HIGH PERFORMANCE SCIENTIFIC COMPUTING IN C++

PATC HPC C++ course 2021

21–24 June 2021 | Sandipan Mohanty | Forschungszentrum Jülich, Germany

High performance scientific computing in C++

Online couse infrastructure: establish your workflow!

- External participants: please download material as mentioned in the mail and prepare your set up.
- Regular participants, please login to the Jupyter-JSC system
- After logging in, try to add a new jupyterlab. Choose JUSUF as the system and training2119 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/set_vars.sh
```

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

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

You should see GCC version 11.1 and Clang version 12.0.

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
  - `cxx2021` : (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) Compiler wrapper for `g++` using common options `-std=c++20 -pedantic -Wall -O3 -march=native`
  - `A` : (SC) Similar to `G` , but for Clang. It also uses Clang's own implementation of the standard library, `libc++`
  - `B` : (SC) Similar to `A` . But it uses GCC's implementation of the standard library.

High performance scientific computing in C++

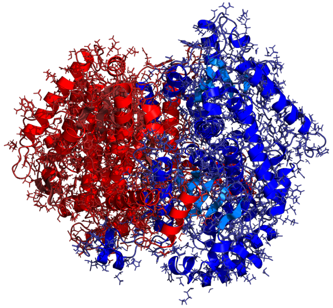
- The scripts `G`, `A` and `B` default to producing executables whose names are deduced from the names of the source files. E.g.: `G hello.cc` produces the executable `hello.g`, `A hello.cc` produces `hello.l` and `B hello.cc` produces `hello.b`
- They recognize libraries we need during the course: `G -tbb xyz.cc` compiles `xyz.cc` with suitable include and library options to use TBB
- The folder `yourworkspace/software`: Any software you build and install with this installation prefix will be found by the compilers
- 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:

  ```
  srun --nodes=1 --cpus-per-task=64 a.out [options]
  ```
- For external participants: 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 scripts `G`, `A` and `B` can be found in `bin`.
- 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 C++17 parallel algorithms and Intel (R) Threading Building Blocks
- GPU programming with NVidia CUDA and Thrust

The default C++ standard for code samples, examples exercises etc. is C++20.

A brief introduction to C++20

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

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

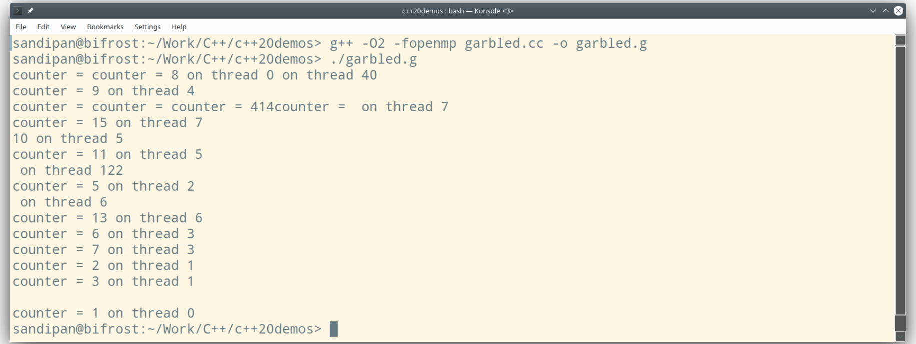
std::osyncstream

```cpp
#include <iostream>
#include <omp.h>

auto main() -> int
{
    #pragma omp parallel for
    for (auto i = 0UL; i < 100UL; ++i) {
       std::cout << "counter = " << i << " on thread "
                 << omp_get_thread_num() << "\n";
    }
}
```

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

std::osyncstream


c++20demos : bash — Konsole <3>
File Edit View Bookmarks Settings Help

```
sandipan@bifrost:~/Work/C++/c++20demos> g++ -O2 -fopenmp garbled.cc -o garbled.g
sandipan@bifrost:~/Work/C++/c++20demos> ./garbled.g
counter = counter = 8 on thread 0 on thread 40
counter = 9 on thread 4
counter = counter = counter = 414counter =  on thread 7
counter = 15 on thread 7
10 on thread 5
counter = 11 on thread 5
 on thread 122
counter = 5 on thread 2
 on thread 6
counter = 13 on thread 6
counter = 6 on thread 3
counter = 7 on thread 3
counter = 2 on thread 1
counter = 3 on thread 1

counter = 1 on thread 0
sandipan@bifrost:~/Work/C++/c++20demos> 
```

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

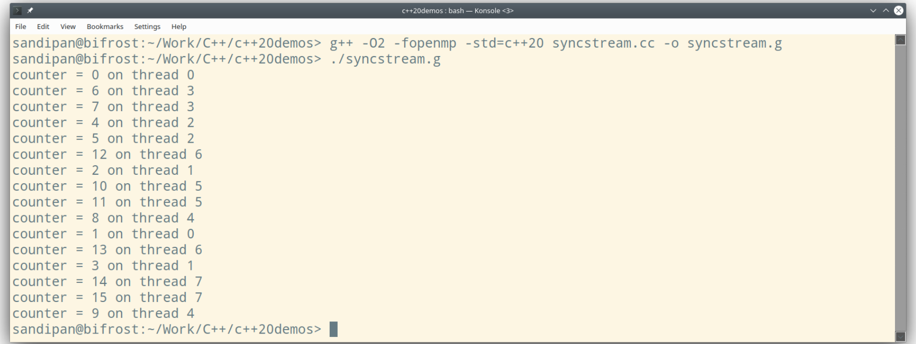
std::osyncstream

```cpp
#include <iostream>
#include <syncstream>
#include <omp.h>

auto main() -> int
{
    #pragma omp parallel for
    for (auto i = 0UL; i < 100UL; ++i) {
        std::osyncstream{std::cout} << "counter = " << i << " on thread "
                                   << omp_get_thread_num() << "\n";
    }
}
```

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

std::osyncstream


c++20demos : bash — Konsole <3>
File Edit View Bookmarks Settings Help

```
sandipan@bifrost:~/Work/C++/c++20demos> g++ -O2 -fopenmp -std=c++20 syncstream.cc -o syncstream.g
sandipan@bifrost:~/Work/C++/c++20demos> ./syncstream.g
counter = 0 on thread 0
counter = 6 on thread 3
counter = 7 on thread 3
counter = 4 on thread 2
counter = 5 on thread 2
counter = 12 on thread 6
counter = 2 on thread 1
counter = 10 on thread 5
counter = 11 on thread 5
counter = 8 on thread 4
counter = 1 on thread 0
counter = 13 on thread 6
counter = 3 on thread 1
counter = 14 on thread 7
counter = 15 on thread 7
counter = 9 on thread 4
sandipan@bifrost:~/Work/C++/c++20demos>
```

Immediate functions

```cpp
// examples/immediate.cc
constexpr auto cxpr_sqr(auto x) { return x * x; }
consteval auto cevl_sqr(auto x) { return x * x; }

auto main(int argc, char* argv[]) -> int
{
    std::array<double, cxpr_sqr(14)> A;
    std::array<double, cevl_sqr(14)> B;
    std::cout << cxpr_sqr(argc) << "\n";
    std::cout << cevl_sqr(argc) << "\n";
}
```

- `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 makes possible certain accidental use of these functions so that they are not evaluated at compile time.
- 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

```
// 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 initialized 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

- `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.
- `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.

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.

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

Concepts

Named requirements on template parameters

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

constexpr auto flag1 = PowerOfTwo<2048>; // Compiler sets flag1 to True
constexpr auto flag2 = PowerOfTwo<2056>; // Compiler sets flag2 to False
```

Concepts

Named requirements on template parameters

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

template <class T, int N> requires PowerOfTwo<N>
struct MyMatrix {
    // code which assumes that the square matrix size is a power of two
};
```

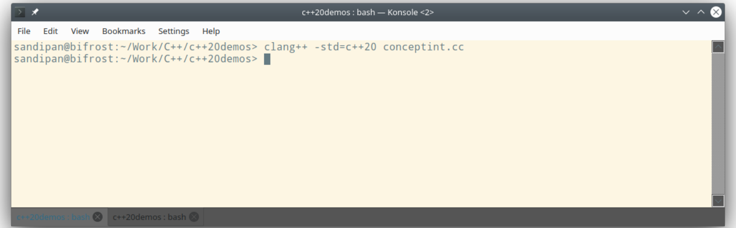
Concepts

Named requirements on template parameters

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

template <class T, int 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, 16>{};
}
```


c++20demos : bash — Konsole <2>
File Edit View Bookmarks Settings Help
c++20demos : bash
c++20demos : bash

```
sandipan@bifrost:~/Work/C++/c++20demos> clang++ -std=c++20 conceptint.cc
sandipan@bifrost:~/Work/C++/c++20demos> 
```

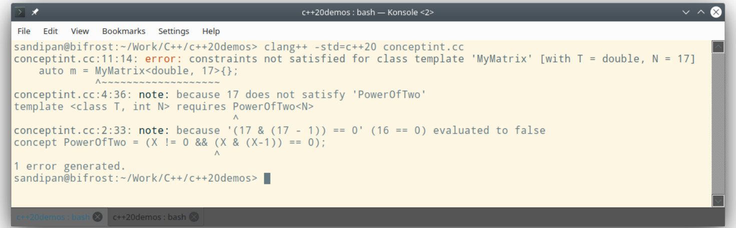
Concepts

Named requirements on template parameters

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

template <class T, int 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, 17>{};
}
```


c++20demos : bash — Konsole <2>
File Edit View Bookmarks Settings Help
c++20demos : bash
c++20demos : bash

```
sandipan@bifrost:~/Work/C++/c++20demos> clang++ -std=c++20 conceptint.cc
conceptint.cc:11:14: error: constraints not satisfied for class template 'MyMatrix' [with T = double, N = 17]
    auto m = MyMatrix<double, 17>{};
             ^~~~~~~~~~~~~~~~~~~~
conceptint.cc:4:36: note: because 17 does not satisfy 'PowerOfTwo'
template <class T, int N> requires PowerOfTwo<N>
                                   ^
conceptint.cc:2:33: note: because '(17 & (17 - 1)) == 0' (16 == 0) evaluated to false
concept PowerOfTwo = (X != 0 && (X & (X-1)) == 0);
                                ^
1 error generated.
sandipan@bifrost:~/Work/C++/c++20demos> 
```

Concepts

Named requirements on template parameters

```
template <int X> concept PowerOfTwo = (X != 0 && (X & (X-1)) == 0);
template <class T> concept Number = std::is_integral_v<T> or std::is_floating_point_v<T>;
```

Concepts

Named requirements on template parameters

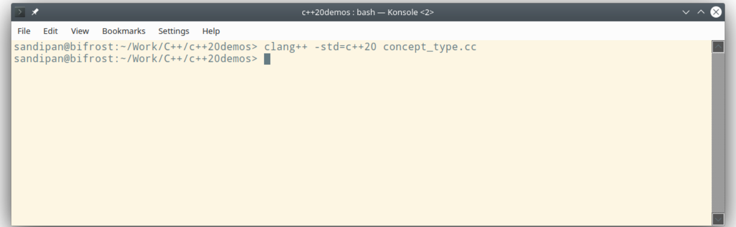
```
template <int X> concept PowerOfTwo = (X != 0 && (X & (X-1)) == 0);
template <class T> concept Number = std::is_integral_v<T> or std::is_floating_point_v<T>;

template <class T, int 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

```
template <int X> concept PowerOfTwo = (X != 0 && (X & (X-1)) == 0);
template <class T> concept Number = std::is_integral_v<T> or std::is_floating_point_v<T>;
template <class T, int 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, 16>{};
}
```

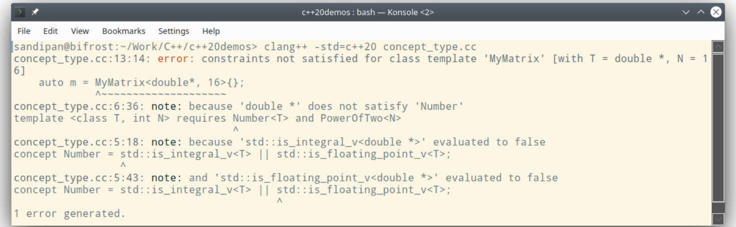

c++20demos : bash — Konsole <2>
File Edit View Bookmarks Settings Help

```
sandipan@bifrost:~/Work/C++/c++20demos> clang++ -std=c++20 concept_type.cc
sandipan@bifrost:~/Work/C++/c++20demos> 
```

Concepts

Named requirements on template parameters

```
template <int X> concept PowerOfTwo = (X != 0 && (X & (X-1)) == 0);
template <class T> concept Number = std::is_integral_v<T> or std::is_floating_point_v<T>;
template <class T, int 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*, 16>{};
}
```


c++20demos : bash — Konsole <2>
File Edit View Bookmarks Settings Help

```
sandipan@bifrost:~/Work/C++/c++20demos> clang++ -std=c++20 concept_type.cc
concept_type.cc:13:14: error: constraints not satisfied for class template 'MyMatrix' [with T = double *, N = 1
6]
    auto m = MyMatrix<double*, 16>{};
             ^~~~~~~~~~~~~~~~~~~~~~
concept_type.cc:6:36: note: because 'double *' does not satisfy 'Number'
template <class T, int N> requires Number<T> and PowerOfTwo<N>
                                   ^
concept_type.cc:5:18: note: because 'std::is_integral_v<double *>' evaluated to false
concept Number = std::is_integral_v<T> || std::is_floating_point_v<T>;
                 ^
concept_type.cc:5:43: note: and 'std::is_floating_point_v<double *>' evaluated to false
concept Number = std::is_integral_v<T> || std::is_floating_point_v<T>;
                                          ^
1 error generated.
```

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<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<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<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<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;
```

- 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

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

```
template <class T> requires Indexable<T>
auto f(T&& x) -> unsigned long;
void elsewhere() {
    std::vector<Protein> v;
    std::array<NucleicAcidType, 4> NA;
    f(v); // OK
    f(NA); // OK
    f(4); // No match!
}
```

Using concepts

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

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

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

```cpp
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

Overloading based on concepts

```
template <class N>
concept Number = std::is_floating_point_v<N>
                  or std::is_integral_v<N>;
template <class N>
concept NotNumber = not Number<N>;
void proc(Number auto&& x) {
    std::cout << "Called proc for numbers";
}
void proc(NotNumber auto&& x) {
    std::cout << "Called proc for non-numbers";
}
auto main() -> int {
    proc(-1);
    proc(88UL);
    proc("0118 999 88199 9119725   3");
    proc(3.141);
    proc("eighty"s);
}
```

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

Overloading based on concepts

```cpp
template <class N>
concept Number = std::is_floating_point_v<N>
                 or std::is_integral_v<N>;
template <class N>
concept NotNumber = not Number<N>;
void proc(Number auto&& x) {
    std::cout << "Called proc for numbers";
}
void proc(NotNumber auto&& x) {
    std::cout << "Called proc for non-numbers";
}
auto main() -> int {
    proc(-1);
    proc(88UL);
    proc("0118 999 88199 9119725   3");
    proc(3.141);
    proc("eighty"s);
}
```

- Constraints on template parameters are not just "documentation" or decoration.
- 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 nonTemplateMemberFunction() -> std::enable_if_t<std::is_integral_v<T>, int> {
        return 42;
    }
    auto other() -> std::string { return "something else"; }
};
auto main() -> int {
    ClassTemp<int> x;
    std::cout << x.nonTemplateMemberFunction() << "\n";
    std::cout << x.other() << "\n";
}
```

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

Constraining non-template members of class templates

```
template <class T> struct ClassTemp {
    auto nonTemplateMemberFunction() -> std::enable_if_t<std::is_integral_v<T>, int> {
        return 42;
    }
    auto other() -> std::string { return "something else"; }
};
auto main() -> int {
    ClassTemp<double> x;
    std::cout << x.nonTemplateMemberFunction() << "\n";
    std::cout << x.other() << "\n";
}
```

```
$ g++ -std=c++20 nontempconstr.cc
error: no type named ‘type’ in ‘struct std::enable_if<false, int>’
 2514 |     using enable_if_t = typename enable_if<_Cond, _Tp>::type;
nontempconstr.cc:17:20: error: ‘struct ClassTemp<double>’ has no member named
     ‘nonTemplateMemberFunction’
     |     std::cout << x.nonTemplateMemberFunction() << "\n";
$
```

Constraining non-template members of class templates

```cpp
template <class T> struct ClassTemp {
    auto nonTemplateMemberFunction() -> std::enable_if_t<std::is_integral_v<T>, int> {
        return 42;
    }
    auto other() -> std::string { return "something else"; }
};
auto main() -> int {
    ClassTemp<double> x;
    // std::cout << x.nonTemplateMemberFunction() << "\n";
    std::cout << x.other() << "\n";
}
```

```
$ g++ -std=c++20 nontempconstr.cc
error: no type named ‘type’ in ‘struct std::enable_if<false, int>’
 2514 |     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

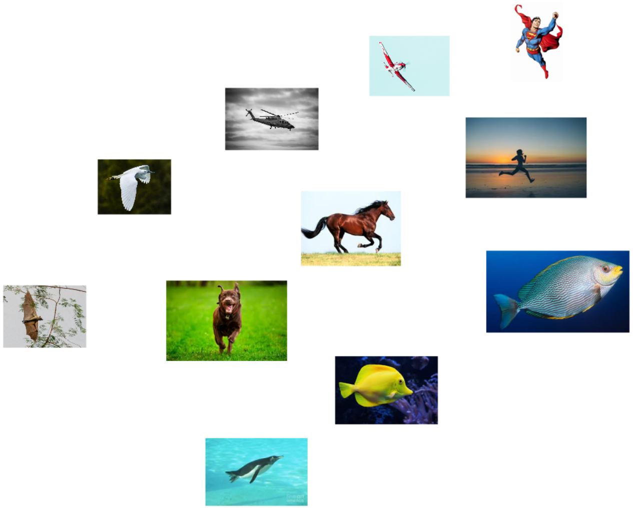
```cpp
template <class N> concept Integer = std::is_integral_v<N>;

template <class T>
struct ClassTemp {
    auto nonTemplateMemberFunction() -> int requires Integer<T> { return 42; }
    auto other() -> std::string { return "something else"; }
};
auto main() -> int {
    ClassTemp<double> x;
    //std::cout << x.nonTemplateMemberFunction() << "\n";
    std::cout << x.other() << "\n";
}
```

```
$ 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)

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

The range concept

```python
def python_sum(Container, start=0):
    res = start
    for x in Container:
        res += x
    return res
```

The range concept

```cpp
auto sum(auto&& Container, auto start = 0) {
    for (auto&& el : Container) start += el;
    return start;
}
```

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

```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 sr::forward_range<T>
auto sum(T&& a, element<T> start) {
    for (auto&& el : a) start += el;
    return start;
}
```

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

```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 sr::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);
}
```

- 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

```
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 sr::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);
}
```

```
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";
}
```

Exercise 1.1:

For exercises in this course, I recommend making a copy of the entire `examples` folder to your `work` folder, and doing all edits, compilations etc. there. Leave the original material as it is, because the `setup.sh` script needs it to be unchanged for it to update the material for you.

- Build and run the examples `immediate.cc` , `desig2.cc` , `conceptint.cc` , `concept_type.cc` , `overload_w_concepts.cc` , `nontempconstr.cc` , `cpp_sum_2.cc` and `syncstream.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++ desig2.cc
a.out
```

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

```
A desig2.cc && ./desig2.l
```

- Some programs using `<ranges>` will have to be compiled using GCC

Recap of elementary features with an example

```
// Trivial piece of code as a background for discussions
// examples/saxpy_0.cc
// includes ...
auto main() -> int
{
    const std::vector inp1 { 1., 2., 3., 4., 5. };
    const std::vector inp2 { 9., 8., 7., 6., 5. };
    std::vector outp(inp1.size(), 0.);

    auto saxpy = [](double a,
                    const std::vector<double>& x,
                    const std::vector<double>& y,
                    std::vector<double>&z) {
        std::transform(x.begin(), x.end(), y.begin(), z.begin(),
                   [a](double X, double Y){ return a * X + Y; });
    };

    std::ostream_iterator<double> cout { std::cout, "\n" };
    saxpy(10., inp1, inp2, outp);
    copy(outp.begin(), outp.end(), cout);
}
```

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

A. 4
B. 3
C. 2
D. 0

```cpp
// examples/saxpy_1.cc
// includes ...

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

    auto saxpy = [](double a, auto&& x, auto&& y, auto& z) {
        std::transform(x.begin(), x.end(), y.begin(), z.begin(),
                       [a](auto X, auto Y){ return a * X + Y; });
    };

    std::ostream_iterator<double> cout { std::cout, "\n" };
    saxpy(10., inp1, inp2, outp);
    copy(outp.begin(), outp.end(), cout);
}
```

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` ?

```cpp
// examples/saxpy_2.cc
// includes ...

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

    auto saxpy = [] <class T, class T_in, class T_out> (T a,
                                                         const T_in& x,
                                                         const T_in& y,
                                                         T_out& z) {
        std::transform(x.begin(), x.end(), y.begin(), z.begin(),
                       [a](T X, T Y){ return a * X + Y; });
    };

    std::ostream_iterator<double> cout { std::cout, "\n" };
    saxpy(10., inp1, inp2, outp);
    copy(outp.begin(), outp.end(), cout);
}
```

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

```
// examples/saxpy_3.cc
    const std::vector inp1 { 1., 2., 3., 4., 5. };
    const std::vector inp2 { 9., 8., 7., 6., 5. };
    std::vector outp(inp1.size(), 0.);

    auto saxpy = []<class T_in, class T_out>
        (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; });
    };
//...
    std::ostream_iterator<double> cout { std::cout, "\n" };
    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.

```cpp
// examples/saxpy_3b.cc
    const std::vector inp1 { 1., 2., 3., 4., 5. };
    const std::vector inp2 { 9., 8., 7., 6., 5. };
    std::vector outp(inp1.size(), 0);

    auto saxpy = []<class T_in, class T_out>
        (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; });
    };

    std::ostream_iterator<double> cout { std::cout, "\n" };
    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!"
```

```cpp
const std::array inp1 { 1., 2., 3., 4., 5. };
const std::array inp2 { 9., 8., 7., 6., 5. };
std::vector outp(inp1.size(), 0.);

auto saxpy = []<class T_in, class T_out>
    (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; });
};

std::ostream_iterator<double> cout { std::cout, "\n" };
saxpy(10., inp1 , inp2 , outp );
copy(outp.begin(), outp.end(), cout);
```

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

```
// examples/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.

```cpp
// examples/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 { ... }
```

- `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 1.2:

The examples used in these slides are all present in the `examples` 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 so far are explicit template syntax for lambdas and `std::span`.

```cpp
// examples/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!

```cpp
// examples/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});
}
```

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

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

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...

```cpp
template <class T>
requires std::is_floating_point_v<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; });
}
```

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/saxpy_6.cc
template <class T>
concept Number = std::is_floating_point_v<T> or std::is_integral_v<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

```cpp
// examples/saxpy_6b.cc
template <class T>
concept Number = std::is_floating_point_v<T> or std::is_integral_v<T> ;

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

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

Predefined useful concepts

Many concepts useful in building our own concepts are available in the standard library header `<concepts>`. Compiler support in October 2020 is far from uniform though.

- `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

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

Using standard concepts and ranges in our example

```cpp
// examples/saxpy_7.cc
namespace sr = std::ranges;
auto 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; });
}
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());
}
```

```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...

```cpp
namespace sr = std::ranges;
template <std::floating_point D, sr::input_range IR, std::weakly_incrementable OI>
requires std::is_same_v<D, std::iter_value_t<IR>> and std::indirectly_writable<OI, D>
void saxpy(D a, IR x, IR y, OI z)
{
    sr::transform(x, y, z, [a](auto X, auto Y) { return a * X + Y; });
}

template <std::floating_point D, sr::input_range IR, std::weakly_incrementable OI>
requires std::is_same_v<D, std::iter_value_t<IR>> and std::indirectly_writable<OI, D>
void saxpy(OI z, D a, IR x, IR y)
{
    sr::transform(x, y, z, [a](const auto& X, const auto& Y) { return a * X + Y; });
}
```

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

Exercise 1.3:

The program `examples/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.

Use `g++` version 10 or later for this exercise. As of version 12.0, `clang++` does not have an adequate implementation of `<ranges>`

Reorganize with modules

Traditionally, C++ projects are organised into header and source files. For our `saxpy` program ...

```
#ifndef SAXPY_HH
#define SAXPY_HH
#include <algorithm>
#include <span>
template <class T>concept Number = std::is_floating_point_v<T> or std::is_integral_v<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.
- Any source `import` ing the module immediately has access to the precompiled syntax tree in the precompiled module files. This leads to faster compilation
- Modules also allow fine grained control over which parts of a module are visible from the importer source code. Let’s apply that to our `saxpy` code. In the following slide, we show the header from a few slides back, and the corresponding module file. Explanations follow…

```
#ifndef SAXPY_HH
#define SAXPY_HH
#include <algorithm>
#include <span>
template <class T>concept Number = std::is_floating_point_v<T> or std::is_integral_v<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(...);
}
#endif
```

```
export module saxpy;
import <algorithm>;
import <span>;
template <class T> concept Number = std::is_floating_point_v<T> or std::is_integral_v<T>;
export template <Number T>
auto saxpy(T a, std::span<const T> x, std::span<const T> y, std::span<T> z){
    std::transform(...);
}
```

Reorganize with modules

- No include guards necessary.
- `export module xyz;` indicates that we are introducing an importable module in this file
- Instead of including, we import other modules we need. The import statement ends with a semi-colon, like all regular C++ statements! It is not a pre-processor directive.
- The current implementation of modules in Clang makes standard library headers “importable” as shown here. But officially, the standard library does not consist of modules in C++20.
- Any names of variables, functions, classes, templates and concepts which need to be visible outside, must be “exported” with the `export` keyword.

Reorganize with modules

```
import <iostream>;
import <array>;
import <vector>;
import <iterator>;
import <span>;
import saxpy;
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.);
    std::ostream_iterator<double> cout { std::cout, "\n" };
    saxpy(10., {inp1}, {inp2}, {outp});
```

- To use facilities exported by a module, we have to import that module with the `import` keyword
- Observe that even if we import the module `saxpy` here, we have no access to the concept `Number` as that was not exported.

Exercise 1.4:

A version of the `saxpy` program used in these demonstrations is written using modules. The module file is called `examples/saxpy.ccm` (for C++ module). The file using the module is called `examples/usesaxpy.cc`. Familiarize yourself with the process of building applications with modules. Building procedure for modules differs quite a bit between the implementations of modules in Clang and GCC. We will use `clang++`, because that implementation has been around longer for us to experiment with, and this example has not been adapted to GCC's way of building modules. Check that you have a minimum version 10.0 for your `clang++`

```
clang++ -std=c++20 -stdlib=libc++ -fmodules --precompile saxpy.ccm -o saxpy.pcm
clang++ -std=c++20 -stdlib=libc++ -fmodules -fprebuilt-module-path=. usesaxpy.cc
```

Experiment by inserting a second overload of the `saxpy` function which takes the output location as the first argument instead of the scalar. Access it from `main`.

Ranges and views

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

Ranges and views

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

Ranges and views

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

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

Ranges and views

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

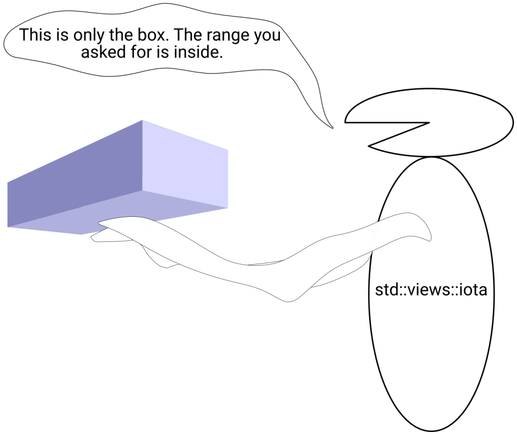
- All containers are ranges, but not all ranges are containers
- `std::string_view` is a perfectly fine range. Has iterators with the right properties. Has `begin()` and `end()` functions. It does not own the contents, but "ownership" is not part of the idea of a range.
- We could take this further by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

Ranges and views

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

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

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.

Ranges and views

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

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

Borrowed ranges

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

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

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

Borrowed ranges

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

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

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

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

Borrowed ranges

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

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

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

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

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

Borrowed ranges

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

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

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

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

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

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

Borrowed ranges

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

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

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

- The ranges algorithms are written with overloads such that when you pass an R-value reference of a container as input, the output type is `ranges::dangling`, an empty `struct` with no operations defined.
- `iter` here will be deduced to be of type `ranges::dangling`, and hence `*iter` leads to that insightful error message.
- When the input is an L-value reference, the algorithm returning the iterator does the work and returns an iterator.
- Valid use cases work painlessly, and invalid ones result in actionable insights from the compiler!

Borrowed ranges

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

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

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

View adaptors

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

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

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

View adaptors

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

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

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

View adaptors

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

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

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

View adaptors

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

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

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

View adaptors

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

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

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

View adaptors

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

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

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

So what are we going to do with this ?

View adaptors

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

View adaptors

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

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

View adaptors

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

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

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

View adaptors

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

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

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

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

View adaptors

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

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

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

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

View adaptors

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

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

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

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

View adaptors

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

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

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

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

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

View adaptors

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

- Command line of Linux, Mac OS ...
- 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 is_bad = [=](double x){ return std::fabs(x) > eps; };
    auto res = sv::iota(0UL, npoints)
                | sv::transform([=](size_t idx) -> double {
                      return std::lerp(0., 2*pi, idx * 1.0 / npoints);
                  })
                | sv::transform([ ](double x) -> double {
                      return sin(x) * sin(x) + cos(x) * cos(x) - 1.0;
                  });
    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.
- 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 1.5:

The code used for the demonstration of view adaptors is `examples/trig_views.cc`. Build this code with GCC and Clang. As it happens, `clang 11` is able to compile this, but it does not yet provide the `<ranges>` library. To compile with g++, you would do this (please use a 2020 version of gcc at the minimum!):

```
g++ -std=c++20 trig_views.cc
```

Exercise 1.6:

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

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

View adaptors

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

View adaptors

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

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

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

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

constexpr algorithms

STL algorithms, including those from `<ranges>` are now available at compile time, and can be used in `constexpr` functions.

```cpp
// examples/cxpr_algo0.cc
constexpr auto poly(double x) { return 3. * x * x * x - 2. * x * x + 0.877 * x - 1.0; }

constexpr auto bounds = []{
    std::array pre{ 1.2, 3.0, 0.99, 0.05, 1.44, 0.71, 0.881 };
    return std::ranges::minmax( pre | std::views::transform(poly) );
}();

auto main() -> int
{
    std::cout << bounds.min << "\t" << bounds.max << "\n";
    static_assert(bounds.min > -1. and bounds.min < -0.94);
}
```