

Department I - C Plus Plus

Modern and Lucid C++ Advanced for Professional Programmers

Week 5 – Iterators & Tags

Thomas Corbat / Felix Morgner
Rapperswil / St. Gallen, 21.03.2024
FS2024



- **Topics:**

- Recap Week 4
- Tags for Dispatching
- Iterators

- **Participants should...**

- ... understand the purpose of tag types
- ... be able to explain how tag types can be used to control overload resolution of a function template
- ... be able to implement their own iterator types
- ... be able to facilitate boost to reduce boilerplate code for implementing iterator

Recap Week 4



- **Explicit heap memory allocation**

- new expression

- **Syntax**

```
new <type> <initializer>
```

- **Allocates memory for an instance of <type>**

- **Returns a pointer to the object or array created (on the heap)**

- of type <type> *

- **The arguments in the <initializer> are passed to the constructor of <type>**

```
struct Point {  
    Point(int x, int y) :  
        x {x}, y{y}{}  
    int x, y;  
};  
  
auto createPoint(int x, int y) -> Point* {  
    return new Point{x, y}; //constructor  
}  
  
auto createCorners(int x, int y) -> Point* {  
    return new Point[2]{{0, 0}, {x, y}};  
}
```

- **Explicit heap memory deallocation**

- delete expression

- **Syntax**

`delete <pointer>`

- **Deallocates the memory (of a single object) pointed to by the <pointer>**

- **Calls the Destructor of the destroyed type**

- **delete nullptr is well defined**

- it does nothing

- **Deleting the same object twice is Undefined Behavior!**

```
struct Point {  
    Point(int x, int y) :  
        x {x}, y{y} {}  
    int x, y;  
}  
  
auto funWithPoint(int x, int y) -> void {  
    Point * pp = new Point{x, y};  
    //pp member access with pp->  
    //pp is the pointer value  
    delete pp; //destructor  
}
```

DANGER

Double
Delete



- **Simple rules**

- Delete every object you allocated
- Do not delete an object twice
- Do not access a deleted object

- **Just don't do the following**

```
auto foo() -> void {  
    int * ip = new int{5};  
    //exit without deleting  
    //location ip points to  
}
```



```
auto foo() -> void {  
    int * ip = new int{5};  
    delete ip;  
    delete ip;  
}
```



```
auto foo() -> void {  
    int * ip = new int{5};  
    delete ip;  
    int dead = *ip;  
}
```



Tags for Dispatching



- **Template parameters don't require a specified type hierarchy**

- but they expect an argument to satisfy a concept

- **Example**

```
template <typename Type>  
auto max(Type value1, Type value2) -> Type {  
    return value1 < value2 ? value2 : value1;  
}
```

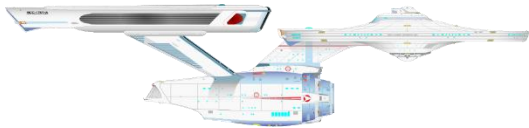
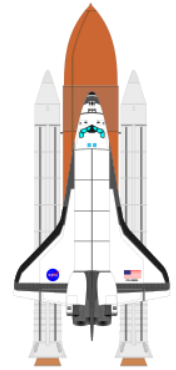
- Operator < must be available to compare Type objects and the result must be convertible to bool
- Type must be (move- or copy-)constructible
- int or std::string would satisfy the concept of Type

- If the same operation can be implemented more/less efficiently depending on the capabilities of the argument, tags can be used to find the "best" implementation
- Let's have a look at a hypothetical example
 - Different types of space ships have different means of travel (API)

```
template <typename SpaceShip>  
auto travelTo(Galaxy destination, SpaceShip& ship) -> void {  
    ship.$functionUsedToTravel$(destination);  
}
```

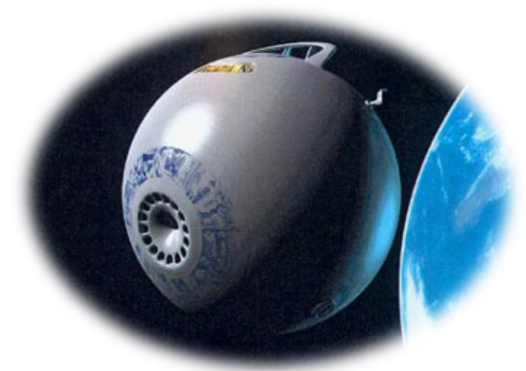


```
struct MultiPurposeCrewVehicle {  
    auto travelThroughSpace(Galaxy destination) -> void;  
};
```



```
struct GalaxyClassShip {  
    auto travelThroughSpace(Galaxy destination) -> void;  
    auto travelThroughHyperspace(Galaxy destination) -> void;  
};
```

```
struct HeartOfGold {  
    auto travelThroughSpace(Galaxy destination) -> void;  
    Galaxy travelImprobably();  
};
```



- For all space ships `travelThroughSpace()` would work

```
struct MultiPurposeCrewVehicle {  
    auto travelThroughSpace(Galaxy destination) -> void;  
};
```

```
template <typename SpaceShip>  
auto travelTo(Galaxy destination, SpaceShip& ship) -> void {  
    ship.travelThroughSpace(destination);  
}
```

- Some space ships might have faster means of travel, which we would like to use!
- However, we cannot implement the same generic template twice with the same parameters.

```
struct GalaxyClassShip {  
    auto travelThroughSpace(Galaxy destination) -> void ;  
    auto travelThroughHyperspace(Galaxy destination) -> void;  
};
```

```
template <typename SpaceShip>  
auto travelTo(Galaxy dest,  
              SpaceShip& ship) -> void {  
    ship.travelThroughSpace(dest);  
}
```

```
template <typename SpaceShip>  
auto travelTo(Galaxy dest,  
              SpaceShip& ship) -> void {  
    ship.travelThroughHyperspace(dest);  
}
```

```
template <typename Spaceship>
auto travelTo(Galaxy destination, Spaceship& ship, A) -> void {
    ship.travelThroughSpace(destination);
}

template <typename Spaceship>
auto travelTo(Galaxy destination, Spaceship& ship, B) -> void {
    ship.travelThroughHyperspace(destination);
}

template <typename Spaceship>
auto travelTo(Galaxy destination, Spaceship& ship, C) -> void {
    while(destination != ship.travelImprobably());
}
```

- Different signatures are required to provide multiple overloads for arbitrary space ships.
- We would like to hide that in the actual call.

```
//Provides travelThroughSpace
struct SpaceDriveTag{};

//Provides travelThroughSpace and travelThroughHyperspace
struct HyperspaceDriveTag : SpaceDriveTag{};

//Provides travelThroughSpace and travelImprobably
struct InfiniteProbabilityDriveTag : SpaceDriveTag{};
```

- Tag types are used mark capabilities of associated types
- Such tag types do not contain any members
- It is possible to derive tag types from each other to "inherit" the capabilities

```
struct SpaceDriveTag {};  
  
struct HyperspaceDriveTag : SpaceDriveTag {};  
  
struct InfiniteProbabilityDriveTag : SpaceDriveTag {};
```

```
struct MultiPurposeCrewVehicle;  
  
struct GalaxyClassShip;  
  
struct HeartOfGoldPrototype;
```

- **Approach 1: Derive space ship from the associated tag type**

```
struct MultiPurposeCrewVehicle : SpaceDriveTag{};  
  
struct GalaxyClassShip : HyperDriveTag{};  
  
struct HeartOfGoldPrototype : InfiniteProbabilityDriveTag{};
```

- This is **not** applicable for all types (e.g. for primitive types)
- This is **not** extensible (i.e. you cannot specify new kinds of tags as a user of the API)


```
struct SpaceDriveTag {};  
  
struct HyperspaceDriveTag : SpaceDriveTag {};  
  
struct InfiniteProbabilityDriveTag : SpaceDriveTag {};
```

```
struct MultiPurposeCrewVehicle;  
  
struct GalaxyClassShip;  
  
struct HeartOfGoldPrototype;
```

● Approach 2: SpaceshipTraits template

- Define a template for the default case
- Specialize the template for specific cases

```
template <typename>  
struct SpaceshipTraits {  
    using Drive = SpaceDriveTag;  
};
```

```
template <>  
struct SpaceshipTraits<GalaxyClassShip> {  
    using Drive = HyperspaceDriveTag;  
};
```

```
template <typename Spaceship>
auto travelToDispatched(Galaxy destination, Spaceship& ship, SpaceDriveTag) -> void {
    ship.travelThroughSpace(destination);
}

template <typename Spaceship>
auto travelToDispatched(Galaxy destination, Spaceship& ship, InfniteProbabilityDriveTag) -> void {
    while(destination != ship.travelImprobably());
}

template <typename Spaceship>
auto travelTo(Galaxy destination, Spaceship& ship) -> void {
    typename SpaceShipTraits<SpaceShip>::Drive drive{}; //instance of the spaceship's Drive
    travelToDispatched(destination, ship, drive);          //call overloaded function
}
```

- A call of `travelTo` with a `HeartOfGold` space ship instance as argument will use the `travelImprobably` function
- A call of `travelTo` with any other space ship with the `SpaceDriveTag` in the `SpaceShipTraits` template will use `travelThroughSpace`

Iterators



- **Different algorithms require different strengths of iterators**
 - OutputIterator - write results, without specifying an end
 - `operator *` returns an lvalue reference for assignment of the value
 - InputIterator - read sequence once
 - `operator *` returns const lvalue reference, or rvalue
 - ForwardIterator - read/write sequence, multiple passes
 - const version: `operator *` returns const lvalue reference or rvalue
 - non-const: `operator *` returns lvalue
 - BidirectionalIterator - read/write sequence, back-forth
 - RandomAccessIterator - read/write/indexed sequence
- **More versatile iterators can be used for more efficient algorithm (like space ships)**
- **Iterator's capabilities can be determined at compile time (with tag types)**

- **Provide member types**

Member	Description
iterator_category	Specifies the iterator category by tag
value_type	Specifies the type of the elements the iterator iterates over
difference_type	Specifies the type used to specify iterator distance (usually ptrdiff_t)
pointer	Specifies the pointer type for the elements the iterator iterates over
reference	Specifies the reference type for the elements the iterator iterates

- **Example:**

```
struct IntIterator {  
    using iterator_category = std::input_iterator_tag;  
    using value_type = int;  
    using difference_type = ptrdiff_t;  
    using pointer = int *;  
    using reference = int &;  
};
```

- Implement members required by your `?_iterator_tag`
- Example: InputIterator (Concept)

```
struct IntIterator {  
    using iterator_category = std::input_iterator_tag;  
    using value_type = int;  
    using difference_type = ptrdiff_t;  
    using pointer = int *;  
    using reference = int &;  
  
    //operator *  
    //operator ->  
    //operator ++ (prefix)  
    //operator ++ (postfix)  
    //operator ==  
    //operator !=  
};
```

```
struct IntIterator { /* Member Types Omitted */
    explicit IntIterator(value_type start = 0) :
        value{start} {}

    auto operator==(IntIterator const& r) const = default;

    value_type operator*() const {
        return value;
    }

    IntIterator& operator++() {
        ++value;
        return *this;
    }

    IntIterator operator++(int) {
        auto old = *this;
        ++(*this);
        return old;
    }

private:
    value_type value;
};
```

Explicit constructor

Default == and != operators

Implement postfix through
prefix operators

Reuse pre-defined type

```
struct input_iterator_tag{};
struct output_iterator_tag{};
struct forward_iterator_tag : public input_iterator_tag{};
struct bidirectional_iterator_tag : public forward_iterator_tag{};
struct random_access_iterator_tag : public bidirectional_iterator_tag{};
```

- Iterators define type aliases for common usage
- `std::iterator<>` base class provides defaults (Pre C++17)

C++03 Style
with typedef

```
template <typename Category, typename Tp, typename Distance = ptrdiff_t,
        typename Pointer = Tp *, typename Reference = Tp &>
struct iterator {
    /// One of the iterator_tags tag types.
    typedef Category iterator_category;
    /// The type "pointed to" by the iterator.
    typedef Tp value_type;
    /// Distance between iterators is represented as this type.
    typedef Distance difference_type;
    /// This type represents a pointer-to-value_type.
    typedef Pointer pointer;
    /// This type represents a reference-to-value_type.
    typedef Reference reference;
};
```

using iterator_category = Category

using value_type = Tp


```
template <typename InputIterator, typename Tp>
auto count(InputIterator first, InputIterator last, const Tp& value)
    -> typename iterator_traits<InputIterator>::difference_type {
    ...
}
```

- STL algorithms often want to determine the type of some specific thing related to an iterator -> use optimal solution!
- However, not all iterator types are actually classes, i.e., subclasses of `std::iterator<>`.
- Default `iterator_traits` just pick the type aliases from those provided by base class `std::iterator`
- Specialization of `iterator_traits` also allows "naked pointers" to be used as iterators in algorithms (that is the main reason for the separate traits mechanism)

```
template <typename _Tp>
struct iterator_traits<_Tp*> {
    typedef random_access_iterator_tag    iterator_category;
    typedef _Tp                          value_type;
    typedef ptrdiff_t                    difference_type;
    typedef _Tp *                        pointer;
    typedef _Tp &                        reference;
};
```

- Simple function overload resolution determines which implementation to use

```
template <typename InputIter, typename Distance>
auto advanceImpl(InputIter& i, Distance d, std::input_iterator_tag) -> void {
    while (d--) { i++; }
}

template <typename RandomAccessIter, typename Distance>
auto advanceImpl(RandomAccessIter& i, Distance d, std::random_access_iterator_tag) -> void {
    i += d;
}

template <typename InputIter, typename Distance>
auto advance(InputIter& i, Distance n) -> void {
    typename std::iterator_traits<InputIter>::difference_type d = n;
    advanceImpl(i, d, typename std::iterator_traits<InputIter>::iterator_category{});
}
```

```
::advance(iter, 15);
```

```
#include <boost/iterator/counting_iterator.hpp>  
#include <boost/iterator/filter_iterator.hpp>  
#include <boost/iterator/transform_iterator.hpp>
```

- **Several pre-defined adapters with factory functions, for example**

- Counting
- Filtering
- Transforming

- **See also**

- http://www.boost.org/doc/libs/1_72_0/libs/iterator/doc/index.html

```
struct odd {  
    auto operator()(int n) const -> bool {  
        return n % 2;  
    }  
};  
  
auto main() -> int {  
    using counter = boost::counting_iterator<int>;  
    std::vector<int> v(counter{1}, counter{11});  
    std::ostream_iterator<int> out { std::cout, ", " };  
    copy(v.begin(), v.end(), out);  
    std::cout << '\n';  
  
    copy(boost::make_filter_iterator(odd{}, v.begin(), v.end()),  
         boost::make_filter_iterator(odd{}, v.end(), v.end()), out);  
    std::cout << '\n';  
  
    auto sq = [](auto i) { return i * i; };  
    copy(boost::make_transform_iterator(v.begin(), sq),  
         boost::make_transform_iterator(v.end(), sq), out);  
}
```

Functor for filtering

Counting iterator

Filter iterator only odd values
provided

transform iterator applies
function/functor/lambda for
each value

- **Inherit and provide own iterator**
 - Class as first template argument
 - Second argument for `value_type`
 - Other template arguments are usually defaulted and OK
- **`input_iterator_helper<T, V>`**
- **`forward_iterator_helper<T, V>`**
- **`bidirectional_iterator_helper<T, V>`**
- **`random_access_iterator_helper<T, V>`**
- **`output_iterator_helper<T>`**
 - Output only is special, no value type, special requirements!

- Using `boost/operators.hpp` shortens definition

Pass own type
CRTP = Curiously Recurring Template Parameter

Explicit
Constructor

```
struct IntIteratorBoost
: boost::input_iterator_helper<IntIteratorBoost, int> {

    explicit IntIteratorBoost(int start = 0)
    : value { start } {}

    auto operator==(IntIteratorBoost const& r) const -> bool {
        return value == r.value;
    }

    auto operator*() const -> value_type { return value; }

    auto operator ++() -> IntIteratorBoost& {
        ++value;
        return *this;
    }

private:
    value_type value;
};
```

Inherit to obtain types and
operations (through CRTP)

`operator==`
required

Reuse
predefined type

- Rarely needed, special tricks required
- `operator*` just returns `this`, `operator++` is a no-op
- `operator=` defines output of value
- `*out++ = 42; // works`

```
struct MyIntOutIter {  
  
    using iterator_category = std::output_iterator_tag;  
    using value_type = int;  
    /* Other Member Types Omitted */  
  
    auto operator++() -> MyIntOutIter& {  
        return *this;  
    }  
    auto operator++(int) -> MyIntOutIter {  
        return *this;  
    }  
    auto operator*() const -> MyIntOutIter const& {  
        return *this;  
    }  
    auto operator=(value_type val) const -> void {  
        std::cout << "val = " << val << '\n';  
    }  
};
```

- Even simpler with Boost
- `operator=` defines output of value
- `*out++ = 42; // works`

```
struct MyIntOutIterBoost : boost::output_iterator_helper<MyIntOutIterBoost> {  
    auto operator=(int val) const -> void {  
        std::cout << "value = " << val << '\n';  
    }  
};
```


- An output iterator for summing and averaging

```
struct SummingIter {  
    using iterator_category = std::output_iterator_tag;  
    using value_type = int;  
    /* Other Member Types Omitted */  
  
    auto operator++() -> void { ++counter; }  
    auto operator*() -> SummingIter& {  
        return *this;  
    }  
    auto operator=(int val) -> void {  
        sum += val;  
    }  
    auto average() const -> double{  
        return sum / counter;  
    }  
    double sum{};  
    size_t counter{};  
};
```

```
std::vector<int> v {3, 1, 4, 1, 5, 9, 2};  
auto res = copy(v.begin(), v.end(), SummingIter{});  
std::cout << res.sum << " average: " << res.average();
```

```
IntInputter();  
value_type operator*();  
auto operator==(IntInputter const & other) const -> bool;
```

- How do we initialize the reference in the default constructor?
- What should happen in operator*()?
- How can we compare iterators to guarantee equality for EOF-condition?

```
IntInputter();
```

- **We need a dirty trick: A global variable to initialize the reference!**
 - Put it into an anonymous namespace to hide it
 - Not good for multi-threading -> bad for production code

```
namespace {  
    // a global helper needed...  
    std::istream empty{};  
}  
IntInputter::IntInputter()  
    : input { empty } {  
    // guarantee the empty stream is not good()  
    input.clear(std::ios_base::eofbit);  
}
```

```
value_type operator*();  
auto operator==(IntInputter const & other) const -> bool;
```

- Just input

```
auto IntInputter::operator*() -> IntInputter::value_type {  
    value_type value{};  
    input >> value;  
    return value;  
}
```

- And Compare. Only equal if both are !good()

- Both eof() would result in problems on failing input when using standard algorithms

```
auto IntInputter::operator==(const IntInputter & other) const -> bool {  
    return !input.good() && !other.input.good();  
}
```

```
struct IntInputter {
    using iterator_category = std::input_iterator_tag;
    using value_type = int;
    /* Other Member Types Omitted */

    IntInputter();
    explicit IntInputter(std::istream & in)
        : input { in } {}
    auto operator*() -> value_type;
    auto operator++() -> IntInputter& {
        return *this;
    }
    auto operator++(int) -> IntInputter {
        IntInputter old{*this};
        ++(*this);
        return old;
    }
    auto operator==(IntInputter const & other) const -> bool;
    auto operator!=(IntInputter const & other) const -> bool {
        return !(*this == other);
    }
private:
    std::istream & input;
};
```

Default Constructor
for EOF

++ does nothing

Equal only if both
EOF

Caller must guarantee survival
of object, otherwise "dangling"
reference!

- **Alternative to global variable: Naked pointer that might point to "nothing"**
 - A pointer can be empty, which requires a check
- **Boost can be used for brevity**

```
struct IntInputterPtrBoost
: boost::input_iterator_helper<IntInputterPtrBoost, int> {
IntInputterPtrBoost() = default;
explicit IntInputterPtrBoost(std::istream& in)
: input {&in} {}
auto operator*() -> IntInputterPtrBoost::value_type;
auto operator++() -> IntInputterPtrBoost& {
return *this;
}
auto operator==(IntInputterPtrBoost const & other) const -> bool;
private:
std::istream * input{};
};
```

Caller must guarantee survival of object, otherwise "dangling" reference!

Initialized with nullptr

```
value_type operator*();  
auto operator==(IntInputter const & other) const -> bool;
```

- **Input only if defined**

```
auto IntInputterPtrBoost::operator*() -> IntInputterPtrBoost::value_type {  
    value_type value{};  
    if (input) (*input) >> value;  
    return value;  
}
```

- **And Compare. Only equal if both are either nullptr or !good()**

```
auto IntInputterPtrBoost::operator==(IntInputterPtrBoost const & other) const -> bool {  
    return (!input || !input->good()) && (!other.input || !other.input->good());  
}
```

- **Tag types can be used for static dispatching (For example in iterators)**
- **DIY Iterators are used much less often than functors for parameterizing the standard library algorithm**
 - Try one of the boost adapters first
- **They need pre-defined member types to work with the standard algorithms**
 - as well as a set of operators
 - if DIY `<boost/operators.hpp>` provides boilerplate code