Department I - C Plus Plus

Modern and Lucid C++ for Professional Programmers

Week 14 – Outlook C++20 Features

Felix Morgner

Rapperswil, 18.12.2018

HS2018

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS
INSTITUTE FOR
SOFTWARE

Concepts

Constraining Generic Functions

- **Remember: To be useable as an argument for a specific template parameter, a type has to fulfill some requirements depending on its usage**
- **What are the requirements of the Type `T` in our `sum` function template?**

```
template<typename T>
T sum(T left, T right) {
  return left + right;
}
```

  - \+ Addable with itself
  - Copy/Move-Constructible, to return `T` by value
- **What happens if the type does not meet the requirements?**

```
int main() {
  std::vector<int> a { 1, 2, 3 };
  std::vector<int> b { 2, 3, 4 };
  sum(a, b);
}
```

Review: Concept of a Function Template Parameter (contd.)


```
In file included from C:/Program Files/mingw-w64/x86_64-8.1.0-posix-seh-rt_v6-rev0/mingw64/lib/gcc/x86_64-w64-mingw32/8.1.0/include/c++/bits/stl_algobase.h:67,

                 from C:/Program Files/mingw-w64/x86_64-8.1.0-posix-seh-rt_v6-rev0/mingw64/lib/gcc/x86_64-w64-mingw32/8.1.0/include/c++/vector:60,

                 from ..\main.cpp:1:

C:/Program Files/mingw-w64/x86_64-8.1.0-posix-seh-rt_v6-rev0/mingw64/lib/gcc/x86_64-w64-mingw32/8.1.0/include/c++/bits/stl_iterator.h:970:5: note: candidate: 'template<class _Iterator, class _Container> __gnu_cxx::__normal_iterator<_Iterator, _Container> __gnu_cxx::operator+(typename __gnu_cxx::__normal_iterator<_Iterator, _Container>::difference_type, const __gnu_cxx::__normal_iterator<_Iterator, _Container>&)’

     operator+(typename __normal_iterator<_Iterator, _Container>::difference_type
     ^~~~~~~~

C:/Program Files/mingw-w64/x86_64-8.1.0-posix-seh-rt_v6-rev0/mingw64/lib/gcc/x86_64-w64-mingw32/8.1.0/include/c++/bits/stl_iterator.h:970:5: note:   template argument deduction/substitution failed:

..\main.cpp:5:14: note:   'std::vector<int>' is not derived from 'const __gnu_cxx::__normal_iterator<_Iterator, _Container>’

  return left + right;
         ~~~~~^~~~~~~
```

- **Obviously, the compiler is not happy with us**
  - What is the problem?

```
..\main.cpp:5:14: note:   'std::vector<int>' is not derived from 'const
__gnu_cxx::__normal_iterator<_Iterator, _Container>’

  return left + right;
         ~~~~~^~~~~~~
```

  - Remember: Always look at the top of the error message!

```
..\main.cpp:5:14: error: no match for 'operator+' (operand types are
'std::vector<int>' and 'std::vector<int>’)

  return left + right;
         ~~~~~^~~~~~~
```

- **Provide “Documentation Comment”?**

```
/**
 * Sum two objects.
 *
 * @note T must be addable!
 */
template<typename T>
  T sum(T left, T right) {
  return left + right;
}
```

- **Compilers don’t read comments!**
- **Does not change the error message => Does not help the programmer**

- **Provide Overloads?**

```
int sum(int left, int right) {
  return left + right;
}

float sum(float left, float right) {
  return left + right;
}

//...
```

- **Cumbersome and duplicated effort**
- **What if the user needs more overloads?**

- **A generic facility to formulate constraints**
  - Simple for the compiler to check
    - Enabling better error reporting
  - Easy to understand for Humans
    - Enabling better error comprehension
  - Reusable!

```
template<typename T>
T sum(T left, T right) /* requires Addable<T> */ {
    return left + right;
}
```

- **A generic facility to formulate constraints**
  - Simple for the compiler to check
    - Enabling better error reporting
  - Easy to understand for Humans
    - Enabling better error comprehension
  - Reusable!

```
template<typename T>
T sum(T left, T right) requires Addable<T> {
    return left + right;
}
```

```
int main() {
  std::vector<int> a { 1, 2, 3 };
  std::vector<int> b { 2, 3, 4 };
  sum(a, b);
}
```

```
<source>:17:5: error: no matching function for call to 'sum’
    sum(a, b);
    ^~~

<source>:9:3: note: candidate template ignored: constraints not satisfied [with T = std::__1::vector<int,
std::__1::allocator<int> >]
T sum(T left, T right) requires Addable<T> {
  ^

<source>:9:33: note: because 'std::__1::vector<int, std::__1::allocator<int> >' does not satisfy 'Addable’
T sum(T left, T right) requires Addable<T> {
                                ^

<source>:5:9: note: because 'val + val' would be invalid: invalid operands to binary expression
('std::__1::vector<int, std::__1::allocator<int> >' and 'std::__1::vector<int, std::__1::allocator<int>
>’)
    val + val;
```

- **The Standard Library will include concept definitions for concepts used in the Standard Library itself**
  - Helping to diagnose misuse of Standard Library types and functions
  - E.g:
    - `CopyConstructible`
    - `Movable`
    - `EqualityComparable`
- **You can also define your own concepts!**
  - For example for specific concepts of your own libraries

- **Concepts are templates themselves!**

```
template<typename T>
concept Addable = requires (T val) {
    { val + val } -> T;
};
```

- **Can constrain expressions and types**

- **Can also be defined ad-hoc**

```
template<typename T>
T sum(T left, T right) requires requires { { right + left } -> T; } {
    return left + right;
}
```

- **Oftentimes a type needs to fulfill multiple requirements**

```
template<typename T>
T sum(T left, T right) {
  return left + right;
}
```

  - + Addable with itself
  - Copy/Move-Constructible, to return `T` by value

- **Therefore, constraints can be combined**

```
template<typename T>
T sum(T left, T right) requires Addable<T> && CopyConstructible<T> {
    return left + right;
}
```

- **Template errors are often hard to figure out**
  - Cannot be Unit Tested
  - Cannot be debugged
- **Concepts will be able to help you**
  - By providing meaningful error messages
  - By allowing you to express requirements in code

Ranges

Generate, Combine, Process

Cevelop
Your C++ deserves it

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS
INSTITUTE FOR
SOFTWARE

- **Iterators are a core component of all data processing in C++**
  - All standard containers provide them
  - All standard algorithms use them
- **You have used them a lot!**

```
std::vector<int> v{5, 4, 3, 2, 1};
std::cout << std::accumulate(std::begin(v), std::end(v), 0) << " = sum\n";
```

- **Combined with algorithms, iterators provide a powerful abstraction**
- **But…**

What Is Wrong with Iterators

- **Iterators are very “verbose”**
  - You always need to provide the `begin` and `end` of the range you want to iterate
  - Oftentimes you want to process to full range
- **Iterators don’t always mean the same thing**
  - `begin()` marks the first element
  - `end()` marks the element behind the last one
  - Sometimes, iterators mark positions, not elements
- **Iterators are hard to compose**

```
void word_tally_classic(std::istream & in, std::ostream & out) {
  using input = std::istream_iterator<Word>;
  using output = std::ostream_iterator<std::string>;

  std::map<Word, int> words{};

  std::for_each(input{in}, input{}, [&](auto && word) {
    words[word]++;
  });

  std::transform(words.cbegin(), words.cend(), output{out, "\n"}, [](auto const & e) {
    return e.first.str() + ": " + std::to_string(e.second);
  });
}
```

- **Ranges generalize the idea of a “range”**
  - Everything that has `begin` and `end` is a range!
  - This of course includes standard containers
  - As well as `std::initializer_list`
  - And your own containers as well!
    - If you implement the standard container interface
- **Ranges will work with all standard algorithms**
- **Ranges are composable**

```
void word_tally_ranges(std::istream & in, std::ostream & out) {
  namespace rng = ranges;
  using input = rng::istream_range<Word>;
  using output = rng::ostream_iterator<std::string>;

  std::map<Word, int> words{};

  rng::for_each(input{in}, [&](auto && word) {
    words[word]++;
  });

  rng::transform(words, output{out, "\n"}, [](auto const & e) {
    return e.first.str() + ": " + std::to_string(e.second);
  });
}
```

- **Ranges introduce a new concept called “views”**
  - Not to be confused with `std::string_view`!
- **Views can be used to**
  - Generate data
    - `std::vector<int> numbers{rng::view::ints(0,20)};`
  - To transform ranges
    - `rng::view::transform(numbers, [](int n){ return n + 2; });`
  - And to filter ranges
    - `rng::view::filter(numbers, [](int n){ return n % 2; });`
- **Views can also be composed using “`operator|`”**

- **Consider the following challenge:**
  - Given is function `bool is_prime(int number)` that tells you if a number is prime
  - Write a function that:
    - Prints ten numbers, whereby
    - Each number is the square of every third prime number
    - Starting with 3, ergo the first number printed is 9
  - Without writing a single loop!
  - In 2 minutes!

Composing Ranges (contd.)

```
int main() {
  namespace rng = ranges;

  auto numbers = rng::view::ints(3)
    | rng::view::filter(is_prime)
    | rng::view::stride(3)
    | rng::view::transform([](auto i){ return i * i; })
    | rng::view::take(10);

  rng::copy(numbers, rng::ostream_iterator<int>{std::cout, "\n"});
}
```

- **Ranges also introduce a new algorithm for processing input**
  - `ranges::getlines(std::cin)`
- **Simplifies reading line based input**
- **No more code like this:**

```
std::string buffer{};
while(std::getline(in, buffer)) {
    // process line
}
```

```
void module_sort::read(std::istream& input) {
  std::string line;
  while (std::getline(input, line)) {
    std::istringstream is { line };
    std::string name;
    if (is >> name) {
      auto current = insert_or_update_module(name);
      while (is >> name) {
        auto dep = insert_or_update_module(name);
        current->add(dep);
      }
    }
  }
}
```

```
void module_sort::read(std::istream& input) {
  rng::for_each(rng::getlines(input), [&](auto const & line) {

    auto elements = rng::view::split(line, " ");

    rng::for_each(rng::view::take(elements, 1), [&](auto const & mod){

      auto current = insert_or_update_module(mod);

      rng::for_each(rng::view::drop(elements, 1), [&](auto const & dep){

        current->add(insert_or_update_module(dep));

      });
    });
  });
}
```

Ranges Summary

- **Ranges generalize an already existing concept**
- **Ranges add new powerful abstraction called “views**
  - Simplify data generation and processing
- **Ranges will work with standard algorithms**
  - Make using them less verbose
  - Provide easy composition

Operator “Spaceship”

Simplifying comparison operators

- **In lecture 5 we talked about the following implementation of a simple date class**

```
struct Date {
  int year, month, day;

  bool operator<(Date const & rhs) const {
    return year < rhs.year ||
      (year == rhs.year && (month < rhs.month ||
        (month == rhs.month && day == rhs.day)));
  }
};
```

- **Implementing the remaining operator is easy enough**
  - But cumbersome
  - And (almost) always the same

```
struct Date {
  int year, month, day;

  bool operator<(Date const & rhs) const {
    return year < rhs.year ||
      (year == rhs.year && (month < rhs.month ||
        (month == rhs.month && day == rhs.day)));
  }

  bool operator>(Date const & rhs) const { return *this < rhs; }

  bool operator==(Date const & rhs) const { return !(*this > rhs || *this < rhs);}

  bool operator!=(Date const & rhs) const { return !(*this == rhs); }

  bool operator>=(Date const & rhs) const { return !(*this < rhs); }

  bool operator<=(Date const & rhs) const { return !(*this > rhs); }
};
```

- **(most) Humans are bad at repetitive tasks but computers excel at it!**
- **One solution are mixins**
  - For example Boost Operator Helpers
  - But they rely on inheritance
- **Idea: Can we get the compiler to generate the operators for us?**
- **Idea: Can we convey more information about our type?**

- **C++20 introduces a new operator `<=>`**
  - Similar to Java’s `compare(…)` function but more powerful!
  - Allows to convey an additional property of our type: The ordering
- **std::strong_ordering**
  - `a > b` implies `f(a) > f(b)`
- **std::weak_ordering**
  - `a > b` **does not** imply `f(a) > f(b)`
- **std::partial_ordering**
  - `a > b` **does not** imply `f(a) > f(b)`
  - There might be a value a so that `a < b` and `a == b` and `a > b` are all **`false`**!

- **Since all integral types support operator <=>, we could implement one for Date**

```
struct Date {
  int year, month, day;

  std::strong_ordering operator<=>(Date const & other) const {
    return ((year - other.year) * 10000
      + (month - other.month) * 100
      + day - other.day) <=> 0;
  }
};
```

- **The compiler will generate the rest!**

- **C++ is alive and well**
- **C++20 will be a huge step towards making the language easier to work with**
- **There is lots more to come**
  - Networking and asynchronous I/O
  - Contracts
  - Simpler string formatting
  - More `constexpr`
  - Coroutines (post C++20)
  - Reflection (post C++20)