PROGRAMMING IN C++

Jülich Supercomputing Centre

May 13, 2022 | Sandipan Mohanty | Forschungszentrum Jülich, Germany

JÜLICH Forschungszentrum

Type erasure

TYPE ERASURE TECHNIQUE

```
auto f(int i) -> PolyVal;
void elsewhere() {
    std::vector<PolyVal> v;
    v.push_back(1);
    v.push_back(2.0);
    v.push_back("Green"s);

    for (auto&& elem : v) {
        func1(elem);
    }
    PolyVal X = f(i);
}
```

- Polymorphic behaviour attained using a class hierarchy and virtual functions...
  - is extensible by simply inheriting from the `Base` type and overriding the virtual functions
  - But, it has “reference symantics”, so that we can not return those polymorphic objects by value from functions
  - Built in types can not be accommodated into the same hierarchy
- `variant` provides a solution to the two problems above, but we need to commit to a fixed number of polymorphic types in the problem, from the outset
- `std::any` is a library provided facility for type erasure

TYPE ERASURE TECHNIQUE

```cpp
void func1(int x);
void func1(double x);
void func1(std::string x);
auto f(int i) -> PolyVal;
void elsewhere() {
    std::vector<PolyVal> v;
    v.push_back(1);
    v.push_back(2.0);
    v.push_back("Green"s);

    for (auto&& elem : v) {
        func1(elem);
    }
    PolyVal X{3.141};
    // func1(X) should go to func1(double)
    X = PolyVal{"some string"s};
    // func1(X) should now go to func1(string)
    X = f(i);
    // func1(X) should redirect according to what
    // polymorphic value f happens to return
}
```

- We want a type `PolyVal`, so that we can store different types of entities in it
- A uniform container of `PolyVal` should be able to hold values of different types
- When a certain instance is used, it should still be able to behave according to the value it is currently holding.
- We should be able to copy a `PolyVal` object using normal copy construction or copy assignment in such a way that the copy of a `PolyVal` storing a `Triangle` would still behave as a `Triangle`

JÜLICH Forschungszentrum

TYPE ERASURE TECHNIQUE

```
class PolyVal {
    struct Internal {
        virtual ~Internal() noexcept = default;
        virtual auto clone() const -> std::unique_ptr<Internal> = 0;
        virtual void func1_() const = 0;
    };
    template <class T>
    struct Wrapped : public Internal // continued...

public:
    template <class T>
    PolyVal(const T& var) : ptr{ std::make_unique<Wrapped<T>>(var) } {}
    PolyVal(const PolyVal& other) : ptr { other.ptr->clone() } {}
private:
    std::unique_ptr<Internal> ptr;
};
```

- Make a normal class with an internal class with virtual functions defining the desired interface, and another internal wrapper class template deriving from the internal base
- Give the outer class one template constructor (unrestrained here to isolate the TE technique)

TYPE ERASURE TECHNIQUE

```cpp
class PolyVal {
    struct Internal {
        virtual ~Internal() noexcept = default;
        virtual auto clone() const -> std::unique_ptr<Internal> = 0;
        virtual void func1_() const = 0;
    };
    template <class T>
    struct Wrapped : public Internal // continued...

public:
    template <class T>
    PolyVal(const T& var) : ptr{ std::make_unique<Wrapped<T>>(var) } {}
    PolyVal(const PolyVal& other) : ptr { other.ptr->clone() } {}
private:
    std::unique_ptr<Internal> ptr;
};
```

- Let the class contain a smart pointer to this base, but initialize that member using a class template which inherits from the internal base.
- Implement a copy constructor for `PolyVal` by using a virtual `clone()` function for the internal class
- Use the template constructor to create a wrapped object containing a copy of the input parameter

TYPE ERASURE TECHNIQUE

```
class PolyVal {
    template <class T>
    struct Wrapped : public Internal {
        Wrapped(T ex) : obj{ex} {}
        ~Wrapped() noexcept override {}
        auto clone() const -> std::unique_ptr<Internal> override
        {
            return std::make_unique<Wrapped>(obj);
        }
        void func1_() const override { func1(obj); }
        T obj;
    };
};
```

- The internal wrapper should store an object of the template parameter type
- It should provide copy, clone etc.
- It should redirect function calls in our original requirement to free functions

TYPE ERASURE TECHNIQUE

```cpp
class PolyVal {
    template <class T>
    struct Wrapped : public Internal {
        Wrapped(T ex) : obj{ex} {}
        ~Wrapped() noexcept override {}
        auto clone() const -> std::unique_ptr<Internal> override
        {
            return std::make_unique<Wrapped>(obj);
        }
        void func1_() const override { func1(obj); }
        T obj;
    };
};
```

- As long as those free functions exist for a type `F`, it will be possible to create objects of `PolyVal` type from type `F`

Example 1.1:

`examples/PolyVal.cc` contains the code corresponding to the slides shown here. Verify that we achieve our purpose of having a copyable object preserving polymorphic behaviour. Add a function `func1()` for a new type into the mix, and extend the existing setup.

Example 1.2:

Sequences of objects with polymorphic behaviour is a frequently occuring programming problem. We have seen one example before, with a vector of `unique_ptr<Shape>`, filled with newly created instances of types inheritted from `Shape`, such as `Circle`, `Triangle` etc. The problem can be solved in many alternative ways. `examples/polymorphic` contains 4 subdirectories with different approaches to the geometric object example. (i) Inherittance with virtual functions (ii) `std::variant` with visitors (iii) Using `std::any` (iv) Custom type erasure.

VALARRAY

```
#include <valarray>

void varray_ops()
{
    std::valarray V1(0., 1000000UL);
    std::valarray<double> V2;
    v2.resize(1000000UL, 0.);
    auto x = exp(-V1 * V1) * sin(V2);
    if (x.sum() < 100.0) {
    //
    }
}
```

- Another dynamic array type
- Mostly intended for numeric operations
- Expression template based whole array math operations
- Algorithms through `std::begin(v)` etc., instead of own member functions
- Bizarre constructor with different convention compared to any other container in the STL.

NUMERIC ALGORITHMS

```cpp
#include <numeric>

using std::reduce;
using std::transform_reduce;

auto res = reduce(v.begin(), v.end());
auto res = reduce(v.begin(), v.end(), init);
auto res = reduce(v.begin(), v.end(),
    init, std::plus<double>{});
auto res = transform_reduce(
    u.begin(), u.end(),
    v.begin(), init);
auto res = transform_reduce(
    u.begin(), u.end(),
    v.begin(), init, reduce_op, transf_op);
auto res = transform_reduce(
    std::execution::par,
    u.begin(), u.end(),
    v.begin(), init, reduce_op, transf_op);
```

- Algorithms focused on numeric calculations are in the `numeric` header
- Given `b`, `e` as iterators in a range V, `reduce(b, e)`: $\sum_{i=b}^{e} V_i$
- `transform_reduce(b, e)`: $\sum_{i=b}^{e} f(V_i)$
- `adjacent_difference(b, e)`: $\{V_b, (V_{b+1} - V_b), (V_{b+2} - V_{b+1}), \ldots\}$
- Parallel versions also in the library
- To run the numeric operations in parallel, use the parallel execution policy

SPAN

```cpp
using std::span;
using std::transform_reduce;
using std::plus;
using std::multiplies;
auto compute(span<const double> u,
     span<const double> v) -> double
{
    return transform_reduce(
        u.begin(), u.end(),
        v.begin(), 0., plus<double>{},
        multiplies<double>{});
}

void elsewhere(double* x, double* y,
               unsigned N)
{
    return compute(span(x, N), span(y, N));
}
```

- Non-owning view type for a contiguous range
- No memory management
- Numeric operations can often be expressed in terms of existing arrays in memory, irrespective of how they got there and who cleans up after they expire
- `span` is designed to be that input for such functions
- Cheap to copy: essentially a pointer and a size
- STL container like interface

Example 1.3:

`examples/spans` is a directory containing the compute function as shown here. Notice how this function is used directly using C++ array types as arguments instead of spans, and indirectly when we only have pointers.

RANGES

```cpp
std::vector v{ 1, 2, 3, 4, 5, 6, 7, 8, 9 };
// before std::ranges we did this...
std::reverse(v.begin(), v.end());
std::rotate(v.begin(), v.begin() + 3, v.end());
std::sort(v.begin(), v.end());
```

```cpp
std::vector v{ 1, 2, 3, 4, 5, 6, 7, 8, 9 };
namespace sr = std::ranges;
sr::reverse(v);
sr::rotate(v, v.begin() + 3);
sr::sort(v);
```

- The `<ranges>` header defines a set of algorithms taking “ranges” as inputs instead of pairs of iterators
- A `range` is a **concept** : something with `sr::begin()`, which returns an entity which can be used to iterate over the elements, and `sr::end()` which returns a sentinel which is equality comparable with an iterator, and indicates when the iteration should stop.
- `sr::sized_range` : the range knows its size in constant time
- `input_range`, `output_range` etc. based on the iterator types
- `borrowed_range` : a type such that its iterators can be returned without the danger of dangling.
- `view` is a range with constant time copy/move/assignment

USING RANGES FROM STD OR FROM RANGE-V3

```
// cxx220ranges
#include <version>
#ifdef __cpp_lib_ranges
  #include<ranges>
  namespace sr = std::ranges;
  namespace sv = sr::views;
#elif __has_include (<range/v3/all.hpp>)
  #include<range/v3/all.hpp>
  namespace sr = ranges;
  namespace sv = sr::views;
  #warning Using ranges-v3 3rd party library
#else
#error No suitable header for C++20 ranges was found!
#endif
```

- The C++20 `<ranges>` library is based on the open source `range-v3` library. Parts of the `range-v3` library were adopted for C++20, more might be added in C++23.
- Even if the standard library shipping with some compilers do not have many features of `<ranges>`, one can start using them, with a redirecting header, which makes use of another standard library feature
- Including `<version>` results in the definition of library feature test macros, which can be used to choose between different header files

Our examples are actually written using a redirecting header as shown here. Compilation with GCC uses the compiler's own version. Compilation with Clang uses the `range-v3` version.

FUN WITH RANGES AND VIEWS

```
// examples/ranges0.cc
#include <ranges>
#include <span>
auto sum(std::ranges::input_range auto&& seq) {
    std::iter_value_t<decltype(seq)> ans{};
    for (auto x : seq) ans += x;
    return ans;
}
auto main() -> int
{
    //using various namespaces;
    cout << "vector   : " << sum(vector(  { 9,8,7,2 } )) << "\n";
    cout << "list     : " << sum(list(    { 9,8,7,2 } )) << "\n";
    cout << "valarray : " << sum(valarray({ 9,8,7,2 } )) << "\n";
    cout << "array    : "
        << sum(array<int,4>({ 9,8,7,2 } )) << "\n";
    cout << "array    : "
        << sum(array<string, 4>({ "9"s,"8"s,"7"s,"2"s } )) << "\n";
    int A[]{1,2,3};
    cout << "span(built-in array) : " << sum(span(A)) << "\n";
}
```

FUN WITH RANGES AND VIEWS

- The `ranges` library gives us many useful concepts describing sequences of objects.
- The function template `sum` in `examples/ranges0.cc` accepts any input range, i.e., some entity whose iterators satisfy the requirements of an `input_iterator`.
- Notice how we obtain the value type of the range
- Many STL algorithms have `range` versions in C++20. They are functions like `sum` taking various kinds of ranges as input.
- The range concept is defined in terms of
  - the existence of an iterator type and a sentinel type.
  - the iterator should behave like an iterator, e.g., allow `++it` `*it` etc.
  - it should be possible to compare the iterators with other iterators or with a sentinel for equality.
  - A `begin()` function returning an iterator and an `end()` function returning a sentinel

FUN WITH RANGES AND VIEWS

```cpp
// examples/iota.cc
#include <ranges>
#include <iostream>
auto main() -> int {
    namespace sv = std::views;
    for (auto i : sv::iota(1UL)) {
        if ((i+1) % 10000UL == 0UL) {
            std::cout << i << ' ';
            if ((i+1) % 100000UL == 0UL)
                std::cout << '\n';
            if (i >= 100000000UL) break;
        }
    }
}
```

- All containers are ranges, but not all ranges are containers
- `std::string_view` is a perfectly fine range. Has iterators with the right properties. Has `begin()` and `end()` functions. It does not own the contents, but "ownership" is not part of the idea of a range.
- We could take this further by creating `views` which need not actually contain any objects of the sequence, but simply fake it when we dereference the iterators.
- Example: the standard view `std::views::iota(integer)` gives us an infinite sequence of integers starting at a given value.

FUN WITH RANGES AND VIEWS

```
// examples/iota.cc
#include <ranges>
#include <iostream>
auto main() -> int {
    namespace sv = std::views;
    for (auto i : sv::iota(1UL)) {
        if ((i+1) % 10000UL == 0UL) {
            std::cout << i << ' ';
            if ((i+1) % 100000UL == 0UL)
                std::cout << '\n';
            if (i >= 100000000UL) break;
        }
    }
}
```

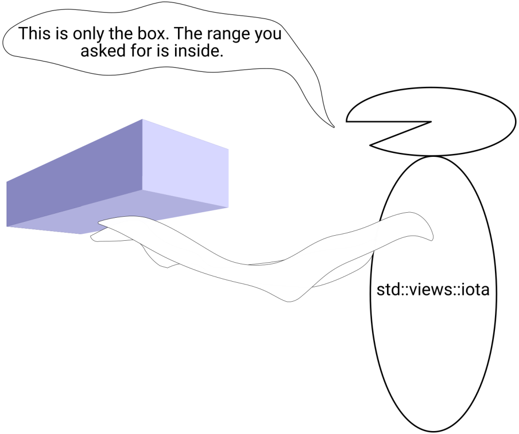
- All containers are ranges, but not all ranges are containers
- `std::string_view` is a perfectly fine range. Has iterators with the right properties. Has `begin()` and `end()` functions. It does not own the contents, but "ownership" is not part of the idea of a range.
- We could take this further by creating `views` which need not actually contain any objects of the sequence, but simply fake it when we dereference the iterators.
- Example: the standard view `std::views::iota(integer)` gives us an infinite sequence of integers starting at a given value.

FUN WITH RANGES AND VIEWS

```
// examples/iota.cc
#include <ranges>
#include <iostream>
auto main() -> int {
    namespace sv = std::views;
    for (auto i : sv::iota(1UL)) {
        if ((i+1) % 10000UL == 0UL) {
            std::cout << i << ' ';
            if ((i+1) % 100000UL == 0UL)
                std::cout << '\n';
            if (i >= 100000000UL) break;
        }
    }
}
```

- All containers are ranges, but not all ranges are containers
- `std::string_view` is a perfectly fine range. Has iterators with the right properties. Has `begin()` and `end()` functions. It does not own the contents, but "ownership" is not part of the idea of a range.
- We could take this further by creating `views` which need not actually contain any objects of the sequence, but simply fake it when we dereference the iterators.
- Example: the standard view `std::views::iota(integer)` gives us an infinite sequence of integers starting at a given value.

FUN WITH RANGES AND VIEWS



- All containers are ranges, but not all ranges are containers
- `std::string_view` is a perfectly fine range. Has iterators with the right properties. Has `begin()` and `end()` functions. It does not own the contents, but "ownership" is not part of the idea of a range.
- We could take this further by creating `views` which need not actually contain any objects of the sequence, but simply fake it when we dereference the iterators.
- Example: the standard view `std::views::iota(integer)` gives us an infinite sequence of integers starting at a given value.

FUN WITH RANGES AND VIEWS

```cpp
#include <ranges>
#include <iostream>
auto main() -> int {
    namespace sv = std::views;
    for (auto i : sv::iota(1UL) ) {
        if ((i+1) % 10000UL == 0UL) {
            std::cout << i << ' ';
            if ((i+1) % 100000UL == 0UL)
                std::cout << '\n';
            if (i >= 100000000UL) break;
        }
    }
}
```

- All containers are ranges, but not all ranges are containers
- `std::string_view` is a perfectly fine range. Has iterators with the right properties. Has `begin()` and `end()` functions. It does not own the contents, but “ownership” is not part of the idea of a range.
- We could take this further by creating `views` which need not actually contain any objects of the sequence, but simply fake it when we dereference the iterators.
- Example: the standard view `std::views::iota(integer)` gives us an infinite sequence of integers starting at a given value.

BORROWED RANGES

```cpp
// examples/dangling0.cc
auto get_vec() {
    std::vector v{ 2, 4, -1, 8, 0, 9 };
    return v;
}
auto main() -> int {
    auto v = get_vec();
    auto iter = std::min_element(v.begin(),
                                 v.end());
    std::cout << "Minimum " << *iter << "\n";
}
```

- The `min_element` function finds the minimum element in a range and returns an iterator

Example from a CPPCon 2020 talk by Tristan Brindle. Link.

BORROWED RANGES

```cpp
// examples/dangling0.cc
auto get_vec() {
    std::vector v{ 2, 4, -1, 8, 0, 9 };
    return v;
}
auto main() -> int {
    auto v = get_vec();
    auto iter = sr::min_element(v);

    std::cout << "Minimum " << *iter << "\n";
}
```

- The `min_element` function finds the minimum element in a range and returns an iterator
- The version from the ranges library takes only a range

Example from a CPPCon 2020 talk by Tristan Brindle. Link.

BORROWED RANGES

```cpp
// examples/dangling0.cc
auto get_vec() {
    std::vector v{ 2, 4, -1, 8, 0, 9 };
    return v;
}
auto main() -> int {

    auto iter = sr::min_element(get_vec());

    std::cout << "Minimum " << *iter << "\n";
}
```

- The `min_element` function finds the minimum element in a range and returns an iterator
- The version from the ranges library takes only a range
- It may be tempting to directly feed the output from a function to the algorithm. But, we would receive an iterator to a container that is already destructed, i.e., a dangling iterator. Dereferencing should therefore lead to a SEGFAULT.

Example from a CPPCon 2020 talk by Tristan Brindle. Link.

BORROWED RANGES

```cpp
// examples/dangling0.cc
auto get_vec() {
    std::vector v{ 2, 4, -1, 8, 0, 9 };
    return v;
}
auto main() -> int {

    auto iter = sr::min_element(get_vec());

    std::cout << "Minimum " << *iter << "\n";
}
```

- The `min_element` function finds the minimum element in a range and returns an iterator
- The version from the ranges library takes only a range
- It may be tempting to directly feed the output from a function to the algorithm. But, we would receive an iterator to a container that is already destructed, i.e., a dangling iterator. Dereferencing should therefore lead to a SEGFAULT.
- In reality, what happes is this!

Example from a CPPCon 2020 talk by Tristan Brindle. Link.

```
error: no match for ‘operator*’ (operand type is 'std::ranges::dangling')
   19 |     std::cout << "Minimum value is " << *iter << "\n";
```

BORROWED RANGES

```cpp
// examples/dangling0.cc
auto get_vec() {
    std::vector v{ 2, 4, -1, 8, 0, 9 };
    return v;
}
auto main() -> int {

    auto iter = sr::min_element(get_vec());

    std::cout << "Minimum " << *iter << "\n";
}
```

- The ranges algorithms are written with overloads such that when you pass an R-value reference of a container as input, the output type is `ranges::dangling`, an empty `struct` with no operations defined.
- `iter` here will be deduced to be of type `ranges::dangling`, and hence `*iter` leads to that insightful error message.

```
error: no match for 'operator*' (operand type is 'std::ranges::dangling')
   19 |     std::cout << "Minimum value is " << *iter << "\n";
```

- When the input was an L-value reference, the algorithm returning the iterator returned a valid iterator.
- Therefore: valid use cases work painlessly, and invalid ones result in actionable insights from the compiler!

BORROWED RANGES

```cpp
// examples/dangling1.cc
static std::vector u{2, 3, 4, -1, 9};
static std::vector v{3, 1, 4, 1, 5};
auto get_vec(int c) -> std::span<int> {
    return { (c % 2 == 0) ? u : v };
}
auto main(int argc, char* argv[]) -> int {
    auto iter = sr::min_element(get_vec(argc));
    // iter is valid, even if its parent span
    // has expired.
    std::cout << "Minimum " << *iter << "\n";
}
```

- Sometimes, an iterator can point to a valid element even when the "container" (imposter) has been destructed. `span`, `string_view` etc. do not own the elements in their range.
- No harm in returning real iterators of these objects, even if they are R-values. Even in this case, there is no danger of dangling.
- A `borrowed_range` is a range so that its iterators can be returned from a function without the danger of dangling, i.e.,
  it is an L-value reference or
  has been explicitly certified to be a borrowed range .

BORROWED RANGES

```cpp
// examples/dangling1.cc
static std::vector u{2, 3, 4, -1, 9};
static std::vector v{3, 1, 4, 1, 5};
auto get_vec(int c) -> std::span<int> {
    return { (c % 2 == 0) ? u : v };
}
auto main(int argc, char* argv[]) -> int {
    auto iter = sr::min_element(get_vec(argc));
    // iter is valid, even if its parent span
    // has expired.
    std::cout << "Minimum " << *iter << "\n";
}
```

- Sometimes, an iterator can point to a valid element even when the "container" (imposter) has been destructed. `span` , `string_view` etc. do not own the elements in their range.
- No harm in returning real iterators of these objects, even if they are R-values. Even in this case, there is no danger of dangling.
- A `borrowed_range` is a range so that its iterators can be returned from a function without the danger of dangling, i.e., it is an L-value reference or has been explicitly certified to be a borrowed range .

```cpp
template <class T>
concept borrowed_range = range<T> &&
          ( is_lvalue_reference_v<T> || enable_borrowed_range<remove_cvref_t<T>> )
```

VIEW ADAPTORS

```cpp
namespace sv = std::views;
std::vector v{1,2,3,4,5};
auto v3 = sv::take(v, 3);
// v3 is some sort of object so
// that it represents the first
// 3 elements of v. It does not
// own anything, and has constant
// time copy/move etc. It's a view.

// sv::take() is a view adaptor
```

- A `view` is a range with constant time copy, move etc. Think `string_view`
- A view adaptor is a function object, which takes a “viewable” range as an input and constructs a view out of it. `viewable` is defined as “either a `borrowed_range` or already a view.
- View adaptors in the `<ranges>` library have very interesting properties, and make some new ways of coding possible.

VIEW ADAPTORS

```
Adaptor(Viewable) -> View
Viewable | Adaptor -> View
V | A1 | A2 | A3 ... -> View

Adaptor(Viewable, Args...) -> View
Adaptor(Args...)(Viewable) -> View
Viewable | Adaptor(Args...) -> View
```

- A `view` itself is trivially viewable.
- Since a view adaptor produces a view, successive applications of such adaptors makes sense.
- If an adaptor takes only one argument, it can be called using the pipe operator as shown. These adaptors can then be chained to produce more complex adaptors.
- For adaptors taking multiple arguments, there exists an equivalent adaptor taking only the viewable range.

VIEW ADAPTORS

```
Adaptor(Viewable) -> View
Viewable | Adaptor -> View
V | A1 | A2 | A3 ... -> View

Adaptor(Viewable, Args...) ->  View
Adaptor(Args...)(Viewable) ->  View
Viewable | Adaptor(Args...) -> View
```

- A `view` itself is trivially viewable.
- Since a view adaptor produces a view, successive applications of such adaptors makes sense.
- If an adaptor takes only one argument, it can be called using the pipe operator as shown. These adaptors can then be chained to produce more complex adaptors.
- For adaptors taking multiple arguments, there exists an equivalent adaptor taking only the viewable range.

JÜLICH Forschungszentrum

VIEW ADAPTORS

```
Adaptor(Viewable) -> View
Viewable | Adaptor -> View
V | A1 | A2 | A3 ... -> View
```

```
Adaptor(Viewable, Args...) ->  View
Adaptor(Args...)(Viewable) ->  View
Viewable | Adaptor(Args...) -> View
```

- A `view` itself is trivially viewable.
- Since a view adaptor produces a view, successive applications of such adaptors makes sense.
- If an adaptor takes only one argument, it can be called using the pipe operator as shown. These adaptors can then be chained to produce more complex adaptors.
- For adaptors taking multiple arguments, there exists an equivalent adaptor taking only the viewable range.

VIEW ADAPTORS

```
Adaptor(Viewable) -> View
Viewable | Adaptor -> View
V | A1 | A2 | A3 ... -> View
```

```
Adaptor(Viewable, Args...) ->  View
Adaptor(Args...)(Viewable) ->  View
Viewable | Adaptor(Args...) -> View
```

- A `view` itself is trivially viewable.
- Since a view adaptor produces a view, successive applications of such adaptors makes sense.
- If an adaptor takes only one argument, it can be called using the pipe operator as shown. These adaptors can then be chained to produce more complex adaptors.
- For adaptors taking multiple arguments, there exists an equivalent adaptor taking only the viewable range.

VIEW ADAPTORS

```
Adaptor(Viewable) -> View
Viewable | Adaptor -> View
V | A1 | A2 | A3 ... -> View
```

```
Adaptor(Viewable, Args...) ->  View
Adaptor(Args...)(Viewable) ->  View
Viewable | Adaptor(Args...) -> View
```

- A `view` itself is trivially viewable.
- Since a view adaptor produces a view, successive applications of such adaptors makes sense.
- If an adaptor takes only one argument, it can be called using the pipe operator as shown. These adaptors can then be chained to produce more complex adaptors.
- For adaptors taking multiple arguments, there exists an equivalent adaptor taking only the viewable range.

So what are we going to do with this ?

VIEW ADAPTORS

Pretend that you want to verify that $sin^2(x) + cos^2(x) = 1$

VIEW ADAPTORS

Pretend that you want to verify that $sin^2(x) + cos^2(x) = 1$

- Start with a range of integers from 0 to $N = 10000$.
- $R_0 = \{0, 1, 2, 3...\}$

VIEW ADAPTORS

Pretend that you want to verify that $sin^2(x) + cos^2(x) = 1$

- Start with a range of integers from 0 to $N = 10000$.
- Map the integer range to real numbers in the range $[0, 2\pi)$

- $R_0 = \{0, 1, 2, 3...\}$
- $R_1 = T_{10} R_0 = T(n \mapsto \frac{n\pi}{N}) R_0$

VIEW ADAPTORS

Pretend that you want to verify that $sin^2(x) + cos^2(x) = 1$

- Start with a range of integers from 0 to $N = 10000$.
- Map the integer range to real numbers in the range $[0, 2\pi)$
- Evaluate $sin^2(x) + cos^2(x) - 1$ over the resulting range

- $R_0 = \{0, 1, 2, 3...\}$
- $R_1 = T_{10} R_0 = T(n \mapsto \frac{n\pi}{N}) R_0$
- $R_2 = T_{21} R_1 = T(x \mapsto (sin^2(x) + cos^2(x) - 1)) R_1$

$$\begin{aligned} R_2 &= T_{21} R_1 = T_{21} T_{10} R_0 \\ &= R_0 | T_{10} | T_{21} \\ &= R_0 | (T_{10} | T_{21}) \end{aligned}$$

VIEW ADAPTORS

Pretend that you want to verify that $sin^2(x) + cos^2(x) = 1$

- Start with a range of integers from 0 to $N = 10000$.
- Map the integer range to real numbers in the range $[0, 2\pi)$
- Evaluate $sin^2(x) + cos^2(x) - 1$ over the resulting range
- If absolute value of any of the values in the result exceeds ϵ, we have found a counter example

- $R_0 = \{0, 1, 2, 3...\}$
- $R_1 = T_{10} R_0 = T(n \mapsto \frac{n\pi}{N}) R_0$
- $R_2 = T_{21} R_1 = T(x \mapsto (sin^2(x) + cos^2(x) - 1)) R_1$

$$\begin{aligned} R_2 &= T_{21} R_1 = T_{21} T_{10} R_0 \\ &= R_0 | T_{10} | T_{21} \\ &= R_0 | (T_{10} | T_{21}) \end{aligned}$$

VIEW ADAPTORS

Pretend that you want to verify that $sin^2(x) + cos^2(x) = 1$

- Start with a range of integers from 0 to $N = 10000$.
- Map the integer range to real numbers in the range $[0, 2\pi)$
- Evaluate $sin^2(x) + cos^2(x) - 1$ over the resulting range
- If absolute value of any of the values in the result exceeds ϵ, we have found a counter example
- Intuitive left-to-right readability

- $R_0 = \{0, 1, 2, 3...\}$
- $R_1 = T_{10} R_0 = T(n \mapsto \frac{n\pi}{N}) R_0$
- $R_2 = T_{21} R_1 = T(x \mapsto (sin^2(x) + cos^2(x) - 1)) R_1$

$$\begin{aligned} R_2 &= T_{21} R_1 = T_{21} T_{10} R_0 \\ &= R_0 | T_{10} | T_{21} \\ &= R_0 | (T_{10} | T_{21}) \end{aligned}$$

VIEW ADAPTORS

Pretend that you want to verify that $sin^2(x) + cos^2(x) = 1$

- Start with a range of integers from 0 to $N = 10000$.
- Map the integer range to real numbers in the range $[0, 2\pi)$
- Evaluate $sin^2(x) + cos^2(x) - 1$ over the resulting range
- If absolute value of any of the values in the result exceeds ϵ, we have found a counter example
- Intuitive left-to-right readability

- $R_0 = \{0, 1, 2, 3...\}$
- $R_1 = T_{10} R_0 = T(n \mapsto \frac{n\pi}{N}) R_0$
- $R_2 = T_{21} R_1 = T(x \mapsto (sin^2(x) + cos^2(x) - 1)) R_1$

$$\begin{aligned} R_2 &= T_{21} R_1 = T_{21} T_{10} R_0 \\ &= R_0 | T_{10} | T_{21} \\ &= R_0 | (T_{10} | T_{21}) \end{aligned}$$

```
find . -name "*.cc" | xargs grep "if" | grep -v "constexpr" | less
```

VIEW ADAPTORS

```
find . -name "*.cc" | xargs grep "if" | grep -v "constexpr" | less
```

- Command line of Linux, Mac OS ...

VIEW ADAPTORS

```
find . -name "*.cc" | xargs grep "if" | grep -v "constexpr" | less
```

- Command line of Linux, Mac OS ...
- Small utilities. Each program does one thing, and does it well.

VIEW ADAPTORS

```
find . -name "*.cc" | xargs grep "if" | grep -v "constexpr" | less
```

- Command line of Linux, Mac OS ...
- Small utilities. Each program does one thing, and does it well.
- There is a way to chain them together with the pipe

VIEW ADAPTORS

```
find . -name "*.cc" | xargs grep "if" | grep -v "constexpr" | less
```

- Command line of Linux, Mac OS ...
- Small utilities. Each program does one thing, and does it well.
- There is a way to chain them together with the pipe
- Overall usefulness of the tool set is amplified exponentially!

VIEW ADAPTORS

```
find . -name "*.cc" | xargs grep "if" | grep -v "constexpr" | less
```

- Command line of Linux, Mac OS ...
- Small utilities. Each program does one thing, and does it well.
- There is a way to chain them together with the pipe
- Overall usefulness of the tool set is amplified exponentially!
- What about writing something similar in C++ ?

VIEW ADAPTORS

Pretend that you want to verify that $sin^2(x) + cos^2(x) = 1$

VIEW ADAPTORS

Pretend that you want to verify that $sin^2(x) + cos^2(x) = 1$

- Start with a range of integers from 0 to $N = 10000$. `R0 = iota(0, N)`

VIEW ADAPTORS

Pretend that you want to verify that $sin^2(x) + cos^2(x) = 1$

- Start with a range of integers from 0 to $N = 10000$. `R0 = iota(0, N)`
- Map the integer range to real numbers in the range $[0, 2\pi)$, i.e., perform the transformation $n \mapsto \frac{2\pi n}{N}$ over the range: `R1 = R0 | transform([](int n) -> double { return 2*pi*n/N; })`

JÜLICH Forschungszentrum

VIEW ADAPTORS

Pretend that you want to verify that $sin^2(x) + cos^2(x) = 1$

- Start with a range of integers from 0 to $N = 10000$. `R0 = iota(0, N)`
- Map the integer range to real numbers in the range $[0, 2\pi)$, i.e., perform the transformation $n \mapsto \frac{2\pi n}{N}$ over the range: `R1 = R0 | transform([](int n) -> double { return 2*pi*n/N; })`
- `R2 = R1 | transform([](double x) -> double { return sin(x)*sin(x)+cos(x)*cos(x); } );`

VIEW ADAPTORS

Pretend that you want to verify that $sin^2(x) + cos^2(x) = 1$

- Start with a range of integers from 0 to $N = 10000$. `R0 = iota(0, N)`
- Map the integer range to real numbers in the range $[0, 2\pi)$, i.e., perform the transformation $n \mapsto \frac{2\pi n}{N}$ over the range: `R1 = R0 | transform([](int n) -> double { return 2*pi*n/N; })`
- `R2 = R1 | transform([](double x) -> double { return sin(x)*sin(x)+cos(x)*cos(x); } );`
- If absolute value of any of the values in the result exceeds ϵ, we have found a counter example `if (any_of(R2, [](auto x){return fabs(x) > eps;})) ...`

JÜLICH Forschungszentrum

VIEW ADAPTORS

```cpp
auto main() -> int {
    namespace sr = std::ranges;
    namespace sv = std::views;
    const auto pi = std::acos(-1);
    constexpr auto npoints = 10'000'00UL;
    constexpr auto eps = 100 * std::numeric_limits<double>::epsilon();
    auto to_0_2pi = [=](size_t idx) -> double {
        return std::lerp(0., 2*pi, idx * 1.0 / npoints);
    };
    auto x_to_fx = [ ](double x) -> double {
        return sin(x) * sin(x) + cos(x) * cos(x) - 1.0;
    };
    auto is_bad = [=](double x){ return std::fabs(x) > eps; };

    auto res = sv::iota(0UL, npoints) | sv::transform(to_0_2pi)
                | sv::transform(x_to_fx);
    if (sr::any_of(res, is_bad) ) {
        std::cerr << "The relation does not hold.\n";
    } else {
        std::cout << "The relation holds for all inputs\n";
    }
}
```


JÜLICH Forschungszentrum

VIEW ADAPTORS

- The job of each small transform in the previous example was small, simple, easily verified for correctness.

VIEW ADAPTORS

- The job of each small transform in the previous example was small, simple, easily verified for correctness.
- The view adaptors allow us to chain them to produce a resulting range

VIEW ADAPTORS

- The job of each small transform in the previous example was small, simple, easily verified for correctness.
- The view adaptors allow us to chain them to produce a resulting range
- Algorithms like `std::range::any_of` work on ranges, so they can work on the views resulting from chained view adaptors.

VIEW ADAPTORS

- The job of each small transform in the previous example was small, simple, easily verified for correctness.
- The view adaptors allow us to chain them to produce a resulting range
- Algorithms like `std::range::any_of` work on ranges, so they can work on the views resulting from chained view adaptors.
- No operation is done on any range when we create the variable `res` above.

VIEW ADAPTORS

- The job of each small transform in the previous example was small, simple, easily verified for correctness.
- The view adaptors allow us to chain them to produce a resulting range
- Algorithms like `std::range::any_of` work on ranges, so they can work on the views resulting from chained view adaptors.
- No operation is done on any range when we create the variable `res` above.
- When we try to access an element of the range in the `any_of` algorithm, one element is taken on the fly out of the starting range, fed through the pipeline and catered to `any_of`

VIEW ADAPTORS

- The job of each small transform in the previous example was small, simple, easily verified for correctness.
- The view adaptors allow us to chain them to produce a resulting range
- Algorithms like `std::range::any_of` work on ranges, so they can work on the views resulting from chained view adaptors.
- No operation is done on any range when we create the variable `res` above.
- When we try to access an element of the range in the `any_of` algorithm, one element is taken on the fly out of the starting range, fed through the pipeline and catered to `any_of`
- `any_of` does not process the range beyond what is necessary to establish its truth value. The remaining elements in the result array are never calculated.

Exercise 1.1:

The code used for the demonstration of view adaptors is `examples/trig_views.cc`. Build this code with GCC and Clang. As of version 14.0 of Clang, parts of the `<ranges>` header we use are not implemented. We are therefore going to use a redirecting header `<cxx20ranges>` examples. When the compiler implements the ranges library, it includes `<ranges>`. Otherwise, it tries to include equivalent headers from the `rangev3` library. It also defines alias namespaces `sr` and `sv` for `std::ranges` and `std::std::views`. To compile, you would need to have the location of this redirecting header in your include path:

```
g++ -std=c++20 -I course_home/local/include trig_views.cc
./a.out

clang++ -std=c++20 -stdlib=libc++ -I course_home/local/include trig_views.cc
./a.out
```

Exercise 1.2:

The trigonometric relation we used is true, so not all possibilities are explored. In `examples/trig_views2.cc` there is another program trying to verify the bogus claim $sin^2(x) < 0.99$. It's mostly true, but sometimes it isn't, so that our `if` and `else` branches both have work to do. The lambdas in this program have been rigged to print messages before returning. Convince yourself of the following:

- The output from the lambdas come out staggered, which means that the program does not process the entire range for the first transform and then again for the second ...
- Processing stops at the first instance where `any_of` gets a `true` answer.

VIEW ADAPTORS

```cpp
// examples/gerund.cc
    using itertype = std::istream_iterator<std::string>;
    std::ifstream fin { argv[1] };
    auto gerund = [](std::string_view w) { return w.ends_with("ing"); };
    auto in = sr::istream_view<std::string>(fin);
    std::cout << (in | sv::filter(gerund)) << "\n";
```

- `sr::istream_view<T>` creates an (input) iterable range from an input stream. Each element of this range is of the type `T`.
- `sv::filter` is a view adaptor, which when applied to a range, produces another containing only the elements satisfying a given condition
- In the above, `std::cout` is shown writing out a range. This works via a separate header file included in `gerund.cc` called `range_output.hh`, which is provided to you with the course material. Ranges in C++20 are not automatically streamable to the standard output.

VIEW ADAPTORS

A program to print the alphabetically first and last word entered on the command line, excluding the program name.

```
// examples/views_and_span.cc
auto main(int argc, char* argv[]) -> int
{
    if (argc < 2) return 1;
    namespace sr = std::ranges;
    namespace sv = std::views;

    std::span args(argv, argc);
    auto str = [](auto cstr) -> std::string_view { return cstr; };
    auto [mn, mx] = sr::minmax(args | sv::drop(1) | sv::transform(str));

    std::cout << "Alphabetically first = " << mn << " last = " << mx << "\n";
}
```

FORMATTED OUTPUT

```cpp
for (auto i = 0UL; i < 100UL; ++i) {
    std::cout << "i = " << i
        << ", E_1 = " << cos(i * wn)
        << ", E_2 = " << sin(i * wn)
        << "\n";
}
```

```
i = 5, E_1 = 0.55557, E_2 = 0.83147
i = 6, E_1 = 0.382683, E_2 = 0.92388
i = 7, E_1 = 0.19509, E_2 = 0.980785
i = 8, E_1 = 6.12323e-17, E_2 = 1
i = 9, E_1 = -0.19509, E_2 = 0.980785
i = 10, E_1 = -0.382683, E_2 = 0.92388
i = 11, E_1 = -0.55557, E_2 = 0.83147
```

- While convenient and type safe and extensible, the interface of `ostream` objects like `std::cout` isn't by itself conducive to regular well-formatted output

FORMATTED OUTPUT

```
for (auto i = 0UL; i < 100UL; ++i) {
    std::cout << "i = " << i
        << ", E_1 = " << cos(i * wn)
        << ", E_2 = " << sin(i * wn)
        << "\n";
}
```

```
i = 5, E_1 = 0.55557, E_2 = 0.83147
i = 6, E_1 = 0.382683, E_2 = 0.92388
i = 7, E_1 = 0.19509, E_2 = 0.980785
i = 8, E_1 = 6.12323e-17, E_2 = 1
i = 9, E_1 = -0.19509, E_2 = 0.980785
i = 10, E_1 = -0.382683, E_2 = 0.92388
i = 11, E_1 = -0.55557, E_2 = 0.83147
```

- While convenient and type safe and extensible, the interface of `ostream` objects like `std::cout` isn't by itself conducive to regular well-formatted output
- C `printf` often has a simpler path towards visually uniform columnar output, although it is neither type safe nor extensible
- C++ `<iomanip>` header allows formatting with a great deal of control, but has a verbose and inconsistent syntax

FORMATTED OUTPUT

```cpp
for (auto i = 0UL; i < 100UL; ++i) {
    std::cout << fmt::format(
        "i = {:>4d}, E_1 = {:< 12.8f}, "
        "E_2 = {:< 12.8f}\n",
         i, cos(i * wn), sin(i * wn));
}
```

```
i =    5, E_1 =  0.55557023 , E_2 =  0.83146961
i =    6, E_1 =  0.38268343 , E_2 =  0.92387953
i =    7, E_1 =  0.19509032 , E_2 =  0.98078528
i =    8, E_1 =  0.00000000 , E_2 =  1.00000000
i =    9, E_1 = -0.19509032 , E_2 =  0.98078528
i =   10, E_1 = -0.38268343 , E_2 =  0.92387953
i =   11, E_1 = -0.55557023 , E_2 =  0.83146961
```

- While convenient and type safe and extensible, the interface of `ostream` objects like `std::cout` isn't by itself conducive to regular well-formatted output
- C `printf` often has a simpler path towards visually uniform columnar output, although it is neither type safe nor extensible
- C++ `<iomanip>` header allows formatting with a great deal of control, but has a verbose and inconsistent syntax
- C++20 introduced the `<format>` header, which introduces Python like string formatting
- Based on the open source `fmt` library.

FORMATTED OUTPUT

```cpp
for (auto i = 0UL; i < 100UL; ++i) {
    std::cout << fmt::format(
        "i = {:>4d}, E_1 = {:< 12.8f}, "
        "E_2 = {:< 12.8f}\n",
         i, cos(i * wn), sin(i * wn));
}
```

```
i =    5, E_1 =  0.55557023 , E_2 =  0.83146961
i =    6, E_1 =  0.38268343 , E_2 =  0.92387953
i =    7, E_1 =  0.19509032 , E_2 =  0.98078528
i =    8, E_1 =  0.00000000 , E_2 =  1.00000000
i =    9, E_1 = -0.19509032 , E_2 =  0.98078528
i =   10, E_1 = -0.38268343 , E_2 =  0.92387953
i =   11, E_1 = -0.55557023 , E_2 =  0.83146961
```

Perfectly aligned, as all numeric output should be.

- While convenient and type safe and extensible, the interface of `ostream` objects like `std::cout` isn't by itself conducive to regular well-formatted output
- C `printf` often has a simpler path towards visually uniform columnar output, although it is neither type safe nor extensible
- C++ `<iomanip>` header allows formatting with a great deal of control, but has a verbose and inconsistent syntax
- C++20 introduced the `<format>` header, which introduces Python like string formatting
- Based on the open source `fmt` library.
- Elegant. Safe. Fast. Extensible.

FORMATTED OUTPUT

```cpp
// Example of a redirecting header
#include <version>
#ifdef __cpp_lib_format
  #include <format>
  namespace fmt = std;
#elif __has_include(<fmt/format.h>)
  #define FMT_HEADER_ONLY
  #include <fmt/core.h>
  #include <fmt/format.h>
#else
  #error No suitable header for C++20 format!
#endif
```

- As of May 2022, no implementation in GCC
- We can use a redirecting header to use the `fmt` library when the compiler does not have the library feature
- Code simplification and compilation (and runtime) speed $\Longrightarrow$ useful to learn it. Eventually all compilers will have it.

FORMATTED OUTPUT

- `std::format("format string {}, {} etc.", args...)` takes a compile time constant format string and a parameter pack to produce a formatted output string
- `std::vformat` can be used if the format string is not known at compilation time
- If instead of receiving output as a newly created string, output into a container or string is desired, `std::format_to` or `std::format_to_n` are available
- The format string contains python style placeholder braces to be filled with formatted values from the argument list
- The braces can optionally contain `id : spec` descriptors. `id` is a 0 based index to choose an argument from `args...` for that slot. `spec` controls how the value is to be written: width, precision, alignment, padding, base of numerals etc. Details of the format specifiers can be found here!

Example 1.4:

A simple example demonstrating the text formatting library of C++20 is in `examples/format1.cc`. When this C++20 header is not available in the standard library implementation, we use headers from the `fmt` library giving us approximately the same functionality. Although `fmt` is usually compiled to a static or shared library to link, we define the macro `FMT_HEADER_ONLY` to pretend that we got everything from the standard library.

Example 1.5:

The program `examples/word_count.cc` is an improved version of the word counter program from day 4. Here we clear any trailing non-alphabetic characters from strings read as words, e.g., treat "instance," as "instance". We use the ranges algorithms to clean up the string. We then use the formatting library to write the histogram.

REGULAR EXPRESSIONS USING C++20

```
constexpr ctll::fixed_string re{ R"xpr(^(https:|http:|www\.)\S*)xpr" };
auto urls_in_input = args | sv::drop(1)
                          | sv::transform([=](auto inp) { return str(inp); })
                          | sv::filter([re](auto inp) { return ctre::search<re>(inp); });
if (auto m = ctre::match<trx>(diststr); m) {
    auto numstr = m.get<1>().to_string();
    // and so on...
}
```

- CTRE: "Compile time regular expressions", header only open source library
- Regular expressions parsed at compile time.
- Smaller binaries than `std::regex`
- Syntax makes excellent use of C++20 features for intuitive handling of regular expressions
- Compile time regex processing is possible, with great performance

REGULAR EXPRESSIONS USING CTRE

Example 1.6:

`examples/dist.cc` contains a rudimentary Distance class. Distances can be constructed by giving a value with a unit. Overloaded literal operators allow writing code like `auto d = 14.5_km;`. It is possible to write distances using `std::cout`, or read using `std::cin`. E.g.,

```
$ Enter distance: 13.99_cm
That is 0.1399_m

$ Enter distance: "23   km"
That is 23000_m
```

To read and interpret the input string in the correct units, we make use of regular expressions. Since these can be known at when writing the source code, we use the CTRE library to process our regular expressions. The example demonstrates many different topics explored during the course.

Modules

A PREVIEW OF C++20 MODULES

Traditionally, C++ projects are organised into header and source files. As an example, consider a simple `saxpy` program ...

```cpp
#ifndef SAXPY_HH
#define SAXPY_HH
#include <algorithm>
#include <span>
template <class T> concept Number = std::floating_point<T> or std::integral<T>;
template <class T> requires Number<T>
auto saxpy(T a, std::span<const T> x, std::span<const T> y, std::span<T> z){
    std::transform(x.begin(), x.end(), y.begin(), z.begin(),
        [a](T X, T Y) { return a * X + Y; });
}
#endif
```

```cpp
#include "saxpy.hh"
auto main() -> int {
    //declarations
    saxpy(10., {inp1}, {inp2}, {outp});
}
```

PROBLEMS WITH HEADER FILES

- Headers contain declarations of functions, classes etc., and definitions of inline functions.
- Source files contain implementations of other functions, such as `main`.
- Since function templates and class templates have to be visible to the compiler at the point of instantiation, these have traditionally lived in headers.
- Standard library, TBB, Thrust, Eigen ... a lot of important C++ libraries consist of a lot of template code, and therefore in header files.
- The `#include <abc>` mechanism is essentially a copy-and-paste solution. The preprocessor inserts the entire source of the headers in each source file that includes it, creating large translation units.
- The same template code gets re-parsed over and over for every new tranlation unit.
- If the headers contain expression templates, CRTP, metaprogramming repeated processing of the templates is a waste of resources.

MODULES

- The `module` mechanism in C++20 offers a better organisation
- All code, including template code can now reside in source files
- Module source files will be processed once to produce "precompiled modules", where the essential syntactic information has been parsed and saved.
- These compiled module interface (binary module interface) files are to be treated as caches generated during the compilation process. There are absolutely no guarantees of them remaining compatible between different versions of the same compiler, different machine configurations etc.
- Any source `import`ing the module immediately has access to the precompiled syntax tree in the precompiled module files. This leads to faster compilation

USING MODULES

```cpp
// examples/hello_m.cc
import <iostream>;

auto main() -> int
{
    std::cout << "Hello, world!\n";
}
```

- If a module is available, not much special needs to be done to use it. `import` the module instead of `#include`ing a header. Use the exported classes, functions and variables from the module.
- As of C++20, the standard library is not available as a module. But standard library headers can be imported as “header units”.

```
$ clang++ -std=c++20 -stdlib=libc++ -fmodules hello_m.cc
$ ./a.out
$ g++ -std=c++20 -fmodules-ts -xc++-system-header iostream
$ g++ -std=c++20 -fmodules-ts hello_m.cc
$ ./a.out
$
```

- GCC requires that the header units needed are first generated in a separate explicit step.
- If `iostream` had been the name of a module, we would have written `import iostream;` instead of `import <iostream>`

USING MODULES

Example 1.7:

Convert a few of the example programs you have seen during the course to use modules syntax instead. At the moment it means no more than replacing the `#include` lines with the corresponding `import` lines for the standard library headers. The point is to get used to the extra compilation options you need with modules at the moment. Use, for instance, the date time library functions like `feb.cc` and `advent.cc` from the day 4 examples.

CREATING A MODULE (EXAMPLE)

```cpp
// saxpy.hh
#ifndef SAXPY_HH
#define SAXPY_HH
#include <algorithm>
#include <span>

template <class T>
concept Number = std::floating_point<T>
              or std::integral<T>;
template <Number T>
auto saxpy(T a, std::span<const T> x,
           std::span<const T> y,
           std::span<T> z)
{
    std::transform(x.begin(), x.end(),
                   y.begin(), z.begin(),
        [a](T X, T Y) {
            return a * X + Y;
        });
}
#endif
```

- A header file contains a function template `saxpy`, and a `concept` necessary to define that function
- A source file, `main.cc` which includes the header and uses the function

CREATING A MODULE (EXAMPLE)

```cpp
// usesaxpy.cc
#include <iostream>
#include <array>
#include <vector>
#include <span>
#include "saxpy.hh"

auto main() -> int
{
    using namespace std;
    const array inp1 { 1., 2., 3., 4., 5. };
    const array inp2 { 9., 8., 7., 6., 5. };
    vector outp(inp1.size(), 0.);

    saxpy(10., {inp1}, {inp2}, {outp});
    for (auto x : outp) cout << x << "\n";
    cout << ":::::::::::::::::::::\n";
}
```

- A header file contains a function template `saxpy`, and a **concept** necessary to define that function
- A source file, `main.cc` which includes the header and uses the function

CREATING A MODULE (EXAMPLE)

```cpp
// saxpy.hh -> saxpy.ixx
#ifndef SAXPY_HH
#define SAXPY_HH
#include <algorithm>
#include <span>

template <class T>
concept Number = std::floating_point<T>
              or std::integral<T>;
template <Number T>
auto saxpy(T a, std::span<const T> x,
           std::span<const T> y,
           std::span<T> z)
{
    std::transform(x.begin(), x.end(),
                   y.begin(), z.begin(),
        [a](T X, T Y) {
            return a * X + Y;
        });
}
#endif
```

Make a module interface unit

CREATING A MODULE (EXAMPLE)

```
// saxpy.hh -> saxpy.ixx
#ifndef SAXPY_HH
#define SAXPY_HH
#include <algorithm>
#include <span>

template <class T>
concept Number = std::floating_point<T>
              or std::integral<T>;
template <Number T>
auto saxpy(T a, std::span<const T> x,
           std::span<const T> y,
           std::span<T> z)
{
    std::transform(x.begin(), x.end(),
                   y.begin(), z.begin(),
        [a](T X, T Y) {
            return a * X + Y;
        });
}
#endif
```

Make a module interface unit

- Include guards are no longer required, since importing a module does not transitively import things used inside the module

CREATING A MODULE (EXAMPLE)

```
// saxpy.hh -> saxpy.ixx


#include <algorithm>
#include <span>

template <class T>
concept Number = std::floating_point<T>
              or std::integral<T>;
template <Number T>
auto saxpy(T a, std::span<const T> x,
           std::span<const T> y,
           std::span<T> z)
{
    std::transform(x.begin(), x.end(),
                   y.begin(), z.begin(),
        [a](T X, T Y) {
            return a * X + Y;
        });
}
```

Make a module interface unit

- Include guards are no longer required, since importing a module does not transitively import things used inside the module

CREATING A MODULE (EXAMPLE)

```cpp
// saxpy.hh -> saxpy.ixx

export module saxpy;
#include <algorithm>
#include <span>

template <class T>
concept Number = std::floating_point<T>
              or std::integral<T>;
template <Number T>
auto saxpy(T a, std::span<const T> x,
           std::span<const T> y,
           std::span<T> z)
{
    std::transform(x.begin(), x.end(),
                   y.begin(), z.begin(),
        [a](T X, T Y) {
            return a * X + Y;
        });
}
```

Make a module interface unit

- Include guards are no longer required, since importing a module does not transitively import things used inside the module
- A module interface unit is a file which exports a module

CREATING A MODULE (EXAMPLE)

```
// saxpy.hh -> saxpy.ixx

export module saxpy;
import <algorithm>;
import <span>;

template <class T>
concept Number = std::floating_point<T>
              or std::integral<T>;
template <Number T>
auto saxpy(T a, std::span<const T> x,
           std::span<const T> y,
           std::span<T> z)
{
    std::transform(x.begin(), x.end(),
                   y.begin(), z.begin(),
        [a](T X, T Y) {
            return a * X + Y;
        });
}
```

Make a module interface unit

- Include guards are no longer required, since importing a module does not transitively import things used inside the module
- A module interface unit is a file which exports a module
- Replace `#include` lines with corresponding `import` lines. Obs: `import` lines end with a semi-colon!

CREATING A MODULE (EXAMPLE)

```
// saxpy.hh -> saxpy.ixx

export module saxpy;
import <algorithm>;
import <span>;

template <class T>
concept Number = std::floating_point<T>
                 or std::integral<T>;
export template <Number T>
auto saxpy(T a, std::span<const T> x,
           std::span<const T> y,
           std::span<T> z)
{
    std::transform(x.begin(), x.end(),
                   y.begin(), z.begin(),
        [a](T X, T Y) {
            return a * X + Y;
        });
}
```

Make a module interface unit

- Include guards are no longer required, since importing a module does not transitively import things used inside the module
- A module interface unit is a file which exports a module
- Replace `#include` lines with corresponding `import` lines. Obs: `import` lines end with a semi-colon!
- Explicitly export any definitions (classes, functions...) you want for users of the module. Anything not exported by a module is automatically private to the module

JÜLICH Forschungszentrum

CREATING A MODULE (EXAMPLE)

Use your module

```
// usesaxpy.cc
#include <iostream>
#include <array>
#include <vector>
#include <span>
#include "saxpy.hh"

auto main() -> int
{
    using namespace std;
    const array inp1 { 1., 2., 3., 4., 5. };
    const array inp2 { 9., 8., 7., 6., 5. };
    vector outp(inp1.size(), 0.);

    saxpy(10., {inp1}, {inp2}, {outp});
    for (auto x : outp) cout << x << "\n";
    cout << ":::::::::::::::::::::\n";
}
```

CREATING A MODULE (EXAMPLE)

```
// usesaxpy.cc
import <iostream>
import <array>
import <vector>
import <span>
#include "saxpy.hh"

auto main() -> int
{
    using namespace std;
    const array inp1 { 1., 2., 3., 4., 5. };
    const array inp2 { 9., 8., 7., 6., 5. };
    vector outp(inp1.size(), 0.);

    saxpy(10., {inp1}, {inp2}, {outp});
    for (auto x : outp) cout << x << "\n";
    cout << ":::::::::::::::::::::\n";
}
```

Use your module

- Replace `#include` lines with corresponding `import` lines. Obs: `import` lines end with a semi-colon!

CREATING A MODULE (EXAMPLE)

```cpp
// usesaxpy.cc
#import <iostream>
#import <array>
#import <vector>
#import <span>
import saxpy;

auto main() -> int
{
    using namespace std;
    const array inp1 { 1., 2., 3., 4., 5. };
    const array inp2 { 9., 8., 7., 6., 5. };
    vector outp(inp1.size(), 0.);

    saxpy(10., {inp1}, {inp2}, {outp});
    for (auto x : outp) cout << x << "\n";
    cout << ":::::::::::::::::::::\n";
}
```

Use your module

- Replace `#include` lines with corresponding `import` lines. Obs: `import` lines end with a semi-colon!
- When importing actual modules, rather than header units, use the module name without angle brackets or quotes

CREATING A MODULE (EXAMPLE)

```cpp
// usesaxpy.cc
#import <iostream>
#import <array>
#import <vector>
#import <span>
import saxpy;

auto main() -> int
{
    using namespace std;
    const array inp1 { 1., 2., 3., 4., 5. };
    const array inp2 { 9., 8., 7., 6., 5. };
    vector outp(inp1.size(), 0.);

    saxpy(10., {inp1}, {inp2}, {outp});
    for (auto x : outp) cout << x << "\n";
    cout << ":::::::::::::::::::::\n";
}
```

Use your module

- Replace `#include` lines with corresponding `import` lines. Obs: `import` lines end with a semi-colon!
- When importing actual modules, rather than header units, use the module name without angle brackets or quotes
- Importing `saxpy` here, only grants us access to the explicitly exported function `saxpy`. Not other functions, classes, concepts etc. defined in the module `saxpy`, not any other module imported in the module interface unit.

COMPILATION OF PROJECTS WITH MODULES

- Different compilers require different (sets of) options
- GCC:
  - Auto-detects if a file is a module interface unit (exports a module), and generates the CMI as well as an object file together.
  - No special file extension required for module interface units(Just `.cc` , `.cpp` , . . . like regular source files).
  - Requires that standard library header units needed by the project are explicitly generated
  - Does not recognise module interface file extensions used by other compilers (`.ixx` , `.ccm` etc.)
  - Rather crashy at the moment, even for toy code
- Clang:
  - Provides standard library header units.
  - Comparatively stable for module based code.
  - Lots of command line options required
  - Different options to translate module interfaces depending on file extensions!
    - `.ccm` or `.cppm` : `--precompile`
    - `.ixx` : `--precompile -xc++-module`
    - `.cc` or `.cpp` : `-Xclang -emit-module-interface`
  - Separate generation of object file required
  - Module partitions not implemented

JÜLICH Forschungszentrum

Exercise 1.3:

Versions of the `saxpy` program written using header files and then modules can be found in the `examples/modules/saxpy/`. The recipe for building is described in the README.md files in the respective sub-folders. Familiarize yourself with the process of building applications with modules. Experiment by writing a new inline function in the module interface file without exporting it. Try to call that function from `main`. Check again after exporting it in the module.

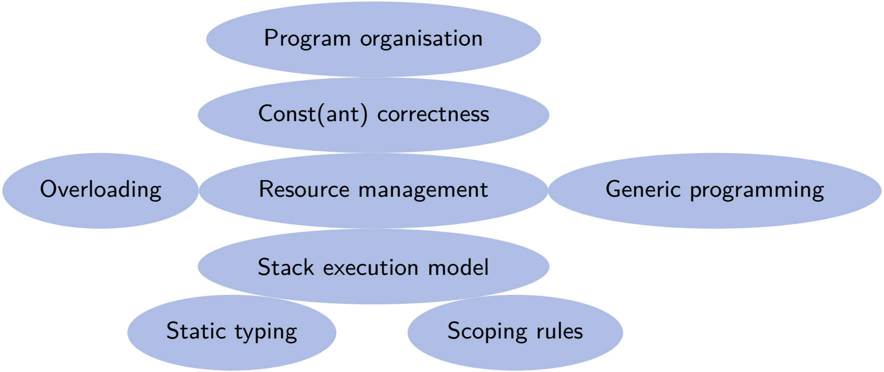
Exercise 1.4:

As a more complicated example, we have in `examples/modules/2_any` the second version of our container with polymorphic geometrical objects. The header and source files for each class `Point`, `Circle` etc have been rewritten for modules. Compare the two versions, build them, run them. The recipes for building are in the README.md files.
PS: GCC almost succeeds here. Clang should have no difficulties.

JÜLICH Forschungszentrum

Closing remarks

C++ "GENES"


Program organisation
Const(ant) correctness
Overloading
Resource management
Generic programming
Stack execution model
Static typing
Scoping rules

CLOSING REMARKS

- Most examples were simply demo code to show you how it works

CLOSING REMARKS

- Most examples were simply demo code to show you how it works
- To really internalise the ideas, you have to solve those or similar problems yourself

CLOSING REMARKS

- Most examples were simply demo code to show you how it works
- To really internalise the ideas, you have to solve those or similar problems yourself
- Information summarised for you, so that you hear it once, and then hopefully when you need it, you will remember having heard about a feature, and then look it up

CLOSING REMARKS

- Most examples were simply demo code to show you how it works
- To really internalise the ideas, you have to solve those or similar problems yourself
- Information summarised for you, so that you hear it once, and then hopefully when you need it, you will remember having heard about a feature, and then look it up
- Rapidly evolving language

CLOSING REMARKS

- Most examples were simply demo code to show you how it works
- To really internalise the ideas, you have to solve those or similar problems yourself
- Information summarised for you, so that you hear it once, and then hopefully when you need it, you will remember having heard about a feature, and then look it up
- Rapidly evolving language
- en.cppreference.com

CLOSING REMARKS

- Most examples were simply demo code to show you how it works
- To really internalise the ideas, you have to solve those or similar problems yourself
- Information summarised for you, so that you hear it once, and then hopefully when you need it, you will remember having heard about a feature, and then look it up
- Rapidly evolving language
- en.cppreference.com
- isocpp.org

CLOSING REMARKS

- Most examples were simply demo code to show you how it works
- To really internalise the ideas, you have to solve those or similar problems yourself
- Information summarised for you, so that you hear it once, and then hopefully when you need it, you will remember having heard about a feature, and then look it up
- Rapidly evolving language
- en.cppreference.com
- isocpp.org
- YouTube channel: Jason Turner's C++ weekly

JÜLICH Forschungszentrum

CLOSING REMARKS

- Most examples were simply demo code to show you how it works
- To really internalise the ideas, you have to solve those or similar problems yourself
- Information summarised for you, so that you hear it once, and then hopefully when you need it, you will remember having heard about a feature, and then look it up
- Rapidly evolving language
- en.cppreference.com
- isocpp.org
- YouTube channel: Jason Turner's C++ weekly
- YouTube channel: CppCon conference talks