PROGRAMMING IN C++

Jülich Supercomputing Centre

13 – 17 May 2024 | Sandipan Mohanty | Forschungszentrum Jülich, Germany

Chapter 6

Standard Template Library

Standard Template Library

DEPENDENT TYPES

```cpp
struct MyClass {
    // data members
    // member functions

    // member types
    using innter_type = int;

    // static data members, one for the whole class
    static const auto max_index = 777;
};

// This is how you might refer to the inner type elsewhere
std::vector<typename Myclass::innter_type> v;

for (int i = 0; i < MyClass::max_index; ++i) { }
```

STANDARD TEMPLATE LIBRARY

- Utilities
  - `pair`, `tuple`
  - `optional`, `expected`, `variant`, `any`
  - `bitset`, `bit`, `endian`, `bit_cast` safe integral comparisons
  - `initializer_list`, `type_traits`, `concepts`,
  - `filesystem`, `stacktrace`
  - `bind`, `placeholders`, `apply`, `invoke` ...
- Date and Time
- Random numbers
- Smart pointers
- File system
- Regular expressions
- Containers and Strings
- Algorithms, ranges
- Iterators
- `span` and `string_view`
- Fast character conversions
- Multi-threading, atomic types
- Parallel algorithms
- Text formatting

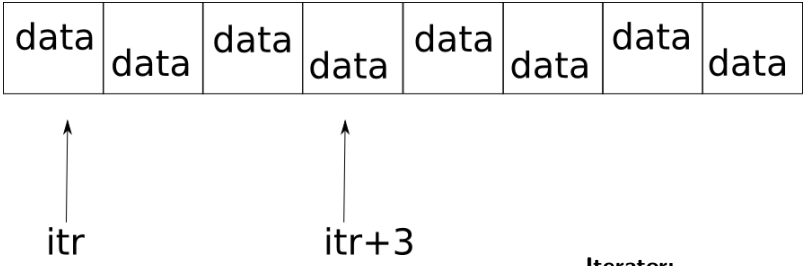
STL CONTAINERS

```
1  using namespace std;
2  int sz;
3  cin >> sz;
4  // vector<double> B(sz,3.0); // <- C++17 ->
5  vector B(sz, 3.0); // C++17 ->
6  vector c{1, 2, 3, 4};
7  c.push_back(5); // append
8  map<string, int> rank;
9  rank["Sirius"] = 1;
10 rank["Canopus"] = 2;
11 for (auto&& el : B) cout << el << "\n";
12 for (auto&& [key, val] : rank)
13     print("{} -> {}\n", key, val);
```

- Form: `container<datatype>`. Include file `containername`

- Many easy-to-use sequence types available in the STL
  - `vector` : Dynamic array type
  - `list` : Linked list
  - `map` : Sorted associative container
  - `unordered_map` : Hash table
- Not always necessary to explicitly state the element type. If there is an initialiser, element type can be inferred.
- Store a fixed kind of elements, determined at the point of declaration.
- They can grow at run time (except `std::array`)
- Whenever possible, prefer `array` or `vector`

VECTOR: DYNAMIC ARRAY CLASS TEMPLATE



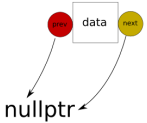
- Element type is a template parameter
- Consecutive elements in memory
- Can be accessed using an "iterator"

Iterator:

- Iterators are classes which pretend to be pointers
- They can be dereferenced with overloaded `*` and `->` operators to retrieve an element
- They can be moved forward or backward using overloaded `++` and `--` operators
- They can be compared for equality or inequality

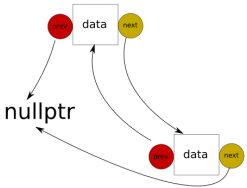
A LINKED LIST

A linked list is a collection of connected nodes. Each node has some data, and one or two pointers to other nodes. They are the "next" and "previous" nodes in the linked list. When "next" or "previous" does not exist, the pointer is set to `nullptr`



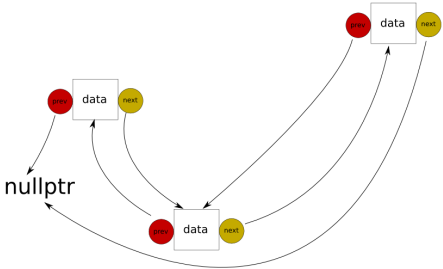
A LINKED LIST

When a new element is added to the end of a list, its "previous" pointer is set to the previous end of chain, and it becomes the target of the "next" pointer of the previous end.



A LINKED LIST

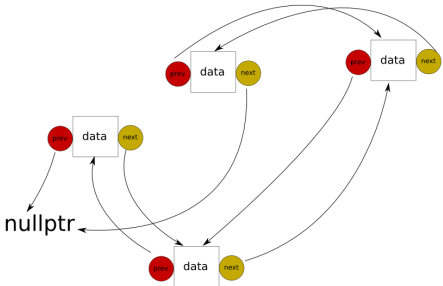
New elements can be added to the front or back of the list with only a few pointers needing rearrangement.



JÜLICH Forschungszentrum

A LINKED LIST

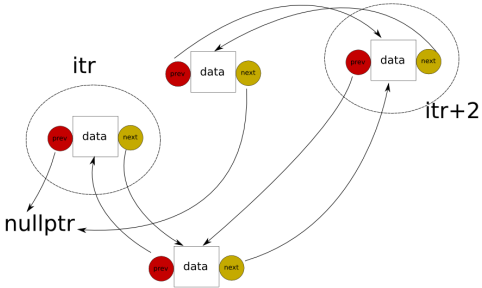
Any element in the list can be reached, if one kept track of the beginning or end of the list, and followed the "next" and "previous" pointers.



JÜLICH Forschungszentrum

A LINKED LIST

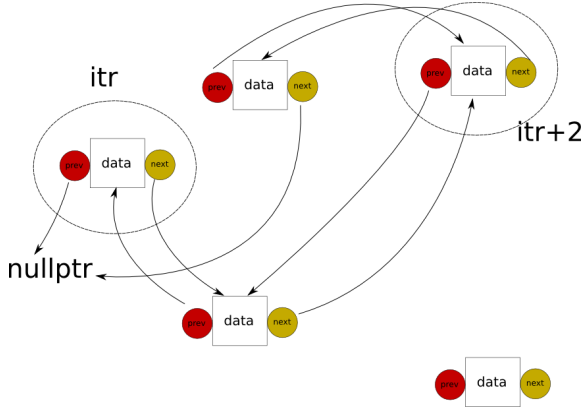
A concept of an "iterator" can be devised, where the `++` and `--` operators move to the next and previous nodes.



JÜLICH Forschungszentrum

A LINKED LIST

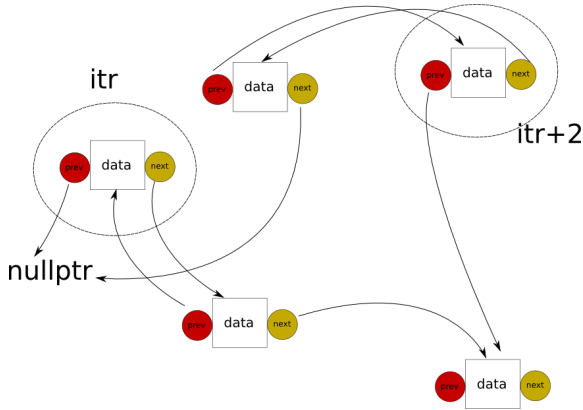
Inserting a new element in the middle of the list does not require moving the existing nodes in memory.



JÜLICH Forschungszentrum

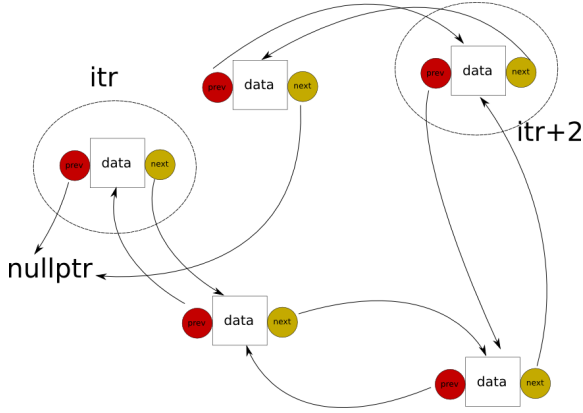
A LINKED LIST

Just rearranging the next and previous pointers of the elements between which the new element must go, is enough. This gives efficient $O(1)$ insertions and deletions.

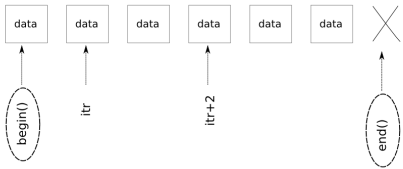


A LINKED LIST

Just rearranging the next and previous pointers of the elements between which the new element must go, is enough. This gives efficient $O(1)$ insertions and deletions.



GENERIC "CONTAINERS"



- Can be accessed through a suitably designed "iterator"
- The data type does not affect the design $\Longrightarrow$ template

- Similarity of interface is by design
- With a standard container `c` of type `C`, it's always possible to use `std::begin(c)` to access the start and `std::end(c)` to access the end
- `std::begin()` and `std::end()` return `C::iterator` or `C::const_iterator` depending on whether `c` is const qualified.
- `std::cbegin(c)` and `std::cend(c)` return `C::const_iterator` types irrespective of whether `c` is a const
- Similarly, `std::size(c)` always returns the size of the container, i.e., the number of elements it contains

STL CONTAINERS

- `std::vector<>` : dynamic arrays
- `std::list<>` : linked lists
- `std::queue<>` : queue
- `std::deque<>` : double ended queue
- `std::map<A,B>` : associative container

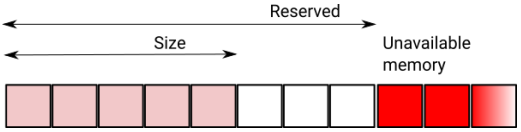
- Include file names correspond to class names
- All of them provide corresponding iterator classes
- If `iter` is an iterator, `*iter` is data.
- All of them provide member functions like `begin()`, `end()`, `size()`, initialiser list constructors, deduction rules for class template argument deduction

```
1       list L{1, 2, 3, 4, 5}; // std::list<int>, initialised to 1, 2, 3, 4, 5
2       auto pp = partition(begin(L), end(L), [](auto i){ return i % 3 == 0; });
3       decltype(L) M;
4       M.splice(end(M), L, begin(L), pp);
```

USING STD::VECTOR

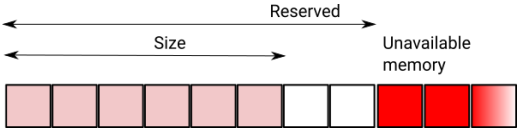
- `vector<int> v(10);` makes a dynamic array of 10 integers, `vector v(10, 0.)` creates a vector of 10 `doubles` initialised to 0, `vector v{1u, 2u, 3u}` creates a vector of `unsigned int` with values 1, 2 and 3.
- Efficient indexing operator `[]`, for unchecked element access
- `v.at(i)` provides range checked access. An exception is thrown if `at(i)` is called with an out-of-range `i`
- `std::vector<std::list<userinfo>> vu(10) ;` array of 10 linked lists.
- Supports `push_back` and insert operations, but sometimes has to relocate the all the elements because of one `push_back` operation (next slide)

STD::VECTOR



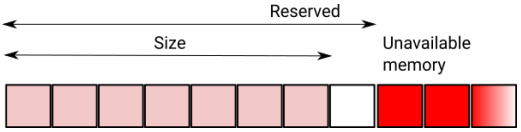
- `std::vector` may reserve a few extra memory blocks to allow a few quick `push_back` operations.
- New items are simply placed in the previously reserved but unused memory and the size member adjusted.

STD::VECTOR



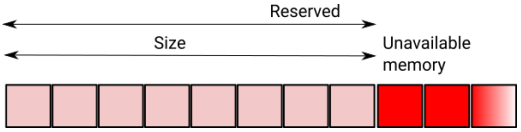
- `std::vector` may reserve a few extra memory blocks to allow a few quick `push_back` operations.
- New items are simply placed in the previously reserved but unused memory and the size member adjusted.

STD::VECTOR



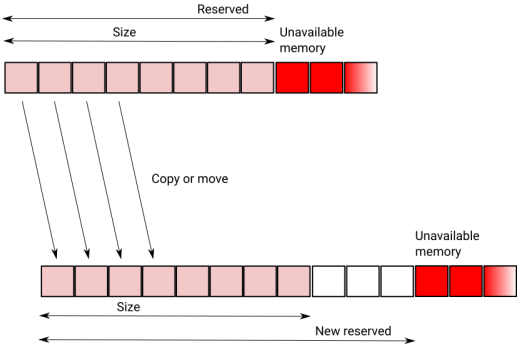
- `std::vector` may reserve a few extra memory blocks to allow a few quick `push_back` operations.
- New items are simply placed in the previously reserved but unused memory and the size member adjusted.

STD::VECTOR



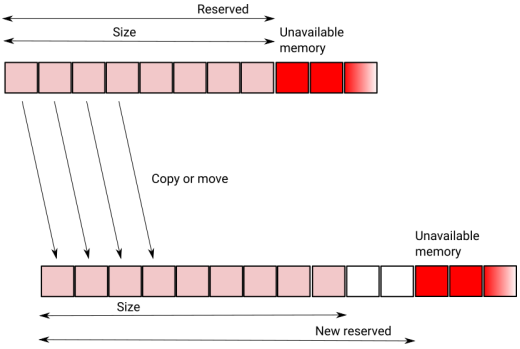
- `std::vector` may reserve a few extra memory blocks to allow a few quick `push_back` operations.
- New items are simply placed in the previously reserved but unused memory and the size member adjusted.

STD::VECTOR



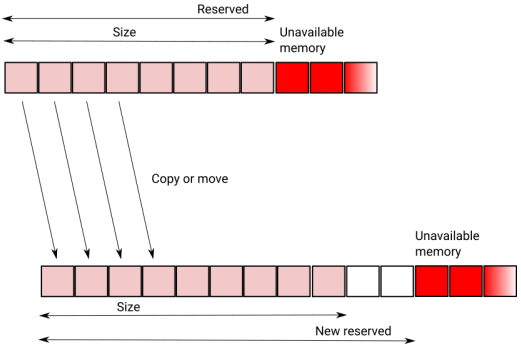
- When this is no longer possible, a new larger memory block is reserved, and all previous content is moved or copied to it.
- A few more quick `push_back` operations are again possible.

STD::VECTOR



- When this is no longer possible, a new larger memory block is reserved, and all previous content is moved or copied to it.
- A few more quick `push_back` operations are again possible.

STD::VECTOR



- When `push_back` is no longer possible, a new larger memory block is reserved, and all previous content is moved or copied to it.
- A few more quick `push_back` operations are again possible.

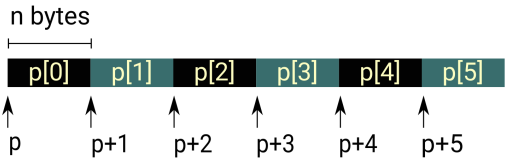
Exercise 6.1:

Construct a list and a vector of 3 elements of the `Vbose` class from your earlier exercise. Add new elements one by one and pause to examine the output.

STD::ARRAY : ARRAYS WITH FIXED COMPILE TIME CONSTANT SIZE

- `std::array<T,N>` is a fixed length array of size N holding elements of type `T`
- It implements functions like `begin()` and `end()` and is therefore usable with STL algorithms like `transform`, `generate` etc.
- The array size is a template parameter, and hence a compile time constant.
- `std::array<std::string,7> week{"Mon","Tue","Wed","Thu","Fri","Sat","Sun"};`

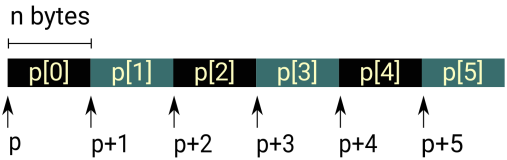
ARRAYS




```
1  double A[10]; // Built-in or C-style array
2  int sz;
3  std::cin >> sz;
4  int M[sz]; // Not allowed!
5  #include <array>
6  ...
7  std::array<double,10> A; // On stack
8  // Like a built-in array, but obeys
9  // C++ standard library conventions.
10 for (size_t i = 0; i < A.size(); ++i) {
11     P *= A[i];
12 }
13 std::vector<double> B(sz,3.0);
```

- Sequence of N objects stored consecutively in memory, with no gaps
- If `p` is a pointer to the first object of such a sequence, `p+1`, `p+2` etc, will point to the subsequent elements. Elements of the sequence can therefore be accessed as `*(p+0)`, `*(p+1)`, `*(p+2)` ... another notation for that is `p[0]`, `p[1]` ...

ARRAYS




```
1  double A[10]; // Built-in or C-style array
2  int sz;
3  std::cin >> sz;
4  int M[sz]; // Not allowed!
5  #include <array>
6  ...
7  std::array<double,10> A; // On stack
8  // Like a built-in array, but obeys
9  // C++ standard library conventions.
10 for (size_t i = 0; i < A.size(); ++i) {
11     P *= A[i];
12 }
13 std::vector<double> B(sz,3.0);
```

- Built-in or "C-style" arrays consist of blocks of memory large enough to hold a fixed number of elements. The array, thought of as a pointer, points to the first element in the sequence. The elements are stored consecutively, but the number of elements is never stored anywhere
- `std::array<type,size>` is a compile-time fixed length array obeying STL conventions. The size is available through a function, although it does not have to be stored with the array data!
- `std::array<type,size>` retains its "personality" (does not decay into a pointer) when used as input to function or when received as the output from a function. This should be your default choice when you need fixed length arrays.

ASSOCIATIVE CONTAINERS: STD::MAP

```
1  std::map<std::string, int> flsize;
2  flsize["S.dat"]=123164;
3  flsize["D.dat"]=423222;
4  flsize["A.dat"]=1024;
```

- Think of it as a special kind of "vector" where you can have things other than integers as indices.
- Template arguments specify the key and data types
- Could be thought of as a container storing (key,value) pairs :
  $\{("S.dat", 123164), ("D.dat", 423222), ("A.dat", 1024)\}$
- The less than comparison operation must be defined on the key type
- Implemented as a tree, which keeps its elements sorted

A WORD COUNTER PROGRAM

Exercise 6.2:

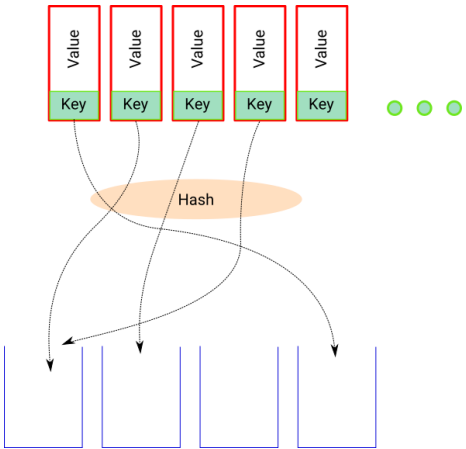
Fake exercise: Write a program that counts all different words in a text file and prints the statistics.

```
1  #include <fstream>
2  #include <print>
3  #include <map>
4  #include <string>
5  auto main(int argc, char* argv[]) -> int
6  {
7      std::ifstream fin(argv[1]);
8      std::map<std::string, unsigned> freq;
9      std::string s;
10     while (fin >> s) freq[s]++;
11     for (auto&& [word, count] : freq)
12         std::print("{:20}   :{:12}\n",
13             word, count);
14 }
```

A quick histogram!

- `std::map<string, unsigned>` is a container which stores an integer, for each unique `std::string` key.
- The iterator for `std::map` "points to" a `pair<key,value>`

STD::UNORDERED_MAP AND STD::UNORDERED_SET



Unordered map

- Like `std::map<k,v>` and `std::set<v>`, but do not sort the elements
- Internally, these are hash tables, providing faster element access than `std::map` and `std::set`
- Additional template arguments to specify hash functions

VALARRAY

```
1  #include <valarray>
2
3  void varray_ops()
4  {
5      std::valarray V1(0., 1000000UL);
6      std::valarray<double> V2;
7      v2.resize(1000000UL, 0.);
8      auto x = exp(-V1 * V1) * sin(V2);
9      if (x.sum() < 100.0) {
10     //
11     }
12 }
```

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

SMART POINTERS

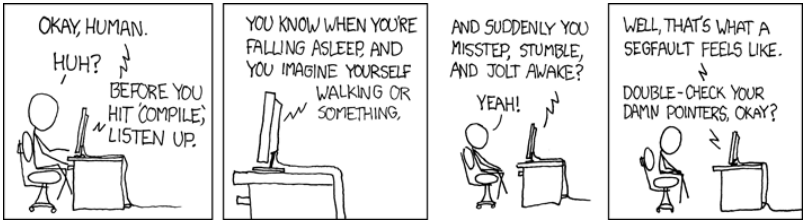


Figure: Source: XKCD (`http://xkcd.com`)

- 3 kinds of smart pointers were introduced in C++11
- Like pointers, but with added clean-up functionality
- Ordinary (raw) pointers are also allowed, but direct heap resource management through those is discouraged

UNIQUE POINTER

```cpp
// examples/uniqueptr.cc
auto main() -> int
{
    auto u1 = std::make_unique<MyStruct>(1);
    //auto u2 = u1; //won't compile
    auto u3 = std::move(u1);
    std::cout << "Data value for u3 is u3->v1 = " << u3->v1 <<'\n';
    auto u4 = std::make_unique<MyStruct[]>(4);
}
```

- Smart pointer: The data pointed to is freed when the pointer expires
- Exclusive access to resource
- Can not be copied (deleted copy constructor and assignment operator)
- Data ownership can be transferred with `std::move`
- Can create single instances as well as arrays through `make_unique`

SHARED POINTER

```cpp
// examples/sharedptr.cc
auto main() -> int
{
    auto u1 = std::make_shared<MyStruct>(1);
    std::shared_ptr<MyStruct> u2 = u1; // Copy is ok
    std::shared_ptr<MyStruct> u3 = std::move(u1);
    std::cout << "Reference count of u3 is "
              << u3.use_count() << '\n';
}
```

- Smart pointer: The data pointed to is freed when the pointer expires
- Can share resource with other shared/weak pointers
- Can be copy assigned/constructed
- Maintains a reference count `ptr.use_count()`

WEAK POINTER

```cpp
// examples/weakptr.cc
auto main() -> int
{
    auto s1 = std::make_shared<MyStruct>(1);
    std::weak_ptr<MyStruct> w1(s1);
    std::cout << "Ref count of s1 = " << s1.use_count() << '\n';
    std::shared_ptr<MyStruct> s3(s1);
    std::cout << "Ref count of s1 = " << s1.use_count() << '\n';
}
```

- Does not own resource
- Can "kind of" share data with shared pointers, but does not change reference count

Exercise 6.3: uniqueptr.cc, sharedptr.cc

Read the 3 smart pointer example files, and try to understand the output. Observe when the constructors and destructors for the data objects are being called.

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 6.4:

`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.

STL ALGORITHMS

```
1   ...
2   std::vector<YourClass> vc(inp.size());
3   std::copy(inp.begin(), inp.end(), vc.begin());
4   //Copy contents of list to a vector
5   auto pos = std::find(vc.begin(), vc.end(), elm);
6   //Find an element in vc which equals elm
7   std::sort(vc.begin(),vc.end());
8   //Sort the vector vc. The operator "<"
9   //must be defined
10  ...
11  std::transform(inp.begin(), inp.end(), out.begin(), rotate);
12  //apply rotate() to each input element,
13  //and store results in output sequence
```

The similarity of the interface, e.g. `begin()` , `end()` etc., among STL containers allows generic algorithms to be written as template functions, performing common tasks on collections

STL ALGORITHMS

- Typically, the algorithms in the namespace `std` accept one or more ranges as (start, stop) pairs, some other inputs which may include callable objects
- New algorithms were introduced in C++20 in the namespace `std::ranges`, where the input ranges are given as single objects rather than iterator pairs. Think

```
std::ranges::for_each(v, [](auto&& elem){ std::cout << elem << "\n"; })
```

rather than

```
std::for_each(v.begin(), v.end(), [](auto&& elem){ std::cout << elem << "\n"; })
```

Exercise 6.5:

- The standard library provides a large number of template functions to work with containers
- Look them up in `www.cplusplus.com` or `en.cppreference.com`
- Use the suitable STL algorithms to generate successive permutations of the vector

STL ALGORITHMS: SORTING

- `std::sort(iter_1, iter_2)` sorts the elements between iterators `iter_1` and `iter_2`

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
auto main() -> int
{
  vector v{2, -3, 7, 4, -1, 9, 0};
  sort(v.begin(), v.end());
  //Sort using "<" operator
  for (auto el : v) cout << el << "\n";
  sort(v.begin(), v.end(),
         [](int i, int j) {
      return i * i < j * j;
  });
  //Sort using custom comparison
  for (auto el: v) cout << el << "\n";
}
```

STL ALGORITHMS: SORTING

- `std::sort(iter_1, iter_2)` sorts the elements between iterators `iter_1` and `iter_2`
- `std::sort(iter_1, iter_2, lt)` sorts the elements between iterators `iter_1` and `iter_2` using a custom comparison method `lt`, which could be any callable object

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
auto main() -> int
{
  vector v{2, -3, 7, 4, -1, 9, 0};
  sort(v.begin(), v.end());
  //Sort using "<" operator
  for (auto el : v) cout << el << "\n";
  sort(v.begin(), v.end(),
         [](int i, int j) {
      return i * i < j * j;
  });
  //Sort using custom comparison
  for (auto el: v) cout << el << "\n";
}
```

STL ALGORITHMS: SORTING

- `std::sort(iter_1, iter_2)` sorts the elements between iterators `iter_1` and `iter_2`
- `std::sort(iter_1, iter_2, lt)` sorts the elements between iterators `iter_1` and `iter_2` using a custom comparison method `lt`, which could be any callable object
- `std::ranges::sort(range)` and `std::ranges::sort(range, lt)` are corresponding versions using a range as an argument instead of a pair of iterators

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
auto main() -> int
{
  vector v{2, -3, 7, 4, -1, 9, 0};
  sort(v.begin(), v.end());
  //Sort using "<" operator
  for (auto el : v) cout << el << "\n";
  ranges::sort(v, [](int i, int j) {
      return i * i < j * j;
  });
  //Sort using custom comparison
  for (auto el: v) cout << el << "\n";
}
```

STD::TRANSFORM

- `std::transform(begin_1 , end_1, begin_res, unary_function);`
- `std::transform(begin_1 , end_1, begin_2, begin_res, binary_function);`
- Apply callable object to the sequence and write result starting at a given iterator location
- The container holding result must be previously resized so that it has the right number of elements
- The "result" container can be (one of the) input container(s)

```
1  std::vector v{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9};
2  std::list L1(v.size(), 0), L2(v.size(), 0);
3  std::transform(v.begin(), v.end(), L1.begin(), sin);
4  std::transform(v.begin(), v.end(), L1.begin(), L2.begin(), std::max);
```

Result: `L1` contains `sin(x)` for each `x` in `v`, and `L2` contains the `greater(x,sin(x))`

STD::RANGES::TRANSFORM

- `std::ranges::transform(range1, begin_res, unary_function);`
- `std::transform(range1, range2, begin_res, binary_function);`
- Apply callable object to the sequence and write result starting at a given iterator location
- The container holding result must be previously resized so that it has the right number of elements
- The "result" container can be (one of the) input container(s)

```
1  std::vector v{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9};
2  std::list L1(v.size(), 0), L2(v.size(), 0);
3  std::ranges::transform(v, L1.begin(), sin);
4  std::ranges::transform(v, L1, L2.begin(), std::max);
```

Result: `L1` contains `sin(x)` for each `x` in `v`, and `L2` contains the `greater(x,sin(x))`

ALL_OF, ANY_OF, NONE_OF

```
1  auto valid(std::string name) -> bool
2  {
3    return all_of(name.begin(),name.end(),
4        [](char c) { return (isalpha(c)) || isspace(c); });
5  }
```

- `std::all_of(begin_ , end_ , condition)` checks if all elements in a given range satisfy `condition`
- `condition` is a callable object
- `std::any_of(begin_ , end_ , condition)` checks if any single element in a given range satisfies `condition`
- `std::none_of(begin_ , end_ , condition)` returns true if not a single element in a given range satisfies `condition`

ALL_OF, ANY_OF, NONE_OF

```
1  auto valid(std::string name) -> bool
2  {
3    return all_of(name,
4        [](char c) { return (isalpha(c)) || isspace(c); });
5  }
```

- `std::ranges::all_of(range , condition)` checks if all elements in a given range satisfy `condition`
- `condition` is a callable object
- `std::ranges::any_of(range , condition)` checks if any single element in a given range satisfies `condition`
- `std::ranges::none_of(range , condition)` returns true if not a single element in a given range satisfies `condition`

ALGORITHMS

```
1  vector v{ 1, 2, 3, 4, 5, 6, 7, 8, 9 }, w{ 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };
2  vector<int>  x, y, z, m;
3  if (is_sorted(begin(v), end(v)))
4      cout << "The sequence is sorted in the increasing order.\n";
5  reverse(v.begin(), v.end());
6  rotate(v.begin(), v.begin() + 3, v.end());
7  sort(begin(v), end(v));
8  merge(v.begin(), v.end(), w.begin(), w.end(), back_inserter(m));
9  set_union(v.begin(), v.end(), w.begin(), w.end(), back_inserter(x));
10 set_intersection(w.begin(), w.end(), v.begin(), v.end(), back_inserter(y));
11 set_symmetric_difference(v.begin(), v.end(), w.begin(), w.end(), back_inserter(z));
12 if (is_permutation(z.begin(), z.end(), v.begin(), v.end())) // do something
```

Exercise 6.6:

A whole lot of operations available for sequence types. The file `seqops.cc` contains the operations shown here.

ALGORITHMS

- `for_each(start, end, operation)` : As it sounds
- `find(start, end, what)` : returns the location of the looked for value, "end" if not found
- `find_if(start, end, condition)` , find the first element satisfying a condition
- `copy(start1, end1, start2)` : As it sounds
- `copy_if(start1, end1, start2, criterion)` : `criterion` is a unary function taking a value of the type found in the sequence and returning true or false
- `transform(start1, end1, start2, operation)` : applies `operation` on every element in the input sequence and writes the results starting at `start2`

CONSTRAINED ALGORITHMS (RANGES)

- `for_each(range, operation)` : As it sounds
- `find(range, what)` : returns the location of the looked for value, "end" if not found
- `find_if(range, condition)` , find the first element satisfying a condition
- `copy(range1, iterator2)` : As it sounds
- `copy_if(range1, iterator2, criterion)` : `criterion` is a unary function taking a value of the type found in the sequence and returning true or false
- `transform(range1, iterator2, operation)` : applies `operation` on every element in the input sequence and writes the results starting at `iterator2`

ALGORITHMS

```
1  vector v{ 1, 2, 3, 4, 5, 6, 7, 8, 9 }, w{ 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };
2  vector<int>  x, y, z, m;
3  if (is_sorted(v))
4      cout << "The sequence is sorted in the increasing order.\n";
5  reverse(v);
6  rotate(v, v.begin() + 3);
7  sort(v);
8  merge(v, w, back_inserter(m));
9  set_union(v, w, back_inserter(x));
10 set_intersection(w, v, back_inserter(y));
11 set_symmetric_difference(v, w, back_inserter(z));
12 if (is_permutation(zv)) // do something
```

Exercise 6.7:

The file `seqops_range.cc` contains the operations shown here. Explore by making modifications.

NUMERIC ALGORITHMS

```
1  #include <numeric>
2
3  using std::reduce;
4  using std::transform_reduce;
5
6  auto res = reduce(v.begin(), v.end());
7  auto res = reduce(v.begin(), v.end(), init);
8  auto res = reduce(v.begin(), v.end(),
9      init, std::plus<double>{});
10 auto res = transform_reduce(
11     u.begin(), u.end(),
12     v.begin(), init);
13 auto res = transform_reduce(
14     u.begin(), u.end(),
15     v.begin(), init, reduce_op, transf_op);
16 auto res = transform_reduce(
17     std::execution::par,
18     u.begin(), u.end(),
19     v.begin(), init, reduce_op, transf_op);
```

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

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);
sr::sort(v);
```

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

USING RANGES FROM STD OR FROM RANGE-V3

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

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

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

FUN WITH RANGES AND VIEWS

```
1  // examples/ranges0.cc
2  #include <ranges>
3  #include <span>
4  auto sum(std::ranges::input_range auto&& seq) {
5      std::iter_value_t<decltype(seq)> ans{};
6      for (auto x : seq) ans += x;
7      return ans;
8  }
9  auto main() -> int
10 {
11     //using various namespaces;
12     cout << "vector   : " << sum(vector(  { 9,8,7,2 } )) << "\n";
13     cout << "list     : " << sum(list(    { 9,8,7,2 } )) << "\n";
14     cout << "valarray : " << sum(valarray({ 9,8,7,2 } )) << "\n";
15     cout << "array    : "
16         << sum(array<int,4>({ 9,8,7,2 } )) << "\n";
17     cout << "array    : "
18         << sum(array<string, 4>({ "9"s,"8"s,"7"s,"2"s } )) << "\n";
19     int A[]{1,2,3};
20     cout << "span(built-in array) : " << sum(span(A)) << "\n";
21 }
```

FUN WITH RANGES AND VIEWS

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

FUN WITH RANGES AND VIEWS

```cpp
// examples/iota.cc
#include <ranges>
#include <iostream>
auto main() -> int {
    namespace sv = std::ranges::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` and `std::span` are perfectly fine ranges. They have iterators with the right properties, `begin()` and `end()` functions.
- We could take this further by creating `views` which need not actually contain any objects of the sequence, but simply fake it when we dereference the iterators.
- Example: the standard view `std::views::iota(integer)` gives us an infinite sequence of integers starting at a given value.

FUN WITH RANGES AND VIEWS

```cpp
// examples/iota.cc
#include <ranges>
#include <iostream>
auto main() -> int {
    namespace sv = std::ranges::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;
        }
    }
}
```

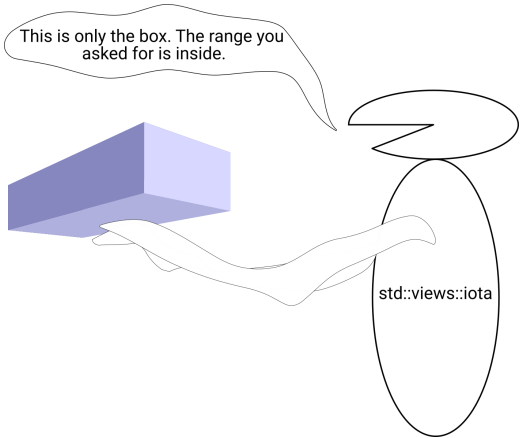
- All containers are ranges, but not all ranges are containers
- `std::string_view` and `std::span` are perfectly fine ranges. They have iterators with the right properties, `begin()` and `end()` functions.
- We could take this further by creating `views` which need not actually contain any objects of 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/iota.cc
2  #include <ranges>
3  #include <iostream>
4  auto main() -> int {
5      namespace sv = std::ranges::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` and `std::span` are perfectly fine ranges. They have iterators with the right properties, `begin()` and `end()` functions.
- 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` and `std::span` are perfectly fine ranges. They have iterators with the right properties, `begin()` and `end()` functions.
- 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` and `std::span` are perfectly fine ranges. They have iterators with the right properties, `begin()` and `end()` functions.
- We could take this further by creating `views` which need not actually contain any objects of the sequence, but simply fake it when we dereference the iterators.
- Example: the standard view `std::views::iota(integer)` gives us an infinite sequence of integers starting at a given value.

BORROWED RANGES

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

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

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

BORROWED RANGES

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

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

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

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

BORROWED RANGES

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

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

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

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

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

BORROWED RANGES

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

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

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

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

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

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

BORROWED RANGES

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

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

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

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

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

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

BORROWED RANGES

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

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

BORROWED RANGES

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

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

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

VIEW ADAPTORS

```
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 $[0, 2\pi)$

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

VIEW ADAPTORS

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

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

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

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

VIEW ADAPTORS

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

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

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

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

VIEW ADAPTORS

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

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

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

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

VIEW ADAPTORS

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

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

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

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

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

VIEW ADAPTORS

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

- Command line of Linux, Mac OS ...

VIEW ADAPTORS

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

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

VIEW ADAPTORS

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

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

VIEW ADAPTORS

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

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

VIEW ADAPTORS

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

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

VIEW ADAPTORS

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

VIEW ADAPTORS

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

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

VIEW ADAPTORS

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

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

VIEW ADAPTORS

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

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

VIEW ADAPTORS

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

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

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 6.8:

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

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

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

Exercise 6.9:

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

```
1  // examples/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::cout << (in | sv::filter(gerund)) << "\n";
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::cout` is shown writing out a range. This works via a separate header file included in `gerund.cc` called `range_output.hh`, which is provided to you with the course material. Ranges in C++20 are not automatically streamable to the standard output.

VIEW ADAPTORS

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

```
1  // examples/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 }
```

Exercise 6.10:

Rewrite the first day exercise about printing the command line arguments using ranges, views and span.

STL utilities

CHRONO: THE TIME LIBRARY

- `namespace std::chrono` defines many time related functions and classes (include file: `chrono`)
- `system_clock`: System clock
- `steady_clock`: Steady monotonic clock
- `high_resolution_clock`: To the precision of your computer's clock
- `steady_clock::now()`: nanoseconds since 1.1.1970
- `duration<double>`: Abstraction for a time duration. Uses `std::ratio<>` internally

Exercise 6.11: chrono_demo.cc

THE TIME LIBRARY

```cpp
// examples/chrono_demo.cc
#include <iostream>
#include <chrono>
#include <vector>
#include <algorithm>
#include <ranges>
bool is_prime(unsigned n);
auto main() -> int
{
    using namespace std::chrono;
    namespace sr = std::ranges;
    namespace sv = std::views;
    std::vector<unsigned> primes;
    auto t = steady_clock::now();
    sr::copy(sv::iota(0UL, 10000UL) | sv::filter(is_prime), back_inserter(primes));
    std::cout << "Primes till 10000 are ... " << '\n';
    for (unsigned i : primes) std::cout << i << '\n';
    auto d = steady_clock::now() - t;
    std::cout<<"Prime search took " << duration<double>(d).count() << " seconds\n";
}
```

CALENDAR AND DATES WITH `STD::CHRONO`

```
1  auto current_year() -> std::chrono::year
2  {
3      using namespace std::chrono;
4      year_month_day date { floor<days>(system_clock::now()) };
5      return date.year();
6  }
7  auto main(int argc, char* argv[]) -> int
8  {
9      using namespace std::chrono;
10     using namespace std::chrono_literals;
11     auto Y0 { current_year() };
12     auto Y1 = Y0 + years{100};
13     if (argc > 1) Y1 = year{std::stoi(argv[1])};
14     if (argc > 2) Y0 = year{std::stoi(argv[2])};
15     if (Y1 < Y0) std::swap(Y1, Y0);
16
17     for (auto y = Y0; y < Y1; ++y) {
18         auto d = y / February / Sunday[5];
19         if (d.ok())
20             std::cout << static_cast<int>(y) << "\n";
21     }
22 }
```

CALENDAR...

Exercise 6.12:

The programs `examples/feb.cc` and `examples/advent.cc` demonstrate the use of the calendar facilities of the C++ standard library. Familiarise yourself with them.

RANDOM NUMBER GENERATION

- Convenient, flexible, powerful random number library providing high quality (pseudo-)random numbers in standard C++ without any external libraries.
- Include `random`. Namespace `std::random`

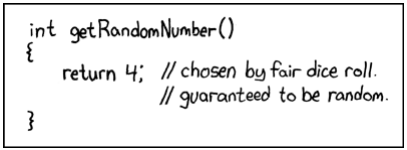


Figure: Source XKCD: `http://xkcd.com`

RANDOM NUMBER GENERATION

- Share a common structure
- Uniform random generator engine with (hopefully) well tested properties
- Distribution generator which adapts its input to a required distribution

$$p(n) = \frac{m^n e^{-m}}{n!}$$

Random distribution

Randomness engine

glue with lambda

```
1  auto gen = [
2      engine = std::mt19937_64{},
3      dist=std::poisson_distribution<>{8.5}
4             ]() mutable {
5      return dist(engine);
6  };
7  r = gen();
```

```
1  std::mt19937_64 engine;
2  std::poisson_distribution<> dist{8.5};
3  auto gen = [&dist, &engine] {
4      return dist(engine);
5  };
6  r = gen();
7  // if engine or dist are required elsewhere
```

RANDOM NUMBER GENERATORS

```cpp
#include <random>
#include <iostream>
#include <map>
auto main() -> int
{
    auto gen = [ dist=std::poisson_distribution<> {8.5}, engine=std::mt19937_64{} ]
        () mutable { return dist(engine); };
    std::map<int,unsigned> H;
    for (auto i = 0UL; i < 5000000UL; ++i) H[gen()]++;
    for (auto [i, fi] : H) std::cout << i << " " << fi << '\n';
}
```

- `std::mt19937_64` is a 64 bit implementation of Mersenne Twister 19937
- The template `std::poisson_distribution` is a functional implementing the Poission distribution

RANDOM NUMBER GENERATORS

```
1  std::normal_distribution<> G{3.5, 1.2}; // Gaussian mu = 3.5, sig = 1.2
2  std::uniform_real_distribution<> U{3.141, 6.282};
3  std::binomial_distribution<> B{13};
4  std::discrete_distribution<> dist{0.3, 0.2, 0.2, 0.1, 0.1, 0.1};
5  // The following is an engine like std::mt19937, but is non-deterministic
6  std::random_device seed; // int i = seed() will be a random integer
```

- Lots of useful distributions available in the standard
- With one or two lines of code, it is possible to create a high quality generator with good properties and the desired distribution
- `std::random_device` is a non-deterministic random number generator.
  - It is good for setting seeds for the used random number engine
  - It is slower than the pseudo-random number generators

RANDOM NUMBER GENERATOR: EXERCISES

Exercise 6.13:

Make a program to generate normally distributed random numbers with user specified mean and variance, and make a histogram to demonstrate that the correct distribution is produced. Start from `examples/normal_distribution.cc`.

Exercise 6.14:

Make a program to implement a "biased die", i.e., with user specified non-uniform probability for different faces. You will need `std::discrete_distribution<>` Start from `examples/weighted_die.cc`.

EXERCISES

Exercise 6.15:

For a real valued random variable X with normal distribution of a given mean μ and standard deviation σ, calculate the following quantity:

$$K[X] \quad = \quad \frac{\left\langle (X-\mu)^4 \right\rangle}{(\langle (X-\mu)^2 \rangle)^2}$$

Fill in the random number generation parts of the program `examples/K.cc` . Run the program a few times varying the mean and standard deviation. What do you observe about the quantity in the equation above ?

Exercise 6.16: Probabilities with playing cards

The program `examples/cards_problem.cc` demonstrates many topics discussed during this course. It has a `constexpr` function to create a fixed length array to store results, several standard library containers and algorithms as well as the use of the random number machinery for a Monte Carlo simulation. It has extensive comments explaining the use of various features. Read the code and identify the different techniques used, and run it to solve a probability question regarding playing cards.

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

```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
- 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 << fmt::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 << fmt::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

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

- GCC 13 has an implementation. Our redirecting header can help us work with clang as well.
- We can use a redirecting header to use the `fmt` library when the compiler does not have the library feature
- Code simplification and compilation (and runtime) speed $\Longrightarrow$ useful to learn it. Eventually all compilers will have it.

FORMATTED OUTPUT

- `std::format("format string {}, {} etc.", args...)` takes a compile time constant format string and a parameter pack to produce a formatted output string
- `std::vformat` can be used if the format string is not known at compilation time
- If instead of receiving output as a newly created string, output into a container or string is desired, `std::format_to` or `std::format_to_n` are available
- The 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!

Exercise 6.17:

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

Exercise 6.18:

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

std::optional and std::variant

STD::OPTIONAL

- `std::optional<T>` manages an optional value of type `T`, which may or may not be present
- Another way to handle errors during computations to determine a value of some kind
- If the `optional` object has a value, the value resides in the object, i.e., the `optional` type does not do any dynamic memory allocation of its own
- The operators `*` and `->` are given for convenience, so that we can pretend we are dealing with a pointer type when using an `optional`
- If converted to a `bool`, we get **true** if there is a value, **false** otherwise
- Default initialisation as well as initialisation with `nullopt_t` create `optional` objects without value.

```
1  auto solve_quadratic(double a, double b,
2                       double c)
3  {
4      using namespace std;
5      optional<pair<double, double>> solution;
6      auto D = b * b - 4 * a * c;
7      if (D >= 0) {
8          auto q = -0.5 * ( b +
9                            copysign(sqrt(D), b) );
10         solution = make_pair(q / a, c / q );
11     }
12     return solution;
13 }
```

Exercise 6.19:

`examples/opt_qsolve.cc` is a small program demonstrating the use of `std::optional`.

STD::VARIANT : A TYPE SAFE UNION

- A `union` is a special kind of class where all the members occupy the same bytes in memory

```
union sameplace { size_t ulong; double real; };
static_assert(sizeof(sameplace) ==
              sizeof(double));
sameplace s;
s.ulong = 0UL;
s.real = 1.0;
cout << s.ulong << "\n";
```

- We can access the elements of a `union` the same way as a `struct` (above).
- Since both members occupy the same bytes, changes to one can affect the other
- If the union contains, e.g., `std::string`, such overriding of bytes would be dangerous.

- `std::variant` is a type safe `union`.
- Unlike the `union`, we don't get to name the different members. The different "members" can be accessed through functions like `std::get<int>(V)`, i.e., we can use the types to select the stored type. We also don't need to say what we are assigning to, since that can be deduced from the type of the object on the right of the `=`
- A `variant` knows what type is currently stored, and calls the destructors etc. when we assign something that would change the stored type

```
variant<double, int, long, string> V;
V = "let's assign a string";
V = 3.141;
// call string destructor and store a double
```

STD::VARIANT : A TYPE SAFE UNION

- A variant type stores one value of any one of a few pre-specified alternatives. To create a `variant` with an integer, a long, a string and a boolean, we would write

```
std::variant<int, long, string, bool> V;
```

- A variant can be assigned a value of any one of its contained types. The variant then remembers the value and the type of the value.

```
V = "0118 999 881 99 9119 725  3"s;
assert(std::holds_alternative<string>(V));
```

- The member function `index()` tells us the zero based index of the currently held type in the list of alternatives for the variant

```
assert(V.index() == 2);
```

- Since the type of the contained object can be changed by an assignment at run time, the variant can not simply have a function `get()` to return the contained value. We have to specify the type of value we want to read as a template argument:

```
cout << get<string>(V);
```

- Unlike the union, we can't store one type and read another

```
V = "0118 999 881 99 9119 725  3"s;
auto num = get<int>(V); //throws exception!
```

- There is also a non throwing version of the accessor:

```
if (auto iptr = get_if<int>(&V); iptr) {
// use iptr as pointer to int value
// Does not get here because get_if<int>
// returns a nullptr in this case.
}
```

STD::VARIANT : A TYPE SAFE UNION

```
1  using member_t = variant<int, long, string, bool>;
2  vector<member_t> pop{true, 91, "Monday"s};
3  for (auto & el : pop) {
4      if (auto iptr = get_if<int>(&el)) {
5      // *iptr is the int value in the variant el
6      } else if (auto lptr = get_if<long>(&el)) {
7      // *lptr is the long value in el
8      } else if (auto sptr = get_if<string>(&el)) {
9      // *sptr is the string value in el
10     }
11 }
```

- Variants can be made to model members of heterogeneous collections, much like pointers to base class in a class hierarchy. The difference is, we can even use built in type like `int`, `double` etc. in a variant based heterogeneous container, because it does not need a class hierarchy!

- Easiest way to model polymorphic behaviour is using a chain of `if ... else if ... else` statements using the `get_if<T>(&v)` function for the different types `T` in the `variant`. `get_if<T>(&v)` returns a valid `T *` if the variant `v` currently holds type `T`. Otherwise it returns `nullptr`.

Exercise 6.20:

The two example programs `examples/variant_0.cc` and `examples/variant_1.cc` demonstrate basic variant usage, such as assignment of values of different types, performing actions based on the content type.

STD::VARIANT : USING STD::VISIT TO SELECT ACTIONS

- Another way to perform different actions based on the currently held type is to use `std::visit`.
- If we have `variant<int, double> V`, `std::visit(F, V)` calls `F(int)` if `V` currently holds an `int` and `F(double)` if `V` currently holds a `double`. `std::visit` unpacks the variant before calling `F` with the stored value. The callable object `F` must have an overload capable of handling the alternatives in the variant
- The overloaded function to be used with `std::visit` can be created in many ways. Three examples in the following boxes:

```
1  struct my_action {
2      auto operator()(int i) { // ... }
3      auto operator()(double x) { //  }
4  };
5  // ...
6  std::visit(my_action{}, V);
```

STD::VARIANT : USING STD::VISIT TO SELECT ACTIONS

- Another way to perform different actions based on the currently held type is to use `std::visit`.
- If we have `variant<int, double> V`, `std::visit(F, V)` calls `F(int)` if `V` currently holds an `int` and `F(double)` if `V` currently holds a `double`. `std::visit` unpacks the variant before calling `F` with the stored value. The callable object `F` must have an overload capable of handling the alternatives in the variant
- The overloaded function to be used with `std::visit` can be created in many ways. Three examples in the following boxes:

```
1  std::visit([](auto upkd) {
2      if constexpr (is_same_v<int, decltype(upkd)>) {
3          // handle int input
4      } else if constexpr (is_same_v<double, decltype(upkd)>) {
5          // handle double input
6      }
7    }, V
8  );
```

STD::VARIANT : USING STD::VISIT TO SELECT ACTIONS

- Another way to perform different actions based on the currently held type is to use `std::visit`.
- If we have `variant<int, double> V`, `std::visit(F, V)` calls `F(int)` if `V` currently holds an `int` and `F(double)` if `V` currently holds a `double`. `std::visit` unpacks the variant before calling `F` with the stored value. The callable object `F` must have an overload capable of handling the alternatives in the variant
- The overloaded function to be used with `std::visit` can be created in many ways. Three examples in the following boxes:

```
1  template <class ... Ts> struct stapler : Ts ... { using Ts::operator()... ; };
2  template <class ... Ts> stapler(Ts ...) -> stapler<Ts...>;
3  std::visit(stapler{
4      [](int i) { /* handle int input */ },
5      [](double d) { /* handle double */ }
6      }, V
7  );
```

USING VARIANTS WITH STD::VISITOR

Exercise 6.21:

Example programs `examples/variant_2.cc`, `examples/variant_3.cc` and `examples/variant_4.cc` demonstrate the use of `std::visit` to dispatch different actions depending on the type of the currently held value in a variant. They parallel the approaches in the 3 boxes in the previous slide.

STD::ANY : A TYPESAFE CONTAINER FOR SINGLE VALUES

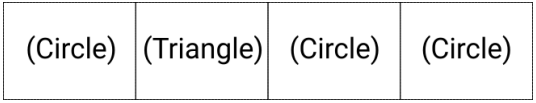
- A variable of type `std::any` can store 1 value of any type
- Simply by assigning a new value, the contained object is replaced with another of the new type. The variable of type `std::any` is like a box, whose type remains unchanged as the content is swapped. The contained object is indirectly accessed, leading to some overhead.

Exercise 6.22:

`examples/any_demo.cc` demonstrates basic usage of `std::any`.

```
1  any var = 1;
2  cout << "Reading int after storing int ... "
3       << any_cast<int>(var) << "\n"; // That works
4  try {
5      cout << "Reading float after storing an int ... "
6           << any_cast<float>(var) << "\n";
7           // This doesn't
8  } catch (const exception & err) {
9      cout << "Float cast after storing int failed. "
10          << "Error : " << err.what() << "\n";
11 }
12 var = "Europa"s;
13 map<string, any> config;
14 config["max_frequency_ghz"] = 3.3;
15 config["memory_MB"] = 16384;
16 config["fingerprint_reader"] = true;
```

SEQUENCES OF POLYMORPHIC OBJECTS



Exercise 6.23:

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

REGULAR EXPRESSIONS USING C++20

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

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

REGULAR EXPRESSIONS USING CTRE

Exercise 6.24:

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

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

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

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