Department I - C Plus Plus

Modern and Lucid C++ Advanced for Professional Programmers

Week 5 – Iterators & Tags

Thomas Corbat / Felix Morgner
Rapperswil, 16.03.2020
FS2020

HSR HOCHSCHULE FÜR TECHNIK RAPPERSWIL
FHO Fachhochschule Ostschweiz
IFS INSTITUTE FOR SOFTWARE

Cevelop
Your C++ deserves it

Recap Week 4

Your C++ deserves it

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

INSTITUTE FOR
SOFTWARE

- **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 const x, y;
};

Point * createPoint(int x, int y) {
  return new Point{x, y}; //constructor
}

Point * createCorners(int x, int y) {
  return new Point[2]{{0, 0}, {x, y}};
}
```

Memory Deallocation (with **delete**)

- **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!**

```cpp
struct Point {
  Point(int x, int y) :
        x {x}, y {y} {}
  int const x, y;
}

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

- **Topics:**
  - Tags for Dispatching
  - DIY Iterators

Tags for Dispatching

Cevelop
Your C++ deserves it

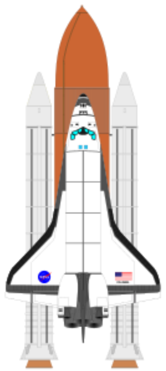
HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS INSTITUTE FOR SOFTWARE

- **Template parameters don't require a specified type hierarchy**
  - but they expect an argument to satisfy a concept
- **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)

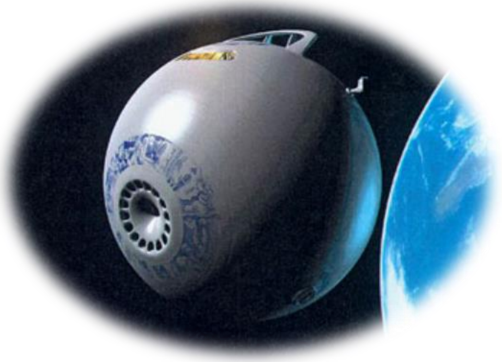
```
template<typename SpaceShip>
void travelTo(Galaxy destination, SpaceShip & ship) {
  ship.$functionUsedToTravel$(destination);
}
```

Tags for Dispatching: Space Ship Types



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

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



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

Heart of Gold Image from: http://hitchhikers.wikia.com/wiki/File:Heartofgold_exterior1.jpg

Tags for Dispatching: Drive Technology Tags

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

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

//Provides travelThroughSpace and travelImprobably
struct InfniteProbabilityDriveTag : 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**

- **Space ship example: Every space ship can somehow travel through space, but some space ships have more advanced technology**

```
struct SpaceDriveTag {};

struct HyperspaceDriveTag : SpaceDriveTag {};

struct InfniteProbabilityDriveTag : SpaceDriveTag {};
```

```
struct MultiPurposeCrewVehicle;

struct GalaxyClassShip;

struct HeartOfGoldPrototype;
```

- **Approach 1: Derive space ship from the associated tag typ**
  - This is not applicable for all types (e.g. for primitive types)
  - This is not extensible (i.e. you cannot specify new kinds of tag kinds as a user of the API)
- **Approach 2: `SpaceshipTraits` template**

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

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

Tags for Dispatching: Dispatching Applied

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

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

template<typename Spaceship>
void travelTo(Galaxy destination, Spaceship & ship) {
  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

Your C++ deserves it

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

INSTITUTE FOR
SOFTWARE

- **Different algorithms require different strengths of iterators**
  - InputIterator - read sequence once
    - `operator *` returns const lvalue reference, or rvalue
  - OutputIterator - write results, without designating an end
    - `operator *` returns lvalue reference
  - 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 (ususally `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 &;
};
```

Provide Expected Members

- **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 !=
};
```

Example DIY Iterator for Integers

```
struct IntIterator { /* Member Types Omitted */
  explicit IntIterator(int const start = 0) :
    value { start } {}
  bool operator==(IntIterator const & r) const {
    return value == r.value;
  }
  bool operator!=(IntIterator const & r) const {
    return !(*this == r);
  }
  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

Implement != through operator ==

Implement postfix through prefix operators

Reuse pre-defined type

Iterator Member Expectations

```
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
```

What Are `iterator_traits<>`?


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

- **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>
void advanceImpl(InputIter & i, Distance d, std::input_iterator_tag) {
  while (d--) { i++; }
}

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

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

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

Boost Iterator Library Provides More


```
#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_66_0/libs/iterator/doc/index.html

Example: Using Boost Iterator

```
struct odd {
  bool operator()(int n) const {
    return n % 2;
  }
};

int main() {
  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

`#include <boost/operators.hpp>`

- **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!

Example: DIY Iterator for `int` with Boost

- **Using `boost/operators.hpp` shortens definition**

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

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

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

  value_type operator*() const { return value; }

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

private:
  value_type value;
};
```

Pass own type
CRTP = Curiously Recurring Template Parameter

Explicit Constructor

Inherit to obtain types and operations (through CRTP)

Reuse predefined type

`operator==` required

- **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 */

  MyIntOutIter & operator++() {
    return *this;
  }
  MyIntOutIter operator++(int) {
    return *this;
  }
  MyIntOutIter const & operator*() const {
    return *this;
  }
  void operator=(value_type val) const {
    std::cout << "val = " << val << '\n';
  }
};
```

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

```
struct MyIntOutIterBoost : boost::output_iterator_helper<MyIntOutIterBoost> {
  void operator=(int val) const {
   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 */

  void operator++() { ++counter; }
  SummingIter & operator*() {
    return *this;
  }
  void operator=(int val) {
    sum += val;
  }
  double average() const {
    return sum / coutner;
  }
  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();
```

```
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 } {}
  value_type operator*();
  IntInputter & operator++() {
    return *this;
  }
  IntInputter operator++(int) {
    IntInputter old{*this};
    ++(*this);
    return old;
  }
  bool operator==(IntInputter const & other) const;
  bool operator!=(IntInputter const & other) const {
    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!

Problems with Stream Input Iterator

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

- **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::istringstream empty{};
}
IntInputter::IntInputter()
    : input { empty } {
  // guarantee the empty stream is not good()
  input.clear(std::ios_base::eofbit);
}
```

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

- **Just input**

```
IntInputter::value_type IntInputter::operator*() {
  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

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

- **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} {}
  IntInputterPtrBoost::value_type operator*();
  IntInputterPtrBoost & operator++() {
    return *this;
  }
  bool operator==(IntInputterPtrBoost const & other) const;
private:
  std::istream * input{};
};
```

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

Initialized with `nullptr`

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

- **Input only if defined**

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

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

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

Summary

- **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