` rather than `reverse(container)`. The `ReversalObject` class is defined in the `gtl` namespace.

```

template <typename Iterator>
class ReversalObject
{
public:
    ReversalObject(Iterator inBegin, Iterator inEnd)
        : mBegin(inBegin),
          mEnd(inEnd)
    {}

```

```

    {
    }

    Iterator begin() const
    {
        return mBegin;
    }

    Iterator end() const
    {
        return mEnd;
    }

private:
    Iterator mBegin, mEnd;
};

template
<
    typename Iterable,
    typename Iterator = decltype(std::begin(std::declval<Iterable>())),
    typename ReverselIterator = std::reverse_iterator<Iterator>
>
ReversalObject<ReverselIterator> reverse(Iterable&& range)
{
    return ReversalObject<ReverselIterator>(
        ReverselIterator(std::end(range)),
        ReverselIterator(std::begin(range)));
}

```

---

For a `std::vector<int>` container named `range`, the compiler resolves the expression `gtl::reverse(range)` as follows.

1. `Iterable` resolves to `std::vector<int>`.
2. `std::declval<std::vector<int>>()` resolves to an unevaluated operand `std::vector<int>()`.
3. `decltype` determines that `Iterator` is `std::vector<int>::iterator`, the type of `std::begin(std::vector<int>())`.
4. `std::reverse_iterator` reverses the iterator type so that `ReverselIterator` is `std::vector<int>::reverse_iterator`.
5. The `reverse` function now creates an object of type `ReversalObject<Iterator>`, where `Iterator` is type `std::vector<int>::reverse_iterator`. In the constructor, the `begin` parameter is set to the reversed iterator corresponding to `std::end(range)`, which evaluates to `range.rbegin()`. The `end` parameter is set to the reversed iterator corresponding to `std::begin(range)`, which evaluates to `range.rend()`. Be aware that when an iterator is reversed, `rbegin()` points to the last element of the container and *not* the past-the-end element. Similarly, `rend()` is past-the-end of the reversed container and *not* the first element of the container.
6. The range-based for-loop visits the first element by calling `ReversalObject<Iterator>::begin()`, which returns the last element of `range`. As the loop increments the reversed iterator, the elements of `range` are visited from last to first. The incremented reversed iterator is internally tested for past-the-end by comparing it to returned value of the call to `ReversalObject<Iterator>::end()`.



## Chapter 5

# Pointer Comparison

[GTL/Utility/SharedPtrCompare.h](#)  
[GTL/Utility/WeakPtrCompare.h](#)

### 5.1 Shared Pointer Comparison

The class `std::shared_ptr<T>` already has comparison operators, but these compare pointer values instead of the objects referenced by the pointers.

The GTL structs `SharedPtrOPERATOR<T>` implement comparisons between objects referenced by shared pointers, where `OPERATOR` is one of `EQ` (equal), `NE` (not equal), `LT` (less than), `LE` (less than or equal), `GT` (greater than) or `GE` (greater than or equal). The type `T` must implement comparison operators. Care is required when managing containers of referenced objects. The underlying objects can change, which invalidates the container ordering. If the objects do not change, these containers are safe to use.

Listing 10 contains the source code in the file `SharedPtrCompare.h`.

---

**Listing 10.** Comparisons of objects via shared pointers that reference the objects. The structs are defined in the `gtl` namespace.

```
// sp0 == sp1
template <typename T>
struct SharedPtrEQ
{
    bool operator()(std::shared_ptr<T> const& sp0, std::shared_ptr<T> const& sp1) const
    {
        return (sp0 ? (sp1 ? *sp0 == *sp1 : false) : !sp1);
    }
};

// sp0 != sp1
template <typename T>
struct SharedPtrNE
{
    bool operator()(std::shared_ptr<T> const& sp0, std::shared_ptr<T> const& sp1) const
    {
        return !SharedPtrEQ<T>()(sp0, sp1);
    }
};
```

```

    }
};

// sp0 < sp1
template <typename T>
struct SharedPtrLT
{
    bool operator()(std::shared_ptr<T> const& sp0, std::shared_ptr<T> const& sp1) const
    {
        return (sp1 ? (!sp0 || *sp0 < *sp1) : false);
    }
};

// sp0 <= sp1
template <typename T>
struct SharedPtrLE
{
    bool operator()(std::shared_ptr<T> const& sp0, std::shared_ptr<T> const& sp1) const
    {
        return !SharedPtrLT<T>()(sp1, sp0);
    }
};

// sp0 > sp1
template <typename T>
struct SharedPtrGT
{
    bool operator()(std::shared_ptr<T> const& sp0, std::shared_ptr<T> const& sp1) const
    {
        return SharedPtrLT<T>()(sp1, sp0);
    }
};

// sp0 >= sp1
template <typename T>
struct SharedPtrGE
{
    bool operator()(std::shared_ptr<T> const& sp0, std::shared_ptr<T> const& sp1) const
    {
        return !SharedPtrLT<T>()(sp0, sp1);
    }
};

```

---

Only the equality and less-than comparisons have specific implementations. The other comparisons are based on these using basic logic. The comparisons of two objects referenced by nonnull shared pointers are what you expect. However, if one or both shared pointers are null, additional semantics are required.

For the equality comparison, if the both shared pointers are null, the comparison returns `true`. If one shared pointer is null and the other is not null, the comparison returns `false`. If both shared pointers are not null, they are dereferenced and the objects are compared for equality.

For the less-than comparison, if both shared pointers are null, the comparison returns `false`. If the first shared pointer is null and the second shared pointer is not null, the comparison returns `true`. If the first shared pointer is not null and the second shared pointer is null, the comparison returns `false`. These two choices lead to sorted containers having all the null shared pointers occurring first followed by all the nonnull shared pointers. The ordering of the latter elements are based on the ordering of the referenced objects.

Listing 11 illustrates how to use the structs.

---

**Listing 11.** Examples of how to use comparisons for objects referenced by shared pointers.

```
auto sp0 = std::make_shared<int>(0);
```



```

auto sp1 = std::make_shared<int>(1);
auto sp2 = std::make_shared<int>(1);
bool result;

result = SharedPtrEQ<int>()(sp0, sp1); // false
result = SharedPtrNE<int>()(sp0, sp1); // true
result = SharedPtrLT<int>()(sp0, sp1); // true
result = SharedPtrLE<int>()(sp0, sp1); // true
result = SharedPtrGT<int>()(sp0, sp1); // false
result = SharedPtrGE<int>()(sp0, sp1); // false

result = SharedPtrEQ<int>()(sp1, sp2); // true
result = SharedPtrNE<int>()(sp1, sp2); // false
result = SharedPtrLT<int>()(sp1, sp2); // false
result = SharedPtrLE<int>()(sp1, sp2); // true
result = SharedPtrGT<int>()(sp1, sp2); // false
result = SharedPtrGE<int>()(sp1, sp2); // true

result = SharedPtrEQ<int>()(sp1, sp0); // false
result = SharedPtrNE<int>()(sp1, sp0); // true
result = SharedPtrLT<int>()(sp1, sp0); // false
result = SharedPtrLE<int>()(sp1, sp0); // false
result = SharedPtrGT<int>()(sp1, sp0); // true
result = SharedPtrGE<int>()(sp1, sp0); // true

result = SharedPtrEQ<int>()(nullptr, sp1); // false
result = SharedPtrEQ<int>()(sp0, nullptr); // false
result = SharedPtrEQ<int>()(nullptr, nullptr); // true

result = SharedPtrLT<int>()(nullptr, sp1); // true
result = SharedPtrLT<int>()(sp0, nullptr); // false
result = SharedPtrLT<int>()(nullptr, nullptr); // false

```

---

## 5.2 Weak Pointer Comparison

The comparison of `std::weak_ptr<T>` objects is identical to that of `std::shared_ptr<T>` objects, except that the weak pointers must be locked first before the comparisons. See Section 5.1 for the comparison details.

Listing 12 contains the source code in the file `WeakPtrCompare.h`.

**Listing 12.** Comparisons of objects via weak pointers that reference the objects. The structs are defined in the `gtl` namespace.

```

// wp0 == wp1
template <typename T>
struct WeakPtrEQ
{
    bool operator()(std::weak_ptr<T> const& wp0, std::weak_ptr<T> const& wp1) const
    {
        auto sp0 = wp0.lock(), sp1 = wp1.lock();
        return (sp0 ? (sp1 ? *sp0 == *sp1 : false) : !sp1);
    }
};

// wp0 != wp1
template <typename T>
struct WeakPtrNE
{
    bool operator()(std::weak_ptr<T> const& wp0, std::weak_ptr<T> const& wp1) const
    {
        return !WeakPtrEQ<T>()(wp0, wp1);
    }
};

```

```

};

// wp0 < wp1
template <typename T>
struct WeakPtrLT
{
    bool operator()(std::weak_ptr<T> const& wp0, std::weak_ptr<T> const& wp1) const
    {
        auto sp0 = wp0.lock(), sp1 = wp1.lock();
        return (sp1 ? (!sp0 || *sp0 < *sp1) : false);
    }
};

// wp0 <= wp1
template <typename T>
struct WeakPtrLE
{
    bool operator()(std::weak_ptr<T> const& wp0, std::weak_ptr<T> const& wp1) const
    {
        return !WeakPtrLT<T>()(wp1, wp0);
    }
};

// wp0 > wp1
template <typename T>
struct WeakPtrGT
{
    bool operator()(std::weak_ptr<T> const& wp0, std::weak_ptr<T> const& wp1) const
    {
        return WeakPtrLT<T>()(wp1, wp0);
    }
};

// wp0 >= wp1
template <typename T>
struct WeakPtrGE
{
    bool operator()(std::weak_ptr<T> const& wp0, std::weak_ptr<T> const& wp1) const
    {
        return !WeakPtrLT<T>()(wp0, wp1);
    }
};

```

---

Listing 13 illustrates how to use the structs.

---

**Listing 13.** Examples of how to use comparisons for objects referenced by weak pointers.

```

auto sp0 = std::make_shared<int>(0);
auto sp1 = std::make_shared<int>(1);
auto sp2 = std::make_shared<int>(1);
std::weak_ptr<int> wp0(sp0);
std::weak_ptr<int> wp1(sp1);
std::weak_ptr<int> wp2(sp2);
bool result;

result = WeakPtrEQ<int>()(wp0, wp1); // false
result = WeakPtrNE<int>()(wp0, wp1); // true
result = WeakPtrLT<int>()(wp0, wp1); // true
result = WeakPtrLE<int>()(wp0, wp1); // true
result = WeakPtrGT<int>()(wp0, wp1); // false
result = WeakPtrGE<int>()(wp0, wp1); // false

result = WeakPtrEQ<int>()(wp1, sp2); // true
result = WeakPtrNE<int>()(wp1, sp2); // false
result = WeakPtrLT<int>()(wp1, sp2); // false
result = WeakPtrLE<int>()(wp1, sp2); // true
result = WeakPtrGT<int>()(wp1, sp2); // false

```

```
result = WeakPtrGE<int>()(wp1, sp2); // true

result = WeakPtrEQ<int>()(wp1, wp0); // false
result = WeakPtrNE<int>()(wp1, wp0); // true
result = WeakPtrLT<int>()(wp1, wp0); // false
result = WeakPtrLE<int>()(wp1, wp0); // false
result = WeakPtrGT<int>()(wp1, wp0); // true
result = WeakPtrGE<int>()(wp1, wp0); // true

result = WeakPtrEQ<int>()(nullptr, wp1); // false
result = WeakPtrEQ<int>()(wp0, nullptr); // false
result = WeakPtrEQ<int>()(nullptr, nullptr); // true

result = WeakPtrLT<int>()(nullptr, wp1); // true
result = WeakPtrLT<int>()(wp0, nullptr); // false
result = WeakPtrLT<int>()(nullptr, nullptr); // false
```

---



## Chapter 6

# HashCombine

[GTL/Utility/HashCombine.h](#)

GTL allows creating hash values for a list of types, each such type `T` having a valid `std::hash<T>()` function. The code in GTL comes from the Nicolai M. Josuttis book [Jos12, Section 7.9.2, pp. 364-465]. Credit for the hash-combine concept is from the Boost library [Lib08]. The magic number and shifts are based on the paper [HZ03].

Listing 14 shows the concise implementation for combining the hash values.

---

**Listing 14.** Code to combine hash values to obtain a single hash value for a list of types.

```
template <typename T>
inline void HashCombine(std::size_t& seed, T const& value)
{
    seed ^= std::hash<T>()(value) + 0x9e3779b9 + (seed << 6) + (seed >> 2);
}

// Functions to create a hash value using a seed.
template <typename T>
inline void HashValue(std::size_t& seed, T const& value)
{
    HashCombine(seed, value);
}

template <typename T, typename... Tail>
inline void HashValue(std::size_t& seed, T const& value, Tail const&... arguments)
{
    HashCombine(seed, value);
    HashValue(seed, arguments...);
}

// Functions to create a hash value from a list of arguments.
template <typename... Tail>
inline std::size_t HashValue(Tail const&... arguments)
{
    std::size_t seed = 0;
    HashValue(seed, arguments...);
    return seed;
}
```

---



## Chapter 7

# ContainerAdapter

GTL/Utility/ContainerAdapter.h  
GTL/Utility/RawIterators.h

Some classes and functions in the GTL have interfaces involving inputs that are `std::array` or `std::vector` objects. Typically the template type involves geometric vectors or integer-valued indices. If a user must integrate the GTL with packages that have their own arrays or vectors of objects, the adapter pattern must be used to allow passing raw pointers from the other package to GTL functions.

The `ContainerAdapter` class is an adapter mechanism for 1-dimensional data. It provides the most common member functions that are found in `std::array` and `std::vector`, including the ability to iterate over objects using range-based iteration. It also provides constructors to wrap the raw pointers from another package and it has common member accessors. Listing 15 shows the basic templates for defining container adapters.

---

**Listing 15.** The templates for container adapters.

```
// The primary template for a container that has most of the interface for
// std::vector, but the source data is a raw pointer. This supports
// functions that can accept arguments of type std::array<T,N> or of type
// ContainerAdapter<T,N> for N > 0 (size known at compile time). And it
// supports functions that can accept arguments of type std::vector<T> or
// of type ContainerAdapter<T> (size known only at run time).
template <typename T, size_t...>
class ContainerAdapter;

// The template for number of elements known at compile time.
template <typename T, size_t N>
class ContainerAdapter<T, N>
{
public:
    using value_type = T;
    using size_type = size_t;
    using difference_type = ptrdiff_t;
    using pointer = T*;
    using const_pointer = T const*;
    using reference = T&;
    using const_reference = T const&;
    using iterator = RawIterator<T>;
    using const_iterator = RawConstIterator<T>;
    using reverse_iterator = std::reverse_iterator< iterator >;
```

```

using const_reverse_iterator = std::reverse_iterator<const_iterator>;

// Construction. The input is a non-null pointer to a contiguous array
// of elements. The caller must ensure that the number of elements is
// at least N. You may pass a null pointer, but you are then expected
// to call 'reset(elements)' with a non-null pointer before using the
// object. The last default argument allows generic implementation in
// adapter operations; it is ignored by the constructor.
ContainerAdapter(T* elements = nullptr, size_t = 0);

// Destruction. Nothing is destroyed because the class does not know
// anything about the input raw data. If the data was dynamically
// allocated elsewhere in the application, the caller is responsible
// for deallocating it.
~ContainerAdapter() = default;

// Copy semantics. The copy constructor is disabled to avoid having
// the raw pointer shared between objects, a simplified design for
// the adapter system. Assignment is allowed between objects of the
// same size.
ContainerAdapter(ContainerAdapter const&) = delete;
ContainerAdapter& operator=(ContainerAdapter const& other);

// Move semantics are disabled.
ContainerAdapter(ContainerAdapter&&) noexcept = delete;
ContainerAdapter& operator=(ContainerAdapter&&) noexcept = delete;

// Call this function with a non-null pointer before manipulating a
// ContainerAdapter object created with a null pointer. The default
// argument allows generic implementation in adapter operations; it
// is ignored by the function.
void reset(T* elements, size_t = 0) noexcept;

// Size and data access.
size_t constexpr size() const noexcept;
T* data() noexcept;
T const* data() const noexcept;

// Element access. The 'at' functions throw std::runtime_error
// exceptions when the storage pointer is null, and they throw
// std::out_of_range exceptions for an invalid index. The
// 'operator[]' functions do not throw exceptions.
T const& at(size_t i) const;
T& at(size_t i);
T const& operator[](size_t i) const noexcept;
T& operator[](size_t i) noexcept;

// Support for iteration.
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;
const_iterator cbegin() const;
const_iterator cend() const;
const_reverse_iterator crbegin() const;
const_reverse_iterator crend() const;

// Set all elements to the specified value.
void fill(T const& value);
};
}

// The template for number of elements known only at runtime.
template <typename T>
class ContainerAdapter<T>
{
public:
    using value_type = T;

```



```

using size_type = size_t;
using difference_type = ptrdiff_t;
using pointer = T*;
using const_pointer = T const*;
using reference = T&;
using const_reference = T const&;
using iterator = RawIterator<T>;
using const_iterator = RawConstIterator<T>;
using reverse_iterator = std::reverse_iterator<iterator>;
using const_reverse_iterator = std::reverse_iterator<const_iterator>;

// Construction. Generally, the input is a non-null pointer to a
// contiguous array of elements. The caller must ensure that the
// number of elements is at least 'numElements'. You may pass a null
// pointer, but you are then expected to call
// 'reset(numElements, elements)' with numElements > 0 and a non-null
// pointer before using the object.
ContainerAdapter(T* elements = nullptr, size_t numElements = 0);

// Destruction. Nothing is destroyed because the class does not know
// anything about the input raw data. If this data was dynamically
// allocated elsewhere in the application, the caller is responsible
// for deallocating it.
~ContainerAdapter() = default;

// Copy semantics. The copy constructor is disabled to avoid having
// the raw pointer shared between objects, a simplified design for
// the adapter system. Assignment is allowed between objects of the
// same size.
ContainerAdapter(ContainerAdapter const&) = delete;
ContainerAdapter& operator=(ContainerAdapter const& other);

// Move semantics are disabled.
ContainerAdapter(ContainerAdapter&&) noexcept = delete;
ContainerAdapter& operator=(ContainerAdapter&&) noexcept = delete;

// Call this function with numElements > 0 and a nonnull pointer
// before manipulating a ContainerAdapter object created with no
// elements and a null pointer.
void reset(T* elements, size_t numElements);

// Size and data access.
size_t size() const noexcept;
T* data() noexcept;
T const* data() const noexcept;

// Element access. The 'at' functions throw std::runtime_error
// exceptions when the storage pointer is null, and they throw
// std::out_of_range exceptions for an invalid index. The
// 'operator[]' functions do not throw exceptions.
T const& at(size_t i) const;
T& at(size_t i);
T const& operator[](size_t i) const noexcept;
T& operator[](size_t i) noexcept;

// Support for iteration.
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;
const_iterator cbegin() const;
const_iterator cend() const;
const_reverse_iterator crbegin() const;
const_reverse_iterator crend() const;

// Set all elements to the specified value.
void fill(T const& value);
};

```

---

The container adapter classes depend on an implementation of iterators for raw pointers. The classes `RawConstIterator` and `RawIterator` provide this. Listing 16 shows the template interfaces.

---

**Listing 16.** The templates for raw iterators.

```
template <typename T>
class RawConstIterator
{
public:
    using iterator_category = std::random_access_iterator_tag;
    using value_type = T;
    using difference_type = ptrdiff_t;
    using pointer = T const*;
    using reference = T const&;

    // Construction.
    RawConstIterator();
    explicit RawConstIterator(pointer inPointer, size_t offset = 0);

    // Pointer access.
    reference operator*() const;
    pointer operator->() const;
    reference operator[](ptrdiff_t const offset) const;

    // Pointer arithmetic.
    RawConstIterator& operator++();
    RawConstIterator operator++(int32_t);
    RawConstIterator& operator--();
    RawConstIterator operator--(int32_t);
    RawConstIterator& operator+=(ptrdiff_t const offset);
    RawConstIterator operator+(ptrdiff_t const offset) const;
    RawConstIterator& operator-=(ptrdiff_t const offset);
    RawConstIterator operator-(ptrdiff_t const offset) const;
    ptrdiff_t operator-(RawConstIterator const& other) const;

    // Pointer comparisons.
    bool operator==(RawConstIterator const& other) const;
    bool operator!=(RawConstIterator const& other) const;
    bool operator<(RawConstIterator const& other) const;
    bool operator<=(RawConstIterator const& other) const;
    bool operator>(RawConstIterator const& other) const;
    bool operator>=(RawConstIterator const& other) const;
};

template <typename T>
class RawIterator : public RawConstIterator<T>
{
public:
    using iterator_category = std::random_access_iterator_tag;
    using value_type = T;
    using difference_type = ptrdiff_t;
    using pointer = T*;
    using reference = T&;

    // Construction.
    RawIterator() noexcept;
    explicit RawIterator(pointer inPointer, size_t offset = 0);

    // Pointer access.
    reference operator*() const;
    pointer operator->() const;

    // Pointer arithmetic.
    RawIterator operator++();
    RawIterator operator++(int32_t);
```

```

RawIterator operator--();
RawIterator operator--(int32_t);
RawIterator& operator+=(ptrdiff_t const offset);
RawIterator operator+(ptrdiff_t const offset) const;
RawIterator& operator-=(ptrdiff_t const offset);
RawIterator operator-(ptrdiff_t const offset) const;
};

```

---

An example for using ContainerAdapter is shown in Listing 17.

---

**Listing 17.** This example uses a container adapter to wrap a `std::vector` with `std::array<double, 2>` elements. The average of the elements is computed.

```

template <typename Container, typename ElementType = typename Container::value_type>
ElementType ComputeAverage(Container const& container)
{
    using T = typename ElementType::value_type;
    ElementType average{ static_cast<T>(0), static_cast<T>(0) };
    for (auto& point : container)
    {
        for (size_t i = 0; i < point.size(); ++i)
        {
            average[i] += point[i];
        }
    }
    for (size_t i = 0; i < average.size(); ++i)
    {
        average[i] /= static_cast<T>(container.size());
    }
    return average;
}

void Experiment0()
{
    std::vector<std::array<double, 2>> container0(4);
    container0[0] = { 1.0, 2.0 };
    container0[1] = { 3.0, 4.0 };
    container0[2] = { 5.0, 6.0 };
    container0[3] = { 7.0, 8.0 };
    std::array<double, 2> average0 = ComputeAverage(container0); // (4.0, 5.0)

    ContainerAdapter<std::array<double, 2>, 4> container1(container0.data());
    std::array<double, 2> average1 = ComputeAverage(container1); // (4.0, 5.0)

    ContainerAdapter<std::array<double, 2>> container2(container0.data(), 4);
    std::array<double, 2> average2 = ComputeAverage(container2); // (4.0, 5.0)
}

```

---

Another example for using ContainerAdapter is shown in Listing 18.

---

**Listing 18.** This example uses a container adapter to wrap a `std::vector` with `Vector<double, 2>` elements. The average of the elements is computed. The `Vector<double, 2>` class has support for addition. The inner loop of the range-based container loop in `ComputeAverage` of Listing 17 for `std::array` is replaced by a single statement that uses that addition.

```

template <typename Container, typename ElementType = typename Container::value_type>
ElementType AnotherComputeAverage(Container const& container)
{
    using T = typename ElementType::value_type;

```

```
ElementType average{}; // default constructor sets this to (0,0)
for (auto& point : container)
{
    average += point;
}
average /= static_cast<T>(container.size());
return average;
}

void Experiment1()
{
    std::vector<Vector<double, 2>> container0(4);
    container0[0] = { 1.0, 2.0 };
    container0[1] = { 3.0, 4.0 };
    container0[2] = { 5.0, 6.0 };
    container0[3] = { 7.0, 8.0 };
    Vector<double, 2> average0 = AnotherComputeAverage(container0); // (4.0, 5.0)

    ContainerAdapter<Vector<double, 2>, 4> container1(container0.data());
    Vector<double, 2> average1 = AnotherComputeAverage(container1); // (4.0, 5.0)

    ContainerAdapter<Vector<double, 2>> container2(container0.data(), 4);
    Vector<double, 2> average2 = AnotherComputeAverage(container2); // (4.0, 5.0)
}
```

---

## Chapter 8

# Multiaarray

[GTL/Utility/Lattice.h](#)  
[GTL/Utility/Multiaarray.h](#)  
[GTL/Utility/MultiaarrayAdapter.h](#)

Class `Multiaarray` is a generalization of 1-dimensional contiguous arrays of values to  $n > 1$  dimensions. For the number of elements known at compile time, it is an  $n$ -dimensional generalization of `std::array<T,N>`. For the number of elements known only at runtime, it is a  $n$ -dimensional generalization of `std::vector<T>`.

The indexing or more generally *multiindexing* is managed by class `Lattice`. The `Multiaarray` class provides data storage and access to the data via the indexing. It also provides two forms of lexicographical ordering, one which is a generalization of row-major order for 2-dimensional tables and one which is a generalization of column-major order for 2-dimensional tables. Other orderings are possible, but the complexity of the code increases significantly to support them. One of the two provided orderings is sufficient for applications.

Similar to the container adapter of Chapter 7, it is convenient to have adapters from raw pointers to multidimensional arrays. Class `MultiaarrayAdapter` provides this support. The most common use for such adapters is multiindexing of the raw pointer data. To keep the adapter simple, iterator support is not provided by `MultiaarrayAdapter`.

### 8.1 Lattices

An  $n$ -dimensional lattice is a grid of multiindices  $(x_0, \dots, x_{n-1})$  with constraints  $0 \leq x_d < b_d$  for all  $d$  where the bounds  $(b_0, \dots, b_{n-1})$  are specified, either at compile time or at run time. A multiindex is  $n$ -dimensional and can be mapped to a 1-dimensional index  $i$  where  $0 \leq i < \prod_{d=0}^{n-1} b_d$ . One mapping is  $i = x_0$  for  $n = 1$ ,  $i = x_0 + b_0 x_1$  for  $n = 2$ ,  $i = x_0 + b_0(x_1 + b_1 x_2)$  for  $n = 3$  or generally

$$i = \sum_{i=0}^{n-1} x_i \left( \prod_{j=0}^{n-2} b_j \right) \quad (8.1.1)$$

where the convention is the product  $\prod_{j=0}^{n-2} b_j = 1$  when  $j > n - 2$ . The indexing corresponds to row-major order for  $n = 2$ . I refer to this as left-to-right ordering. Another mapping is  $i = x_0$  for  $n = 1$ ,  $i = x_1 + b_1 x_0$  for  $n = 2$ ,  $i = x_2 + b_2(x_1 + b_1 x_0)$  for  $n = 3$  or generally

$$i = \sum_{i=0}^{n-1} x_i \left( \prod_{j=i+1}^{n-1} b_j \right) \quad (8.1.2)$$

where the convention is the product  $\prod_{j=i+1}^{n-1} b_j = 1$  when  $j + 1 > n - 1$ . The indexing corresponds to column-major order for  $n = 2$ . I refer to this as right-to-left ordering.

The implementation uses template metaprogramming to support efficient compilation of the multiindexing. The details are significant and can be viewed in the source file itself. The public interface for `Lattice` is shown in Listing 19 for bounds known at compile time.

---

**Listing 19.** The public interface of class `Lattice` for bounds known at compile time. The index member functions convert a multidimensional index to a 1-dimensional index. The coordinate member functions convert a 1-dimensional index to a multidimensional index.

```
// Implementation for a lattice whose sizes are known at compile time.
// The class has no data members. Its member functions use template
// metaprogramming for mapping indices to and from multiindices. The
// Sizes parameter pack has n >= 1 elements and represents the bounds
// (b[0], ..., b[n-1]).
template<bool OrderLtoR, size_t... Sizes>
class Lattice
{
public:
    Lattice();
    ~Lattice() = default;

    // The number of dimensions is the number of arguments in the Sizes
    // parameter pack. This is 'n' in the comments about lattices.
    size_t constexpr dimensions() const noexcept;

    // Get the number of elements for dimension d. This is 'b[d]' in the
    // comments about lattices.
    size_t constexpr size(size_t d) const noexcept;

    // Get the number of elements. This is 'product{d=0}^{n-1} b[d]' in
    // the comments about lattices.
    size_t constexpr size() const noexcept;

    // *** Conversions for left-to-right ordering.

    // Convert from an n-dimensional index to a 1-dimensional index for
    // left-to-right ordering.
    template<typename... IndexTypes, bool LeftToRight = OrderLtoR>
    typename std::enable_if<LeftToRight, size_t>::type constexpr
    index(IndexTypes... ntuple) const noexcept;

    template<bool LeftToRight = OrderLtoR>
    typename std::enable_if<LeftToRight, size_t>::type constexpr
    index(std::array<size_t, sizeof...(Sizes)> const& coordinate) const noexcept;

    // Convert from a 1-dimension index to an n-dimensional index for
    // left-to-right ordering.
    template<bool LeftToRight = OrderLtoR>
    typename std::enable_if<LeftToRight, std::array<size_t, sizeof...(Sizes)>>::type constexpr
    coordinate(size_t i) const noexcept;

    // *** Conversions for right-to-left ordering.
```

```

// Convert from an n-dimensional index to a 1-dimensional index for
// right-to-left ordering.
template <typename... IndexTypes, bool LeftToRight = OrderLtoR>
typename std::enable_if<!LeftToRight, size_t >::type constexpr
index(IndexTypes... ntuple) const noexcept;

template <bool LeftToRight = OrderLtoR>
typename std::enable_if<!LeftToRight, size_t >::type constexpr
index(std::array<size_t, sizeof...(Sizes)> const& coordinate) const noexcept;

// Convert from a 1-dimension index to an n-dimensional index for
// right-to-left ordering.
template <bool LeftToRight = OrderLtoR>
typename std::enable_if<!LeftToRight, std::array<size_t, sizeof...(Sizes)>>::type constexpr
coordinate(size_t i) const noexcept;
}

```

---

The public interface for `Lattice` is shown in Listing 20 for bounds known only at runtime. Currently, there is no support for a mixture of compile-time-known bounds for some dimensions and runtime-known bounds for the remaining dimensions.

---

**Listing 20.** The public interface of class `Lattice` for bounds known at runtime. The index member functions convert a multidimensional index to a 1-dimensional index. The coordinate member functions convert a 1-dimensional index to a multidimensional index.

```

template <bool OrderLtoR>
class Lattice<OrderLtoR>
{
public:
    // The lattice has no elements.
    Lattice();

    // The lattice has the specified sizes.
    Lattice(std::vector<size_t> const& sizes);

    // The lattice has the specified sizes.
    Lattice(std::initializer_list<size_t> const& sizes);

    // Support for deferred construction where the initial lattice is
    // created by the default constructor. During later execution, the
    // lattice sizes can be set as needed.
    void reset(std::vector<size_t> const& sizes);
    void reset(std::initializer_list<size_t> const& sizes);

    ~Lattice() = default;

    // Copy semantics.
    Lattice(Lattice const& other);
    Lattice& operator=(Lattice const& other);

    // Move semantics.
    Lattice(Lattice&& other) noexcept;
    Lattice& operator=(Lattice&& other) noexcept;

    // The number of dimensions is the number of elements of mSizes. This
    // is 'n' in the comments about lattices.
    size_t dimensions() const noexcept;

    // Get the number of elements for dimension d. This is 'b[d]' in the
    // comments about lattices.
    size_t size(size_t d) const;

    // Get the number of elements. This is 'product{d=0}^{n-1} b[d]' in
    // the comments about lattices.
    size_t size() const noexcept;

```

```

// Convert from an n-dimensional index to a 1-dimensional index for
// left-to-right ordering.
template <typename... IndexTypes, bool LeftToRight = OrderLtoR>
typename std::enable_if<LeftToRight, size_t >::type
index(IndexTypes... ntuple) const;

template <bool LeftToRight = OrderLtoR>
typename std::enable_if<LeftToRight, size_t >::type
index(std::vector<size_t> const& coordinate) const;

// Convert from an n-dimensional index to a 1-dimensional index for
// right-to-left ordering.
template <typename... IndexTypes, bool LeftToRight = OrderLtoR>
typename std::enable_if<!LeftToRight, size_t >::type
index(IndexTypes... ntuple) const;

template <bool LeftToRight = OrderLtoR>
typename std::enable_if<!LeftToRight, size_t >::type
index(std::vector<size_t> const& coordinate) const;

// Convert from a 1-dimension index to an n-dimensional index for
// left-to-right ordering.
template <bool LeftToRight = OrderLtoR>
typename std::enable_if<LeftToRight, std::vector<size_t >>::type
coordinate(size_t i) const;

// Convert from a 1-dimension index to an n-dimensional index for
// right-to-left ordering.
template <bool LeftToRight = OrderLtoR>
typename std::enable_if<!LeftToRight, std::vector<size_t >>::type
coordinate(size_t i) const;

// Support for sorting and comparing Lattice objects.
bool operator==(Lattice const& other) const;
bool operator!=(Lattice const& other) const;
bool operator<(Lattice const& other) const;
bool operator<=(Lattice const& other) const;
bool operator>(Lattice const& other) const;
bool operator>=(Lattice const& other) const;
}

```

---

Some examples for using Lattice are shown in Listing 21.

---

**Listing 21.** Several examples for using a 3-dimensional lattice with bounds (2,3,5).

```

// This code is independent of storage order. It works for any of the
// declarations of a lattice (the active one or the commented-out ones).
Lattice<true, 2, 3, 5> lattice{};
// Lattice<false, 2, 3, 5> lattice{};
// Lattice<true> lattice{2, 3, 5};
// Lattice<false> lattice{2, 3, 5};
for (size_t x2 = 0, i = 0; x2 < lattice.size(2); ++x2)
{
    for (size_t x1 = 0; x1 < lattice.size(1); ++x1)
    {
        for (size_t x0 = 0; x0 < lattice.size(0); ++x0, ++i)
        {
            size_t k0 = lattice.index(x0, x1, x2);
            GTL_RUNTIME_ASSERT(k0 == i, "invalid index");

            std::array<size_t, 3> x{ x0, x1, x2 };
            size_t k1 = lattice.index(x);
            GTL_RUNTIME_ASSERT(k1 == i, "invalid index");
        }
    }
}

```



```

// This verification code depends on storage order.
Lattice<true, 2, 3, 5> lattice{};
// Lattice<true> lattice{2, 3, 5};
for (size_t i = 0; i < lattice.size(); ++i)
{
    std::array<size_t, 3> x = lattice.coordinate(i);
    GTL_RUNTIME_ASSERT(
        x[0] == (i % 2) && x[1] == ((i / 2) % 3) && x[2] == ((i / 2) / 3),
        "Invalid coordinate.");
}

// This verification code depends on storage order.
Lattice<false, 2, 3, 5> lattice{};
// Lattice<false> lattice{2, 3, 5};
for (size_t i = 0; i < lattice.size(); ++i)
{
    std::array<size_t, 3> x = lattice.coordinate(i);
    GTL_RUNTIME_ASSERT(
        x[2] == (i % 5) &&
        x[1] == ((i / 5) % 3) &&
        x[0] == ((i / 5) / 3),
        "Invalid coordinate.");
}

```

---

## 8.2 Multidimensional Arrays

The Lattice class is responsible for multindexing into a contiguous chunk of memory. The Multiarray class is responsible for providing the memory. It derives from Lattice. The interface for compile-time-known bounds is shown in Listing 22.

**Listing 22.** The interface of class Multiarray for bounds known at compile time. It is similar to Container-Adapter but without the raw iterators.

```

template <typename T, bool OrderLtoR, size_t... Sizes>
class Multiarray : public Lattice<OrderLtoR, Sizes...>
{
public:
    // All elements of the multiarray are uninitialized for native data
    // but are initialized when the default T constructor initializes
    // its data.
    Multiarray();
    ~Multiarray() = default;

    // Copy semantics.
    Multiarray(Multiarray const& other);
    Multiarray& operator=(Multiarray const& other);

    // Move semantics.
    Multiarray(Multiarray&& other) noexcept;
    Multiarray& operator=(Multiarray&& other) noexcept;

    // Get a pointer to the array of elements.
    T const* data() const noexcept;
    T* data() noexcept;

    // Access the elements at the specified index.
    T const& at(size_t i) const;
    T& at(size_t i);
    T const& operator[](size_t i) const;
    T& operator[](size_t i);

```

```

// Set all elements to the specified value.
void fill(T const& value);

// Get the element corresponding to the n-dimensional parameter pack
// of indices.
template <typename... IndexTypes>
T const& operator()(IndexTypes... ntuple) const;

template <typename... IndexTypes>
T& operator()(IndexTypes... ntuple);

// Get the element corresponding to the n-dimensional coordinate.
T const& operator()(std::array<size_t, sizeof...(Sizes)> const& coordinate) const;
T& operator()(std::array<size_t, sizeof...(Sizes)> const& coordinate);

// Support for sorting and comparing Multiarray objects.
bool operator==(Multiarray const& other) const;
bool operator!=(Multiarray const& other) const;
bool operator<(Multiarray const& other) const;
bool operator<=(Multiarray const& other) const;
bool operator>(Multiarray const& other) const;
bool operator>=(Multiarray const& other) const;

private:
    std::array<T, Lattice<OrderLtoR, Sizes...>::NumElements> mContainer;
};

```

---

The Multiarray interface for runtime-known bounds is shown in Listing 23.

**Listing 23.** The interface of class Multiarray for bounds known at runtime. It is similar to ContainerAdapter but without the raw iterators.

```

template <typename T, bool OrderLtoR>
class Multiarray<T, OrderLtoR> : public Lattice<OrderLtoR>
{
public:
    // The multiarray has no elements.
    Multiarray();

    // The multiarray has the specified sizes and all elements are
    // uninitialized.
    Multiarray(std::vector<size_t> const& sizes);
    Multiarray(std::initializer_list<size_t> const& sizes);

    // Support for deferred construction where the initial multiarray is
    // created by the default constructor. During later execution, the
    // multiarray sizes can be set as needed.
    void reset(std::vector<size_t> const& sizes);
    void reset(std::initializer_list<size_t> const& sizes);

    ~Multiarray() = default;

    // Copy semantics.
    Multiarray(Multiarray const& other);
    Multiarray& operator=(Multiarray const& other);

    // Move semantics.
    Multiarray(Multiarray&& other) noexcept;
    Multiarray& operator=(Multiarray&& other) noexcept;

    // Get a pointer to the array of elements.
    T const* data() const noexcept;
    T* data() noexcept;

    // Access the elements at the specified index.
    T const& at(size_t i) const;

```

```

T& at(size_t i);
T const& operator[](size_t i) const;
T& operator[](size_t i);

// Set all elements to the specified value.
void fill(T const& value);

// Get the element corresponding to the n-dimensional tuple of
// indices.
template <typename... IndexTypes>
T const& operator()(IndexTypes... ntuple) const;

template <typename... IndexTypes>
T& operator()(IndexTypes... ntuple);

// Get the element corresponding to the n-dimensional coordinate.
T const& operator()(std::vector<size_t> const& coordinate) const;
T& operator()(std::vector<size_t> const& coordinate);

// Support for sorting and comparing Multiarray objects.
bool operator==(Multiarray const& other) const;
bool operator!=(Multiarray const& other) const;
bool operator<(Multiarray const& other) const;
bool operator<=(Multiarray const& other) const;
bool operator>(Multiarray const& other) const;
bool operator>=(Multiarray const& other) const;

private:
    std::vector<T> mContainer;
};

```

---

An example for using Multiarray is shown in Listing 24.

---

**Listing 24.** Several examples for using a 3-dimensional multiarray with bounds (2,3,5).

```

// This code is independent of storage order. It works for any of the
// declarations of a multiarray (the active one or the commented-out ones).
Multiarray<float, true, 2, 3, 5> multiarray{};
// Multiarray<float, false, 2, 3, 5> multiarray{};
// Multiarray<float, true> multiarray{2, 3, 5};
// Multiarray<float, false> multiarray{2, 3, 5};
for (size_t x2 = 0, i = 0; x2 < multiarray.size(2); ++x2)
{
    for (size_t x1 = 0; x1 < multiarray.size(1); ++x1)
    {
        for (size_t x0 = 0; x0 < multiarray.size(0); ++x0, ++i)
        {
            multiarray(x0, x1, x2) = static_cast<float>(i + 1);
        }
    }
}

for (size_t i = 0; i < multiarra.size(); ++i)
{
    multiarray[i] += 2.0f;
}

for (size_t x2 = 0, i = 0; x2 < multiarray.size(2); ++x2)
{
    for (size_t x1 = 0; x1 < multiarray.size(1); ++x1)
    {
        for (size_t x0 = 0; x0 < multiarray.size(0); ++x0, ++i)
        {
            GTL_RUNTIME_ASSERT(
                multiarray(x0, x1, x2) == static_cast<float>(i + 3),
                "invalid index");
        }
    }
}

```

```

        std::array<size_t, 3> x{ x0, x1, x2 };
        GTL_RUNTIME_ASSERT(
            multiarray(x) == static_cast<float>(i + 3),
            "invalid index");
    }
}

```

---

## 8.3 Multidimensional Array Adapters

Class `MultiarrayAdapter` has effectively the same interface as that of `Multiarray` except that the storage is provided from an external source by raw pointer. Because the source is external and the class shares the raw pointer, copy and move semantics are disabled. Listing 25 shows the interfaces for the classes. There is a template class for compile-time-known bounds and for runtime-known bounds.

---

**Listing 25.** The interfaces for `MultiarrayAdapter`.

```

// Implementation for multiarray adapters whose sizes are known at compile time.
template <typename T, bool OrderLtoR, size_t... Sizes>
class MultiarrayAdapter : public Lattice<OrderLtoR, Sizes...>
{
public:
    // All elements of the multiarray are uninitialized for native data
    // but are initialized when the default T constructor initializes
    // its data.
    MultiarrayAdapter(T* container = nullptr);

    // Support for deferred construction where the initial multiarray is
    // created by the default constructor whose input is the null pointer.
    // During later execution, the pointer can be set to point to an
    // actual block of memory.
    void reset(T* container);

    ~MultiarrayAdapter() = default;

    // Disallow copy semantics and move semantics.
    MultiarrayAdapter(MultiarrayAdapter const&) = delete;
    MultiarrayAdapter& operator=(MultiarrayAdapter const&) = delete;
    MultiarrayAdapter(MultiarrayAdapter&&) = delete;
    MultiarrayAdapter& operator=(MultiarrayAdapter&&) = delete;

    // Get a pointer to the array of elements.
    T const* data() const noexcept;
    T* data() noexcept;

    // Access the elements at the specified index.
    T const& at(size_t i) const;
    T& at(size_t i);
    T const& operator[](size_t i) const;
    T& operator[](size_t i);

    // Set all elements to the specified value.
    void fill(T const& value);

    // Get the element corresponding to the n-dimensional parameter pack
    // of indices.
    template <typename... IndexTypes>
    T const& operator()(IndexTypes... ntuple) const;

    template <typename... IndexTypes>
    T& operator()(IndexTypes... ntuple);

```

```

// Get the element corresponding to the n-dimensional coordinate.
T const& operator()(std::array<size_t, sizeof...(Sizes)> const& coordinate) const;
T& operator()(std::array<size_t, sizeof...(Sizes)> const& coordinate);

// Support for sorting and comparing Multiarray objects.
bool operator==(MultiarrayAdapter const& other) const;
bool operator!=(MultiarrayAdapter const& other) const;
bool operator<(MultiarrayAdapter const& other) const;
bool operator<=(MultiarrayAdapter const& other) const;
bool operator>(MultiarrayAdapter const& other) const;
bool operator>=(MultiarrayAdapter const& other) const;

private:
// The pointer must be to a block of memory storing
// Lattice<OrderLtoR, Sizes...>::size() T-objects.
T* mContainer;
};

// Implementation for multiarrays whose sizes are known only at runtime.
template <typename T, bool OrderLtoR>
class MultiarrayAdapter<T, OrderLtoR> : public Lattice<OrderLtoR>
{
public:
// The multiarray has no elements.
MultiarrayAdapter();

// The multiarray has the specified sizes and all elements are
// uninitialized.
MultiarrayAdapter(std::vector<size_t> const& sizes, T* container);

// Support for deferred construction where the initial multiarray is
// created by the default constructor whose input is the null pointer.
// During later execution, the lattice sizes can be set as needed.
void reset(std::vector<size_t> const& sizes, T* container);

~MultiarrayAdapter() = default;

// Disallow copy semantics and move semantics.
MultiarrayAdapter(MultiarrayAdapter const&) = delete;
MultiarrayAdapter& operator=(MultiarrayAdapter const&) = delete;
MultiarrayAdapter(MultiarrayAdapter&&) = delete;
MultiarrayAdapter& operator=(MultiarrayAdapter&&) = delete;

// Get a pointer to the array of elements.
T const* data() const noexcept;
T* data() noexcept;

// Access the elements at the specified index.
T const& at(size_t i) const;
T& at(size_t i);
T const& operator[](size_t i) const;
T& operator[](size_t i);

// Set all elements to the specified value.
void fill(T const& value);

// Get the element corresponding to the n-dimensional tuple of
// indices.
template <typename... IndexTypes>
T const& operator()(IndexTypes... ntuple) const;

template <typename... IndexTypes>
T& operator()(IndexTypes... ntuple);

// Get the element corresponding to the n-dimensional coordinate.
T const& operator()(std::vector<size_t> const& coordinate) const;

T& operator()(std::vector<size_t> const& coordinate);

// Support for sorting and comparing MultiarrayAdapter objects.
bool operator==(MultiarrayAdapter const& other) const;
bool operator!=(MultiarrayAdapter const& other) const;

```

```

bool operator< (MultiarrayAdapter const& other) const;
bool operator<=(MultiarrayAdapter const& other) const;
bool operator> (MultiarrayAdapter const& other) const;
bool operator>=(MultiarrayAdapter const& other) const;

private:
    // The pointer must be to a block of memory storing
    // Lattice<OrderLtoR>::size() T-objects.
    T* mContainer;
};

```

---

Listing 26 shows an example for using MultiarrayAdapter.

---

**Listing 26.** An example for using MultiarrayAdapter to interpret a raw pointer as storage for a 3-dimensional multiarray.

```

MultiarrayAdapter<int32_t, true, 2, 3, 5> multiarray{}; // null storage

std::array<int32_t, 30> storage{};
for (size_t i = 0; i < storage.size(); ++i)
{
    storage[i] = static_cast<int32_t>(i + 1);
}

multiarray.reset(storage.data());

for (size_t x2 = 0, i = 0; x2 < multiarray.size(2); ++x2)
{
    for (size_t x1 = 0; x1 < multiarray.size(1); ++x1)
    {
        for (size_t x0 = 0; x0 < multiarray.size(0); ++x0, ++i)
        {
            GTL_RUNTIME_ASSERT(
                multiarray(x0, x1, x2) == static_cast<int32_t>(i + 1);
                "invalid index");

            std::array<size_t, 3> x{ x0, x1, x2 };
            GTL_RUNTIME_ASSERT(
                multiarray(x) == static_cast<int32_t>(i + 1),
                "invalid index");
        }
    }
}

```

---

## Chapter 9

# MinHeap

[GTL/Utility/MinHeap.h](#)

A min-heap is a binary tree whose nodes have weights and with the constraint that the weight of a parent node is less than or equal to the weights of its children. This data structure may be used as a priority queue. If the `std::priority_queue` interface suffices for your needs, use that instead. The priority queue allows you to insert new weights into the queue and to remove the minimum weight from the queue. It does not allow write-access to weights already in the queue. For some geometric algorithms, that interface is insufficient for optimal performance.

For example, consider a polyline of vertices  $\{\mathbf{V}_i\}_{i=0}^{n-1}$  for which the segments are  $\langle \mathbf{V}_j, \mathbf{V}_{j+1} \rangle$ . A small number of vertices are to be removed to smooth the polyline, each vertex nearly collinear with its neighbors. When a vertex is removed, the two segments sharing it are removed from the polyline and the vertex's neighbors are considered to be the endpoints of a new segment. Various weight functions can be defined that provide measures of how close a triple of vertices  $\langle \mathbf{V}_{i-1}, \mathbf{V}_i, \mathbf{V}_{i+1} \rangle$  is to being collinear. The function can incorporate angles formed by the segments shared by  $\mathbf{V}_i$  or distance from  $\mathbf{V}_i$  to the segment connecting neighboring points. It can also try to include geometric information to make the measurement invariant to scale. For the discussion here, the function choice is not relevant. Define  $w_i \geq 0$  to be the weight for the aforementioned triple of vertices. If  $w_i = 0$ , then the vertices are collinear. The weights are  $\{w_i\}_{i=0}^{n-1}$ . For an open polyline, the endpoints are typically chosen to be immovable, so  $w_0 = w_{n-1} = \infty$ . For a closed polyline, an additional segment is represented by  $\langle \mathbf{V}_{n-1}, \mathbf{V}_0 \rangle$ . The weight  $w_0$  is associated with the triple  $\langle \mathbf{V}_{n-1}, \mathbf{V}_0, \mathbf{V}_1 \rangle$  and the weight  $w_{n-1}$  is associated with the triple  $\langle \mathbf{V}_{n-2}, \mathbf{V}_{n-1}, \mathbf{V}_0 \rangle$ .

If the template type `T` for the `std::priority_queue<T>` object is chosen to be a floating-point type, say, `double`, a weight that occurs for two vertices will appear twice in the queue. When that weight is the minimum and removed from the queue, the index of the vertex associated with it is not known. Instead choose `T` to be `std::pair<double, size_t>`, where `size_t` is the type of the indices. The comparison `std::less<std::pair<double, size_t>>` is used so that the priority queue will return the minimum weight from a `top()` call and remove that element on a `pop()` call.

The polyline smoothing can be attained by inserting the weight-index pairs into the priority queue. The vertices can be removed from the polyline, one vertex at a time that corresponds to the minimum weight in the priority queue. The problem with this approach is that when a vertex is removed, each of its neighbors

are part of a new triple of vertices. The new weights of those neighbors are usually different from the old weights, but only the old weights are in the priority queue.

For example, consider the polyline with 5 vertices  $\mathbf{V}_0 = (0, 2)$ ,  $\mathbf{V}_1 = (1, 0.9)$ ,  $\mathbf{V}_2 = (2, 0)$ ,  $\mathbf{V}_3 = (3, 0)$  and  $\mathbf{V}_4 = (4, 0.95)$ . Let the weight function be  $\cos(\theta_i)$ , where  $\theta_i \in [0, \pi]$  is the angle between  $\mathbf{V}_{i-1} - \mathbf{V}_i$  and  $\mathbf{V}_{i+1} - \mathbf{V}_i$ . The endpoints of the polyline are immovable, so only the weights of the other 4 points are relevant. The weights  $w_1$ ,  $w_2$  and  $w_3$  are computed using

$$w_i = \frac{\mathbf{V}_{i-1} - \mathbf{V}_i}{|\mathbf{V}_{i-1} - \mathbf{V}_i|} \cdot \frac{\mathbf{V}_{i+1} - \mathbf{V}_i}{|\mathbf{V}_{i+1} - \mathbf{V}_i|} \quad (9.0.1)$$

The weights are  $w_1 \doteq -0.994988$ ,  $w_2 \doteq -0.743294$  and  $w_3 \doteq -0.724999$ . If inserted into a priority queue and then removed one at a time, the order of vertex removal is  $\mathbf{V}_1$ ,  $\mathbf{V}_2$  and  $\mathbf{V}_3$ . However, consider that  $\mathbf{V}_1$  is removed first, so the smoothed polyline now has vertices  $\mathbf{V}_0$ ,  $\mathbf{V}_2$  and  $\mathbf{V}_3$ . The angle at  $\mathbf{V}_2$  is different from that before the removal. If the angle is now computed, it is

$$w_2 = \frac{\mathbf{V}_0 - \mathbf{V}_2}{|\mathbf{V}_0 - \mathbf{V}_2|} \cdot \frac{\mathbf{V}_3 - \mathbf{V}_2}{|\mathbf{V}_3 - \mathbf{V}_2|} \doteq -0.707107 \quad (9.0.2)$$

If we were able to update the priority-queue weight of  $\mathbf{V}_2$  from  $-0.743294$  to  $-0.707107$ , the new weight is larger than  $w_3$ , so in fact we would want to remove  $\mathbf{V}_3$  from the polyline instead of  $\mathbf{V}_2$ .

It is desirable to modify the old weights and then re-sort to restore the data structure to a priority queue, but `std::priority_queue` does not support this. GTL provides a min-heap class called `MinHeap`. The min-heap is represented as a complete binary tree that allows for modification of an element at any node of the tree. After weight modification, the tree itself must be updated so that it represents a min-heap. Listing 27 shows the class interface for `MinHeap`.

---

**Listing 27.** The class interface for `MinHeap<T>`. The type `T` must have a less-than comparison operator `operator< (...)`. All comparisons in `MinHeap<T>` are implemented in terms of this operator, so you do not need to provide others. The class is defined in the `gtl` namespace.

```
template <typename T>
class MinHeap
{
public:
    static size_t constexpr invalid = std::numeric_limits<size_t>::max();

    // The node weight values are default constructed. For native T,
    // the weights are uninitialized. For non-native T, the weights are
    // whatever the default constructor creates.
    MinHeap(size_t maxElements = 0);

    // Copy semantics.
    MinHeap(MinHeap const& other) = default;
    MinHeap& operator=(MinHeap const& other) = default;

    // Move semantics.
    MinHeap(MinHeap&& other) = default;
    MinHeap& operator=(MinHeap&& other) = default;

    // Resize the min-heap so that it has the specified maximum number of
    // elements. The previous state of the min-heap is not preserved. The
    // node weights values are default constructed. For native T, the
    // weights are uninitialized. For non-native T, the weights are
    // whatever the default constructor creates.
    void Reset(size_t maxElements);
```



```

// Get the maximum number of elements allowed in the min-heap.
size_t GetMaxElements() const;

// Get the current number of elements in the min-heap. This number
// is in the range {0..maxElements}.
size_t GetNumElements() const;

// Get the root of the min-heap. The function reads the root but does
// not remove the element from the min-heap. The function return is
// 'key'. If the min-heap is not empty, the 'key' is valid, 'weight'
// corresponds to the root of the min-heap, and 'handle' is the
// user-provided identifier for the corresponding application object.
// If the min-heap is empty, 'key' is invalid and the operation is
// unsuccessful, in which case both 'handle' and 'weight' are invalid
// and should not be used.
size_t GetMinimum(size_t& handle, T& weight) const;

// Insert (handle,weight) into the min-heap. The function return is
// 'key'. If the min-heap is not full before the insertion, 'key' is
// valid and (handle,weight) is stored in the appropriate node. If the
// min-heap is full before the insertion, 'key' is invalid, the
// operation is unsuccessful, and the min-heap is not modified. When
// the insertion is successful, the 'key' may be used later in a call
// to Update.
size_t Insert(size_t handle, const& weight);

// Remove the root of the min-heap, which contains the minimum weight.
// The function return is 'key'. If the min-heap is not empty before
// the removal, 'key' is valid and corresponds to the node storing
// (handle, weight). If the min-heap is empty before the removal,
// 'key' is invalid, the operation is unsuccessful, and the min-heap
// is not modified. On return, 'key' is valid until another insert,
// remove or update call is made. The intent is for the caller to use
// 'key' and, if necessary, clean up any resources that are associated
// with 'key' before any additional heap-modifying calls.
size_t Remove(size_t& handle, T& weight);

// The value of a min-heap node must be modified via this function
// call. The side effect is that the binary tree is restored to a
// min-heap. The input 'updateKey' should be a key returned by some
// call to "key = Insert(handle, oldWeight)". The input 'updateWeight'
// is the new value to be associated with that key (and handle). The
// function return is 'updateKey' when it is valid; that is, it is
// required that 0 <= updateKey < GetMaxElements(), and internally
// 0 <= updateIndex < GetNumElements(). The function returns 'true'
// if and only if the update key is in the required range, in which
// case the min-heap was not modified.
bool Update(size_t updateKey, T const& updateWeight);

// Support for debugging. The functions test whether the data
// structure is a valid min-heap.
bool IsValid() const;

// The user-specified information stored in the binary tree nodes.
// The first of the pair is a handle for the application object
// involved in the min-heap operation. The second of the pair is
// the weight associated with that object.
struct Node
{
    Node();
    Node(size_t inHandle, T const& inWeight);
    size_t handle;
    T weight;
};

inline std::vector<Node> const& GetNodes() const;
inline Node const& GetNode(size_t key) const;
inline size_t GetHandle(size_t key) const;
inline T const& GetWeight(size_t key) const;

private:
    size_t mNumElements;

```

```

    std::vector<size_t> mKeys, mIndices;
    std::vector<Node> mNodes;
};

```

---

The node values are stored in `mNodes[]`, the ordering based on the order of `Insert` and `Remove` calls. The min-heap is stored as a binary tree `mKeys[]` where the array elements are lookups into `mNodes[]`. In the following discussion `index`, `key`, `handle` and `weight` refer to the following lookups: `key = mKeys[index]`, `handle = mNodes[key].handle`, `weight = mNodes[key].weight` and `index = mIndices[key]`. The array `mKeys[]` is a permutation of indices  $(0, \dots, N - 1)$ , where  $N$  is the maximum number of nodes in the binary tree. The array `mIndices[]` is the inverse of that permutation.

The `handle` value is a user-defined quantity that associates the min-heap node with an application-specific object. Typically, the handle is an index into an array of objects (such as vertices of a polygon).

The root of the binary tree has parameters `index = 0`, `key = mKeys[0]`, `handle = mNodes[mKeys[0]].handle`, `value = mNodes[mKeys[0]].weight` and `index = mIndices[mKeys[0]]`. The left child of the root has parameters `index = 1`, `key = mKeys[1]`, `handle = mNodes[mKeys[1]].handle`, `weight = mNodes[mKeys[1]].weight` and `index = mIndices[mKeys[1]]`. The right child of the root has parameters `index = 2`, `key = mKeys[2]`, `handle = mNodes[mKeys[2]].handle`, `weight = mNodes[mKeys[2]].weight` and `index = mIndices[mKeys[2]]`. Generally, let a node of the binary tree have index  $p$ . The left child of the node has index  $2p + 1$  and the right child of the node has index  $2p + 2$ . If a node of the binary tree has index  $c$ , the parent node has index  $(c - 1)/2$ .

The design has two goals. First, the `mNodes[]` elements are never copied internally in the algorithm during insertion or removal. The `mKeys[]` elements act as a level of indirection for sorting the values. This is useful for performance when copying objects of type `T` is expensive. Second, the `mIndices[]` elements support the `Update` function, where a node's value can be modified and the binary tree restored to a min-heap. The `Insert` function returns a key that the caller manages during the min-heap lifetime. The value associated with the key can be modified and a call made to `Update` using the key and new weight. The min-heap becomes invalid on the weight change, but internally `Update` restores the binary tree to a min-heap. To do so requires looking up the index from the input key. The update capability makes `MinHeap` different from a priority queue. A priority queue allows you to read and remove the root of the heap, but it does not allow you to modify non-root values and then restore the data structure to a heap.

The constructor `MinHeap(size_t)` requires you to specify the maximum number of elements in the min-heap. This is typically known for many geometric algorithms, but if you are uncertain about the exact size required, you can choose a number sufficiently large. The `mNodes[]`.`weight` elements are uninitialized for native types chosen for `T`. If `T` is a class or struct, the default constructor of `T` is used to set the `mNodes[]`.`weight` elements.

The copy semantics (copy constructor and assignment operator) are the default ones. The `mNumElements` is copied and the array members are deep-copied.

The move semantics (move constructor and move operator) are the default ones. The `mNumElements` is copied and the other array members are moved to this object.

The `Reset` function resizes the min-heap so that it has the specified maximum number of elements. The previous state of the min-heap is not preserved. That is, the keys and indices are reset to the identity permutations.

The `GetMaxElements` and `GetNumElements` functions are simple accessors. The first function returns the maximum number of elements specified by the user in the constructor. The second function returns the current number of elements in the min-heap.

The function `GetMinimum` reads the minimum weight from the min-heap, which is the `weight` stored at the root, and it also reads the associated user-defined handle. The min-heap is not modified by this call. The function return is `key`. If the min-heap is not empty, `mNodes[key]` corresponds to the root of the min-heap. If the min-heap is empty, `key` is the maximum of type `size_t`. In this case, `weight` is undefined.

The function `Insert` is used to insert a new element into the min-heap. The `(handle,weight)` is inserted into the first unoccupied leaf node and then bubbled toward the root of the tree by swaps, stopping when the tree becomes a min-heap. The function return is `key`. If the min-heap is not full before the insertion, `mNodes[key]` stores the `(handle,weight)` pair. If the min-heap is full before the insertion, `key` is the maximum of type `size_t`. In this case, the min-heap is not modified. When the insertion is successful, the `key` may be used later in a call to `Update`.

The function `Remove` is used to remove the `(handle,weight)` pair from the root of the min-heap. The function return is `key`. If the min-heap is not empty before the removal, `key` is smaller than the maximum number of type `size_t`, which indicates the operation is successful. If the min-heap is empty before the removal, `key` is the maximum of type `size_t`. In this case, the min-heap is not modified. On return, `key` is valid until another insert, remove or update call is made. The intent is for the caller to use `key` and, if necessary, clean up any resources that are associated with `key` before any additional heap-modifying calls.

The function `Update` allows the weight of a min-heap node to be modified. After modification, it is possible that the tree no longer represents a min-heap. However, the function bubbles the new value toward the root or toward the leaves as necessary to restore the tree to a min-heap. The input `updateKey` should be a key returned by a call to `Insert`, say `key = Insert(handle, oldWeight)`, that is associated with the old weight. The input `updateWeight` is the new value to be associated with that key. The function return is `true` if and only if the `updateKey` is in the range `{0..GetMaxElements()-1}` and the internal number `updateIndex` is in the range `{0..GetNumElements()-1}`.

The function `IsValid` is used for debugging and verifies that the current `MinHeap` object is indeed a min-heap. The functions `GetNodes`, `GetNode`, `GetHandle` and `GetWeight` are used for debugging, but might be useful in applications.

Listing 28 illustrates the use of min-heap for the polyline smoothing algorithm. The `ComputeWeight` function is chosen to guide which vertices are removed first from the polyline. The general framework is independent of the implementation of `ComputeWeight` but the order of removed vertices is dependent on it.

---

**Listing 28.** A simple illustration for using the min-heap data structure to smooth a  $D$ -dimensional closed polyline.

```
// The data structure for a doubly linked list of vertices representing a simple polygon. The positions
// are stored separately. The key is the location of the vertex (handle,weight) in the min-heap.
struct Vertex
{
    size_t key;
    size_t prev, curr, next;
};

// Distance from a point to a line segment (end0,end1).
double Distance(Vector<double, 2> const& point, Vector<double, 2> const& end0,
    Vector<double, 2> const& end1)
{
    Vector<double, 2> direction = end1 - end0;
    Vector<double, 2> diff = point - end1;
    double t = Dot(direction, diff);
    if (t >= 0.0)
    {
        return Length(diff);
    }
}
```

```

    }

    diff = point - end0;
    t = Dot(direction, diff);
    if (t <= 0.0)
    {
        return Length(diff);
    }

    Vector<double, 2> closest = end0 + (t / Dot(direction, direction)) * direction;
    diff -= (t / Dot(direction, direction)) * direction;
    return Length(diff);
}

// The weight function for the min-heap.
double ComputeWeight(std::vector<Vector<double, 2>> const& positions, Vertex const& vertex)
{
    // The position of the vertex whose weight must be modified.
    Vector<double, 2> posCurr = positions[vertex.curr];

    // The positions of the neighboring vertices.
    Vector<double, 2> posPrev = positions[vertex.prev];
    Vector<double, 2> posNext = positions[vertex.next];

    // Implement your favorite weight function. For example, a scale-invariant
    // weight function is shown here.
    double distance = Distance(posCurr, posPrev, posNext);
    double length = Length(posNext - posPrev);
    double weight = distance / length;
    return weight;
}

void PolygonDecimation()
{
    size_t const N = 8;
    std::vector<double> angles =
    {
        0.0,
        1.3093002290045481,
        1.5752296219476425,
        1.7393568117223051,
        2.1773350778399365,
        3.3789492594255890,
        4.8255294939361066,
        5.4983791260200432
    };

    // The positions for the vertices, assumed to be initialized before
    // the smoothing.
    std::vector<Vector<double, 2>> positions(N);
    for (size_t i = 0; i < N; ++i)
    {
        positions[i] = { std::cos(angles[i]), std::sin(angles[i]) };
    }

    // The abstraction of the vertex weight computations.

    // Create the closed polyline doubly linked list. The vertex members curr
    // and key are the same, because the vertices are inserted into the min-heap
    // in natural order. Generally, if insertions and removals for a polyline are
    // interleaved, the members will be different.
    size_t constexpr invalid = std::numeric_limits<size_t>::max();
    std::vector<Vertex> vertices(N);
    vertices[0] = { invalid, N - 1, 0, 1 };
    for (size_t i = 1; i < N - 1; ++i)
    {
        vertices[i] = { invalid, i - 1, i, i + 1 };
    }
    vertices[N - 1] = { invalid, N - 2, N - 1, 0 };

    // Initialize the min-heap.
    MinHeap<double> minHeap(N);

```

```

for (size_t i = 0; i < N; ++i)
{
    vertices[i].key = minHeap.Insert(i, ComputeWeight(positions, vertices[i]));
    GTL_RUNTIME_ASSERT(
        vertices[i].key != minHeap.invalid,
        "Expecting Insert to succeed.");

    // The vertices are inserted in natural order with no interleaved
    // removals.
    GTL_RUNTIME_ASSERT(
        vertices[i].key == i,
        "Mismatch of key and handle.");

    // NOTE: It is possible to provide handles that are not the
    // consecutive indices into the array of vertices. Generally, the
    // key and the handle can be different.
}

// Smooth by removing one vertex at a time until the final polyline is
// a triangle. The minHeap before the decimation is:
// minHeap.numElements = 8
// minHeap.keys = { 2, 3, 1, 7, 4, 5, 6, 0 }
// minHeap.indices = { 7, 2, 0, 1, 4, 5, 6, 3 }
// minHeap.nodes =
// {
//     (0, 0.26888569665111983),
//     (1, 0.11390513218138611),
//     (2, 0.050930178636699894),
//     (3, 0.060054612309021399),
//     (4, 0.16800021721713373),
//     (5, 0.38583356427562143),
//     (6, 0.25049729962792699),
//     (7, 0.18954933023987491)
// }
size_t handle = invalid;
double weight = -std::numeric_limits<double>::max();
while (minHeap.GetNumElements() > 3)
{
    // Remove the root node of the min-heap. This node represents the
    // vertex of minimum weight. The weight is not needed in this simple
    // illustration, so it is not consumed later in the code.
    size_t key = minHeap.Remove(handle, weight);
    GTL_RUNTIME_ASSERT(
        key != minHeap.invalid,
        "Expecting Remove to succeed.");

    // This is the vertex of minimum weight.
    Vertex& vertexCurr = vertices[handle];

    // Remove the vertex from the doubly linked list.
    Vertex& vertexPrev = vertices[vertexCurr.prev];
    Vertex& vertexNext = vertices[vertexCurr.next];
    vertexPrev.next = vertexCurr.next;
    vertexNext.prev = vertexCurr.prev;
    vertexCurr.key = invalid;
    vertexCurr.prev = invalid;
    vertexCurr.next = invalid;

    // Update the neighbors' weights in the min-heap. The
    bool updated;
    updated = minHeap.Update(vertexPrev.key,
        ComputeWeight(positions, vertexPrev));
    GTL_RUNTIME_ASSERT(
        updated,
        "Expecting Update to succeed.");

    updated = minHeap.Update(vertexNext.key,
        ComputeWeight(positions, vertexNext));
    GTL_RUNTIME_ASSERT(
        updated,
        "Expecting Update to succeed.");
}

```

```

// Iteration 1.
// minHeap.numElements = 7
// minHeap.keys = { 3, 4, 1, 7, 0, 5, 6, 2 }
// minHeap.indices = { 4, 2, 7, 0, 1, 5, 6, 3 }
// minHeap.nodes =
// {
//     (0, 0.26888569665111983),
//     (1, 0.17002554753747984), // modified
//     (2, removed),
//     (3, 0.11023099461064528), // modified
//     (4, 0.16800021721713373),
//     (5, 0.38583356427562143),
//     (6, 0.25049729962792699),
//     (7, 0.18954933023987491)
// }

// Iteration 2.
// minHeap.numElements = 6
// minHeap.keys = { 7, 6, 1, 4, 0, 5, 3, 2 }
// minHeap.indices = { 4, 2, 7, 6, 3, 5, 1, 0 }
// minHeap.nodes =
// {
//     (0, 0.26888569665111983),
//     (1, 0.28898821550388476), // modified
//     (2, *),
//     (3, removed),
//     (4, 0.27649515486339993), // modified
//     (5, 0.38583356427562143),
//     (6, 0.25049729962792699),
//     (7, 0.18954933023987491)
// }

// Iteration 3.
// minHeap.numElements = 5
// minHeap.keys = { 4, 5, 1, 6, 0, 7, 3, 2 }
// minHeap.indices = { 4, 2, 7, 6, 0, 1, 3, 5 }
// minHeap.nodes =
// {
//     (0, 0.41273251413336454), // modified
//     (1, 0.28898821550388476),
//     (2, *),
//     (3, *),
//     (4, 0.27649515486339993),
//     (5, 0.38583356427562143),
//     (6, 0.44391487866790591), // modified
//     (7, removed)
// }

// Iteration 4.
// minHeap.numElements = 4
// minHeap.keys = { 0, 6, 5, 1, 4, 7, 3, 2 }
// minHeap.indices = { 0, 3, 7, 6, 4, 2, 1, 5 }
// minHeap.nodes =
// {
//     (0, 0.41273251413336454),
//     (1, 0.52720830925252427), // modified
//     (2, *),
//     (3, *),
//     (4, removed),
//     (5, 0.57917600588083229), // modified
//     (6, 0.44391487866790591),
//     (7, *)
// }

// Iteration 5.
// minHeap.numElements = 3
// minHeap.keys = { 5, 6, 1, 0, 4, 7, 3, 2 }
// minHeap.indices = { 3, 2, 7, 6, 4, 0, 1, 5 }
// minHeap.nodes =
// {
//     (0, removed),

```

```
//      (1, 1.2763118300752101), // modified
//      (2, *),
//      (3, *),
//      (4, *),
//      (5, 0.57917600588083229),
//      (6, 0.75633580064878481), // modified
//      (7, *)
//  }

// Iteration 6.
// minHeap.numElements = 3
// minHeap.keys = { 5, 6, 1, 0, 4, 7, 3, 2 }
// minHeap.indices = { 3, 2, 7, 6, 4, 0, 1, 5 }
// minHeap.nodes =
// {
//     (0, removed),
//     (1, 1.2763118300752101), // modified
//     (2, *),
//     (3, *),
//     (4, *),
//     (5, 0.57917600588083229),
//     (6, 0.75633580064878481), // modified
//     (7, *)
// }

// The loop exits because the decimated polygon is a triangle
// with vertices whose positions are position[1], position[5] and
// position[6].
}
```

---





## Chapter 10

# Timer

[GTL/Utility/Timer.h](#)

The Timer is a simple wrapper for a 64-bit clock using `std::chrono`. It hides some of the verbose steps in measuring the time. The interface is shown in Listing 29. An example for using the timer is also shown in the listing.

---

**Listing 29.** The interface for class Timer and an example for using the class.

```
class Timer
{
public:
    // Construction of a high-resolution timer (64-bit).
    Timer();

    // Get the current time relative to the initial time.
    int64_t GetNanoseconds() const;
    int64_t GetMicroseconds() const;
    int64_t GetMilliseconds() const;
    double GetSeconds() const;

    // Reset so that the current time is the initial time.
    void Reset();
};

#include <GTL/Utility/Timer.h>
#include <iostream>

void ExampleTiming()
{
    Timer timer{};
    <Do some computing here.>
    int64_t milliseconds = timer.GetMilliseconds();
    std::cout << "Compute time is " << milliseconds << " milliseconds." << std::endl;

    timer.Reset();
    <Do some more computing here.>
    milliseconds = timer.GetMilliseconds();
    std::cout << "Compute time is " << milliseconds << " milliseconds." << std::endl;
}
```

---



# Bibliography

- [HZ03] Timothy C. Hoad and Justin Zobel. Methods for Identifying Versioned and Plagiarised Documents. 54(3), February 2003.
- [Jos12] Nicolai M. Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley, an imprint of Pearson PLC, Upper Saddle River, NJ, 2nd edition, 2012.
- [Lib08] The Boost Library. Combining hash values.  
[https://www.boost.org/doc/libs/1\\_75\\_0/doc/html/hash/combine.html](https://www.boost.org/doc/libs/1_75_0/doc/html/hash/combine.html), 2008.