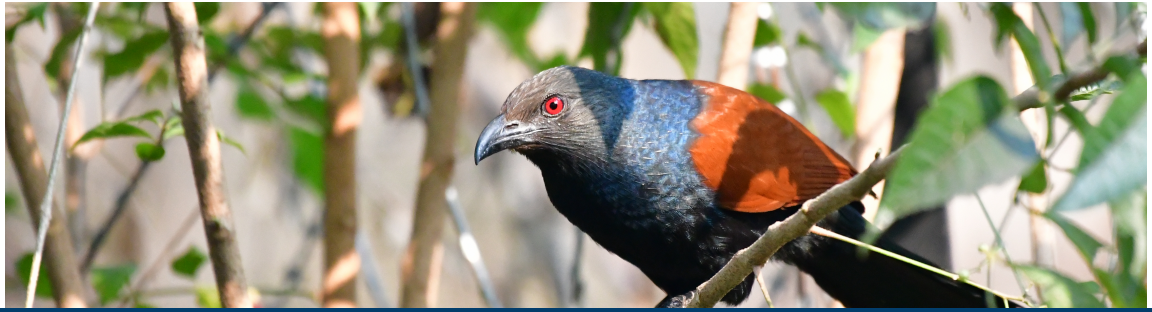


High performance scientific computing in C++

HPC C++ Course 2024

28 October – 31 October 2024 | Sandipan Mohanty | Forschungszentrum Jülich, Germany

JÜLICH Forschungszentrum

Course material

- Logging in to our system as described in the following slides and running the `setup.sh` script downloads/updates the course material.
- If you want to get hold of the material outside the scope of the course, here is the git repo: https://gitlab.jsc.fz-juelich.de/sdlbio-courses/hpcxx2024.git

High performance scientific computing in C++

Supercomputer access for the exercises

- Please login to the Jupyter-JSC system
- After logging in, try to add a new jupyterlab. Choose JUSUF as the system and training2444 as the project.
- For the partition choose **LoginNode**, and then start. Wait until the swirly things stop and you see the panel.
- What we will most need from there is the terminal, which should be at the bottom.
- In the terminal type this:

```
$ source $PROJECT/local/setup.sh
```

- After this, your paths should be set correctly. Test it using

```
$ g++ --version
$ clang++ --version
```

You should see GCC version 15.0 and Clang version 20.0, both compiled from source obtained from their respective git repositories.

Direct SSH connection to JUSUF

- Follow the instructions in the page https://apps.fz-juelich.de/jsc/hps/jusuf/access.html#ssh-login to generate a suitable SSH key for JUSUF. Certain key types are preferred, and the recommended key type is ed25519.

```
$ ssh-keygen -a 100 -t ed25519 -f ~/.ssh/id_ed25519_jsc
$
```

- Upload the generated **public key** to JuDoor
  - Login to JuDoor
  - Click on "Manage SSH-keys" next to JUSUF
  - Select your ed25519 public key file using the "Browse" button. This file has a ".pub" extension.
  - Pay special attention to the "from clause". This goes in the box to the right of the public key filename box. If you are in a hurry, use your current IP address, which is displayed in the colour coded field at the top in the section "Upload your public keys"
  - After filling in a suitable "from clause", click on the Start upload of SSH-keys button
- Wait

Direct SSH connection to JUSUF

- Open SSH session on JUSUF using your ed25519 key like this:

```
$  ssh -i ~/.ssh/id_ed25519_jsc <yourid>@jusuf.fz-juelich.de
```

- ... or, make it a little easier for yourself:
  - Add a section for JUSUF in your SSH config file `~/.ssh/config`

```
1  Host jusuf jusuf??
2    Hostname %h.fz-juelich.de
3    Match Host jusuf.fz-juelich.de,jusuf??.fz-juelich.de
4    User yourid
5    IdentityFile ~/.ssh/id_ed25519_jsc
6    ServerAliveInterval 60
```

  - Open SSH session like this: `$ ssh jusuf`
- After opening a new SSH session, set your project to training2316 and run the setup script:

```
$ jutil env activate -p training2444
$ source $PROJECT/local/setup.sh
$
```

High performance scientific computing in C++

- The setup script must be run at the beginning of every new login to JUSUF for this course.
- It creates user specific working directories, downloads and updates course material and sets up the environment variables for compilers and libraries.
- After the script `setup.sh` is sourced, the following environment variables (EV) and additional shortcuts (SC) are available
  - `cxx2024`: (EV) Location of your private working area for the course
  - `swhome`: (EV) Top level folder for software installations for compilers and libraries
  - `cdp`: (SC) Change directory to the top level of your private workspace
  - `pathadd`: (SC) Prepend a new folder to PATH. E.g., `pathadd /x/y/z/bin`
  - `pathrm`: (SC) Remove a folder from PATH
  - `libpathadd`, `libpathrm`: (SC) Same as above, but for `LD_LIBRARY_PATH`, `LD_RUN_PATH`, `LIBRARY_PATH`
  - `incpathadd`, `incpathrm`: (SC) Same, but for `CPATH`, which is searched by the compilers for include files.
  - `cmpathadd`, `cmpathrm`: Same, but for `CMAKE_PREFIX_PATH`
  - `G`: (SC) Alias for `g++` using common options `-std=c++23 -pedantic -Wall -O3`
  - `C`: (SC) Similar to `G`, but for Clang. It also uses Clang's own implementation of the standard library, `libc++`
  - `I`: (SC) Similar to `G`, but for the Intel compiler. It also uses gcc's implementation of the standard library. To use clang's standard library with the intel compiler, use the shotcut `Ic`

Some notes on the organisation of the course material

- The folder `yourworkspace/software`: Any software you build and install with this installation prefix will be found by the compilers and CMake
- Run simple compilation and small programs on the Login node, as you would on your laptop.
- For heavier workloads, we will use the batch system during the course. Run the executable `a.out` using 64 maximum threads on a JUSUF compute node as follows:
  `batch_run --cpus-per-task=64 a.out [OPTIONS]` The alias `batch_run` will be updated for each course day to always use the reservations made on the supercomputer for that course day.
- To run multiple examples on a compute node, use the shortcut `node`. This takes a bit longer, but then starts a BASH session on a compute node for you, where you can run heavier computations from the command line for an hour
- The path manipulation utilities used in the course are available with the course material in the file `code/bash/pathutils.sh`. It contains only BASH functions like `pathadd`, and nothing specific to our setup on JUSUF. Similarly, the aliases `G`, `C`, minus our JUSUF+course specific options, can be found in `code/bash/aliases.sh`.

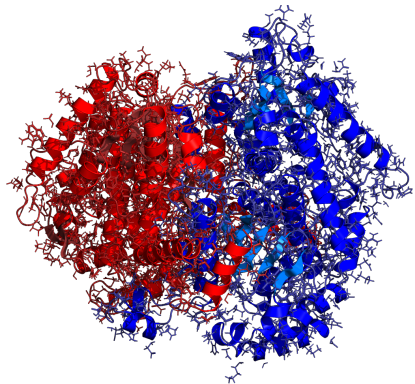
Some notes on the organisation of the course material

- If you log in directly using a terminal, instead of JupyterLab, run the following:
  jutil env activate -p training2444
- For every new terminal window/tab you want to use for any of the exercises here, please run
  source $PROJECT/local/setup.sh
- A working directory with your user name will be created (if it was not already there) inside the project working area
- Type `cdp` to change directory to your private working directory. In that directory, you will see sub-directories `orig` and `work`.
  - `orig` is where we download the course material in the pristine form. **It is recommended that you don't edit anything here**
  - `work` should be empty when you start. You copy any examples you want to work on, to this directory. How you manage things there is up to you.
  - Editing things inside `orig` might interfere with updating the course material during the course.
- The contents of these first few slides, since you may need to look them up later, are placed in the file `utilities.pdf`

HPC, C++ and scientific computing

HPC and C++ in scientific computing

- Handle complexity and do it fast
- Reliablity: catch implementation logic errors before the program runs
- Efficient machine code based on the source: application return time may decide whether or not a research problem is even considered
  - Smart algorithms
  - Hardware aware translation of ideas into code
  - Profiling and tuning



C++: elegant and efficient abstractions

- General purpose: no specialization to specific usage areas
- Compiler as a friend: in a large project, static type checking, data ownership control, const-ness guarantees and user defined compile time checks preclude a lot of possible errors
- No over simplification that precludes direct expert level use of hardware
- Leave no room for a lower level language
- You don't pay for features you don't use

C++ : high level and low level

- High level abstractions to facilitate fast development
- Direct access to low level features when you want them

Outline

- Building blocks for your own efficient code
- Cost of different abstractions
- SIMD programming
- Lessons from writing a matrix multiplication program
- Linear algebra with EIGEN
- Multi-threaded programs using standard parallel algorithms and Intel (R) Threading Building Blocks
- GPU programming with NVidia CUDA and Thrust
- Introduction to single source heterogeneous computing using SYCL and OneAPI

The default C++ standard for code samples, examples exercises etc. is C++23, but a few examples will require older standards.

A brief introduction to C++20 and C++23

C++20

Important refreshing of the language, similar to C++11.

- **Concepts**
- **Ranges**
- **Modules**
- **Coroutines**
- `auto` function parameters to implicitly declare function templates
- Explicit template syntax for lambdas
- Class non-type template parameters
- `try ... catch` and virtual functions in `constexpr` functions
- `consteval` and `constinit`
- `<=>`

- `<span>`
- `<ranges>`
- `<concepts>`
- `std::atomic<double>`
- `constexpr` algorithms
- `std::assume_aligned`
- `constexpr` numeric algorithms

C++23

(Interesting changes are mostly concentrated in the standard library.)

- Multi-dimensional subscript operators
- Deducing `this`
- Static `operator()` and `operator[]`
- `[[assume(expr)]]`

- `import std;`
- `<expected>`
- `ranges::to`, `views::zip`
- `<stacktrace>`
- `std::byteswap`
- `std::mdspan`
- Formatting ranges and containers
- `<print>`
- `std::forward_like`
- `std::generator`: synchronous coroutine generator

First: a couple of small, but interesting changes...

std::osyncstream

```
1  #include <iostream>
2  #include <omp.h>
3
4  auto main() -> int
5  {
6      #pragma omp parallel for
7      for (auto i = 0UL; i < 100UL; ++i) {
8         std::cout << "counter = " << i << " on thread "
9                   << omp_get_thread_num() << "\n";
10     }
11 }
```

First: a couple of small, but interesting changes...

std::osyncstream

JÜLICH Forschungszentrum

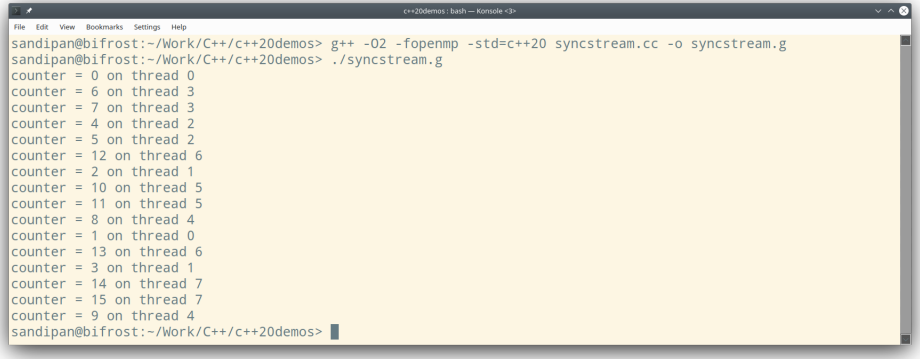
First: a couple of small, but interesting changes...

std::osyncstream

```
1  #include <iostream>
2  #include <syncstream>
3  #include <omp.h>
4
5  auto main() -> int
6  {
7      #pragma omp parallel for
8      for (auto i = 0UL; i < 100UL; ++i) {
9          std::osyncstream{std::cout} << "counter = " << i << " on thread "
10                                     << omp_get_thread_num() << "\n";
11     }
12 }
```

First: a couple of small, but interesting changes...

std::osyncstream



Exercise 1.1:

To work on the examples, please copy the examples folder into the `work` folder of your private work space. Do not modify the content in the `orig` folder, since that is where the course material will be updated. The update does not succeed if any file is modified in the `orig` folder. Suggested work flow...

```
$ cd $cxx2024/work
$ cp -r ../orig/day1/examples ./d1examples
$ cd d1examples
$ G syncstream.cc -fopenmp -o syncstream.g
$ ./syncstream.g
```

`examples/garbled.cc` and `examples/syncstream.cc` demonstrate the use of `std::osyncstream` as shown above. `examples/syncstream_mpi.cc` demonstrates that the synchronisation of output stream also works with output from different MPI processes.

```
$ mpicxx -std=c++23 -O3 -fopenmp syncstream_mpi.cc -o syncstream.mpi
$ OMP_NUM_THREADS=4 batch_run --ntasks=32 --cpus-per-task=4 ./syncstream.mpi
```

Immediate functions

```
1  // examples/immediate.cc
2  constexpr auto cxpr_sqr(auto x) { return x * x; }
3  consteval auto cevl_sqr(auto x) { return x * x; }
4
5  auto main(int argc, char* argv[]) -> int
6  {
7      std::array<double, cxpr_sqr(14)> A;
8      std::array<double, cevl_sqr(14)> B;
9      std::cout << cxpr_sqr(argc) << "\n";
10     std::cout << cevl_sqr(argc) << "\n";
11 }
```

- `constexpr` functions with compile time constant arguments are evaluated at compile time, if the result is needed to initialise a `constexpr` variable
- `constexpr` functions remain available for use with non-constant objects at run-time. This is sometimes desirable, but it also makes certain accidental uses possible, when we intend compile time evaluation but get something else.
- The new `consteval` specifier creates "immediate" functions. It is possible to use them in the compile time context. But it is an error to use them with non-constant arguments.

Designated initialisers

```cpp
// examples/desig2.cc
struct v3 { double x, y, z; };
struct pars { int offset; v3 velocity; };
auto operator<<(std::ostream & os, const v3 & v) -> std::ostream&
{
    return os << v.x << ", " << v.y << ", " << v.z << " ";
}
void example_func(pars p)
{
    std::cout << p.offset << " with velocity " << p.velocity << "\n";
}
auto main() -> int
{
    example_func({.offset = 5, .velocity = {.x=1., .y = 2., .z=3.}});
}
```

- Simple struct type objects can be initialised by designated initialisers for each field.
- Can be used to implement a kind of "keyword arguments" for functions. But remember, at least in C++20, the field order can not be shuffled.

Couple of small, but interesting changes... I

- You can now write `auto` in function parameter lists, e.g.,
  `auto add(auto x, auto y) { return x + y; }`, to create a function template

```
template <class T, class U> auto add(T x, U y) { return x + y; }
```

Couple of small, but interesting changes... II

```
1  if (auto it = find_if(M.begin(), M.end(), [](auto&& el){ return el.first == "key"; });
2      it != M.end()) { ... }
```

- `std::erase(C, element)` and `std::erase_if(C, predicate)` erase elements equal to a given element or elements satisfying a given predicate from a container C. *Same behaviour for different containers.*
- `std::lerp(min, max, t)` : linear interpolation, `std::midpoint(a, b)` : overflow aware mid-point calculation
- `std::assume_aligned<16>(dptr)` returns the input pointer, but the compiler then assumes that the pointer is aligned to a given number of bytes.

Couple of small, but interesting changes...

- `std::span<T>` is a new non-owning view type for contiguous ranges of arbitrary element types `T`. It is like the `string_view`, but for other array like entities such as `vector<T>`, `array<T,N>`, `valarray<T>` or even C-style arrays. Can be used to encapsulate the (pointer, size) pairs often used as function arguments. Benefit: it gives us an STL style interface for the (pointer, size) pair, so that they can be directly used with C++ algorithms.
- Signed size of containers: `std::ssize(C)`, where `C` is a container, returns a signed integer (number of elements in container). Containers like `std::vector`, `std::list`, `std::map` have member functions `ssize()` for the same purpose. Signed sizes are useful, for instance, when iterating backwards through the container.
- Safe integer comparisons: functions like `cmp_less(i1, i2)` will perform integer comparisons without conversions. `cmp_less(-1, 1U)` will return `true`, where as, `-1 < 1U` returns `false`. Similarly, we have, `cmp_less_equal`, `cmp_greater`, `cmp_not_equal` and `cmp_equal`.

Couple of small, but interesting changes...

Bit manipulation:

- `std::bit_cast<>`: `bit_cast<uint64_t>(3.141592653)` reinterprets the bits in the object representation of the input and returns an object of a required type so that the corresponding bits match
- `has_single_bit(UnsignedInteger)`: answers if only one of the bits in the input is **1**, while the rest are **0**
- `std::rotl(UnsignedInteger, amount)`, `std::rotr(UnsignedInteger, amount)`: Rotates the bits in an unsigned integer left or right by a given amount
- `std::bit_floor(UnsignedInteger)`: Largest power of two not greater than input
- `std::bit_ceil(UnsignedInteger)`: Smallest power of two not smaller than input
- Count consecutive **0** bits from the left `countl_zero` or right `countr_zero`, and similarly for **1** bits
- `popcount`, count the total number of **1** bits in the entire input

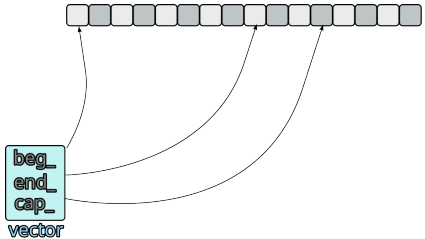
Exercise 1.2:

Some example programs about the minor new features of C++20 and C++23 are `desig.cc`, `desig2.cc`, `cxpr_algo0.cc`, `immediate.cc`, `intcmp.cc`, and `bit0.cc`, in the `examples/` directory. Check them, change them in small ways, ask related questions!

span

std::vector

- `operator[]`
- `size()` and `ssize()`
- `begin()`
- `end()`



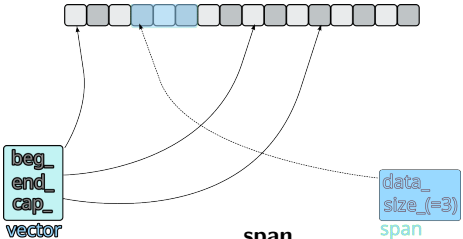
Contiguous containers

- Resource ownership and management
  - Memory allocation in constructors and member functions
  - Cleanup in destructor
  - As long as container exists, elements can be accessed
  - When container has expired, references / pointers / iterators to elements are invalidated

span

std::vector

- `operator[]`
- `size()` and `ssize()`
- `begin()`
- `end()`



std::span

- `operator[]`
- `size()` and `ssize()`
- `begin()`
- `end()`

Contiguous containers

- Resource ownership and management
  - Memory allocation in constructors and member functions
  - Cleanup in destructor
  - As long as container exists, elements can be accessed
  - When container has expired, references / pointers / iterators to elements are invalidated

span

- No resource ownership or management
  - Just stores address (of pre-existing data) and a size
  - Data is managed elsewhere, no cleanup duties
  - Even if span exists, accessibility of data is not guaranteed
  - When span has expired, references / pointers / iterators to elements may remain valid
  - Trivially copyable. Pass to functions and return from functions as values.

span

```
1  using std::transform_reduce;
2  using std::plus;
3  using std::multiplies;
4  using std::vector;
5
6
7  auto compute(const vector<double>& u,
8      const vector<double>& v) -> double
9  {
10     return transform_reduce(
11         u.begin(), u.end(),
12         v.begin(), 0., plus<double>{},
13         multiplies<double>{});
14 }
15 void elsewhere()
16 {
17     vector<double> A(100UL, 0.34);
18     vector<double> B(100UL, 0.87);
19     std::cout << compute(A, B) << "\n";
20 }
```

- We can avoid needlessly restrictive interfaces

span

```
1  using std::transform_reduce;
2  using std::plus;
3  using std::multiplies;
4  using std::vector;
5  using std::valarray;
6
7  auto compute(const vector<double>& u,
8       const vector<double>& v) -> double
9  {
10     return transform_reduce(
11         u.begin(), u.end(),
12         v.begin(), 0., plus<double>{},
13         multiplies<double>{});
14 }
15 void elsewhere()
16 {
17     vector<double> A(100UL, 0.34);
18     valarray<double> B(100UL, 0.87);
19     std::cout << compute(A, B) << "\n";
20 }
```

- We can avoid needlessly restrictive interfaces
- As written here, `std::valarray` wouldn't be an acceptable input

span

```
1  using std::transform_reduce;
2  using std::plus;
3  using std::multiplies;
4  using std::vector;
5  template <class T>
6  using VT = vector<T, tbb::scalable_allocator<T>>;
7
8  auto compute(const vector<double>& u,
9      const vector<double>& v) -> double
10 {
11     return transform_reduce(
12         u.begin(), u.end(),
13         v.begin(), 0., plus<double>{},
14         multiplies<double>{});
15 }
16 void elsewhere()
17 {
18     vector<double> A(100UL, 0.34);
19     VT<double> B(100UL, 0.87);
20     std::cout << compute(A, B) << "\n";
21 }
```

- We can avoid needlessly restrictive interfaces
- As written here, `std::valarray` wouldn't be an acceptable input
- As written here, even `std::vector` with a different allocator wouldn't be an acceptable input

span

```
using std::transform_reduce;
using std::plus;
using std::multiplies;
using std::vector;
template <class T>
using VT = vector<T, tbb::scalable_allocator<T>>;
using std::span;
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()
{
    vector<double> A(100UL, 0.34);
    VT<double> B(100UL, 0.87);
    std::cout << compute(A, B) << "\n";
}
```

- We can avoid needlessly restrictive interfaces
- As written here, `std::valarray` wouldn't be an acceptable input
- As written here, even `std::vector` with a different allocator wouldn't be an acceptable input
- With `std::span` we can write a concrete function, which can be used with any contiguous container!

span

```
using std::transform_reduce;
using std::plus;
using std::multiplies;
using std::vector;
template <class T>
using VT = vector<T, tbb::scalable_allocator<T>>;
using std::span;
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(const double* A, size_t N)
{
     VT<double> B(N, 0.87);
     std::cout << compute(span(A, N), B) << "\n";
}
```

- We can avoid needlessly restrictive interfaces
- As written here, `std::valarray` wouldn't be an acceptable input
- As written here, even `std::vector` with a different allocator wouldn't be an acceptable input
- With `std::span` we can write a concrete function, which can be used with any contiguous container!
- Contiguous data stored anywhere, even C-style arrays, can be easily used for the same function

```
template <class NoBueno>
auto compute(std::span<NoBueno> s) {...}
void elsewhere(const VT& v) {
     compute(v);
} // Template argument deduction failed!
```

span

```
1  using std::span;
2  using std::transform_reduce;
3  using std::plus;
4  using std::multiplies;
5  auto compute(span<const double> u,
6       span<const double> v) -> double
7  {
8      return transform_reduce(
9          u.begin(), u.end(),
10         v.begin(), 0., plus<double>{},
11         multiplies<double>{});
12 }
13
14 void elsewhere(double* x, double* y,
15                unsigned N)
16 {
17     return compute(span(x, N), span(y, N));
18 }
```

- 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

Exercise 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.

The 4 big changes

- Concepts: Named constraints on templates
- Ranges
  - A concept of an iterable range of entities demarcated by an iterator-sentinel pair, e.g., all STL containers, views (like `string_view`s and `span`s), adapted ranges, any containers you might write so long as they have some characteristics
  - Views: ranges which have constant time copy, move and assignment
  - Range adaptors : lazily evaluated functionals taking viewable ranges and producing views.
    Important consequence: UNIX pipe like syntax for composing simple easily verified components for non-trivial functionality
- Modules : Move away from header files, even for template/concepts based code. Consequences: faster build times, easier and more fine grained control over the exposed interface
- Coroutines: functions which can suspend and resume from the middle. Stackless. Consequences: asynchronous sequential code, lazily evaluated sequences, ... departure from pure stack trees at run time.

Concepts

Constrained templates

- Overloaded functions: different strategies for different input types

```
auto power(double x, double y) -> double ;
auto power(double x, int i) -> double ;
```

- Function templates: same steps for different types, e.g.,

```
template <class T> auto power(double x, T i) -> double ;
```

Constrained templates

- Overloaded functions: different strategies for different input types

```
auto power(double x, double y) -> double ;
auto power(double x, int i) -> double ;
```

- Function templates: same steps for different types, e.g.,

```
template <class T> auto power(double x, T i) -> double ;
```

- Can we combine the two, so that we have two (or more) function templates, both looking like the above, but one is automatically selected whenever `T` is an integral type and the other whenever `T` is a floating point type ?

Constrained templates

- Overloaded functions: different strategies for different input types

```
auto power(double x, double y) -> double ;
auto power(double x, int i) -> double ;
```

- Function templates: same steps for different types, e.g.,

```
template <class T> auto power(double x, T i) -> double ;
```

- Can we combine the two, so that we have two (or more) function templates, both looking like the above, but one is automatically selected whenever `T` is an integral type and the other whenever `T` is a floating point type ?
- Is there any way to impose conditions for a given function template to be selected instead of blindly substituting `T` with the type of the input ? Perhaps, something like this ?

```
template <class T> auto power(double x, T i) -> double requires floating_point<T>;
template <class T> auto power(double x, T i) -> double requires integer<T>;
```

Constrained templates

- Overloaded functions: different strategies for different input types

```
auto power(double x, double y) -> double ;
auto power(double x, int i) -> double ;
```

- Function templates: same steps for different types, e.g.,

```
template <class T> auto power(double x, T i) -> double ;
```

- Can we combine the two, so that we have two (or more) function templates, both looking like the above, but one is automatically selected whenever `T` is an integral type and the other whenever `T` is a floating point type ?
- Is there any way to impose conditions for a given function template to be selected instead of blindly substituting `T` with the type of the input ? Perhaps, something like this ?

```
template <class T> auto power(double x, T i) -> double requires floating_point<T>;
template <class T> auto power(double x, T i) -> double requires integer<T>;
```

- We can.

Constrained templates

- Overloaded functions: different strategies for different input types

```
auto power(double x, double y) -> double ;
auto power(double x, int i) -> double ;
```

- Function templates: same steps for different types, e.g.,

```
template <class T> auto power(double x, T i) -> double ;
```

- Can we combine the two, so that we have two (or more) function templates, both looking like the above, but one is automatically selected whenever `T` is an integral type and the other whenever `T` is a floating point type ?
- Is there any way to impose conditions for a given function template to be selected instead of blindly substituting `T` with the type of the input ? Perhaps, something like this ?

```
template <class T> auto power(double x, T i) -> double requires floating_point<T>;
template <class T> auto power(double x, T i) -> double requires integer<T>;
```

- We can. Or rather, we always could with C++ templates. But now the syntax is easier.

Concepts

Named requirements on template parameters

```
template <unsigned X> concept PowerOfTwo = (X != 0 && (X & (X-1)) == 0);
```

Concepts

Named requirements on template parameters

```
template <unsigned X> concept PowerOfTwo = std::has_single_bit(X);
```

Concepts

Named requirements on template parameters

```
template <unsigned X> concept PowerOfTwo = std::has_single_bit(X);
constexpr auto flag1 = PowerOfTwo<2048U>; // Compiler sets flag1 to True
constexpr auto flag2 = PowerOfTwo<2056U>; // Compiler sets flag2 to False
```

Concepts

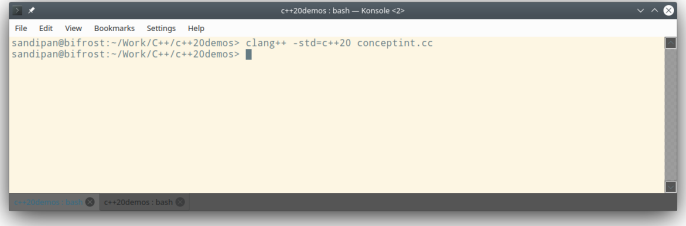
Named requirements on template parameters

```
template <unsigned X> concept PowerOfTwo = std::has_single_bit(X);
template <class T, unsigned N> requires PowerOfTwo<N>
struct MyMatrix {
    // code which assumes that the square matrix size is a power of two
};
```

Concepts

Named requirements on template parameters

```
template <unsigned X> concept PowerOfTwo = std::has_single_bit(X);
template <class T, unsigned N> requires PowerOfTwo<N>
struct MyMatrix {
    // code which assumes that the square matrix size is a power of two
};
auto main() -> int
{
    auto m = MyMatrix<double, 16U>{};
}
```

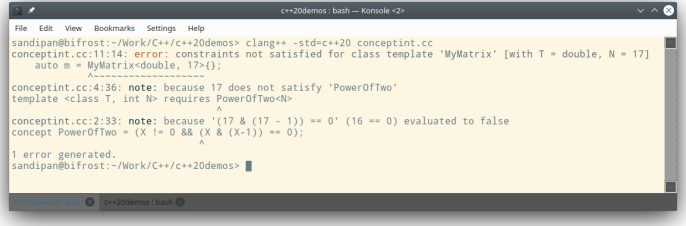


JÜLICH Forschungszentrum

Concepts

Named requirements on template parameters

```
template <unsigned X> concept PowerOfTwo = std::has_single_bit(X);
template <class T, unsigned N> requires PowerOfTwo<N>
struct MyMatrix {
    // code which assumes that the square matrix size is a power of two
};
auto main() -> int
{
    auto m = MyMatrix<double, 17U>{};
}
```



JÜLICH Forschungszentrum

Concepts

Named requirements on template parameters

```
template <unsigned X> concept PowerOfTwo = std::has_single_bit(X);
template <class T> concept Number = std::integral<T> or std::floating_point<T>;
```

Concepts

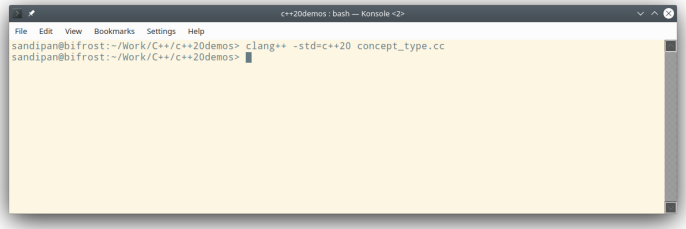
Named requirements on template parameters

```
template <unsigned X> concept PowerOfTwo = std::has_single_bit(X);
template <class T> concept Number = std::integral<T> or std::floating_point<T>;
template <class T, unsigned N> requires Number<T> && PowerOfTwo<N>
struct MyMatrix {
    // assume that the square matrix size is a power of two, and T is a numeric type
};
```

Concepts

Named requirements on template parameters

```cpp
template <unsigned X> concept PowerOfTwo = std::has_single_bit(X);
template <class T> concept Number = std::integral<T> or std::floating_point<T>;
template <class T, unsigned N> requires Number<T> && PowerOfTwo<N>
struct MyMatrix {
    // assume that the square matrix size is a power of two, and T is a numeric type
};
auto main() -> int
{
    auto m = MyMatrix<double, 16U>{};
}
```

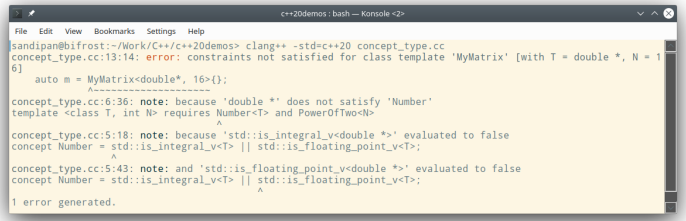


JÜLICH Forschungszentrum

Concepts

Named requirements on template parameters

```
template <unsigned X> concept PowerOfTwo = std::has_single_bit(X);
template <class T> concept Number = std::integral<T> or std::floating_point<T>;
template <class T, unsigned N> requires Number<T> && PowerOfTwo<N>
struct MyMatrix {
    // assume that the square matrix size is a power of two, and T is a numeric type
};
auto main() -> int
{
    auto m = MyMatrix<double*, 16U>{};
}
```



Concepts

Named requirements on template parameters

- `concept`s are named requirements on template parameters, such as `floating_point`, `contiguous_range`
- If `MyAPI` is a `concept`, and `T` is a template parameter, `MyAPI<T>` evaluates at compile time to either true or false.
- Concepts can be combined using conjunctions (`&&`) and disjunctions (`||`) to make other concepts.
- A `requires` *clause* introduces a constraint or requirement on a template type

A suitably designed set of concepts can greatly improve readability of template code

Creating concepts

```
template <template-pars>
concept conceptname = constraint_expr;
```

```
template <class T>
concept Integer = std::is_integral_v<T>;
template <class D, class B>
concept Derived = std::is_base_of_v<B, D>;

class Counters;
template <class T>
concept Integer_ish = Integer<T> ||
                      Derived<T,Counters>;

template <class T>
concept Addable = requires (T a, T b) {
    { a + b };
};
template <class T>
concept Indexable = requires(T A) {
    { A[0UL] };
};
```

- Out of a simple `type_traits` style boolean expression
- Combine with logical operators to create more complex requirements
- The **`requires`** *expression* allows creation of syntactic requirements

Creating concepts

```
template <template-pars>
concept conceptname = constraint_expr;
```

```
template <class T>
concept Integer = std::is_integral_v<T>;
template <class D, class B>
concept Derived = std::is_base_of_v<B, D>;
```

```
class Counters;
template <class T>
concept Integer_ish = Integer<T> ||
                      Derived<T,Counters>;
```

```
template <class T>
concept Addable = requires (T a, T b) {
    { a + b };
};
template <class T>
concept Indexable = requires(T A) {
    { A[0UL] };
};
```

- Out of a simple `type_traits` style boolean expression
- Combine with logical operators to create more complex requirements
- The **`requires`** *expression* allows creation of syntactic requirements

Creating concepts

```
template <template-pars>
concept conceptname = constraint_expr;
```

```
template <class T>
concept Integer = std::is_integral_v<T>;
template <class D, class B>
concept Derived = std::is_base_of_v<B, D>;

class Counters;
template <class T>
concept Integer_ish = Integer<T> ||
                      Derived<T,Counters>;

template <class T>
concept Addable = requires (T a, T b) {
    { a + b };
};
template <class T>
concept Indexable = requires(T A) {
    { A[0UL] };
};
```

- Out of a simple `type_traits` style boolean expression
- Combine with logical operators to create more complex requirements
- The **`requires`** *expression* allows creation of syntactic requirements

Creating concepts

```
template <template-pars>
concept conceptname = constraint_expr;
```

```
template <class T>
concept Integer = std::is_integral_v<T>;
template <class D, class B>
concept Derived = std::is_base_of_v<B, D>;

class Counters;
template <class T>
concept Integer_ish = Integer<T> ||
                      Derived<T,Counters>;

template <class T>
concept Addable = requires (T a, T b) {
    { a + b };
};
template <class T>
concept Indexable = requires(T A) {
    { A[0UL] };
};
```

- Out of a simple `type_traits` style boolean expression
- Combine with logical operators to create more complex requirements
- The `requires` *expression* allows creation of syntactic requirements

`requires` expression: Parameter list and a brace enclosed sequence of requirements:

- type requirements, e.g., `typename T::value_type;`
- simple requirements as shown on the left
- compound requirements with optional return type constraints, e.g., `{ A[0UL] } -> convertible_to<int>;`

Creating concepts

```
template <template-pars>
concept conceptname = constraint_expr;
```

```
template <class T>
concept Integer = std::is_integral_v<T>;
template <class D, class B>
concept Derived = std::is_base_of_v<B, D>;

class Counters;
template <class T>
concept Integer_ish = Integer<T> ||
                      Derived<T,Counters>;

template <class T>
concept Addable = requires (T a, T b) {
    { a + b };
};
template <class T>
concept Indexable = requires(T A) {
    { A[0UL] };
};
```

- Out of a simple `type_traits` style boolean expression
- Combine with logical operators to create more complex requirements
- The **`requires`** *expression* allows creation of syntactic requirements

```
1  // Usage example...
2  template <class T> requires Indexable<T>
3  auto f(T&& x) -> unsigned long;
4  void elsewhere() {
5      std::vector<Protein> v;
6      std::array<NucleicAcidType, 4> NA;
7      f(v); // OK
8      f(NA); // OK
9      f(4); // No match!
10 }
```

Using concepts

```
template <class T>
requires Integer_ish<T>
auto categ0(T&& i, double x) -> T;

template <class T>
auto categ1(T&& i, double x) -> T
    requires Integer_ish<T>;

template <Integer_ish T>
auto categ2(T&& i, double x) -> T;

void erase(Integer_ish auto&& i)
```

To constrain template parameters, one can

- place a `requires` clause immediately after the template parameter list
- place a `requires` clause after the function parameter parentheses
- Use the `concept` name in place of `class` or `typename` in the template parameter list
- Use `ConceptName auto` in the function parameter list

Using concepts

```
template <class T>
requires Integer_ish<T>
auto categ0(T&& i, double x) -> T;

template <class T>
auto categ1(T&& i, double x) -> T
    requires Integer_ish<T>;

template <Integer_ish T>
auto categ2(T&& i, double x) -> T;

void erase(Integer_ish auto&& i)
```

To constrain template parameters, one can

- place a `requires` clause immediately after the template parameter list
- place a `requires` clause after the function parameter parentheses
- Use the `concept` name in place of `class` or `typename` in the template parameter list
- Use `ConceptName auto` in the function parameter list

Using concepts

```
template <class T>
requires Integer_ish<T>
auto categ0(T&& i, double x) -> T;

template <class T>
auto categ1(T&& i, double x) -> T
    requires Integer_ish<T>;

template <Integer_ish T>
auto categ2(T&& i, double x) -> T;

void erase(Integer_ish auto&& i)
```

To constrain template parameters, one can

- place a `requires` clause immediately after the template parameter list
- place a `requires` clause after the function parameter parentheses
- Use the `concept` name in place of `class` or `typename` in the template parameter list
- Use `ConceptName auto` in the function parameter list

Using concepts

```
template <class T>
requires Integer_ish<T>
auto categ0(T&& i, double x) -> T;

template <class T>
auto categ1(T&& i, double x) -> T
    requires Integer_ish<T>;

template <Integer_ish T>
auto categ2(T&& i, double x) -> T;

void erase(Integer_ish auto&& i)
```

To constrain template parameters, one can

- place a `requires` clause immediately after the template parameter list
- place a `requires` clause after the function parameter parentheses
- Use the `concept` name in place of `class` or `typename` in the template parameter list
- Use `ConceptName auto` in the function parameter list

Using concepts

```
template <class T>
requires Integer_ish<T>
auto categ0(T&& i, double x) -> T;

template <class T>
auto categ1(T&& i, double x) -> T
    requires Integer_ish<T>;

template <Integer_ish T>
auto categ2(T&& i, double x) -> T;

void erase(Integer_ish auto&& i)
```

To constrain template parameters, one can

- place a `requires` clause immediately after the template parameter list
- place a `requires` clause after the function parameter parentheses
- Use the `concept` name in place of `class` or `typename` in the template parameter list
- Use `ConceptName auto` in the function parameter list

Declaring function input parameters with auto

```
1  template <class T>
2  auto sqr(const T& x) { return x * x; }
```

- Because of syntax introduced for functions with constrained templates in C++20, we have a new way to write fully unconstrained function templates...

Declaring function input parameters with auto

```
1
2   auto sqr(const auto& x) { return x * x; }
```

- Because of syntax introduced for functions with constrained templates in C++20, we have a new way to write fully unconstrained function templates...
- Functions with `auto` in their parameter list are implicitly function templates

Declaring function input parameters with auto

```
1
2   auto sqr(const auto& x) { return x * x; }
```

- Because of syntax introduced for functions with constrained templates in C++20, we have a new way to write fully unconstrained function templates...
- Functions with `auto` in their parameter list are implicitly function templates

Exercise 1.8:

The program `examples/concepts/gcd_w_concepts.cc` shows a very small concept definition and its use in a function calculating the greatest common divisor of two integers.

Exercise 1.9:

The series of programs `examples/concepts/generic_func1.cc` through `generic_func4.cc` shows some trivial functions implemented with templates with and without constraints. The files contain plenty of comments explaining the rationale and use of concepts.

Overloading based on concepts

```
1  template <class N>
2  concept Number = std::floating_point<N>
3                   or std::integral<N>;
4
5
6  void proc(Number auto&& x) {
7      std::cout << "Called proc for numbers";
8  }
9
10
11
12 auto main() -> int {
13     proc(-1);
14     proc(88UL);
15     proc("0118 999 88199 9119725   3");
16     proc(3.141);
17     proc("eighty"s);
18 }
```

- Constraints on template parameters are not just "documentation" or decoration or annotation.

Overloading based on concepts

```
1  template <class N>
2  concept Number = std::floating_point<N>
3                   or std::integral<N>;
4  template <class N>
5  concept NotNumber = not Number<N>;
6  void proc(Number auto&& x) {
7      std::cout << "Called proc for numbers";
8  }
9  void proc(NotNumber auto&& x) {
10     std::cout << "Called proc for non-numbers";
11 }
12 auto main() -> int {
13     proc(-1);
14     proc(88UL);
15     proc("0118 999 88199 9119725   3");
16     proc(3.141);
17     proc("eighty"s);
18 }
```

- Constraints on template parameters are not just "documentation" or decoration or annotation.
- The compiler can choose between different versions of a function based on concepts

Overloading based on concepts

```
1  template <class N>
2  concept Number = std::floating_point<N>
3                   or std::integral<N>;
4  template <class N>
5  concept NotNumber = not Number<N>;
6  void proc(Number auto&& x) {
7      std::cout << "Called proc for numbers";
8  }
9  void proc(NotNumber auto&& x) {
10     std::cout << "Called proc for non-numbers";
11 }
12 auto main() -> int {
13     proc(-1);
14     proc(88UL);
15     proc("0118 999 88199 9119725   3");
16     proc(3.141);
17     proc("eighty"s);
18 }
```

- Constraints on template parameters are not just "documentation" or decoration or annotation.
- The compiler can choose between different versions of a function based on concepts
- The version of a function chosen depends on *properties* of the input types, rather than their identities. "It's not who you are underneath, it's what you (can) do that defines you."

Overloading based on concepts

```
1  template <class N>
2  concept Number = std::floating_point<N>
3                   or std::integral<N>;
4  template <class N>
5  concept NotNumber = not Number<N>;
6  void proc(Number auto&& x) {
7      std::cout << "Called proc for numbers";
8  }
9  void proc(NotNumber auto&& x) {
10     std::cout << "Called proc for non-numbers";
11 }
12 auto main() -> int {
13     proc(-1);
14     proc(88UL);
15     proc("0118 999 88199 9119725   3");
16     proc(3.141);
17     proc("eighty"s);
18 }
```

- Constraints on template parameters are not just "documentation" or decoration or annotation.
- The compiler can choose between different versions of a function based on concepts
- The version of a function chosen depends on *properties* of the input types, rather than their identities. "It's not who you are underneath, it's what you (can) do that defines you."
- During overload resolution, in case multiple matches are found, the most constrained overload is chosen.

Overloading based on concepts

```
1  template <class N>
2  concept Number = std::floating_point<N>
3                   or std::integral<N>;
4  template <class N>
5  concept NotNumber = not Number<N>;
6  void proc(Number auto&& x) {
7      std::cout << "Called proc for numbers";
8  }
9  void proc(NotNumber auto&& x) {
10     std::cout << "Called proc for non-numbers";
11 }
12 auto main() -> int {
13     proc(-1);
14     proc(88UL);
15     proc("0118 999 88199 9119725   3");
16     proc(3.141);
17     proc("eighty"s);
18 }
```

- Constraints on template parameters are not just "documentation" or decoration or annotation.
- The compiler can choose between different versions of a function based on concepts
- The version of a function chosen depends on *properties* of the input types, rather than their identities. "It's not who you are underneath, it's what you (can) do that defines you."
- During overload resolution, in case multiple matches are found, the most constrained overload is chosen.
- Not based on any inheritance relationships

Overloading based on concepts

```
1  template <class N>
2  concept Number = std::floating_point<N>
3                   or std::integral<N>;
4  template <class N>
5  concept NotNumber = not Number<N>;
6  void proc(Number auto&& x) {
7      std::cout << "Called proc for numbers";
8  }
9  void proc(NotNumber auto&& x) {
10     std::cout << "Called proc for non-numbers";
11 }
12 auto main() -> int {
13     proc(-1);
14     proc(88UL);
15     proc("0118 999 88199 9119725   3");
16     proc(3.141);
17     proc("eighty"s);
18 }
```

- Constraints on template parameters are not just "documentation" or decoration or annotation.
- The compiler can choose between different versions of a function based on concepts
- The version of a function chosen depends on *properties* of the input types, rather than their identities. "It's not who you are underneath, it's what you (can) do that defines you."
- During overload resolution, in case multiple matches are found, the most constrained overload is chosen.
- Not based on any inheritance relationships
- Not a "quack like a duck, or bust" approach either.

Overloading based on concepts

```
1  template <class N>
2  concept Number = std::floating_point<N>
3                   or std::integral<N>;
4  template <class N>
5  concept NotNumber = not Number<N>;
6  void proc(Number auto&& x) {
7      std::cout << "Called proc for numbers";
8  }
9  void proc(NotNumber auto&& x) {
10     std::cout << "Called proc for non-numbers";
11 }
12 auto main() -> int {
13     proc(-1);
14     proc(88UL);
15     proc("0118 999 88199 9119725   3");
16     proc(3.141);
17     proc("eighty"s);
18 }
```

- Constraints on template parameters are not just "documentation" or decoration or annotation.
- The compiler can choose between different versions of a function based on concepts
- The version of a function chosen depends on *properties* of the input types, rather than their identities. "It's not who you are underneath, it's what you (can) do that defines you."
- During overload resolution, in case multiple matches are found, the most constrained overload is chosen.
- Not based on any inheritance relationships
- Not a "quack like a duck, or bust" approach either.
- Entirely compile time mechanism

Selecting a code path based on input properties

```cpp
template <class T>
concept hasAPI = requires( T x ) {
    typename T::value_type;
    typename T::block_type;
    { x[0UL] };
    { x.block(0UL) };
};

template <class C> auto algo(C && x) -> size_t
{
    if constexpr (hasAPI<C>) {
        // Use x.block() etc to calculate
        // using vector blocks
    } else {
        // Some general method, quick to
        // develop but perhaps slow to run
    }
}
```

```cpp
#include "algo.hh"
#include "Machinery.hh"

auto main() -> int
{
    Machinery obj;
    auto res = algo(obj);
    std::cout << "Result = " << res << "\n";
}
```

- General algorithms can be implemented such that a faster method is selected whenever the input has specific properties
- No requirement of any inheritance relationships for the user of the algorithms

Constraining non-template members of class templates

```cpp
template <class T> struct ClassTemp {
    auto operator++() -> std::enable_if_t<std::is_integral_v<T>, ClassTemp&> {
        ++obj;
        return *this;
    }
    auto other() -> std::string { return "something else"; }
    auto val() const -> T { return obj; }
    T obj{};
};
auto main() -> int {
    ClassTemp<int> x;
    std::cout << (++x).val() << "\n";
    std::cout << x.other() << "\n";
}
```

```
$ g++ -std=c++20 nontempconstr.cc
$
```

Constraining non-template members of class templates

```cpp
template <class T> struct ClassTemp {
    auto operator++() -> std::enable_if_t<std::is_integral_v<T>, ClassTemp&> {
        ++obj;
        return *this;
    }
    auto other() -> std::string { return "something else"; }
    auto val() const -> T { return obj; }
    T obj{};
};
auto main() -> int {
    ClassTemp<double> x;
    std::cout << (++x).val() << "\n";
    std::cout << x.other() << "\n";
}
```

```
$ g++ -std=c++20 nontempconstr.cc
error: no type named ‘type’ in ‘struct std::enable_if<false, ClassTemp<double>&>’
 2614 |       using enable_if_t = typename enable_if<_Cond, _Tp>::type;
      |                 ^~~~~~~~~~
nontempconstr1.cc: In function ‘int main()’:
nontempconstr1.cc:19:19: error: no match for ‘operator++’ (operand type is
‘ClassTemp<double>’)
$
```

Constraining non-template members of class templates

```
1  template <class T> struct ClassTemp {
2      auto operator++() -> std::enable_if_t<std::is_integral_v<T>, ClassTemp&> {
3          ++obj;
4          return *this;
5      }
6      auto other() -> std::string { return "something else"; }
7      auto val() const -> T { return obj; }
8      T obj{};
9  };
10 auto main() -> int {
11     ClassTemp<double> x;
12     // std::cout << (++x).val() << "\n";
13     std::cout << x.other() << "\n";
14 }
```

```
$ g++ -std=c++20 nontempconstr.cc
error: no type named ‘type’ in ‘struct std::enable_if<false, ClassTemp<double>&>’
 2614 |       using enable_if_t = typename enable_if<_Cond, _Tp>::type;
$
```

`std::enable_if` can not be used to disable non-template members of class templates.

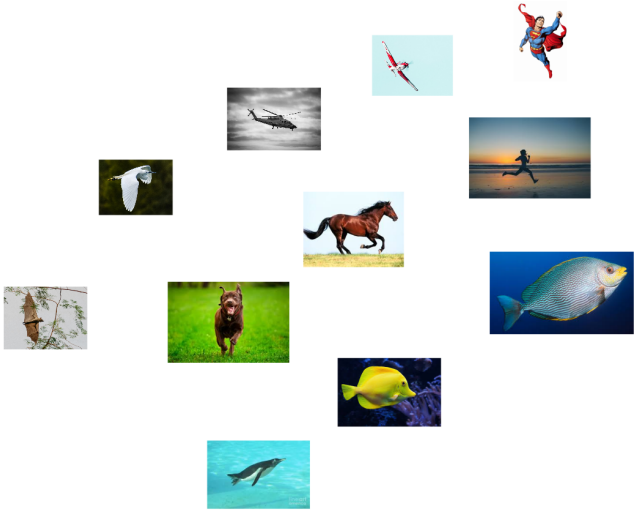
Constraining non-template members of class templates

```
1  template <class N> concept Number = std::integral<N>  std::floating_point<N>;
2  template <class N> concept Integer = Number<N> && std::integral<N>;
3
4  template <class T> struct ClassTemp {
5      auto operator++() -> ClassTemp& requires Integer<T> {
6          ++obj;
7          return *this;
8      }
9      auto other() -> std::string { return "something else"; }
10     auto val() const -> T { return obj; }
11     T obj{};
12 };
13 auto main() -> int {
14     ClassTemp<double> x;
15     // std::cout << (++x).val() << "\n";
16     std::cout << x.other() << "\n";
17 }
```

```
$ g++ -std=c++20 nontempconstr.cc
$
```

But `concepts` can be used as restraints on non-template members of class templates.

Concepts: summary



f(those who can fly)

f(runners)

f(swimmers)

Exercise 1.10:

- Build and run the examples `conceptint.cc`, `concept_type.cc`, `overload_w_concepts.cc`, `nontempconstr.cc`, and `cpp_sum_2.cc`. In some cases the programs illustrate specific types of programming error. The demonstration is that compiler finds them and gives us useful error messages. Example compilation:

```
clang++ -std=c++20 -stdlib=libc++ overload_w_concepts.cc
a.out
```

- Alternatively, you could use one of the shortcuts provided with the course material.

```
C overload_w_concepts.cc -o overload_w_concepts.l && ./overload_w_concepts.l
```

Predefined useful concepts

Many concepts useful in building our own concepts are available in the standard library header `<concepts>`.

- `same_as`
- `convertible_to`
- `signed_ingegral`, `unsigned_integral`
- `floating_point`
- `assignable_from`
- `swappable`, `swappable_with`
- `derived_from`
- `move_constructible`, `copy_constructible`
- `invocable`
- `predicate`
- `relation`

Ranges

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);
std::sort(v);
```

- The `<ranges>` header defines a special kind of concept describing entities with a start and an end.
- The `<algorithm>` header has many 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

The range concept

```
1  def python_sum(Container, start=0):
2      res = start
3      for x in Container:
4          res += x
5      return res
```

The range concept

```
1  auto sum(auto&& Container, auto start = 0) {
2      for (auto&& el : Container) start += el;
3      return start;
4  }
```

As compact as the python version, but with the same problems:

- We did not ensure that the first parameter is in fact a container. Just calling it `Container` isn't good enough
- We did not ensure that the type of the second parameter was the data type of the first

The range concept

```
1  template <class T> using cleanup = std::remove_cvref_t<T>;
2  template <class T> using element = std::iter_value_t<cleanup<T>>;
3  template <class T> requires std::ranges::forward_range<T>
4  auto sum(T&& a, element<T> start) {
5      for (auto&& el : a) start += el;
6      return start;
7  }
```

Slightly more lines, but:

- Only available when `T` really is a sequence where forward iteration is possible
- The second parameter must be the element type of the first one

The range concept

```
1  template <class T> using cleanup = std::remove_cvref_t<T>;
2  template <class T> using element = std::iter_value_t<cleanup<T>>;
3  template <class T> requires std::ranges::::forward_range<T>
4  auto sum(T&& a, element<T> start) {
5      for (auto&& el : a) start += el;
6      return start;
7  }
8  template <class ... T, class U> requires ((std::same_as<T, U>) && ...)
9  auto sum(U&& start, T&& ... a) { return (start + ... + a); }
```

The range concept

```
1  template <class T> using cleanup = std::remove_cvref_t<T>;
2  template <class T> using element = std::iter_value_t<cleanup<T>>;
3  template <class T> requires std::ranges::::forward_range<T>
4  auto sum(T&& a, element<T> start) {
5      for (auto&& el : a) start += el;
6      return start;
7  }
8  template <class ... T, class U> requires ((std::same_as<T, U>) && ...)
9  auto sum(U&& start, T&& ... a) { return (start + ... + a); }
```

- We can overload with a different function template taking the same number of generic parameters, but different constraints
- We can overload with a variadic function template of the same name, so long as the constraints are different

The range concept

```cpp
template <class T> using cleanup = std::remove_cvref_t<T>;
template <class T> using element = std::iter_value_t<cleanup<T>>;
template <class T> requires std::ranges::::forward_range<T>
auto sum(T&& a, element<T> start) {
    for (auto&& el : a) start += el;
    return start;
}
template <class ... T, class U> requires ((std::same_as<T, U>) && ...)
auto sum(U&& start, T&& ... a) { return (start + ... + a); }
```

```cpp
auto main() -> int {
    std::vector v{ 1, 2, 3, 4, 5 };
    std::list l{9.1, 9.2, 9.3, 9.4, 9.5, 9.6};
    std::cout << sum(v, 0) << "\n";
    std::cout << sum(l, 0.) << "\n";
    std::cout << sum(4.5, 9.) << "\n";
    std::cout << sum(4.5, 3.4, 5., 9.) << "\n";
}
```

Fun with ranges and views

```cpp
// examples/ranges/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/ranges/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

```
1  // examples/ranges/iota.cc
2  #include <ranges>
3  #include <iostream>
4  auto main() -> int {
5      namespace sv = std::views;
6      for (auto i : sv::iota(1UL)) {
7          if ((i+1) % 10000UL == 0UL) {
8              std::cout << i << ' ';
9              if ((i+1) % 100000UL == 0UL)
10                 std::cout << '\n';
11             if (i >= 100000000UL) break;
12         }
13     }
14 }
```

- 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

```
1  // examples/ranges/iota.cc
2  #include <ranges>
3  #include <iostream>
4  auto main() -> int {
5      namespace sv = std::views;
6      for (auto i : sv::iota(1UL)) {
7          if ((i+1) % 10000UL == 0UL) {
8              std::cout << i << ' ';
9              if ((i+1) % 100000UL == 0UL)
10                 std::cout << '\n';
11             if (i >= 100000000UL) break;
12         }
13     }
14 }
```

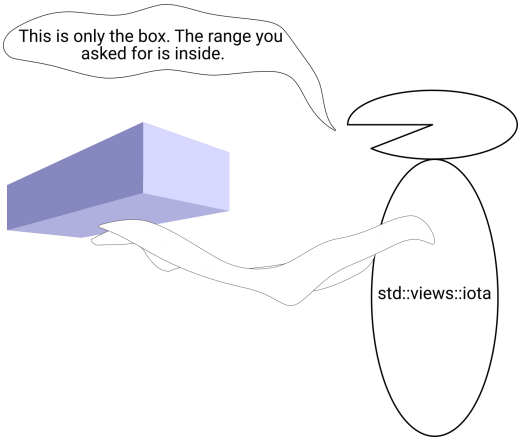
- All containers are ranges, but not all ranges are containers
- `std::string_view` is a perfectly fine range. Has iterators with the right properties. Has `begin()` and `end()` functions. It does not own the contents, but "ownership" is not part of the idea of a range.
- We could take this further by 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

```
1  // examples/ranges/iota.cc
2  #include <ranges>
3  #include <iostream>
4  auto main() -> int {
5      namespace sv = std::views;
6      for (auto i : sv::iota(1UL)) {
7          if ((i+1) % 10000UL == 0UL) {
8              std::cout << i << ' ';
9              if ((i+1) % 100000UL == 0UL)
10                 std::cout << '\n';
11             if (i >= 100000000UL) break;
12         }
13     }
14 }
```

- 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

```
1  #include <ranges>
2  #include <iostream>
3  auto main() -> int {
4      namespace sv = std::views;
5      for (auto i : sv::iota(1UL)) {
6          if ((i+1) % 10000UL == 0UL) {
7              std::cout << i << ' ';
8              if ((i+1) % 100000UL == 0UL)
9                  std::cout << '\n';
10             if (i >= 100000000UL) break;
11         }
12     }
13 }
```

- 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.

View adaptors

```
1  namespace sv = std::views;
2  std::vector v{1,2,3,4,5};
3  auto v3 = sv::take(v, 3);
4  // v3 is some sort of object so
5  // that it represents the first
6  // 3 elements of v. It does not
7  // own anything, and has constant
8  // time copy/move etc. It's a view.
9
10 // 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.

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 $[0, 2\pi)$

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

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

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

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

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

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

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

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

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

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

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

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

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

```
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; })`

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; })`
- Perform the transformation $x \mapsto sin^2(x) + cos^2(x) - 1$ over the resulting range

```
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; })`
- Perform the transformation $x \mapsto sin^2(x) + cos^2(x) - 1$ over the resulting range

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

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";
    }
}
```

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 1.11:

The code used for the demonstration of view adaptors is `examples/ranges/trig_views.cc`. Build this code with GCC and Clang.

```
g++ -std=c++20  trig_views.cc
./a.out

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

Exercise 1.12:

The trigonometric relation we used is true, so not all possibilities are explored. In `examples/ranges/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

```
1  // examples/ranges/gerund.cc
2      using itertype = std::istream_iterator<std::string>;
3      std::ifstream fin { argv[1] };
4      auto gerund = [](std::string_view w) { return w.ends_with("ing"); };
5      auto in = sr::istream_view<std::string>(fin);
6      std::print("{}\n", in | sv::filter(gerund));
7
```

- `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::print` is shown writing out a range. This works with the Clang standard library `libc++`. Since GCC doesn't have an implementation yet, please use `clang++` for this.

View adaptors

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

```
1  // examples/ranges/views_and_span.cc
2  auto main(int argc, char* argv[]) -> int
3  {
4      if (argc < 2) return 1;
5      namespace sr = std::ranges;
6      namespace sv = std::views;
7
8      std::span args(argv, argc);
9      auto str = [](auto cstr) -> std::string_view { return cstr; };
10     auto [mn, mx] = sr::minmax(args | sv::drop(1) | sv::transform(str));
11
12     std::cout << "Alphabetically first = " << mn << " last = " << mx << "\n";
13 }
```

Ranges in C++23

```
1  std::vector v { "apples"s, "oranges"s,
2                  "mangos"s, "bananas"s };
3
4  for (auto [i, fruit] : sv::enumerate(v)) {
5      std::cout << format("{}: {}\n", i, fruit);
6  }
```

```
$ G -std=c++23 enumerate.cc
$ ./a.out
0: apples
1: oranges
2: mangos
3: bananas
```

With the definitions
`namespace sr = std::ranges`,
and `namespace sv = sr::views`

- `sv::enumerate`
- `sv::zip`
- `sv::zip_transform`
- `sv::adjacent`
- `sr::to`

Ranges in C++23

```
1  std::vector v { "apples"s, "oranges"s,
2                  "mangos"s, "bananas"s };
3
4  for (auto [fruit1, fruit2] :
5           sv::zip(v, sv::reverse(v))) {
6      std::cout << format("{}: {}\n", fruit1, fruit2);
7  }
```

```
$ G -std=c++23 zip.cc
$ ./a.out
apples: bananas
oranges: mangos
mangos: oranges
bananas: apples
```

With the definitions
`namespace sr = std::ranges`,
and `namespace sv = sr::views`

- `sv::enumerate`
- `sv::zip`
- `sv::zip_transform`
- `sv::adjacent`
- `sr::to`

Ranges in C++23

```
1  for (auto s : sv::zip_transform(
2            [](auto&& s1, auto&& s2) {
3                return format("{} <--> {}", s1, s2);
4            },
5            v, sv::reverse(v))) {
6                std::cout << s << "\n";
7  }
```

```
$ G -std=c++23 zip_transform.cc
$ ./a.out
apples <--> bananas
oranges <--> mangos
mangos <--> oranges
bananas <--> apples
```

With the definitions
`namespace sr = std::ranges`,
and `namespace sv = sr::views`

- `sv::enumerate`
- `sv::zip`
- `sv::zip_transform`
- `sv::adjacent`
- `sr::to`

Ranges in C++23

```
1  for (auto [i0, i1, i2]:
2           sv::iota(0UL, 15UL) | sv::adjacent<3UL>) {
3      std::print("{}, {}, {}\n", i0, i1, i2);
4  }
```

```
$ G -std=c++23 adjacent.cc
$ ./a.out
0, 1, 2
1, 2, 3
2, 3, 4
3, 4, 5
4, 5, 6
5, 6, 7
...
```

With the definitions `namespace sr = std::ranges`, and `namespace sv = sr::views`

- `sv::enumerate`
- `sv::zip`
- `sv::zip_transform`
- `sv::adjacent`
- `sr::to`

Ranges in C++23

```
1  auto R = sv::iota(0UL, 50UL)
2       | sv::transform([](auto i) { return 2. * pi * i; })
3       | sr::to<std::vector>();
```

With the definitions `namespace sr = std::ranges`, and `namespace sv = sr::views`

- `sv::enumerate`
- `sv::zip`
- `sv::zip_transform`
- `sv::adjacent`
- `sr::to`

Recap of elementary features with an example

```
1  // Trivial piece of code as a background for discussions
2  // examples/demo_saxpy/saxpy_0.cc
3  // includes ...
4  auto main() -> int
5  {
6      const std::vector inp1 { 1., 2., 3., 4., 5. };
7      const std::vector inp2 { 9., 8., 7., 6., 5. };
8      std::vector outp(inp1.size(), 0.);
9
10     auto saxpy = [](double a,
11                     const std::vector<double>& x,
12                     const std::vector<double>& y,
13                     std::vector<double>&z) {
14         std::transform(x.begin(), x.end(), y.begin(), z.begin(),
15                        [a](double X, double Y){ return a * X + Y; });
16     };
17
18     std::ostream_iterator<double> cout { std::cout, "\n" };
19     saxpy(10., inp1, inp2, outp);
20     copy(outp.begin(), outp.end(), cout);
21 }
```

Recap of elementary features with an example

```
1  // Trivial piece of code as a background for discussions
2  // examples/demo_saxpy/saxpy_0.cc
3  // includes ...
4  auto main() -> int
5  {
6      const std::vector inp1 { 1., 2., 3., 4., 5. };
7      const std::vector inp2 { 9., 8., 7., 6., 5. };
8      std::vector outp(inp1.size(), 0.);
9
10     auto saxpy = [](double a,
11                     const std::vector<double>& x,
12                     const std::vector<double>& y,
13                     std::vector<double>&z) {
14         std::transform(x.begin(), x.end(), y.begin(), z.begin(),
15                        [a](double X, double Y){ return a * X + Y; });
16     };
17
18     std::ostream_iterator<double> cout { std::cout, "\n" };
19     saxpy(10., inp1, inp2, outp);
20     copy(outp.begin(), outp.end(), cout);
21 }
```

How many syntax errors are there if we are using C++17 ?

A. 4

B. 3

C. 2

D. 0

Generic lambdas...

```
1  // examples/demo_saxpy/saxpy_1.cc
2  // includes ...
3
4  auto main() -> int
5  {
6      const std::vector inp1 { 1., 2., 3., 4., 5. };
7      const std::vector inp2 { 9., 8., 7., 6., 5. };
8      std::vector outp(inp1.size(), 0.);
9
10     auto saxpy = [](double a, auto&& x, auto&& y, auto& z) {
11         std::transform(x.begin(), x.end(), y.begin(), z.begin(),
12                        [a](auto X, auto Y){ return a * X + Y; });
13     };
14
15     std::ostream_iterator<double> cout { std::cout, "\n" };
16     saxpy(10., inp1, inp2, outp);
17     copy(outp.begin(), outp.end(), cout);
18 }
```

We can make the lambda more compact by making it generic. But now the types of `x`, `y` and `z` are deduced independently. How can we keep it generic, and yet indicate that we want the same types for `x` and `y` ?

Explicit template syntax for lambdas

```
1  // examples/demo_saxpy/saxpy_2.cc
2  // includes ...
3  auto main() -> int
4  {
5      const std::vector inp1 { 1., 2., 3., 4., 5. };
6      const std::vector inp2 { 9., 8., 7., 6., 5. };
7      std::vector outp(inp1.size(), 0.);
8      auto saxpy = []<class T, class T_in, class T_out>
9                    (T a, const T_in& x, const T_in& y, T_out& z) {
10         std::transform(x.begin(), x.end(), y.begin(), z.begin(),
11                        [a](T X, T Y){ return a * X + Y; });
12     };
13
14     std::ostream_iterator<double> cout { std::cout, "\n" };
15     saxpy(10., inp1, inp2, outp);
16     copy(outp.begin(), outp.end(), cout);
17 }
```

For normal function templates, we could easily express relationships among the types of different parameters. Now, we can do that for generic lambdas.

Constraining generic functions

```
1  // examples/demo_saxpy/saxpy_3.cc
2      const std::vector inp1 { 1., 2., 3., 4., 5. };
3      const std::vector inp2 { 9., 8., 7., 6., 5. };
4      std::vector outp(inp1.size(), 0.);
5
6      auto saxpy = []<class T_in, class T_out>
7          (typename std::remove_cvref_t<T_in>::value_type a,
8           T_in&& x, T_in&& y, T_out& z) {
9          using in_element_type = typename std::remove_cvref_t<T_in>::value_type;
10         using out_element_type = typename std::remove_cvref_t<T_out>::value_type;
11         static_assert(std::is_same_v<in_element_type, out_element_type>,
12                       "Input and output element types must match!");
13         std::transform(x.begin(), x.end(), y.begin(), z.begin(),
14                        [a](in_element_type X, in_element_type Y){ return a * X + Y; });
15     };
16 //...
17     std::ostream_iterator<double> cout { std::cout, "\n" };
18     saxpy(10., inp1, inp2, outp);
```

At the least, we can use this to get helpful error messages when we use the function in a way that violates our assumptions.

Constraining generic functions

```
1  // examples/demo_saxpy/saxpy_3b.cc
2      const std::vector inp1 { 1., 2., 3., 4., 5. };
3      const std::vector inp2 { 9., 8., 7., 6., 5. };
4      std::vector outp(inp1.size(), 0);
5
6      auto saxpy = []<class T_in, class T_out>
7          (typename std::remove_cvref_t<T_in>::value_type a,
8           T_in&& x, T_in&& y, T_out& z) {
9          using in_element_type = typename std::remove_cvref_t<T_in>::value_type;
10         using out_element_type = typename std::remove_cvref_t<T_out>::value_type;
11         static_assert(std::is_same_v<in_element_type, out_element_type>,
12                       "Input and output element types must match!");
13         std::transform(x.begin(), x.end(), y.begin(), z.begin(),
14                        [a](in_element_type X, in_element_type Y){ return a * X + Y; });
15     };
16
17     std::ostream_iterator<double> cout { std::cout, "\n" };
18     saxpy(10., inp1, inp2, outp);
```

```
saxpy_3b.cc:16:9: error: static_assert failed due to requirement
'std::is_same_v<double, int>' "Input and output element types must match!"
```

Constraining generic functions

```
1   const std::array inp1 { 1., 2., 3., 4., 5. };
2   const std::array inp2 { 9., 8., 7., 6., 5. };
3   std::vector outp(inp1.size(), 0.);
4
5   auto saxpy = []<class T_in, class T_out>
6       (typename std::remove_cvref_t<T_in>::value_type a,
7        T_in&& x, T_in&& y, T_out& z) {
8       using in_element_type = typename std::remove_cvref_t<T_in>::value_type;
9       using out_element_type = typename std::remove_cvref_t<T_out>::value_type;
10      static_assert(std::is_same_v<in_element_type, out_element_type>,
11                    "Input and output element types must match!");
12      std::transform(x.begin(), x.end(), y.begin(), z.begin(),
13                     [a](in_element_type X, in_element_type Y){ return a * X + Y; });
14  };
15
16  std::ostream_iterator<double> cout { std::cout, "\n" };
17  saxpy(10., inp1, inp2, outp);
18  copy(outp.begin(), outp.end(), cout);
```

Different container types are acceptable as long as element types match! Controlled generic behaviour!

Constraining generic functions

```cpp
// examples/demo_saxpy/saxpy_4.cc
// includes ...
template <class T_in, class T_out>
auto saxpy(typename std::remove_cvref_t<T_in>::value_type a,
    T_in&& x, T_in&& y, T_out& z)
{
    using in_element_type = typename std::remove_cvref_t<T_in>::value_type;
    using out_element_type = typename std::remove_cvref_t<T_out>::value_type;
    static_assert(std::is_same_v<in_element_type, out_element_type>,
        "Input and output element types must match!");

    std::transform(x.begin(), x.end(), y.begin(), z.begin(),
        [a](in_element_type X, in_element_type Y) { return a * X + Y; });
}
auto main() -> int { ... }
```

Constraining normal function templates with template metaprogramming is an old technique. The syntax has become clearer with newer standards. Still, we are not expressing in code that the template parameters `T_in` and `T_out` should be array like objects, with `begin()`, `end()` etc.

std::span as function parameters

```
1  // examples/demo_saxpy/saxpy_5.cc
2  // other includes
3  #include <span>
4  template <class T>
5  void saxpy(T a, std::span<const T> x, std::span<const T> y, std::span<T> z)
6  {
7      std::transform(x.begin(), x.end(), y.begin(), z.begin(),
8          [a](T X, T Y) { return a * X + Y; });
9  }
10
11 auto main() -> int { ... }
```

- `std::span<T>` is a non-owning adaptor ("view") for an existing array of objects in memory. It is like a pointer and a size.
- Provides an STL compatible interface
- Can be constructed from typical array like containers, e.g., `vector` `array`, C-style arrays ...
- Writing the `saxpy` function in terms of the `span` allows us to easily express that the element types in all three containers must be the same as the scalar.
- Still general enough to be used with different container types and different `T`

Exercise 1.13:

The examples used in these slides are all present in the `examples/demo_saxpy` folder of your course material. Check examples `saxpy_1.cc` through `saxpy_5.cc` containing the various version discussed so far. The important C++20 features we have revisited in this section so far, are explicit template syntax for lambdas and `std::span`.

std::span as function parameters

```cpp
// examples/demo_saxpy/saxpy_5.cc
// other includes
#include <span>
template <class T>
void 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; });
}
auto main() -> int
{
    const std::array inp1 { 1., 2., 3., 4., 5. };
    const std::array inp2 { 9., 8., 7., 6., 5. };
    std::vector outp(inp1.size(), 0.);
    saxpy(10., {inp1}, {inp2}, {outp});
}
```

No inheritance relationships between `span` and any other containers!

std::span as function parameters

```
1  // examples/demo_saxpy/saxpy_5.cc
2  // other includes
3  #include <span>
4  template <class T>
5  void saxpy(T a, std::span<const T> x, std::span<const T> y, std::span<T> z)
6  {
7      std::transform(x.begin(), x.end(), y.begin(), z.begin(),
8          [a](T X, T Y) { return a * X + Y; });
9  }
10 auto main() -> int
11 {
12     const std::array inp1 { 1., 2., 3., 4., 5. };
13     const std::array inp2 { 9., 8., 7., 6., 5. };
14     std::vector outp(inp1.size(), 0.);
15     saxpy(10., {inp1}, {inp2}, {outp});
16 }
```

Can we restrict the scalar type to just floating point numbers, like `float` or `double` ?

Constraining templates using concepts

```
1  template <class T>
2  auto saxpy(T a, std::span<const T> x, std::span<const T> y, std::span<T> z)
3      -> std::enable_if_t<std::is_floating_point_v<T>, void>
4  {
5      std::transform(x.begin(), x.end(), y.begin(), z.begin(),
6          [a](T X, T Y) { return a * X + Y; });
7  }
```

SFINAE: "Substitution Failure is not an error" is widely used to achieve the effect in C++.

- If `T` is not a floating point number, `is_floating_point_v` becomes false.
- `enable_if_t<cond, R>` is defined as `R` if `cond` is true. If not it is simply undefined!
- False condition to `enable_if_t` makes the result type, which is used as the output here, vanish.
- The compiler interprets that as : "Stupid substitution! If I do that the function ends up with no return type! That can't be the right function template. Let's look elsewhere!"

Does the job. But, in C++20, we have a better alternative...

Constraining templates using concepts

```
1  template <class T>
2  requires std::floating_point<T>
3  void saxpy(T a, std::span<const T> x, std::span<const T> y, std::span<T> z)
4  {
5      std::transform(x.begin(), x.end(), y.begin(), z.begin(),
6          [a](T X, T Y) { return a * X + Y; });
7  }
```

concepts: Named requirements on template parameters.

- Far easier to read than SFINAE (even the name!)
- If `MyAPI` is a **`concept`**, and `T` is a type, `MyAPI<T>` evaluates at compile time to either true or false.
- Concepts can be combined using conjunctions (`&&`) and disjunctions (`||`) to make other concepts.
- A **`requires`** clause introduces a constraint on a template type

A suitably designed set of concepts can greatly improve readability of template code

```cpp
// examples/demo_saxpy/saxpy_6.cc
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; });
}
auto main() -> int
{
    {
        const std::array inp1 { 1., 2., 3., 4., 5. };
        const std::array inp2 { 9., 8., 7., 6., 5. };
        std::vector outp(inp1.size(), 0.);
        saxpy(10., {inp1}, {inp2}, {outp});
    }
    {
        const std::array inp1 { 1, 2, 3, 4, 5 };
        const std::array inp2 { 9, 8, 7, 6, 5 };
        std::vector outp(inp1.size(), 0);
        saxpy(10, {inp1}, {inp2}, {outp});
    }
}
```

Using concepts for our example

```
1  // examples/demo_saxpy/saxpy_6b.cc
2  template <class T> concept Number = std::floating_point<T> or std::integral<T>;
3
4  template <Number T>
5  auto saxpy(T a, std::span<const T> x, std::span<const T> y, std::span<T> z)
6  {
7      std::transform(x.begin(), x.end(), y.begin(), z.begin(),
8          [a](T X, T Y) { return a * X + Y; });
9  }
```

Our function is still a function template. But it does not accept "anything" as input. Acceptable inputs must have the following properties:

- The scalar type (first argument here) is a number by our definition
- The next two are contiguously stored constant arrays of the same scalar type
- The last is another span of non-const objects of the same scalar type

Using standard concepts and ranges in our example

```
1  // examples/demo_saxpy/saxpy_7.cc
2  namespace sr = std::ranges;
3  auto saxpy(std::floating_point auto a,
4             sr::input_range auto&& x, sr::input_range auto&& y,
5             std::weakly_incrementable auto&& z)
6  {
7      sr::transform(x, y, z, [a](auto X, auto Y) { return a * X + Y; });
8  }
9  auto main() -> int
10 {
11     std::vector inp1 { 1., 2., 3., 4., 5. };
12     std::vector inp2 { 9., 8., 7., 6., 5. };
13     std::array  inp3 { 9., 8., 7., 6., 5. };
14     double cstyle[]  { 1., 2., 3., 4., 5. };
15     std::vector outp( inp1.size(), 0.);
16     saxpy(10., inp1, inp2, outp.begin());
17     saxpy(10., inp1, inp3, outp.begin());
18     saxpy(10., inp1, std::to_array(cstyle), outp.begin());
19 }
```

```cpp
namespace sr = std::ranges;
void saxpy(std::floating_point auto a,
           sr::input_range auto&& x, sr::input_range auto&& y,
           std::weakly_incrementable auto&& z) {
    sr::transform(x, y, z, [a](auto X, auto Y) { return a * X + Y; });
}
void saxpy(std::weakly_incrementable auto&& z, std::floating_point auto a,
           sr::input_range auto&& x, sr::input_range auto&& y) {
    sr::transform(x, y, z, [a](auto X, auto Y) { return a * X + Y; });
}
auto main() -> int {
    std::vector inp1 { 1., 2., 3., 4., 5. };
    std::vector inp2 { 9., 8., 7., 6., 5. };
    std::array  inp3 { 9., 8., 7., 6., 5. };
    double cstyle[]  { 1., 2., 3., 4., 5. };
    std::vector outp( inp1.size(), 0.);
    saxpy(10., inp1, inp2, outp.begin());
    saxpy(10., inp1, inp3, outp.begin());
    saxpy(10., inp1, std::to_array(cstyle), outp.begin());
    saxpy(outp.begin(), 10., inp1, inp3);
}
```

We can now specify our requirements thoroughly...

```
1  namespace sr = std::ranges;
2  template <std::floating_point D, sr::input_range IR, std::weakly_incrementable OI>
3  requires std::is_same_v<D, std::iter_value_t<IR>> and std::indirectly_writable<OI, D>
4  void saxpy(D a, IR x, IR y, OI z)
5  {
6      sr::transform(x, y, z, [a](auto X, auto Y) { return a * X + Y; });
7  }
8
9  template <std::floating_point D, sr::input_range IR, std::weakly_incrementable OI>
10 requires std::same_as<D, std::iter_value_t<IR>> and std::indirectly_writable<OI, D>
11 void saxpy(OI z, D a, IR x, IR y)
12 {
13     sr::transform(x, y, z, [a](const auto& X, const auto& Y) { return a * X + Y; });
14 }
15
```

Look up cppreference.com and find out what pre-defined concepts and ranges are available in the standard library.

Exercise 1.14:

The program `examples/demo_saxpy/saxpy_9.cc` contains this last version with the requirements on template parameters as well as two overloads. Verify that even if the two functions are both function templates with 4 function parameters, they are indeed distinct for the compiler. Depending on the placement of our arguments, one or the other version is chosen. Try changing data types uniformly in all parameters. Try using different numeric types between source, destination arrays. Try changing container types for the 3 containers involved.

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

```
1  for (auto i = 0UL; i < 100UL; ++i) {
2      std::cout << "i = " << i
3          << ", E_1 = " << cos(i * wn)
4          << ", E_2 = " << sin(i * wn)
5          << "\n";
6  }
```

```
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

```
1  for (auto i = 0UL; i < 100UL; ++i) {
2      std::cout << format(
3          "i = {:>4d}, E_1 = {:< 12.8f}, "
4          "E_2 = {:< 12.8f}\n",
5           i, cos(i * wn), sin(i * wn));
6  }
```

```
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

```
1  for (auto i = 0UL; i < 100UL; ++i) {
2      std::cout << format(
3          "i = {:>4d}, E_1 = {:< 12.8f}, "
4          "E_2 = {:< 12.8f}\n",
5           i, cos(i * wn), sin(i * wn));
6  }
```

```
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

- `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!

std::print

- Formats using the `std::format` syntax, but then directs the output to stdout, as if you had written `std::cout << std::format(...);`

Exercise 1.15:

A simple example demonstrating the text formatting library of C++20 is in `examples/format1.cc`. Replace cout+format by the equivalent use of `std::print`!

Optional values

```
1  #include <optional>
2  auto f(double x) -> std::optional<double> {
3      std::optional<double> ans;
4      const auto eps2 = 1.0e-24;
5      if (x >= 0) {
6          auto r0 = 0.5 * (1. + x);
7          auto r1 = x / r0;
8          while ((r0 - r1) * (r0 - r1) > eps2) {
9              r0 = 0.5 * (r0 + r1);
10             r1 = x / r0;
11         }
12         ans = r1;
13     }
14     return ans;
15 }
16 // Elsewhere...
17 std::cout << "Enter number : ";
18 std::cin >> x;
19 if (auto r = f(x); r.has_value()) {
20     std::cout << "The result is "
21               << r.value() << '\n';
22 }
```

- `std::optional<T>` is analogous to a box containing exactly one object of type `T` or nothing at all
- If created without any initialisers, the box is empty
- You store something in the box by assigning to the `optional`
- Evaluating the optional as a boolean gives a **`true`** outcome if there is an object inside, irrespective of the value of that object
- Empty box evaluates to **`false`**

Optional values

```
1  #include <optional>
2  auto f(double x) -> std::optional<double> {
3      std::optional<double> ans;
4      const auto eps2 = 1.0e-24;
5      if (x >= 0) {
6          auto r0 = 0.5 * (1. + x);
7          auto r1 = x / r0;
8          while ((r0 - r1) * (r0 - r1) > eps2) {
9              r0 = 0.5 * (r0 + r1);
10             r1 = x / r0;
11         }
12         ans = r1;
13     }
14     return ans;
15 }
16 // Elsewhere...
17 std::cout << "Enter number : ";
18 std::cin >> x;
19 if (auto r = f(x); r) {
20     std::cout << "The result is "
21               << *r << '\n';
22 }
```

- `std::optional<T>` is analogous to a box containing exactly one object of type `T` or nothing at all
- If created without any initialisers, the box is empty
- You store something in the box by assigning to the `optional`
- Evaluating the optional as a boolean gives a **`true`** outcome if there is an object inside, irrespective of the value of that object
- Empty box evaluates to **`false`**

C++23 std::expected

```cpp
#include <expected>
auto mysqrt(double x) -> std::expected<double, std::string> {
    const auto eps = 1.0e-12;
    const auto eps2 = eps * eps;
    if (x >= 0.) {
        auto r0 = 0.5 * (1. + x);
        auto r1 = x / r0;
        while ((r0 - r1) * (r0 - r1) > eps2) {
            r0 = 0.5 * (r0 + r1);
            r1 = x / r0;
        }
        return { r1 };
    } else {
        return std::unexpected { "Unexpected input!" };
    }
}
// Elsewhere...
if (auto rm = mysqrt(x); rm) std::cout << "Square root = " << rm.value() << "\n";
else std::cout << "Error: " << rm.error() << "\n";
```

- Similar to `std::optional`, but has more capacity to describe the error
- The *unexpected* value can be of a type of our choosing, making it very flexible

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 ...

```
#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
```

```
#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

```
1  // examples/hello_m.cc
2  import <iostream>;
3
4  auto main() -> int
5  {
6      std::cout << "Hello, world!\n";
7  }
```

- 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

Exercise 1.16:

Convert a few small example programs 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, examples demonstrating the date time library functions, like `feb.cc` and `advent.cc`.

Creating a module (example)

```
1  // saxpy.hh
2  #ifndef SAXPY_HH
3  #define SAXPY_HH
4  #include <algorithm>
5  #include <span>
6
7  template <class T>
8  concept Number = std::floating_point<T>
9                or std::integral<T>;
10 template <Number T>
11 auto saxpy(T a, std::span<const T> x,
12            std::span<const T> y,
13            std::span<T> z)
14 {
15     std::transform(x.begin(), x.end(),
16                    y.begin(), z.begin(),
17         [a](T X, T Y) {
18             return a * X + Y;
19         });
20 }
21 #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)

```
1  // usesaxpy.cc
2  #include <iostream>
3  #include <array>
4  #include <vector>
5  #include <span>
6  #include "saxpy.hh"
7
8  auto main() -> int
9  {
10     using namespace std;
11     const array inp1 { 1., 2., 3., 4., 5. };
12     const array inp2 { 9., 8., 7., 6., 5. };
13     vector outp(inp1.size(), 0.);
14
15     saxpy(10., {inp1}, {inp2}, {outp});
16     for (auto x : outp) cout << x << "\n";
17     cout << "::::::::::::::::::::::\n";
18 }
```

- 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)

Make a module interface unit

```
1  // saxpy.hh -> saxpy.ixx
2  #ifndef SAXPY_HH
3  #define SAXPY_HH
4  #include <algorithm>
5  #include <span>
6
7  template <class T>
8  concept Number = std::floating_point<T>
9                or std::integral<T>;
10 template <Number T>
11 auto saxpy(T a, std::span<const T> x,
12            std::span<const T> y,
13            std::span<T> z)
14 {
15     std::transform(x.begin(), x.end(),
16                    y.begin(), z.begin(),
17         [a](T X, T Y) {
18             return a * X + Y;
19         });
20 }
21 #endif
```

Creating a module (example)

```
1  // saxpy.hh -> saxpy.ixx
2  #ifndef SAXPY_HH
3  #define SAXPY_HH
4  #include <algorithm>
5  #include <span>
6
7  template <class T>
8  concept Number = std::floating_point<T>
9                or std::integral<T>;
10 template <Number T>
11 auto saxpy(T a, std::span<const T> x,
12            std::span<const T> y,
13            std::span<T> z)
14 {
15     std::transform(x.begin(), x.end(),
16                    y.begin(), z.begin(),
17         [a](T X, T Y) {
18             return a * X + Y;
19         });
20 }
21 #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)

```
1  // saxpy.hh -> saxpy.ixx
2
3
4  #include <algorithm>
5  #include <span>
6
7  template <class T>
8  concept Number = std::floating_point<T>
9                or std::integral<T>;
10 template <Number T>
11 auto saxpy(T a, std::span<const T> x,
12            std::span<const T> y,
13            std::span<T> z)
14 {
15     std::transform(x.begin(), x.end(),
16                    y.begin(), z.begin(),
17         [a](T X, T Y) {
18             return a * X + Y;
19         });
20 }
```

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)

```
1  // saxpy.hh -> saxpy.ixx
2
3  export module saxpy;
4  #include <algorithm>
5  #include <span>
6
7  template <class T>
8  concept Number = std::floating_point<T>
9                or std::integral<T>;
10 template <Number T>
11 auto saxpy(T a, std::span<const T> x,
12            std::span<const T> y,
13            std::span<T> z)
14 {
15     std::transform(x.begin(), x.end(),
16                    y.begin(), z.begin(),
17         [a](T X, T Y) {
18             return a * X + Y;
19         });
20 }
```

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)

```
1  // saxpy.hh -> saxpy.ixx
2
3  export module saxpy;
4  import <algorithm>;
5  import <span>;
6
7  template <class T>
8  concept Number = std::floating_point<T>
9                or std::integral<T>;
10 template <Number T>
11 auto saxpy(T a, std::span<const T> x,
12            std::span<const T> y,
13            std::span<T> z)
14 {
15     std::transform(x.begin(), x.end(),
16                    y.begin(), z.begin(),
17         [a](T X, T Y) {
18             return a * X + Y;
19         });
20 }
```

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)

```
1  // saxpy.hh -> saxpy.ixx
2
3  export module saxpy;
4  import <algorithm>;
5  import <span>;
6
7  template <class T>
8  concept Number = std::floating_point<T>
9                || std::integral<T>;
10 export template <Number T>
11 auto saxpy(T a, std::span<const T> x,
12            std::span<const T> y,
13            std::span<T> z)
14 {
15     std::transform(x.begin(), x.end(),
16                    y.begin(), z.begin(),
17         [a](T X, T Y) {
18             return a * X + Y;
19         });
20 }
```

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

Creating a module (example)

Use your module

```
1  // usesaxpy.cc
2  #include <iostream>
3  #include <array>
4  #include <vector>
5  #include <span>
6  #include "saxpy.hh"
7
8  auto main() -> int
9  {
10     using namespace std;
11     const array inp1 { 1., 2., 3., 4., 5. };
12     const array inp2 { 9., 8., 7., 6., 5. };
13     vector outp(inp1.size(), 0.);
14
15     saxpy(10., {inp1}, {inp2}, {outp});
16     for (auto x : outp) cout << x << "\n";
17     cout << "::::::::::::::::::::::\n";
18 }
```

Creating a module (example)

```
1  // usesaxpy.cc
2  import <iostream>;
3  import <array>;
4  import <vector>;
5  import <span>;
6  #include "saxpy.hh"
7
8  auto main() -> int
9  {
10     using namespace std;
11     const array inp1 { 1., 2., 3., 4., 5. };
12     const array inp2 { 9., 8., 7., 6., 5. };
13     vector outp(inp1.size(), 0.);
14
15     saxpy(10., {inp1}, {inp2}, {outp});
16     for (auto x : outp) cout << x << "\n";
17     cout << "::::::::::::::::::::::\n";
18 }
```

Use your module

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

Creating a module (example)

```
1  // usesaxpy.cc
2  import <iostream>;
3  import <array>;
4  import <vector>;
5  import <span>;
6  import saxpy;
7
8  auto main() -> int
9  {
10     using namespace std;
11     const array inp1 { 1., 2., 3., 4., 5. };
12     const array inp2 { 9., 8., 7., 6., 5. };
13     vector outp(inp1.size(), 0.);
14
15     saxpy(10., {inp1}, {inp2}, {outp});
16     for (auto x : outp) cout << x << "\n";
17     cout << "::::::::::::::::::::::\n";
18 }
```

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)

```
1  // usesaxpy.cc
2  import <iostream>;
3  import <array>;
4  import <vector>;
5  import <span>;
6  import saxpy;
7
8  auto main() -> int
9  {
10     using namespace std;
11     const array inp1 { 1., 2., 3., 4., 5. };
12     const array inp2 { 9., 8., 7., 6., 5. };
13     vector outp(inp1.size(), 0.);
14
15     saxpy(10., {inp1}, {inp2}, {outp});
16     for (auto x : outp) cout << x << "\n";
17     cout << ":::::::::::::::::::::\n";
18 }
```

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.

Modules: compilation

- 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 a `scan-dep` utility to discover dependencies across sources
  - 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

Modules: Managing compilation using CMake

```
cmake_minimum_required(VERSION 3.28)
# ...
set(CMAKE_CXX_STANDARD 20)
target_sources(use_saxpy
    PUBLIC
    usesaxpy.cc
)
target_sources(use_saxpy
  PUBLIC
    FILE_SET saxpy_module TYPE CXX_MODULES FILES
    saxpy.ixx
)
```

```
$ mkdir build && cd build
$ cmake -DCMAKE_GENERATOR=Ninja ..
$ ninja
$
```

- Source files may produce as well as consume modules
- Modules must be built before they can be consumed
- `CMake 3.28`, `clang++ 18.0` and `Ninja 1.1`
- As of `3.28`, compilation of C++ modules is not considered experimental, even with multiple interdependent modules
- To use standard library modules of C++23, i.e., just writing `import std;` in place of all the `#include <standard library stuff>` headers, a minimum of `CMake 3.30` is required

Exercise 1.17:

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 1.18:

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.

Module interface and implementation units

- A translation unit with a module declaration is a module unit
- In a module unit, the first declaration must be a module declaration, e.g., `export module mname`, `module mname`, or simply `module;` which starts a *global module fragment* which ends with the module declaration.
- The global module fragment is the only place we can use traditional `#include` directives
- A module declaration can only be exported in one source file. When a module unit exports the module, it is called a *module interface unit*.
- Module interface units can export declarations and definitions, so that they become available whenever one imports the module.
- Implementation of functions declared in a module interface unit can be in other source files, which must also contain the module declaration `module mname` without the `export` keyword, so that the definitions are associated with the same module. These translation units are called *module implementation units*.

Module partitions

- Module partitions are satelite modules containing code which helps build up the module.
- They are declared like `export module A:B`, where `B` is a partition of module `A`
- A module partition can **only** be imported by other module units of the **same module**. For instance, to import the partition `B` above, in a module unit belonging to the module `A`, we have to write `import :B`.
- Each module must have one interface unit declaring the module without any partition names. That is then the primary module interface unit for that module.
- Definitions in a module partition can be imported and re-exported to make them available for use, outside of the module
- **Note:** module partitions are not independently importable "sub-modules"! When meaningful, a group of modules can be organised in a module – sub-module hierarchy by simply choosing their names, e.g., `MC`, `MC.MUCA`, `MC.Canon` etc.

Module partitions

```
1  // A-B.cc
2  module;
3  #include <string>
4  export module A:B; // partition module interface unit
5  export auto hello() -> std::string { return "Hello"; }
6  // A-C.cc
7  module;
8  // global module fragment
9  #include <string>
10 module A:C; // partition module implementation unit
11 // visible inside any module unit of 'A' importing ':C'.
12 char const* worldImpl() { return "World"; }
```

Module partitions

```
1  // A.cc
2  module;
3  #include <string>
4  export module A;
5  // primary module interface unit
6  export import :B;
7  // Import symbols from :B and make the exported declarations available
8  // to users of module A
9
10 import :C;
11 // Import symbols from :C. We can use them, since we are in the same
12 // module, but we can not export them, since the partition :C does
13 // not export anything
14 // export import :C; // Error!
15
16 // visible by any translation unit importing 'A'.
17 export auto world() -> std::string
18 {
19     return worldImpl();
20 }
```

Demo: Travelling salesman problem

Exercise 1.19:

The directory `examples/TSP` contains a set of files implementing a brute-force approach to solve a simplified version of the travelling salesman problem: Given the coordinates of N cities on a flat 2D surface, we seek the path $\{c[0], c[i_1], ..., c[i_{N-1}], c[0]\}$, going from the city 0 to the other $N-1$ cities in the order $i_1, i_2 ..., i_{N-1}$ and returning to the city 0 so that the total travelled distance is minimised. The solution is only meant to demonstrate C++ syntax, and uses the algorithm `std::next_permutation` to generate new itineraries. We demonstrate the use of module partitions as well as C++23 standard library modules. As the support for this feature is not mature, we keep the headers in our sources and put them behind a MACRO guard.

```
# Configure without import std
CXX=clang++ cmake -DCMAKE_BUILD_TYPE=Release -DCMAKE_GENERATOR=Ninja ..
# or with import std
CXX=clang++ cmake -DCMAKE_BUILD_TYPE=Release -DCMAKE_GENERATOR=Ninja\
  -DTSP_USE_IMPORT_STD=1 \
  -DCMAKE_CXX_FLAGS="-stdlib=libc++ -std=c++23" ..
ninja
```