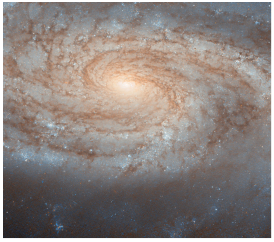
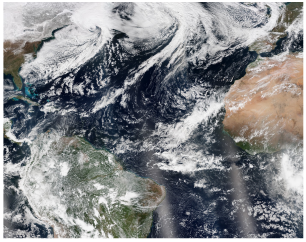
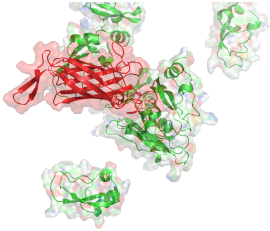
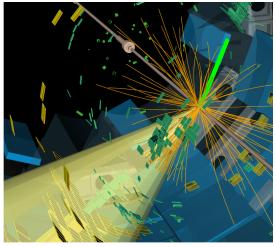





High performance scientific computing in C++
HPC C++ Course 2025

27 October – 30 October 2025 | Sandipan Mohanty | Forschungszentrum Jülich, Germany

Member of the Helmholtz Association

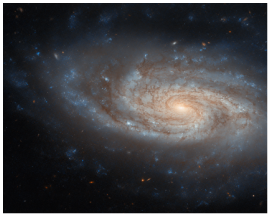
JÜLICH Forschungszentrum

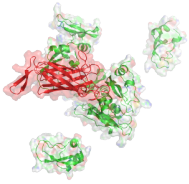
Chapter 1

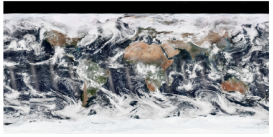
Introduction

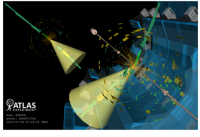
HPC and C++ in scientific computing

- 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









JÜLICH Forschungszentrum

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 of themes

- A "quick" recap of C++20 and C++23
- Revisiting language fundamentals for high performance code
- Expression templates
- Explicit SIMD programming
- Multi-threaded programs using standard parallel algorithms and Intel (R) Threading Building Blocks
- Lessons from writing a matrix multiplication program
- Linear algebra with EIGEN
- GPU programming with NVidia CUDA and Thrust
- Introduction to single source heterogeneous computing using SYCL and OneAPI

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

C++20

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

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

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

C++23

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

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

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

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

std::osyncstream

```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

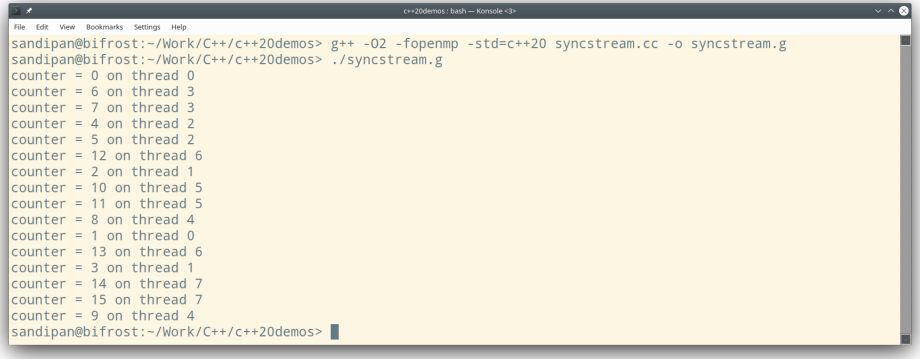
First: a couple of small, but interesting changes...

std::osyncstream

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

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

std::osyncstream



Exercise 1.1:

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

```
$ cd $cxx2025/work
$ cp -r ../orig/day1/examples ./d1examples
$ cd d1examples
$ G syncstream.cc -fopenmp -o syncstream.g
$ ./syncstream.g
```

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

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

Immediate functions

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

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

Designated initialisers

```
1  // examples/desig2.cc
2  struct v3 { double x, y, z; };
3  struct pars { int offset; v3 velocity; };
4  auto operator<<(std::ostream & os, const v3 & v) -> std::ostream&
5  {
6      return os << v.x << ", " << v.y << ", " << v.z << " ";
7  }
8  void example_func(pars p)
9  {
10     std::cout << p.offset << " with velocity " << p.velocity << "\n";
11 }
12 auto main() -> int
13 {
14     example_func({.offset = 5, .velocity = {.x=1., .y = 2., .z=3.}});
15 }
```

- Simple struct type objects can be initialised by designated initialisers for each field.
- Can be used to implement a kind of "keyword arguments" for functions. But remember, at least upto C++23, 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; }`
- You can now use explicit template parameters in lambda functions

```
[]<class T>(T x, const std::vector<T>& v) {
   for (auto el : v) x += el;
   return x;
};
```

- `std::string` can now `reserve()` memory
- `S.starts_with("pre")` checks if a string `S` starts with the prefix "pre". Similarly for `ends_with()`
- `M.contains("key")` answers whether a certain key "key" is present in an associative container. Cleaner than `try { M.at("key"); } catch (auto& err) { }`, or

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

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.

Couple of small, but interesting changes...

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

Couple of small, but interesting changes...

Bit manipulation:

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

Exercise 1.2:

Some example programs about the minor new features of C++20 and C++23 are `desig.cc`, `desig2.cc`, `cxpr_algo0.cc`, `immediate.cc`, `intcmp.cc`, and `bit0.cc`, in the `examples/` directory. Check them, change them in small ways, ask related questions!
Alternatively, double click on the Jupyter Notebook in the examples directory called `MiscSmallFeatures.ipynb`, and work through the notebook.

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 << format(
3          "i = {:>4d}, E_1 = {:< 12.8f}, "
4          "E_2 = {:< 12.8f}\n",
5           i, cos(i * wn), sin(i * wn));
6  }
```

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

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

Formatted output

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

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

Perfectly aligned, as all numeric output should be.

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

Formatted output

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

std::print

- Introduced in C++23
- Formats using the `std::format` syntax, but then directs the output to stdout, as if you had written `std::cout << std::format(...);`
- Formatting capabilities were extended to containers (ranges in general), date/time utilities

```
1  std::print("Hello world!\n");
2  std::print("answer = {: >12.8f}\n", d);
3  std::print("{}\n", v);
4  std::print("{}\n", t1 - t0);
```

Exercise 1.3:

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

Optional values

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

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

Optional values

```
#include <optional>
auto f(double x) -> std::optional<double> {
    std::optional<double> ans;
    const auto eps2 = 1.0e-24;
    if (x >= 0) {
        auto r0 = 0.5 * (1. + x);
        auto r1 = x / r0;
        while ((r0 - r1) * (r0 - r1) > eps2) {
            r0 = 0.5 * (r0 + r1);
            r1 = x / r0;
        }
        ans = r1;
    }
    return ans;
}
// Elsewhere...
std::cout << "Enter number : ";
std::cin >> x;
if (auto r = f(x); r) {
    std::cout << "The result is "
          << *r << '\n';
}
```

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

C++23 std::expected

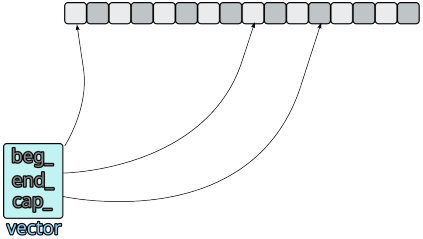
```cpp
#include <expected>
auto mysqrt(double x) -> std::expected<double, std::string> {
    const auto eps = 1.0e-12;
    const auto eps2 = eps * eps;
    if (x >= 0.) {
        auto r0 = 0.5 * (1. + x);
        auto r1 = x / r0;
        while ((r0 - r1) * (r0 - r1) > eps2) {
            r0 = 0.5 *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

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

std::span (C++20)

std::vector

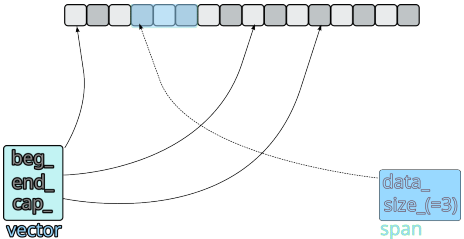
- `operator[]`
- `size()` and `ssize()`
- `begin()`
- `end()`



std::span (C++20)

std::vector

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



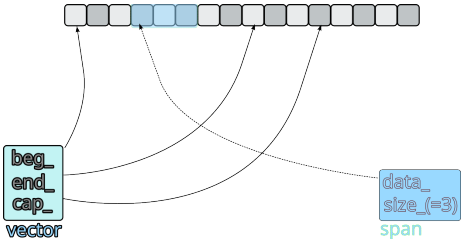
std::span

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

std::span (C++20)

std::vector

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



std::span

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

Contiguous containers

- RAII

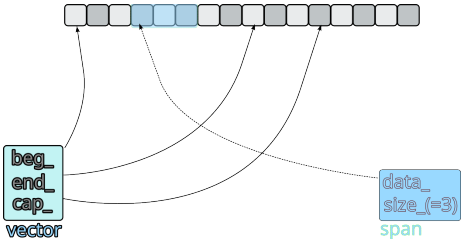
span: address and size of pre-existing data

- No resource ownership or management

std::span (C++20)

std::vector

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



std::span

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

Contiguous containers

- As long as container exists, elements can be accessed

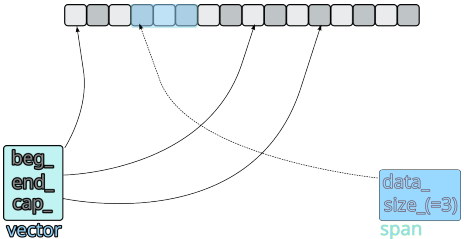
span: address and size of pre-existing data

- Even if span exists, accessibility of data is not guaranteed

std::span (C++20)

std::vector

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



std::span

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

Contiguous containers

- When container has expired, references / pointers / iterators to elements are invalidated

span: address and size of pre-existing data

- When span has expired, references / pointers / iterators to elements may remain valid

span

```
using std::transform_reduce;
using std::plus;
using std::multiplies;
using std::vector;


auto compute(const vector<double>& u,
    const vector<double>& v) -> double
{
    return transform_reduce(
        u.begin(), u.end(),
        v.begin(), 0., plus<double>{},
        multiplies<double>{});
}
void elsewhere()
{
    vector<double> A(100UL, 0.34);
    vector<double> B(100UL, 0.87);
    std::cout << compute(A, B) << "\n";
}
```

- We can avoid needlessly restrictive interfaces

span

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

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

span

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

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

span

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

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

span

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

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

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

span

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

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

Exercise 1.4:

`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.

The 4 big changes

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

Concepts

Constrained templates

- Overloaded functions: different strategies for different input types

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

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

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

Constrained templates

- Overloaded functions: different strategies for different input types

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

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

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

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

Constrained templates

- Overloaded functions: different strategies for different input types

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

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

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

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

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

Constrained templates

- Overloaded functions: different strategies for different input types

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

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

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

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

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

- We can.

Constrained templates

- Overloaded functions: different strategies for different input types

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

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

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

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

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

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

Concepts

Named requirements on template parameters

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

Concepts

Named requirements on template parameters

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

Concepts

Named requirements on template parameters

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

Concepts

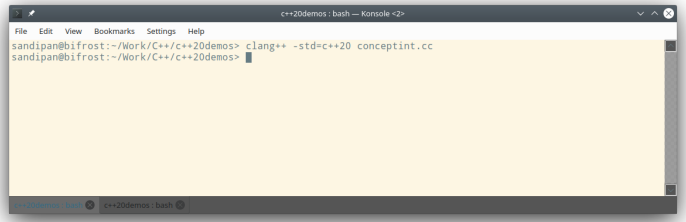
Named requirements on template parameters

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

Concepts

Named requirements on template parameters

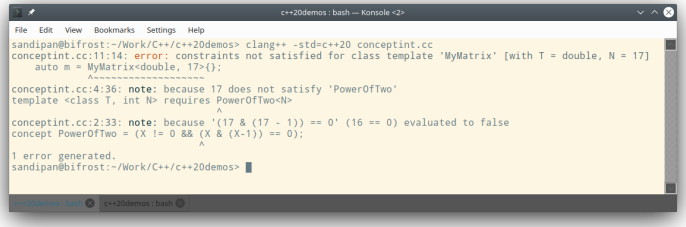
```
template <unsigned X> concept PowerOfTwo = std::has_single_bit(X);
template <class T, unsigned N> requires PowerOfTwo<N>
struct MyMatrix {
    // code which assumes that the square matrix size is a power of two
};
auto main() -> int
{
    auto m = MyMatrix<double, 16U>{};
}
```



Concepts

Named requirements on template parameters

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



JÜLICH Forschungszentrum

Concepts

Named requirements on template parameters

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

Concepts

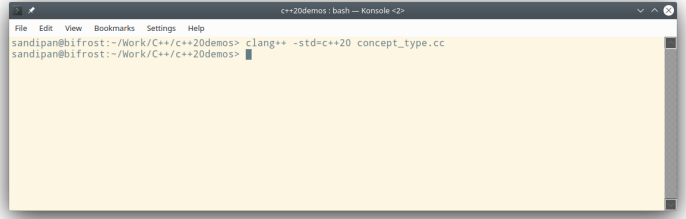
Named requirements on template parameters

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

Concepts

Named requirements on template parameters

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

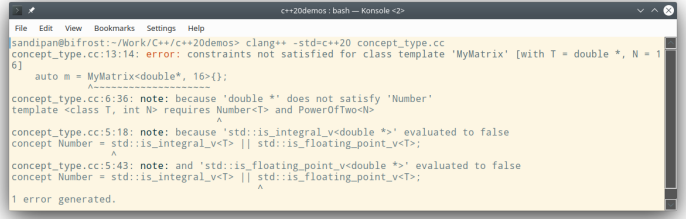


JÜLICH Forschungszentrum

Concepts

Named requirements on template parameters

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



Concepts

Named requirements on template parameters

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

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

Creating concepts

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

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

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

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

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

Creating concepts

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

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

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

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

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

Creating concepts

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

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

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

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

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

Creating concepts

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

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

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

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

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

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

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

Creating concepts

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

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

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

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

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

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

Using concepts

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

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

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

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

To constrain template parameters, one can

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

Declaring function input parameters with auto

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

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

Declaring function input parameters with auto

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

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

Declaring function input parameters with auto

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

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

Exercise 1.9:

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

Exercise 1.10:

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

Overloading based on concepts

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

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

Overloading based on concepts

```cpp
template <class N>
concept Number = std::floating_point<N>
                 or std::integral<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 or annotation.
- The compiler can choose between different versions of a function based on concepts

Overloading based on concepts

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

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

Overloading based on concepts

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

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

Overloading based on concepts

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

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

Overloading based on concepts

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

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

Overloading based on concepts

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

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

Selecting a code path based on input properties

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

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

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

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

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

Constraining non-template members of class templates

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

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

Constraining non-template members of class templates

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

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

Constraining non-template members of class templates

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

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

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

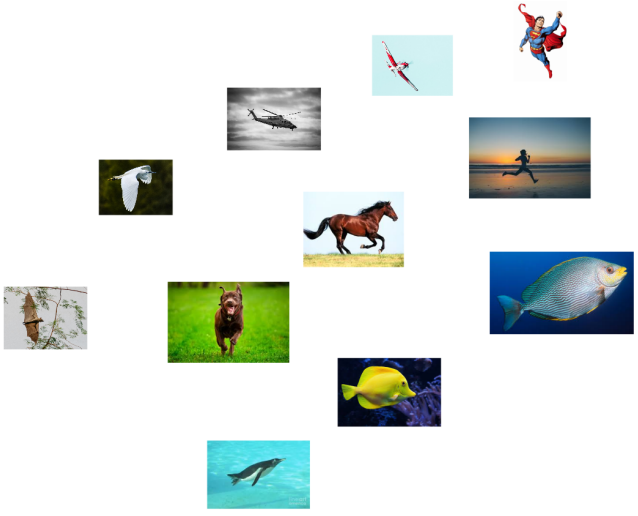
Constraining non-template members of class templates

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

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

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

Concepts: summary



f(those who can fly)

f(runners)

f(swimmers)

Exercise 1.11:

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

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

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

```
C overload_w_concepts.cc -o overload_w_concepts.1 && ./overload_w_concepts.1
```

Predefined useful concepts

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

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

Ranges

The range concept

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

The range concept

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

A C++ version can be as compact as the python version, but then it will also have the same problems:

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

The range concept

```
1  template <class T> requires has_referenceable_begin_end<T>
2  auto sum(T&& Container, element_type_of<T> start = 0) {
3      for (auto&& el : Container) start += el;
4      return start;
5  }
```

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

- Use `concepts` to put constraints on the function template!
- What matters for the code inside `sum` to work is the presence of `begin` and `end` functions, which return iterator types
- The type of the second parameter should somehow be obtained from the first using some kind of meta-function

The range concept

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

Using definitions in the `ranges` header, we have a few more lines, but:

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

The range concept

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

The range concept

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

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

The range concept

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

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

Tremendous flexibility, but still resolved at compilation time!

The range concept

```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 special kind of concept describing entities with a start and an end.
- 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
- Other useful `concepts` defined in the `ranges` header:
  - `view` is a range with constant time copy/move/assignment
  - `sr::sized_range`, `input_range`, `output_range`
  - `borrowed_range` : a type such that its iterators can be returned without the danger of dangling.
- The `<algorithm>` header has many algorithms taking ranges as inputs instead of pairs of iterators

The range concept

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

Exercise 1.12:

The function template `sum` in `examples/ranges/ranges0.cc` accepts any input range, i.e., some entity whose iterators satisfy the requirements of an `input_iterator`. Notice how we obtain the value type of the range

Fun with ranges and views

```cpp
// examples/ranges/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 >= 1000000000UL) break;
        }
    }
}
```

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

Fun with ranges and views

```cpp
// examples/ranges/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 >= 1000000000UL) break;
        }
    }
}
```

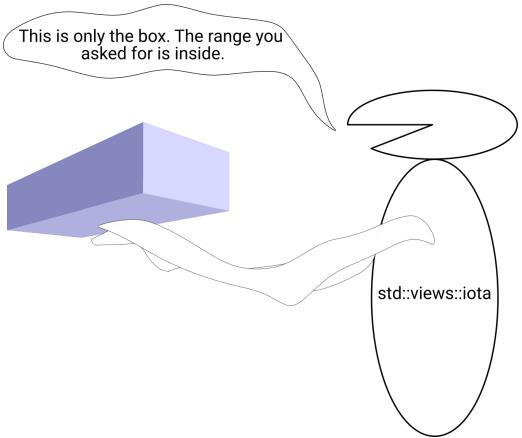
- All containers are ranges, but not all ranges are containers
- `std::string_view` is a perfectly fine range. Has iterators with the right properties. Has `begin()` and `end()` functions. It does not own the contents, but "ownership" is not part of the idea of a range.
- We could take this further by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

Fun with ranges and views

```cpp
// examples/ranges/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 >= 1000000000UL) break;
        }
    }
}
```

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

Fun with ranges and views



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

Fun with ranges and views

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

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

View adaptors

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

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

View adaptors

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

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

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

View adaptors

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

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

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

View adaptors

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

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

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

View adaptors

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

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

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

View adaptors

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

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

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

So what are we going to do with this ?

View adaptors

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

View adaptors

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

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

View adaptors

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

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

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

View adaptors

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

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

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

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

View adaptors

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

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

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

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

View adaptors

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

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

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

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

View adaptors

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

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

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

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

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

View adaptors

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

- Command line of Linux, Mac OS ...

View adaptors

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

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

View adaptors

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

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

View adaptors

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

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

View adaptors

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

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

View adaptors

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

View adaptors

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

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

View adaptors

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

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

View adaptors

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

- Start with a range of integers from 0 to $N = 10000$. `R0 = iota(0, N)`
- Map the integer range to real numbers in the range $[0, 2\pi)$, i.e., perform the transformation $n \mapsto \frac{2\pi n}{N}$ over the range: `R1 = R0 | transform([](int n) -> double { return 2*pi*n/N; })`
- Perform the transformation $x \mapsto sin^2(x) + cos^2(x) - 1$ over the resulting range

```
R2 = R1 | transform([](double x) -> double { return sin(x)*sin(x)+cos(x)*cos(x); } );
```

View adaptors

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

- Start with a range of integers from 0 to $N = 10000$. `R0 = iota(0, N)`
- Map the integer range to real numbers in the range $[0, 2\pi)$, i.e., perform the transformation $n \mapsto \frac{2\pi n}{N}$ over the range: `R1 = R0 | transform([](int n) -> double { return 2*pi*n/N; })`
- Perform the transformation $x \mapsto sin^2(x) + cos^2(x) - 1$ over the resulting range

```
R2 = R1 | transform([](double x) -> double { return sin(x)*sin(x)+cos(x)*cos(x); } );
```

- If absolute value of any of the values in the result exceeds ϵ, we have found a counter example

```
if (any_of(R2, [](auto x){return fabs(x) > eps;})) ...
```

View adaptors

```cpp
auto main() -> int {
    namespace sr = std::ranges;
    namespace sv = std::views;
    using std::numbers::pi;
    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.
- 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 resulting views 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 1.13:

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

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

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

Exercise 1.14:

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

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

View adaptors

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

- `sr::istream_view<T>` creates an (input) iterable range from an input stream. Each element of this range is of the type `T`.
- `sv::filter` is a view adaptor, which when applied to a range, produces another containing only the elements satisfying a given condition
- In the above, `std::print` is shown writing out a range. This works with the Clang standard library `libc++`. Since GCC doesn't have an implementation yet, please use `clang++` for this.

View adaptors

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

```cpp
// examples/ranges/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";
}
```

Ranges improvements in C++23

```cpp
std::vector v { "apples"s, "oranges"s,
                "mangos"s, "bananas"s };

for (auto [i, fruit] : sv::enumerate(v)) {
    std::print("{}: {}\n", i, fruit);
}
```

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

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

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

Ranges improvements in C++23

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

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

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

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

Ranges improvements in C++23

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

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

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

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

Ranges improvements in C++23

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

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

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

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

Ranges improvements in C++23

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

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

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

Ranges improvements in C++23

```cpp
auto main() -> int
{
    using namespace std::literals;
    namespace sr = std::ranges;
    namespace sv = sr::views;
    std::vector v{"One"s, "Two"s, "Three"s, "Four"s};
    std::print("{}\n", v);
    std::print("{}\n", v | sv::reverse);
    std::print("{}\n", sv::zip(v | sv::reverse, v));
    std::print("{}\n", sv::zip(sv::iota(1UL), v));
}
```

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

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

```
["One", "Two", "Three", "Four"]
["Four", "Three", "Two", "One"]
[("Four", "One"), ("Three", "Two"), ("Two", "Three"), ("One", "Four")]
[(1, "One"), (2, "Two"), (3, "Three"), (4, "Four")]
```

Recap of elementary features with an example

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

Recap of elementary features with an example

```cpp
// Trivial piece of code as a background for discussions
// examples/demo_saxpy/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

Generic lambdas...

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

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

Explicit template syntax for lambdas

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

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

Constraining generic functions

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

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

Constraining generic functions

```cpp
// examples/demo_saxpy/saxpy_3b.cc
    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>
            (value_type_of<T_in> a, T_in&& x, T_in&& y, T_out& z) {
        using in_element_type = value_type_of<T_in>;
        using out_element_type = value_type_of<T_out>;
        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!"
```

Constraining generic functions

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

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

Constraining generic functions

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

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

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

std::span as function parameters

```cpp
// examples/demo_saxpy/saxpy_5.cc
// other includes
#include <span>
template <class T>
void saxpy(T a, std::span<const T> x, std::span<const T> y, std::span<T> z)
{
    std::transform(x.begin(), x.end(), y.begin(), z.begin(),
        [a](T X, T Y) { return a * X + Y; });
}

auto main() -> int { ... }
```

- `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 1.15:

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

std::span as function parameters

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

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

std::span as function parameters

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

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

Constraining templates using concepts

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

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

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

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

Constraining templates using concepts

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

concepts: Named requirements on template parameters.

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

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

```cpp
// examples/demo_saxpy/saxpy_6.cc
template <class T> concept Number = std::floating_point<T> or std::integral<T>;
template <class T> requires Number<T>
auto saxpy(T a, std::span<const T> x, std::span<const T> y, std::span<T> z)
{
    std::transform(x.begin(), x.end(), y.begin(), z.begin(),
        [a](T X, T Y) { return a * X + Y; });
}
auto main() -> int
{
    {
        const std::array inp1 { 1., 2., 3., 4., 5. };
        const std::array inp2 { 9., 8., 7., 6., 5. };
        std::vector outp(inp1.size(), 0.);
        saxpy(10., {inp1}, {inp2}, {outp});
    }
    {
        const std::array inp1 { 1, 2, 3, 4, 5 };
        const std::array inp2 { 9, 8, 7, 6, 5 };
        std::vector outp(inp1.size(), 0);
        saxpy(10, {inp1}, {inp2}, {outp});
    }
}
```

Using concepts for our example

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

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

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

Using standard concepts and ranges in our example

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

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

We can now specify our requirements thoroughly...

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

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

Exercise 1.16:

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

Modules

C++20 modules

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

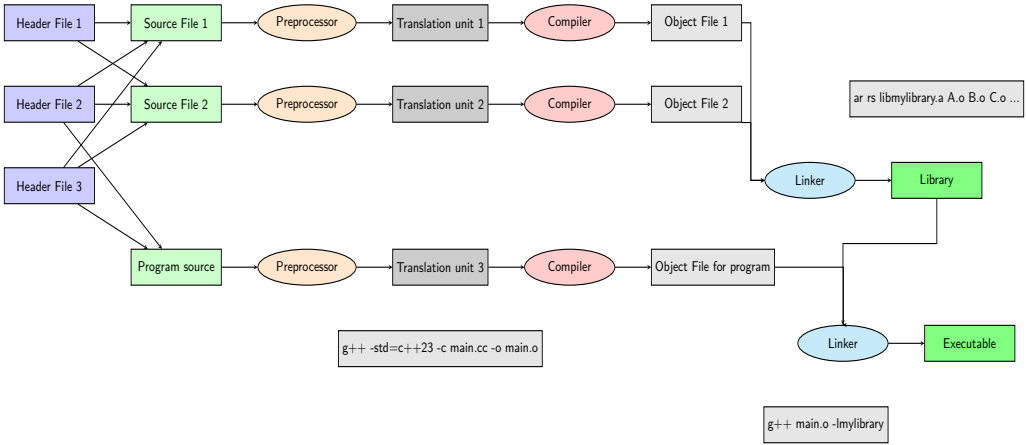
```
#ifndef SAXPY_HH
#define SAXPY_HH
#include <algorithm>
#include <span>
template <class T> concept Number = std::floating_point<T> or std::integral<T>;
template <class T> requires Number<T>
auto saxpy(T a, std::span<const T> x, std::span<const T> y, std::span<T> z){
    std::transform(x.begin(), x.end(),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

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

Problems with header files

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

The compilation process

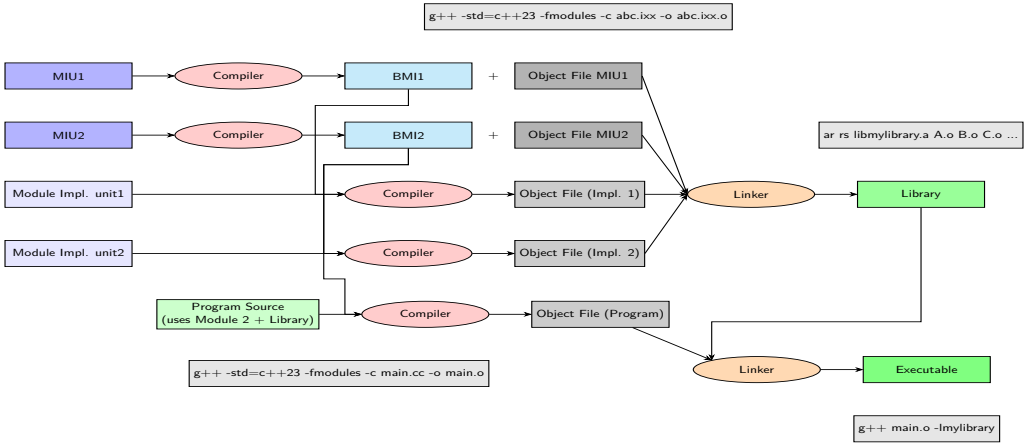


JÜLICH Forschungszentrum

Modules

- The C++20 `module`s offer an alternative (better) organisation, in which all code, including template code, can now reside in source files
- Module sources are processed once to generate the so called compiled module interface (binary module interface, BMI) in addition to an object file
- The BMI caches syntactic information from all entities in the module
- Any source `import`ing the module immediately has access to the precompiled syntax tree in the BMI, leading to faster compilation
- Enforces ordered compilation: since a source file may export a module to be imported by another source file
- C++23 added the standard library as a module

Compilation flow using modules



JÜLICH Forschungszentrum

Using modules

```
// examples/hello_m23.cc
import std;
auto main() -> int
{
  std::print("Hello, world!\n");
}
```

- In C++23, the standard library is also available as a module.
- **Note:** We have set up an alias `Gm='g++ -std=c++23 -fmodules'` in the course

```
$ Gm -c -fsearch-include-path bits/std.cc
$ Gm hello_m23.cc
$ ./a.out
$
```

- The first step generates the BMI for the standard library and the second compiles the actual program

Using modules

```
// examples/hello_m23.cc
import std;
auto main() -> int
{
  std::print("Hello, world!\n");
}
```

- Clang standard library is also avilable as a module, but usually it requires building using build system generators like CMake

```
$ export libcxxsrc=$(find $(dirname $(which clang) )/.. -name std.cppm 2>/dev/null )
$ clang++ -std=c++23 -stdlib=libc++ -Wno-reserved-identifier\
  -Wno-reserved-module-identifier --precompile \
  -o std.pcm $libcxxsrc
$ clang++ -std=c++23 -stdlib=libc++ -fmodules -fmodule-file=std=std.pcm hello_m23.cc
$ ./a.out
```

- Not as elegant as the GCC oneliner, but it's possible to precompile the standard library module with clang
- It is necessary to explicitly specify the location of the module file when using it

Using modules

Exercise 1.17:

Convert a few of the example programs you have seen during the course to use modules syntax instead. At the moment it means no more than replacing the `#include` lines with the `import` line for the standard library. The point is to get used to the extra compilation options you need with modules at the moment. Use, for instance, the date time library demo programs like `feb.cc` and `advent.cc`. In the next step, replace all the `import` lines using individual header units for standard library features by a single `import std;` line. Refer to the slides and compile using `g++`.

Creating a module (example)

```
class NullSimulator {
        bool config_ok{false}, run_ok{false};
public:
    void configure(std::string_view pars)
    {
        if (pars.empty())
            throw std::runtime_error{"..."};
        std::println("config {}", pars);
        config_ok = true;
    }
    void run()
    {
        if (not config_ok)
                throw std::runtime_error{"..."};
        std::println("running NullSimulator!");
        run_ok = true;
    }
    auto summary() const -> std::string
    {
        if (not run_ok)
                throw std::runtime_error{"..."};
        return { "And here are the results:" };
```

```
auto main() -> int {
    using namespace cxx_course;
    try {
        NullSimulator sim;
        sim.configure("Modules demo");
        sim.run();
        std::print("{}\n", sim.summary());
        return 0;
    } catch (std::exception& err) {
        std::print("{}\n", err.what());
        return 1;
    };
}
```

- A simple “do nothing” simulator class, mimicking the top level control flow in many applications
- We want to put our `NullSimulator` in a module and **`import`** it in `main` and use it as shown

Creating a module (example)

- When using our module, like when using the standard library module, there are no changes in syntax
- We just have to import our module
- Notice that the module name, namespace name and the class name are all independent in C++
- `#include "myheader.hh"` searches for the header file and inserts it in place
- In contrast, the file name containing module code is not tied to the module name (which is why we had to explicitly state `-fmodule-file=std=std.pcm` when compiling with clang earlier)

```cpp
// main.cc
import NullSim;
import std;
auto main() -> int {
    using namespace cxx_course;
    try {
        NullSimulator sim;
        sim.configure("Modules demo");
        sim.run();
        std::print("{}\n", sim.summary());
        return 0;
    } catch (std::exception& err) {
        std::print("{}\n", err.what());
        return 1;
    };
}
```

- In the module world, there are **no transitive imports**. Module `std` has to be imported independently wherever it is used

Creating a module (example)

```
// NullSimulator.cc
export module NullSim;
import std;
namespace cxx_course {
export class NullSimulator {
        bool config_ok{false}, run_ok{false};
public:
    void configure(std::string_view pars) {
        // code...
    }
    void run() {
        // code...
    }
    auto summary() const -> std::string {
        // code...
    }
};
void i_am_invisible() {
//...
}
}
```

- The class/function declarations and definitions can all be put in module source files
- The first non-comment line has to declare the module.
  - `export module XYZ;` : Module **interface** unit
  - `module XYZ;` : Module **implementation** unit
  - `module;` : Start of the "global module fragment"

Creating a module (example)

```
// NullSimulator.cc
export module NullSim;
import std;
namespace cxx_course {
export class NullSimulator {
        bool config_ok{false}, run_ok{false};
public:
    void configure(std::string_view pars) {
        // code...
    }
    void run() {
        // code...
    }
    auto summary() const -> std::string {
        // code...
    }
};
void i_am_invisible() {
//...
}
}
```

- Unlike declarations in a header file, those in a module file are not automatically visible if you import the module
- Only exported symbols are visible to code which imports a module
- Unexported declarations remain private

Creating a module (example)

```
// NullSimulator.cc
export module NullSim;
import std;
namespace cxx_course {
export class NullSimulator {
        bool config_ok{false}, run_ok{false};
public:
    void configure(std::string_view pars) {
        // code...
    }
    void run() {
        // code...
    }
    auto summary() const -> std::string {
        // code...
    }
};
void i_am_invisible() {
//...
}
}
```

```
// main.cc
import NullSim;
import std;
auto main() -> int {
    using namespace cxx_course;
    try {
        NullSimulator sim;
        sim.configure("Modules demo");
        sim.run();
        std::print("{}\n", sim.summary());
    } catch (std::exception& err) {
        std::print("{}\n", err.what());
        return 1;
    };
}
```

```
Gm -c -fsearch-include-path bits/std.cc
Gm -c NullSimulator.cc
Gm -c main.cc
g++ main.o NullSimulator.o std.o -o nullsim
```

Building the project using CMake

```
cmake_minimum_required(VERSION 3.30 FATAL_ERROR)
set (CMAKE_CXX_EXTENSIONS OFF)
set (CMAKE_CXX_STANDARD 23)
set (CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_EXPORT_COMPILE_COMMANDS ON)

if(CMAKE_VERSION VERSION_GREATER_EQUAL "4.0.3")
    set(CMAKE_EXPERIMENTAL_CXX_IMPORT_STD
      "d0edc3af-4c50-42ea-a356-e2862fe7a444")
elseif(CMAKE_VERSION VERSION_GREATER_EQUAL "4.0.0")
    set(CMAKE_EXPERIMENTAL_CXX_IMPORT_STD
      "a9e1cf81-9932-4810-974b-6eccaf14e457")
elseif(CMAKE_VERSION VERSION_GREATER_EQUAL "3.30.0")
    set(CMAKE_EXPERIMENTAL_CXX_IMPORT_STD
      "0e5b6991-d74f-4b3d-a41c-cf096e0b2508")
endif()

project(nullsim LANGUAGES CXX)
set(CMAKE_CXX_MODULE_STD 1)
add_executable(nullsim main.cc)
target_sources(nullsim PUBLIC FILE_SET CXX_MODULES
    FILES NullSimulator.cc)
```

- Let CMake handle the necessary compiler options

```
mkdir build && cd build
CXX=g++ cmake -GNinja ..
ninja
```

- For clang, you would configure like this:

```
CXX=clang++ CXXFLAGS="-stdlib=libc++"\
  cmake -GNinja ..
```

- The odd experimental import std guids are temporary as long as the feature is considered “experimental”

Exercise 1.18:

The simple example with `NullSimulator` in the previous slides is in the folder `examples/modules/create0`. Build it using CMake with both Clang and GCC as compilers. Try building it directly using GCC as shown. The idea is to familiarise yourself with the modules based workflow. Insert a simple `I_am_invisible` function as shown in the slides. Try to use it in `main()`. What error do you get, even if we are importing the module containing it? Now, add the keyword **`export`** before that function, and test again!

Creating a module: implementation units

```
// NullSimulator.ixx
export module NullSim;
export class NullSimulator {
public:
    void configure(std::string_view pars);
    void run();
    auto summary() const -> std::string;
};
```

```
// NullSimulator.cc
module NullSim;
import std;
void NullSimulator::configure(std::string_...) {
    // actual implementation
}
void NullSimulator::run() {
    // code...
}
auto NullSimulator::summary() const
    -> std::string {
    // code...
}
```

- Separating implementation is not syntactically necessary, but may sometimes be desirable to offer a clearer overview of the interface without the implementation code
- Only the interface unit shall export the symbols and the module itself, not the implementation unit
- The implementation units are bound to the module by the module lines at the start

Creating a module: implementation units

```
// NullSimulator.ixx
export module NullSim;
export class NullSimulator {
public:
    void configure(std::string_view pars);
    void run();
    auto summary() const -> std::string;
};
```

```
// NullSimulator.cc
module NullSim;
import std;
void NullSimulator::configure(std::string_...) {
    // actual implementation
}
void NullSimulator::run() {
    // code...
}
auto NullSimulator::summary() const
    -> std::string {
    // code...
}
```

```
Gm -c -fsearch-include-path bits/std.cc
Gm -c NullSimulator.ixx
Gm -c NullSimulator.cc
Gm -c main.cc
g++ main.o NullSimulator.o std.o -o nullsim
```

Creating a module: implementation units

```
// NullSimulator.ixx
export module NullSim;
export class NullSimulator {
public:
    void configure(std::string_view pars);
    void run();
    auto summary() const -> std::string;
};
```

```
// NullSimulator.cc
module NullSim;
import std;
void NullSimulator::configure(std::string_...) {
    // actual implementation
}
void NullSimulator::run() {
    // code...
}
auto NullSimulator::summary() const
    -> std::string {
    // code...
}
```

```
Gm -c -fsearch-include-path bits/std.cc
Gm -c NullSimulator.ixx
Gm -c NullSimulator.cc
Gm -c main.cc
g++ main.o NullSimulator.o std.o -o nullsim
```

Works, but masks a big problem! Can you see it?

Creating a module: implementation units

```
Gm -c -fsearch-include-path bits/std.cc
Gm -c NullSimulator.ixx
Gm -c NullSimulator.cc
Gm -c main.cc
g++ main.o NullSimulator.o std.o -o nullsim
```

- The third line overwrites one of the outputs `NullSimulator.o` from the second line
- When we compile the interface unit, the by-products are the BMI and an object file.
  - The BMI contains information about the interface and the pre-compiled inline functions, class and function templates.
  - Since implementation units are not mandatory, the interface units can contain ordinary function definitions, which will result in binary code stored in the object file
- Compiling the implementation unit is like compiling any other C++ file, and results in an object file for the functions defined there.
- Our example works because we separated all implementation code into the implementation unit `NullSimulator.cc`. What if we left one function body in `NullSimulator.ixx`?

Exercise 1.19:

The example directory `modules/create1` contains the code where we have split the `NullSimulator` code between the module interface and implementation units. Compile it with GCC as shown in the previous slides. What happens if you delete the `NullSimulator.o` output after the second step and go through with the rest? Then in `modules/create1b`, we have the same code, but, we have inserted a new free function `check_results`. It will lead to a compiler error in step 2, which you should be able to solve. Does the rest of the build process go through without issues?

Non-inline functions in interface units

- ... end up in the object file when the interface unit is compiled. If we are using those functions, we need that object file when linking
- $\Longrightarrow$ Possible conflict in output filename when compiling the implementation unit
- Common solutions:
  - When compiling with option `-c`, use an explicit output filename, e.g.,

```
Gm -c NullSimulator.ixx -o NullSimulator.ixx.o
Gm -c NullSimulator.cc -o NullSimulator.cc.o
Gm -c main.cc -o main.cc.o
g++ main.cc.o NullSimulator.cc.o NullSimulator.ixx.o std.o
```

  - Abandon suffix based differentiation, and name interface and implementation units differently, e.g., `NullSimulator.cc` and `NullSimulator_impl.cc`.

```
Gm -c NullSimulator.cc
Gm -c NullSimulator_impl.cc
Gm -c main.cc
g++ main.o NullSimulator.o NullSimulator_impl.o std.o
```

Non-inline functions in interface units

- How do the functions defined inside the class declarations behave?
- Answer: as of C++23, like regular non-inline functions!

```
// NullSimulator.ixx
export module NullSim;
import std;
namespace cxx_course {
export class NullSimulator {
        bool config_ok{false}, run_ok{false};
public:
    void configure(std::string_view pars);
    void run();
    auto summary() const -> std::string
    {
        return {"Summary of results:"};
    }
};
}
```

- The function `summary` is compiled and placed in the object code when compiling `NullSimulator.ixx`

```
// NullSimulator.cc
module NullSim;
void NullSimulator::configure(...)
{
    ...
}
void NullSimulator::run()
{
    ...
}
```

- The functions `configure()` and `run()` end up in the object file when `NullSimulator.cc` is compiled.
- Both generated object files are required for linking!

Building process including modules

- Object files from the interface and implementation units can be combined into static/shared libraries.
- The interface units also produce the BMI
- To compile a module user, we need the BMI of all modules it uses
- To link the application, we need the object files or libraries made of them.

Splitting the module interface

- Large module interface units may be split into multiple files
  - Example: different large classes in their own files
  - Reason: maintenance, collaborative development
  - How: module partitions
- Each of the 3 module interface units shown here belong to the same module `Measurements`. Observe how they are named and exported
- Implementation units belong to the module as a whole, not to any partitions

```
// Measurements.ixx
export module Measurements;
export import :RMSD;
export import :Rg;
export import :ContactOrder;
```

```
// RMSD.ixx
export module Measurements:RMSD;
export class RMSD {/*...*/};
```

```
// Rg.ixx
export module Measurements:Rg;
export class Rg {/*...*/};
```

```
// ContactOrder.ixx
export module Measurements:ContactOrder;
export class ContactOrder {/*...*/};
```

- Partitions can be imported by other partitions
- Partitions can not be directly imported outside the module

Splitting the module interface

- Large module interface units may be split into multiple files
  - Example: different large classes in their own files
  - Reason: maintenance, collaborative development
  - How: module partitions
- Each of the 3 module interface units shown here belong to the same module `Measurements`. Observe how they are named and exported
- Implementation units belong to the module as a whole, not to any partitions

```
// Measurements.ixx
export module Measurements;
export import :RMSD;
export import :Rg;
export import :ContactOrder;
```

```
// RMSD.ixx
export module Measurements:RMSD;
export class RMSD {/*...*/};
```

```
// Rg.ixx
export module Measurements:Rg;
export class Rg {/*...*/};
```

```
// ContactOrder.ixx
export module Measurements:ContactOrder;
export class ContactOrder {/*...*/};
```

- The interface exported by the module partitions can be exported by the primary module interface

Organisation of interface partitions vs headers

```cpp
// M.hh
constexpr auto R = 42;
```

```cpp
// A.hh
struct cA {
    auto func() const -> int;
};
```

```cpp
// B.hh
struct cB {
    auto func() const -> int;
};
```

- Quite common to have classes in their own headers
- Must include the header for a class when implementing member functions in separate source files

```cpp
// A.cc
#include "A.hh"
auto cA::func() const -> int {
        return 42;
}
```

```cpp
// B.cc
#include "B.hh"
auto cB::func() const -> int {
        return 43;
}
```

```cpp
// main.cc
#include "A.hh"
#include "B.hh"
#include "M.hh"
auto main() -> int {
    cA a;
    cB b;
    return R + a.func() + b.func();
}
```

Organisation of interface partitions vs headers

```
// M.ixx
export module M;
export constexpr auto R = 42;
export struct cA {
    auto func() const -> int;
};
export struct cB {
    auto func() const -> int;
};
```

- Large module interface units are not considered bad
- Names exported in the primary module interface don't need to be imported inside the module's implementation units

```
// A.cc
module M;
auto cA::func() const -> int {
        return 42;
}
```

```
// B.cc
module M;
auto cB::func() const -> int {
        return 43;
}
```

```
// main.cc
import M;
auto main() -> int {
    cA a;
    cB b;
    return a.func() + b.func();
}
```

Organisation of interface partitions vs headers

```
// M.ixx
export module M;
export constexpr auto R = 42;
export import :A;
export import :B;
```

```
// A.ixx
export module M:A;
export struct cA {
    auto func() const -> int;
};
```

```
// B.ixx
export module M:B;
export struct cB {
    auto func() const -> int;
};
```

- In case the primary interface is split into partitions, the partitions need to be re-exported in it, in order that the declarations in them be visible in implementation units

```
// A.cc
module M;
auto cA::func() const -> int {
        return 42;
}
```

```
// B.cc
module M;
auto cB::func() const -> int {
        return 43;
}
```

```
// main.cc
import M;
auto main() -> int {
    cA a;
    cB b;
    return R + a.func() + b.func();
}
```

Organisation of interface partitions vs headers

```
// M.ixx
export module M;
export constexpr auto R = 42;
export import :A;
export import :B;
```

```
// A.ixx
export module M:A;
export struct cA {
    auto func() const -> int;
};
```

```
// B.ixx
export module M:B;
export struct cB {
    auto func() const -> int;
};
```

- Module exporters and importers must be compiled in a specific order, not always trivially inferred

```
// A.cc
module M;
auto cA::func() const -> int {
        return 42;
}
```

```
// B.cc
module M;
auto cB::func() const -> int {
        return 43;
}
```

```
// main.cc
import M;
auto main() -> int {
    cA a;
    cB b;
    return R + a.func() + b.func();
}
```

Letting the build system generators handle modules

- CMake and other build system generators simplify the management of modules based compilation
- A set of module interface units can be attached to an executable or library target in CMake with the `target_sources` function
- CMake determines the order in which they must be compiled, and sets where the BMI and object files are stored
- The library or executable is then linked using the relevant set of object files
- CMake places the BMI files at the right places so that it can find them while compiling the rest of the project
- The BMI must be generated from the module interface units fresh using the build flags.

Using header files inside module units

- Even if you want to use modules for your project, you might need external dependencies which don't yet support a modules based build
- Including headers is possible, but restricted to a specific segment in a module unit, called the **global module fragment**
- The global module fragment, if present, has to be the first section in a module unit.

```
module;
#include <Eigen/Dense>
#include <boost/type_index.hpp>
module Measurements;

import ...;
```

Using header files inside module units

- Even if you want to use modules for your project, you might need external dependencies which don't yet support a modules based build
- Including headers is possible, but restricted to a specific segment in a module unit, called the **global module fragment**
- The global module fragment, if present, has to be the first section in a module unit.

```
module;
#include <Eigen/Dense>
#include <boost/type_index.hpp>
module Measurements;

import ...;
```

- It starts like an empty or nameless module declaration (`module;`). It ends at the actual module declaration (with `export` for interface and without for implementation)

As of October 2025, mixed mode projects when using the standard library as a module, but also including it directly or indirectly through dependencies, lead to errors with GCC! With versions 15.1, 15.2 and the latest git version.

Converting older header based projects

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

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

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

Converting older header based projects

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

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

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

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

Converting older header based projects

Make a module interface unit

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

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

Converting older header based projects

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

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

Make a module interface unit

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

Converting older header based projects

Make a module interface unit

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


#include <algorithm>
#include <span>

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

Converting older header based projects

```
// saxpy.hh -> saxpy.ixx
module;
#include <algorithm>
#include <span>
export module saxpy;

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

Make a module interface unit

- Start a global module fragment to enclose the headers you have to use.
- Export the module.

Converting older header based projects

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

export module saxpy;
import std;

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

Make a module interface unit

- Start a global module fragment to enclose the headers you have to use.
- Export the module.
- If you can get by with only **`import`**s, replace `#include` lines with corresponding **`import`** lines. Omit the global module fragment in this case

Converting older header based projects

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

export module saxpy;
import std;

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

Make a module interface unit

- Start a global module fragment to enclose the headers you have to use.
- Export the module.
- If you can get by with only **`import`**s, replace `#include` lines with corresponding **`import`** lines. Omit the global module fragment in this case
- Explicitly export any definitions (classes, functions...) you want for users of the module. Anything not exported by a module is automatically private to the module

Using the module

Use your module

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

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

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

Using the module

```
// usesaxpy.cc
import std;
#include "saxpy.hh"

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

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

Use your module

- Optionally replace `#include` lines with corresponding `import` line(s).
- If the source is not a module unit, include directives may be used alongside imports

Using the module

```
// usesaxpy.cc
import std;
import saxpy;

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

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

Use your module

- Optionally replace `#include` lines with corresponding `import` line(s).
- If the source is not a module unit, include directives may be used alongside imports
- Import your module by name

Using the module

```
// usesaxpy.cc
import std;
import saxpy;

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

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

Use your module

- Optionally replace `#include` lines with corresponding `import` line(s).
- If the source is not a module unit, include directives may be used alongside imports
- Import your module by name
- Importing `saxpy` here, only grants us access to the explicitly exported function `saxpy`. Not other functions, classes, concepts etc. defined in the module `saxpy`, not any other module imported in the module interface unit.

Setup building with CMake

```
cmake_minimum_required(VERSION 3.30)
set (CMAKE_CXX_EXTENSIONS OFF)
set (CMAKE_CXX_STANDARD 23)
set (CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_EXPORT_COMPILE_COMMANDS ON)

if(CMAKE_VERSION VERSION_GREATER_EQUAL "4.0.3")
    set(CMAKE_EXPERIMENTAL_CXX_IMPORT_STD
      "d0edc3af-4c50-42ea-a356-e2862fe7a444")
elseif(CMAKE_VERSION VERSION_GREATER_EQUAL "4.0.0")
    set(CMAKE_EXPERIMENTAL_CXX_IMPORT_STD
      "a9e1cf81-9932-4810-974b-6eccaf14e457")
elseif(CMAKE_VERSION VERSION_GREATER_EQUAL "3.30.0")
    set(CMAKE_EXPERIMENTAL_CXX_IMPORT_STD
      "0e5b6991-d74f-4b3d-a41c-cf096e0b2508")
endif()
project(use_saxpy-example LANGUAGES CXX)
set(CMAKE_CXX_MODULE_STD 1)
add_executable(use_saxpy usesaxpy.cc)
target_sources(use_saxpy
  PUBLIC
    FILE_SET CXX_MODULES
    FILES saxpy.ixx
)
```

- `CMake` supports C++ modules since the version 3.28
- Since version 3.30 it supports creation and use of standard library as a module if the compiler + standard library combination supports it
- This means Clang >= 18.1 or GCC >= 15.1.
- The `Ninja` generator is required
- Massive simplification of the build process!

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

Exercise 1.20:

Versions of the `saxpy` program written using header files and then modules can be found in the `examples/saxpy/`. Familiarise yourself with the process of building applications with modules. Experiment by writing a new inline function in the module interface file without exporting it. Try to call that function from `main`. Check again after exporting it in the module.

Exercise 1.21:

As a more complicated example, we have in `examples/2_any` the second version of our container with polymorphic geometrical objects. The header and source files for each class `Point`, `Circle` etc have been rewritten for modules. Compare the two versions, build them, run them.

Modules: Summary

- Status, since around September 2024: Usable!
- There is support from CMake and b2build with GCC and Clang.
- A different organisation of multi-file projects than the one with header and source files
- Promise:
  - easier control over symbol visibility
  - no “copy and paste” solution of headers
  - smaller translation units and hence faster compilation
  - no transitive imports, no import of MACROs defined in imported modules
- Does not change anything about functions, classes, templates or concepts, just how we place them in files
- Module interface units play a similar role to headers, but without their problems

Linkage: determining who is who in multi-file projects

```
namespace app {
    int variable{};
}
void A()
{
    std::printf("%d\n",
      app::variable++);
}
```

Linkage: determining who is who in multi-file projects

```
namespace app {
    int variable{};
}
void A()
{
    std::printf("%d\n",
      app::variable++);
}
```

```
g++ -S -O3 A.cc
```

Linkage: determining who is who in multi-file projects

```
namespace app {
    int variable{};
}
void A()
{
    std::printf("%d\n",
      app::variable++);
}
```

```
g++ -S -O3 A.cc
```

```
A():
    movl    app::variable(%rip), %esi
    movl    $.LC0, %edi
    leal    1(%rsi), %eax
    movl    %eax, app::variable(%rip)
    xorl    %eax, %eax
    jmp     printf
app::variable:
        .zero   4
```

Linkage: determining who is who in multi-file projects

```
namespace app {
    int variable{};
}
void A()
{
    std::printf("%d\n",
      app::variable++);
}
```

```
g++ -S -O3 A.cc
```

```
namespace app {
    int variable{};
}
void B()
{
    std::printf("%d\n",
      app::variable++);
}
```

```
A():
    movl    app::variable(%rip), %esi
    movl    $.LC0, %edi
    leal    1(%rsi), %eax
    movl    %eax, app::variable(%rip)
    xorl    %eax, %eax
    jmp     printf
app::variable:
        .zero   4
```

Linkage: determining who is who in multi-file projects

```cpp
namespace app {
    int variable{};
}
void A()
{
    std::printf("%d\n",
      app::variable++);
}
```

```
g++ -S -O3 A.cc
```

```cpp
namespace app {
    int variable{};
}
void B()
{
    std::printf("%d\n",
      app::variable++);
}
```

```
g++ -S -O3 B.cc
```

```
A():
    movl    app::variable(%rip), %esi
    movl    $.LC0, %edi
    leal    1(%rsi), %eax
    movl    %eax, app::variable(%rip)
    xorl    %eax, %eax
    jmp     printf
app::variable:
        .zero   4
```

Linkage: determining who is who in multi-file projects

```
namespace app {
    int variable{};
}
void A()
{
    std::printf("%d\n",
      app::variable++);
}
```

```
g++ -S -O3 A.cc
```

```
namespace app {
    int variable{};
}
void B()
{
    std::printf("%d\n",
      app::variable++);
}
```

```
g++ -S -O3 B.cc
```

```
A():
    movl    app::variable(%rip), %esi
    movl    $.LC0, %edi
    leal    1(%rsi), %eax
    movl    %eax, app::variable(%rip)
    xorl    %eax, %eax
    jmp     printf
app::variable:
        .zero   4
```

```
B():
    movl    app::variable(%rip), %esi
    movl    $.LC0, %edi
    leal    1(%rsi), %eax
    movl    %eax, app::variable(%rip)
    xorl    %eax, %eax
    jmp     printf
app::variable:
        .zero   4
```

Linkage: determining who is who in multi-file projects

```
namespace app {
    int variable{};
}
void A()
{
    std::printf("%d\n",
      app::variable++);
}
```

```
g++ -S -O3 A.cc
```

```
namespace app {
    int variable{};
}
void B()
{
    std::printf("%d\n",
      app::variable++);
}
```

```
g++ -S -O3 B.cc
```

```
auto main() -> int
{
    A();
    B();
}
```

```
A():
    movl    app::variable(%rip), %esi
    movl    $.LC0, %edi
    leal    1(%rsi), %eax
    movl    %eax, app::variable(%rip)
    xorl    %eax, %eax
    jmp     printf
app::variable:
        .zero   4
```

```
B():
    movl    app::variable(%rip), %esi
    movl    $.LC0, %edi
    leal    1(%rsi), %eax
    movl    %eax, app::variable(%rip)
    xorl    %eax, %eax
    jmp     printf
app::variable:
        .zero   4
```

Linkage: determining who is who in multi-file projects

```cpp
namespace app {
    int variable{};
}
void A()
{
    std::printf("%d\n",
      app::variable++);
}
```

```
g++ -S -O3 A.cc
```

```cpp
namespace app {
    int variable{};
}
void B()
{
    std::printf("%d\n",
      app::variable++);
}
```

```
g++ -S -O3 B.cc
```

```cpp
auto main() -> int
{
    A();
    B();
}
```

```
g++ -S -O3 main.cc
```

```
A():
    movl    app::variable(%rip), %esi
    movl    $.LC0, %edi
    leal    1(%rsi), %eax
    movl    %eax, app::variable(%rip)
    xorl    %eax, %eax
    jmp     printf
app::variable:
        .zero   4
```

```
B():
    movl    app::variable(%rip), %esi
    movl    $.LC0, %edi
    leal    1(%rsi), %eax
    movl    %eax, app::variable(%rip)
    xorl    %eax, %eax
    jmp     printf
app::variable:
        .zero   4
```

Linkage: determining who is who in multi-file projects

```
namespace app {
    int variable{};
}
void A()
{
    std::printf("%d\n",
      app::variable++);
}
```

```
g++ -S -O3 A.cc
```

```
namespace app {
    int variable{};
}
void B()
{
    std::printf("%d\n",
      app::variable++);
}
```

```
g++ -S -O3 B.cc
```

```
auto main() -> int
{
    A();
    B();
}
```

```
g++ -S -O3 main.cc
```

```
A():
    movl    app::variable(%rip), %esi
    movl    $.LC0, %edi
    leal    1(%rsi), %eax
    movl    %eax, app::variable(%rip)
    xorl    %eax, %eax
    jmp     printf
app::variable:
        .zero   4
```

```
B():
    movl    app::variable(%rip), %esi
    movl    $.LC0, %edi
    leal    1(%rsi), %eax
    movl    %eax, app::variable(%rip)
    xorl    %eax, %eax
    jmp     printf
app::variable:
        .zero   4
```

```
main:
    call    A()
    call    B()
    xorl    %eax, %eax
    addq    $8, %rsp
    ret$
```

Linkage: determining who is who in multi-file projects

```
namespace app {
    int variable{};
}
void A()
{
    std::printf("%d\n",
      app::variable++);
}
```

```
g++ -S -O3 A.cc
```

```
namespace app {
    int variable{};
}
void B()
{
    std::printf("%d\n",
      app::variable++);
}
```

```
g++ -S -O3 B.cc
```

```
auto main() -> int
{
    A();
    B();
}
```

```
g++ -S -O3 main.cc
```

```
A():
    movl    app::variable(%rip), %esi
    movl    $.LC0, %edi
    leal    1(%rsi), %eax
    movl    %eax, app::variable(%rip)
    xorl    %eax, %eax
    jmp     printf
app::variable:
        .zero   4
```

```
B():
    movl    app::variable(%rip), %esi
    movl    $.LC0, %edi
    leal    1(%rsi), %eax
    movl    %eax, app::variable(%rip)
    xorl    %eax, %eax
    jmp     printf
app::variable:
        .zero   4
```

```
main:
    call    A()
    call    B()
    xorl    %eax, %eax
    addq    $8, %rsp
    ret$
```

Link

Linkage: determining who is who in multi-file projects

```cpp
namespace app {
    int variable{};
}
void A()
{
    std::printf("%d\n",
      app::variable++);
}
```

```
g++ -S -O3 A.cc
```

```cpp
namespace app {
    int variable{};
}
void B()
{
    std::printf("%d\n",
      app::variable++);
}
```

```
g++ -S -O3 B.cc
```

```cpp
auto main() -> int
{
    A();
    B();
}
```

```
g++ -S -O3 main.cc
```

```
A():
    movl    app::variable(%rip), %esi
    movl    $.LC0, %edi
    leal    1(%rsi), %eax
    movl    %eax, app::variable(%rip)
    xorl    %eax, %eax
    jmp     printf
app::variable:
        .zero   4
```

```
B():
    movl    app::variable(%rip), %esi
    movl    $.LC0, %edi
    leal    1(%rsi), %eax
    movl    %eax, app::variable(%rip)
    xorl    %eax, %eax
    jmp     printf
app::variable:
        .zero   4
```

```
main:
    call    A()
    call    B()
    xorl    %eax, %eax
    addq    $8, %rsp
    ret$
```

Link

- Have to produce one sequence of instructions.

Linkage: determining who is who in multi-file projects

```cpp
namespace app {
    int variable{};
}
void A()
{
    std::printf("%d\n",
      app::variable++);
}
```

```
g++ -S -O3 A.cc
```

```cpp
namespace app {
    int variable{};
}
void B()
{
    std::printf("%d\n",
      app::variable++);
}
```

```
g++ -S -O3 B.cc
```

```cpp
auto main() -> int
{
    A();
    B();
}
```

```
g++ -S -O3 main.cc
```

```
A():
    movl    app::variable(%rip), %esi
    movl    $.LC0, %edi
    leal    1(%rsi), %eax
    movl    %eax, app::variable(%rip)
    xorl    %eax, %eax
    jmp     printf
app::variable:
        .zero   4
```

```
B():
    movl    app::variable(%rip), %esi
    movl    $.LC0, %edi
    leal    1(%rsi), %eax
    movl    %eax, app::variable(%rip)
    xorl    %eax, %eax
    jmp     printf
app::variable:
        .zero   4
```

```
main:
    call    A()
    call    B()
    xorl    %eax, %eax
    addq    $8, %rsp
    ret$
```

Link

- Have to produce one sequence of instructions.
- But there are two versions of `app::variable`, one from the compilation of `A.cc`, one from that of `B.cc`

Linkage: determining who is who in multi-file projects

```cpp
namespace app {
    int variable{};
}
void A()
{
    std::printf("%d\n",
      app::variable++);
}
```

```
g++ -S -O3 A.cc
```

```cpp
namespace app {
    int variable{};
}
void B()
{
    std::printf("%d\n",
      app::variable++);
}
```

```
g++ -S -O3 B.cc
```

```cpp
auto main() -> int
{
    A();
    B();
}
```

```
g++ -S -O3 main.cc
```

```
A():
    movl    app::variable(%rip), %esi
    movl    $.LC0, %edi
    leal    1(%rsi), %eax
    movl    %eax, app::variable(%rip)
    xorl    %eax, %eax
    jmp     printf
app::variable:
        .zero   4
```

```
B():
    movl    app::variable(%rip), %esi
    movl    $.LC0, %edi
    leal    1(%rsi), %eax
    movl    %eax, app::variable(%rip)
    xorl    %eax, %eax
    jmp     printf
app::variable:
        .zero   4
```

```
main:
    call    A()
    call    B()
    xorl    %eax, %eax
    addq    $8, %rsp
    ret$
```

Link

- Have to produce one sequence of instructions.
- But there are two versions of `app::variable`, one from the compilation of `A.cc`, one from that of `B.cc`
- Are they to be treated as the same entity or different ones?

Linkage: determining who is who in multi-file projects

```
// var.hh
namespace app {
    int variable{};
}
```

Linkage: determining who is who in multi-file projects

```
// var.hh
namespace app {
    int variable{};
}
```

```
#include "var.hh"
void A()
{
    std::printf("%d\n",
      app::variable++);
}
```

```
#include "var.hh"
void B()
{
    std::printf("%d\n",
      app::variable++);
}
```

Linkage: determining who is who in multi-file projects

```
// var.hh
namespace app {
    int variable{};
}
```

```
#include "var.hh"
void A()
{
    std::printf("%d\n",
      app::variable++);
}
```

```
#include "var.hh"
void B()
{
    std::printf("%d\n",
      app::variable++);
}
```

```
A():
    movl    app::variable(%rip), %esi
    movl    $.LC0, %edi
    leal    1(%rsi), %eax
    movl    %eax, app::variable(%rip)
    xorl    %eax, %eax
    jmp     printf
app::variable:
        .zero   4
```

```
B():
    movl    app::variable(%rip), %esi
    movl    $.LC0, %edi
    leal    1(%rsi), %eax
    movl    %eax, app::variable(%rip)
    xorl    %eax, %eax
    jmp     printf
app::variable:
        .zero   4
```

Linkage: determining who is who in multi-file projects

```
// var.hh
namespace app {
    int variable{};
}
```

```
#include "var.hh"
void A()
{
    std::printf("%d\n",
      app::variable++);
}
```

```
#include "var.hh"
void B()
{
    std::printf("%d\n",
      app::variable++);
}
```

```
A():
    movl    app::variable(%rip), %esi
    movl    $.LC0, %edi
    leal    1(%rsi), %eax
    movl    %eax, app::variable(%rip)
    xorl    %eax, %eax
    jmp     printf
app::variable:
        .zero   4
```

```
B():
    movl    app::variable(%rip), %esi
    movl    $.LC0, %edi
    leal    1(%rsi), %eax
    movl    %eax, app::variable(%rip)
    xorl    %eax, %eax
    jmp     printf
app::variable:
        .zero   4
```

- Putting it in a header does not fix anything!

Linkage: determining who is who in multi-file projects

```
// var.hh
namespace app {
    int variable{};
}
```

```
#include "var.hh"
void A()
{
    std::printf("%d\n",
      app::variable++);
}
```

```
#include "var.hh"
void B()
{
    std::printf("%d\n",
      app::variable++);
}
```

```
A():
    movl    app::variable(%rip), %esi
    movl    $.LC0, %edi
    leal    1(%rsi), %eax
    movl    %eax, app::variable(%rip)
    xorl    %eax, %eax
    jmp     printf
app::variable:
        .zero   4
```

```
B():
    movl    app::variable(%rip), %esi
    movl    $.LC0, %edi
    leal    1(%rsi), %eax
    movl    %eax, app::variable(%rip)
    xorl    %eax, %eax
    jmp     printf
app::variable:
        .zero   4
```

- Putting it in a header does not fix anything!
- There are 4 types of "linkage" defined in C++, to address the issue of sameness of symbols across multiple translation units.

Linkage: determining who is who in multi-file projects

```
// var.hh
namespace app {
    int variable{};
}
```

```
#include "var.hh"
void A()
{
    std::printf("%d\n",
      app::variable++);
}
```

```
#include "var.hh"
void B()
{
    std::printf("%d\n",
      app::variable++);
}
```

```
A():
    movl    app::variable(%rip), %esi
    movl    $.LC0, %edi
    leal    1(%rsi), %eax
    movl    %eax, app::variable(%rip)
    xorl    %eax, %eax
    jmp     printf
app::variable:
        .zero   4
```

```
B():
    movl    app::variable(%rip), %esi
    movl    $.LC0, %edi
    leal    1(%rsi), %eax
    movl    %eax, app::variable(%rip)
    xorl    %eax, %eax
    jmp     printf
app::variable:
        .zero   4
```

- Putting it in a header does not fix anything!
- There are 4 types of "linkage" defined in C++, to address the issue of sameness of symbols across multiple translation units.
- Connected with a name, not with a file or project

Linkage type

- No-linkage: variables like block-scope entities can be safely excluded when trying to solve the problem of whether or not the same name appearing in different translation units refer to the same entity
- Internal linkage: the name is treated as private to each translation unit it appears in. Different occurences in different translation units are considered independent entities
- External linkage: Anywhere the name appears in all the translation units, it refers to the same entity. There needs to be a single definition of the object (ODR).
- Module linkage: The symbol is to be treated as the same entity everywhere it appears inside a module, but if it appears outside the module, it is another entity

External linkage

```cpp
// A.cc : A.o and B.o can't be linked together
namespace app {
    int variable{};
}
void A()
{
    std::printf("%d\n",
      app::variable++);
}
```

```cpp
// B.cc : A.o and B.o can't be linked together
namespace app {
    int variable{};
}
void A()
{
    std::printf("%d\n",
      app::variable++);
}
```

- "Top-level"/namespace scope (non-constant) variables and functions have external linkage by default

External linkage

```cpp
// A.cc : A.o and B.o can't be linked together
namespace app {
    int variable{};
}
void A()
{
    std::printf("%d\n",
      app::variable++);
}
```

```cpp
// B.cc : A.o and B.o can't be linked together
namespace app {
    int variable{};
}
void A()
{
    std::printf("%d\n",
      app::variable++);
}
```

- "Top-level"/namespace scope (non-constant) variables and functions have external linkage by default
- One and only one definition is permitted for such a name(ODR: **O**ne **D**efinition **R**ule)

External linkage

```cpp
// A.cc
namespace app {
    extern int variable{};
}
void A()
{
    std::printf("%d\n",
      app::variable++);
}
```

```cpp
// B.cc
namespace app {
    int variable{};
}
void A()
{
    std::printf("%d\n",
      app::variable++);
}
```

- "Top-level"/namespace scope (non-constant) variables and functions have external linkage by default
- One and only one definition is permitted for such a name(ODR: **O**ne **D**efinition **R**ule)
  - Variables: Make sure all translation units but one declare the symbol as `extern`, and the last one defines it without the `extern`. (`extern`: Somewhere out there there is an instance of this variable in memory. The linker will find it)

External linkage

```cpp
// A.cc
namespace app {
    extern int variable{};
}
void A()
{
    std::printf("%d\n",
      app::variable++);
}
```

```cpp
// B.cc
namespace app {
    int variable{};
}
void A()
{
    std::printf("%d\n",
      app::variable++);
}
```

- "Top-level"/namespace scope (non-constant) variables and functions have external linkage by default
- One and only one definition is permitted for such a name(ODR: **O**ne **D**efinition **R**ule)
  - Variables: Make sure all translation units but one declare the symbol as `extern`, and the last one defines it without the `extern`. (`extern`: Somewhere out there there is an instance of this variable in memory. The linker will find it)
  - Functions: Make sure all translation units except one have only access to only the function declaration, and only one translation unit contains the actual definition.

External linkage

```hh
// var.hh
namespace app {
    inline int variable{};
}
```

```cc
// A.cc
#include "var.hh"
void A()
{
    std::printf("%d\n",
      app::variable++);
}
```

```cc
// B.cc
#include "var.hh"
void B()
{
    std::printf("%d\n",
      app::variable++);
}
```

- "Top-level"/namespace scope (non-constant) variables and functions have external linkage by default
- One and only one definition is permitted for such a name(ODR: **O**ne **D**efinition **R**ule)
  - Variables: Make sure all translation units but one declare the symbol as `extern`, and the last one defines it without the `extern`. (`extern`: Somewhere out there there is an instance of this variable in memory. The linker will find it)
  - Functions: Make sure all translation units except one have only access to only the function declaration, and only one translation unit contains the actual definition.
- `inline`: Making a variable or function `inline` frees it from the one definition rule. Even if it has external linkage, the linker treats all instances of as the same entity

Internal linkage

```cpp
// var.hh
namespace app {
    const unsigned long max_dim = 1024UL;
    static auto f(int i) {
        // ...
    }
    static int variable{};
}
```

```cpp
// A.cc
#include "var.hh"
void A()
{
    std::printf("%d\n",
      app::variable++);
}
```

```cpp
// B.cc
#include "var.hh"
void B()
{
    std::printf("%d\n",
      app::variable++);
}
```

- "Top-level"/namespace scope constants have internal linkage by default in C++ (but not in C!)

Internal linkage

```cpp
// var.hh
namespace app {
    const unsigned long max_dim = 1024UL;
    static auto f(int i) {
        // ...
    }
    static int variable{};
}
```

```cpp
// A.cc
#include "var.hh"
void A()
{
    std::printf("%d\n",
      app::variable++);
}
```

```cpp
// B.cc
#include "var.hh"
void B()
{
    std::printf("%d\n",
      app::variable++);
}
```

- "Top-level"/namespace scope constants have internal linkage by default in C++ (but not in C!)
- A free function can be declared `static` give it internal linkage

Internal linkage

```cpp
// var.hh
namespace app {
    const unsigned long max_dim = 1024UL;
    static auto f(int i) {
        // ...
    }
    static int variable{};
}
```

```cpp
// A.cc
#include "var.hh"
void A()
{
    std::printf("%d\n",
      app::variable++);
}
```

```cpp
// B.cc
#include "var.hh"
void B()
{
    std::printf("%d\n",
      app::variable++);
}
```

- "Top-level"/namespace scope constants have internal linkage by default in C++ (but not in C!)
- A free function can be declared `static` give it internal linkage
- A non-constant namespace scope variable can be declared `static` to give it both a static storage duration and internal linkage

Internal linkage

```cpp
// var.hh
namespace app {
    const unsigned long max_dim = 1024UL;
    static auto f(int i) {
        // ...
    }
    static int variable{};
}
```

```cpp
// A.cc
#include "var.hh"
void A()
{
    std::printf("%d\n",
      app::variable++);
}
```

```cpp
// B.cc
#include "var.hh"
void B()
{
    std::printf("%d\n",
      app::variable++);
}
```

- "Top-level"/namespace scope constants have internal linkage by default in C++ (but not in C!)
- A free function can be declared `static` give it internal linkage
- A non-constant namespace scope variable can be declared `static` to give it both a static storage duration and internal linkage
- Since a variable or function with internal linkage can not be "required elsewhere", it is possible for the compiler to perform agressive optimizations, sometimes eliminating the symbol altogether from the translation unit

Module linkage

```
// A.cc
export module A;
import std;
namespace app {
    int variable{};
}
void other_func();
export void A()
{
    std::print("{}\n",
      app::variable++);
    other_func();
}
```

```
// A2.cc
module A;
void other_func()
{
    app::variable += 10;
}
```

```
export module B;
import std;
namespace app {
    int variable{};
}
export void B()
{
    std::print("{}\n",
      app::variable++);
}
```

- In between internal and external linkage: external as far as different module units inside the module are concerned, but visible only inside the module

Module linkage

```
// A.cc
export module A;
import std;
namespace app {
    int variable{};
}
void other_func();
export void A()
{
    std::print("{}\n",
      app::variable++);
    other_func();
}
```

```
// A2.cc
module A;
void other_func()
{
    app::variable += 10;
}
```

```
export module B;
import std;
namespace app {
    int variable{};
}
export void B()
{
    std::print("{}\n",
      app::variable++);
}
```

- In between internal and external linkage: external as far as different module units inside the module are concerned, but visible only inside the module
- Unexported names in the primary module interface unit are regarded as the same entity in all implementation units of the same module

Module linkage

```
// A.cc
export module A;
import std;
namespace app {
    int variable{};
}
void other_func();
export void A()
{
    std::print("{}\n",
      app::variable++);
    other_func();
}
```

```
// A2.cc
module A;
void other_func()
{
    app::variable += 10;
}
```

```
export module B;
import std;
namespace app {
    int variable{};
}
export void B()
{
    std::print("{}\n",
      app::variable++);
}
```

- In between internal and external linkage: external as far as different module units inside the module are concerned, but visible only inside the module
- Unexported names in the primary module interface unit are regarded as the same entity in all implementation units of the same module
- The exact same symbol may be used in a different module

Exercise 1.22:

The folder `examples/linkage` contains tiny demos for internal/external linkage (`intext`) and module linkage (`module`).

- `intext`: Build and run the program as is. Observe the output. Replace `inline` with **`static`** in the `intext/var.hh`. Build and run again. Reason about any differences in the output.
- **`module`**: Build and run using the approriate compiler options. Observe how the symbol `app::variable` is regarded the same across the different files of module `A`, but the same symbol is regarded as an independent object in module `B`

Argument Dependent Lookup

Argument Dependent Lookup

```
1  namespace Surrounding {
2      struct AClass {};
3      void one_func(int x, int y) {
4          std::cout << "Surrounding::one_func(int, int) << "\n";
5      }
6      void another_func(int x, AClass y) {
7          std::cout << "Calling Surrounding::another_func...";
8      }
9  }
10 // Elsewhere...
11 Surrounding::AClass obj; // OK
12 Surrounding::one_func(1, 2); // OK
13 one_func(1, 2); // Error!
14 Surrounding::another_func(1, obj); // OK
15 another_func(1, obj); // OK!
```

Argument Dependent Lookup

- If a function call expression involves one or more arguments of class types, the search for the matching function includes functions defined in the namespaces surrounding each of those classes. This is called "Argument Dependent Lookup", or Koenig Lookup
- The functions considered have to be in the immediately surrounding namespace around a class
- Calling such functions is very similar to calling our mental model of a member function, e.g., `norm(x)` instead of `x.norm()`
- **Recommendation:** Write more functions using class type objects in the surrounding namespace instead of making them members!

Argument Dependent Lookup

Exercise 1.23:

The notebook `ADL.ipynb` demonstrates argument dependent lookup. This is an important class related idea. Please go through the notebook and try out your own variations!

Exercise 1.24:

The folder `examples/ADL` contains a series of small programs demonstrating ADL, similar to those in the notebook above. Try them, in addition to the notebooks, as they explore the topic further.

Numeric types

Floating point numbers

Area of a triangle of sides a, b and c...

- Heron's formula (Metrica, Heron of Alexandria, $\approx$ 60 CE)

$$
\begin{aligned}
s &= \frac{a+b+c}{2} \\
\Delta &= \sqrt{s \times (s-a) \times (s-b) \times (s-c)}
\end{aligned}
$$

Floating point numbers

Area of a triangle of sides a, b and c...

- Heron's formula (Metrica, Heron of Alexandria, $\approx$ 60 CE)

$$\begin{aligned} s &= \frac{a+b+c}{2} \\ \Delta &= \sqrt{s \times (s-a) \times (s-b) \times (s-c)} \end{aligned}$$

- Kahan's formula (Miscalculating Area and Angles of a Needle-like Triangle, W. Kahan, 2000 CE, `http://http.cs.berkeley.edu/~wkahan/Triangle.pdf`)

$$\begin{aligned} a \geq b &\geq c \\ \Delta &= \frac{1}{4}\sqrt{(a+(b+c)) \times (c-(a-b)) \times (c+(a-b)) \times (a+(b-c))} \end{aligned}$$

Floating point numbers

```
1  const auto a = 5.0f;
2  const auto b = 4.0f;
3  const auto c = 3.0f;
4  std::cout << "Heron's formula = "
5        << area_heron(a,b,c) << "\n";
6  std::cout << "Kahan's formula = "
7        << area_kahan(a,b,c) << "\n";
```

```
Heron's formula = 6
Kahan's formula = 6
```

- Mathematically, both calculate the same thing

Floating point numbers

```
const auto a = 100'000.000'00f;
const auto b =  99'999.999'79f;
const auto c =       0.000'29f;
std::cout << "Heron's formula = "
      << area_heron(a,b,c) << "\n";
std::cout << "Kahan's formula = "
      << area_kahan(a,b,c) << "\n";
```

```
|
|
```

- Mathematically, both calculate the same thing
- If the triangle becomes very long and thin though, weird things happen

Floating point numbers

```
1  const auto a = 100'000.000'00f;
2  const auto b =  99'999.999'79f;
3  const auto c =       0.000'29f;
4  std::cout << "Heron's formula = "
5        << area_heron(a,b,c) << "\n";
6  std::cout << "Kahan's formula = "
7        << area_kahan(a,b,c) << "\n";
```

```
Heron's formula = 0
Kahan's formula = 14.5
```

- Mathematically, both calculate the same thing
- If the triangle becomes very long and thin though, weird things happen

Floating point numbers

```
1  const auto a = 100'000.000'00f;
2  const auto b =  99'999.999'79f;
3  const auto c =       0.000'29f;
4  std::cout << "Heron's formula = "
5        << area_heron(a,b,c) << "\n";
6  std::cout << "Kahan's formula = "
7        << area_kahan(a,b,c) << "\n";
```

```
Heron's formula = 0
Kahan's formula = 14.5
```

- Mathematically, both calculate the same thing
- If the triangle becomes very long and thin though, weird things happen
- Correct answer is 10.

Representation of floating point numbers

$$-1^{s} \times 1.mantissa \times 2^{exponent}$$

- It is enough to store the coloured parts. We win an extra bit of precision in the mantissa by skipping the 1 before the decimal point.

Representation of floating point numbers

$$-1^{s} \times 1.mantissa \times 2^{exponent}$$

- It is enough to store the coloured parts. We win an extra bit of precision in the mantissa by skipping the 1 before the decimal point.
- For a fixed exponent, there are 2^{23} different floating point numbers. $\Longrightarrow$ There are as many `float`s between 2^{-11} and 2^{-10} as there are between 1024 and 2048

Representation of floating point numbers

$$-1^{s} \times 1.mantissa \times 2^{exponent}$$

- It is enough to store the coloured parts. We win an extra bit of precision in the mantissa by skipping the 1 before the decimal point.
- For a fixed exponent, there are 2^{23} different floating point numbers. $\Longrightarrow$ There are as many `float`s between 2^{-11} and 2^{-10} as there are between 1024 and 2048
- By contrast, integral types have a uniform density throughout their range

Representation of floating point numbers

$$-1^{s} \times 1.mantissa \times 2^{exponent}$$

- Zero = all bits 0. One ?

Representation of floating point numbers

$$-1^{s} \times 1.mantissa \times 2^{exponent}$$

- Zero = all bits 0. One ?
- Exponent is stored *shift-127* encoded. So, $1 \equiv$ [0][01111111][00000000000000000000000]

Representation of floating point numbers

$$-1^{s} \times 1.mantissa \times 2^{exponent}$$

- Zero = all bits 0. One ?
- Exponent is stored *shift-127* encoded. So, $1 \equiv$ [0][01111111][00000000000000000000000]
- To maintain our sanity, we will write it as $1 \equiv$ [0][(2^0)][00000000000000000000000]

Floating point numbers

- Mental exercise: we have two decimal numbers in scientific notation 9.78×10^{2}, and 1.0×10^{-1}. How will you add them ?

Floating point numbers

- Mental exercise: we have two decimal numbers in scientific notation 9.78×10^2, and 1.0×10^{-1}. How will you add them ?
- You shift the decimal point in one of them until the exponents are the same, and then add the mantissas: $9.78 \times 10^2 + 0.001 \times 10^2$. Digits in the smaller number are pushed to the right

Floating point numbers

- $1 \equiv [0][(2^0)][00000000000000000000000]$

Floating point numbers

- $1 \equiv [0][(2^0)][00000000000000000000000]$
- What is the smallest representable n, with $n > 1$?

Floating point numbers

- $1 \equiv [0][(2^0)][00000000000000000000000]$
- What is the smallest representable n, with $n > 1$?
- $[0][(2^0)][00000000000000000000001]$ with the mantissa changing by $2^{-23} \approx 0.00000011920928955078125$

Floating point numbers

- $1 \equiv [0][(2^0)][00000000000000000000000]$
- What is the smallest representable n, with $n > 1$?
- $[0][(2^0)][00000000000000000000001]$ with the mantissa changing by $2^{-23} \approx 0.00000011920928955078125$
- What is 2.0 ? $[0][(2^1)][00000000000000000000000]$. What if you add these two ? What information about the smaller number can we retain ?

Floating point numbers

- What is the smallest representable n, with $n > 1$?
- [0][(2^0)][00000000000000000000001]. Mantissa changes by $2^{-23} \approx 0.000000119209289550781 3$.
- This quantity depends on the floating point type. In C++, you can retrieve it `std::numeric_limits<T>::epsilon()`

Floating point numbers

- What is the smallest representable n, with $n > 1$?
- [0][(2^0)][00000000000000000000001]. Mantissa changes by $2^{-23} \approx 0.000000119209289550781 3$.
- This quantity depends on the floating point type. In C++, you can retrieve it `std::numeric_limits<T>::epsilon()`
- Two quantities with exponent 0 can not be distinguished in this representation, if they differ by less than `epsilon`

Floating point numbers

- What is the smallest representable n, with $n > 1$?
- [0][(2^0)][00000000000000000000001]. Mantissa changes by $2^{-23} \approx 0.00000011920928955078125$.
- This quantity depends on the floating point type. In C++, you can retrieve it `std::numeric_limits<T>::epsilon()`
- Two quantities with exponent 0 can not be distinguished in this representation, if they differ by less than `epsilon`
- In an expression like `(big+small)-big`, if `big` and `small` differ by more than 23 in exponent, all information about `small` is lost, and we get a 0. $2^{23} = 8388608$.

Floating point numbers

- Floating point numbers with all bits in the exponent field at 0, are said to be “denormalised” (remember the shift-127 encoding)
- Not enough bits to represent such small quantities.
- All exponent bits being 1 indicate some special “numbers”:
  - $\pm\infty$: all mantissa bits 0.
  - NaN : at least one mantissa bit non-zero.

Exercise 1.25:

In `examples/floating_fun.cc`, there is a small program "simulating" a calculation involving some large quantities adding up to 0. Eight numbers are stored in an array of floats, and their sum evaluated and printed. The calculation is repeated by permuting the indexes of the array, so that the numbers are added in all possible orders. Observe the output!

Exercise 1.26: std::numeric_limits

What is epsilon for **`float`** and **`double`** on your computer ? Find out by writing a small C++ program and printing out the values from `std::numeric_limits`. Look up the documentation of `numeric_limits`. What other information can you get about numeric types from that header ?

Float: $[1-\text{bit}][8-\text{bits}][23-\text{bits}]$

Maximum	3.40282e+38
Minimum	1.17549e-38
Lowest	-3.40282e+38
Epsilon	1.19209e-07
Rounding error	0.5

Double: $[1-\text{bit}][11-\text{bits}][52-\text{bits}]$

Maximum	1.79769e+308
Minimum	2.22507e-308
Lowest	-1.79769e+308
Epsilon	2.22045e-16
Rounding error	0.5

New floating point types in C++23

Name	typeid	Min	Max	Epsilon
double	d	2.2250738585072014e-308	1.7976931348623157e+308	2.220446049250313e-16
std::float64_t	DF64	2.2250738585072014e-308	1.7976931348623157e+308	2.220446049250313e-16
float	f	1.1754944e-38	3.4028235e+38	1.1920929e-07
std::float32_t	DF32_	1.1754944e-38	3.4028235e+38	1.1920929e-07
std::float16_t	DF16_	6.1035156e-05	65504	0.0009765625
std::bfloat16_t	DF16b	1.1754944e-38	3.3895314e+38	0.0078125

- Two different 16 bit floating point numbers introduced
- `std::float64_t` and `std::float32_t` with different **`typeid`**s compared to built in **`double`** and **`float`**

Chapter 2

Cost of ...

Stack execution model

```
auto main() -> int
{
    auto N = 10;
    if (f(N) < g(N)) {
        h1(N);
    } else {
        h2(N);
    }
}
```

```
auto f(int i)() -> int
{
    return (i * i) %12;
}
auto g(int i)() -> int
{
    return i % 12;
}
auto h1(int i)() -> int
{
    return h11(i);
}
 auto h2(int i)() -> int
{
    return h21(i);
}
```

```
auto h11(int i)() -> int
{
    return i * i;
}
auto h21(int i)() -> int
{
    return i + h211(i);
}
```

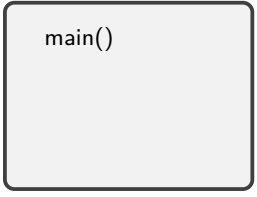
```
auto h211(int i)
    -> int
{
    return -i;
}
```

```
auto main() -> int
{
    auto N = 10;
    if (f(N) < g(N)) {
        h1(N);
    } else {
        h2(N);
    }
}
```

```
auto f(int i)() -> int
{
    return (i * i) %12;
}
auto g(int i)() -> int
{
    return i % 12;
}
auto h1(int i)() -> int
{
    return h11(i);
}
 auto h2(int i)() -> int
{
    return h21(i);
}
```

```
auto h11(int i)() -> int
{
    return i * i;
}
auto h21(int i)() -> int
{
    return i + h211(i);
}
```

```
auto h211(int i)
    -> int
{
    return -i;
}
```

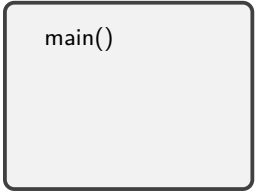


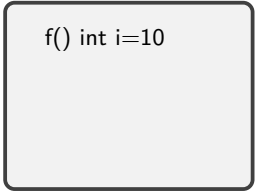
```
auto main() -> int
{
    auto N = 10;
    if (f(N) < g(N)) {
        h1(N);
    } else {
        h2(N);
    }
}
```

```
auto f(int i)() -> int
{
    return (i * i) %12;
}
auto g(int i)() -> int
{
    return i % 12;
}
auto h1(int i)() -> int
{
    return h11(i);
}
 auto h2(int i)() -> int
{
    return h21(i);
}
```

```
auto h11(int i)() -> int
{
    return i * i;
}
auto h21(int i)() -> int
{
    return i + h211(i);
}
```

```
auto h211(int i)
    -> int
{
    return -i;
}
```



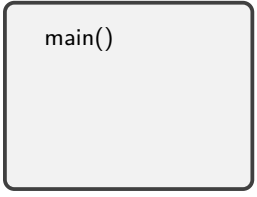


```
auto main() -> int
{
    auto N = 10;
    if (f(N) < g(N)) {
        h1(N);
    } else {
        h2(N);
    }
}
```

```
auto f(int i)() -> int
{
    return (i * i) %12;
}
auto g(int i)() -> int
{
    return i % 12;
}
auto h1(int i)() -> int
{
    return h11(i);
}
 auto h2(int i)() -> int
{
    return h21(i);
}
```

```
auto h11(int i)() -> int
{
    return i * i;
}
auto h21(int i)() -> int
{
    return i + h211(i);
}
```

```
auto h211(int i)
    -> int
{
    return -i;
}
```

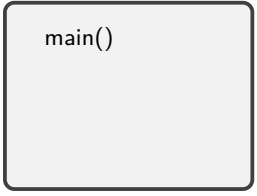


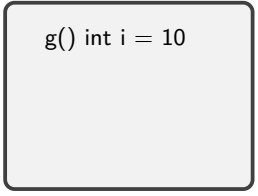
```
auto main() -> int
{
    auto N = 10;
    if (f(N) < g(N)) {
        h1(N);
    } else {
        h2(N);
    }
}
```

```
auto f(int i)() -> int
{
    return (i * i) %12;
}
auto g(int i)() -> int
{
    return i % 12;
}
auto h1(int i)() -> int
{
    return h11(i);
}
 auto h2(int i)() -> int
{
    return h21(i);
}
```

```
auto h11(int i)() -> int
{
    return i * i;
}
auto h21(int i)() -> int
{
    return i + h211(i);
}
```

```
auto h211(int i)
    -> int
{
    return -i;
}
```



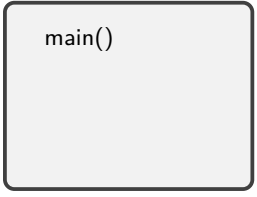


```
auto main() -> int
{
    auto N = 10;
    if (f(N) < g(N)) {
        h1(N);
    } else {
        h2(N);
    }
}
```

```
auto f(int i)() -> int
{
    return (i * i) %12;
}
auto g(int i)() -> int
{
    return i % 12;
}
auto h1(int i)() -> int
{
    return h11(i);
}
 auto h2(int i)() -> int
{
    return h21(i);
}
```

```
auto h11(int i)() -> int
{
    return i * i;
}
auto h21(int i)() -> int
{
    return i + h211(i);
}
```

```
auto h211(int i)
    -> int
{
    return -i;
}
```



```
auto main() -> int
{
    auto N = 10;
    if (f(N) < g(N)) {
        h1(N);
    } else {
        h2(N);
    }
}
```

```
auto f(int i)() -> int
{
    return (i * i) %12;
}
auto g(int i)() -> int
{
    return i % 12;
}
auto h1(int i)() -> int
{
    return h11(i);
}
 auto h2(int i)() -> int
{
    return h21(i);
}
```

```
auto h11(int i)() -> int
{
    return i * i;
}
auto h21(int i)() -> int
{
    return i + h211(i);
}
```

```
auto h211(int i)
    -> int
{
    return -i;
}
```

main()

h1() int i = 10

```
auto main() -> int
{
    auto N = 10;
    if (f(N) < g(N)) {
        h1(N);
    } else {
        h2(N);
    }
}
```

```
auto f(int i)() -> int
{
    return (i * i) %12;
}
auto g(int i)() -> int
{
    return i % 12;
}
auto h1(int i)() -> int
{
    return h11(i);
}
 auto h2(int i)() -> int
{
    return h21(i);
}
```

```
auto h11(int i)() -> int
{
    return i * i;
}
auto h21(int i)() -> int
{
    return i + h211(i);
}
```

```
auto h211(int i)
    -> int
{
    return -i;
}
```

main()

h1() int i = 10

h11() int i = 10

```
auto main() -> int
{
    auto N = 10;
    if (f(N) < g(N)) {
        h1(N);
    } else {
        h2(N);
    }
}
```

```
auto f(int i)() -> int
{
    return (i * i) %12;
}
auto g(int i)() -> int
{
    return i % 12;
}
auto h1(int i)() -> int
{
    return h11(i);
}
 auto h2(int i)() -> int
{
    return h21(i);
}
```

```
auto h11(int i)() -> int
{
    return i * i;
}
auto h21(int i)() -> int
{
    return i + h211(i);
}
```

```
auto h211(int i)
    -> int
{
    return -i;
}
```

main()

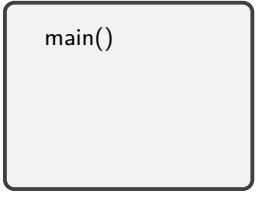
h1() int i = 10

```
auto main() -> int
{
    auto N = 10;
    if (f(N) < g(N)) {
        h1(N);
    } else {
        h2(N);
    }
}
```

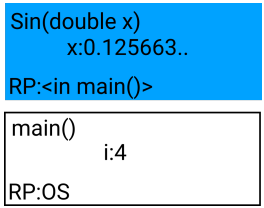
```
auto f(int i)() -> int
{
    return (i * i) %12;
}
auto g(int i)() -> int
{
    return i % 12;
}
auto h1(int i)() -> int
{
    return h11(i);
}
 auto h2(int i)() -> int
{
    return h21(i);
}
```

```
auto h11(int i)() -> int
{
    return i * i;
}
auto h21(int i)() -> int
{
    return i + h211(i);
}
```

```
auto h211(int i)
    -> int
{
    return -i;
}
```



Functions at run time




```
1  auto sin(double x) -> int {
2    // Somehow calculate sin of x
3    return answer;
4  }
5  auto main() -> int {
6      double x{3.141592653589793};
7      for (int i = 0; i < 100; ++i) {
8          std::cout << i * x / 100
9              << sin(i * x / 100) <<"\n";
10     }
11 }
```

When a function is called, e.g., when we write `f(value1,value2,value3)` for a function `f` declared as `ret_type f(type1 x, type2 y, type3 z)`:

- A "workbook" in memory called a stack frame is created for the call
- The local variables `x`, `y`, `z` are created, as if using instructions `type1 x{value1}`, `type2 y{value2}`, `type3 z{value3}`.
- A return address is stored.
- The actual body of the function is executed
- When the function concludes, execution continues at the stored return address, and the stack frame is destroyed
- Memory used for the stack frame is usually cached and can be accessed quickly

Member functions

```
class D {
    int nm;
    double d;
public:
    void val(double x) { d = x; }
    auto val() const -> double { return d; }
    auto name() const { return nm; }
    auto operator+(double x1) const -> double;
};
```

```
auto D::operator+(double x) const -> double
{
    return d + x * x;
}
```

```
0000000000000000 <_ZNK1DplEd>:
  vmulsd xmm0,xmm0,xmm0
  vaddsd xmm0,xmm0,QWORD PTR [rdi+0x8]
  ret
```

- Object of class types are passed using their addresses. The compiler uses the `address` of the class type variable and `offsets` to its parts to find the appropriate values to use.
- Return value is written to the type appropriate registers, e.g., `xmm0`, `eax`...
- Execution continues at the previously stored return address

Aside: reading assembly code

The compiler explorer

Exercise 2.1:

The compiler explorer https://godbolt.org provides a great tool to quickly examine the assembly code corresponding to a code snippet. It is possible to choose different compilers, give compiler options ... Use it to quickly check the assembly code generated for simple functions. Compare different compilers. Try the examples in `examples/assembly`. Vary the compiler and compiler options and see how the assembler changes.

- `class.cc` contains two functions doing the same thing. One operates one a bare **`double`** variable, and another on a **`double`** variable wrapped in a class with simple accessor functions. How different are the generated assembler code for the two functions ?
- `axpy.cc` shows an example of a simple **`struct`** with an internal array (presumably of some numeric type). Notice how separate numeric operations, written over elements of those arrays become *fused multiply-add* or vector `fma` operations, when compiled with more recent compilers. What happens when the compile-time fixed length array has a size 32 or 64 instead of 16? Compare also with the assembly from older compilers!

See also: CppCon 2016: Serge Guelton "C++ Costless Abstractions: the compiler view"

Stack

```
1  class V3 {
2      double x{}, y{}, z{};
3      auto cross(const V3 &) -> V3;
4      auto dot(const V3 &) -> double;
5  };
6  auto prob(int i, const V3& x, const V3& y)
7      -> double
8  {
9      int j = i % 233;
10     V3 tmp{x};
11     for (; j < i; ++j) {
12         tmp = tmp.cross(y);
13     }
14     return tmp.dot(x);
15 }
```

- Heavily reused memory locations
- Likely cached, therefore, fast
- All local (block scope) variables of any type, which come into existence inside a block, and expire at the end of the block, i.e., with *automatically managed* lifetime.

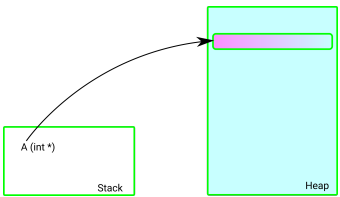
Global storage

```
1  auto prob(int i) -> double
2  {
3      static int c{0};
4      ++c;
5      if (c % 1000==0) {
6          std::cout << "Call count reached "
7            << c << "\n";
8      }
9      static const double L[] = {3.14, 2.71};
10     return L[i % 2];
11 }
```

- Variables outside any function
- Variables marked with the `static` keyword in functions
- Floating point constants, array initializer lists, jump tables, virtual function tables

Heap

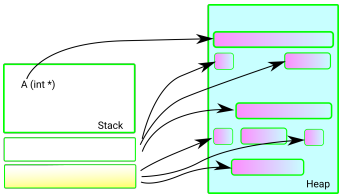
```
1  void f()
2  {
3      int *A = new int[1000000];
4      // calculations with A
5      delete [] A;
6  }
```



- Directly/indirectly managed memory through **new**, **delete**, `malloc` or `free`
- Best practice: managed by container types like `vector`, `list` etc. or by smart pointers `unique_ptr` or `shared_ptr`
- Objects who come into existence with a **new** call, and live until an explicit **delete** call
- Can store very large objects which don't fit in the stack
- Arrays whose size is not known at compile time. C99 style variable length arrays are not standard C++.

Heap

```
1  void f()
2  {
3      int *A = new int[1000000];
4      // calculations with A
5      // What if we throw an exception here
6      // and never actually reach the delete?
7      delete [] A;
8  }
```



- Must remember to free memory before all pointers pointing to that heap block go out of scope. Those pointers may expire either because the program successfully runs past the `}` marking the end of their lifetime, or leaves the scope by throwing an exception. $\Longrightarrow$ RAII: tie the acquiring and releasing of resources to the life time of a suitable object.
- Tends to get fragmented
- Must find a suitably sized unused block, and must keep track of what is and isn't in use $\longmapsto$ allocation and deallocation are expensive
- Objects stored one after the other may end up in very different locations
- Slower than stack storage

Exercise 2.2:

In HPC, we have to carefully monitor our heap allocation/deallocation operations. In the program `examples/alloc_cost.cc`, we compare two nearly identical functions, where the only difference is the use of a heap allocated array as the returned value. We clearly see that the version without the heap allocation runs faster. Reducing the allocation/deallocation operations inside hot code sections improves performance.

```cpp
static void StringCreation(benchmark::State& state) {
  for (auto _ : state) {
    std::string created_string("hello");
    benchmark::DoNotOptimize(created_string);
  }
}
BENCHMARK(StringCreation);
static void StringCopy(benchmark::State& state) {
  std::string x = "hello";
  for (auto _ : state) {
    std::string copy(x);
  }
}
BENCHMARK(StringCopy);
```

Exercise 2.3:

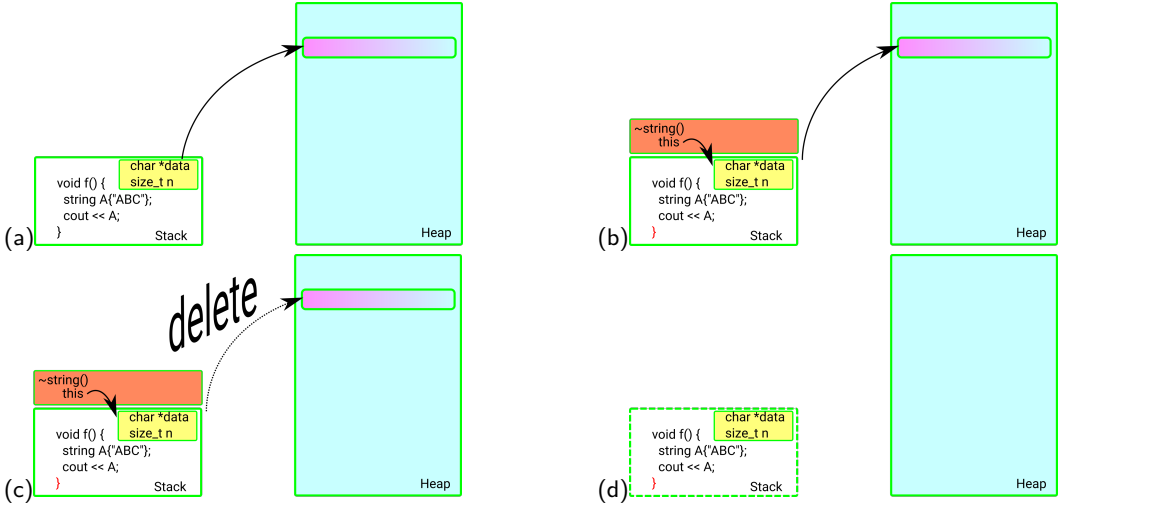
Test the cost of allocation and deallocation using the microbenchmarking site quick-bench.com! Their default example is the code given here (above), comparing string creation and copy. Note down the timings. Then add about 20 'o's at the end of the "hello" in each bench mark, i.e., "hello" $\longmapsto$ "hellooooooooooooooooooooo". Compare the timings again! Reduce the number of o's until the timings are as in the original form. Do you understand the timings?

Resource handles

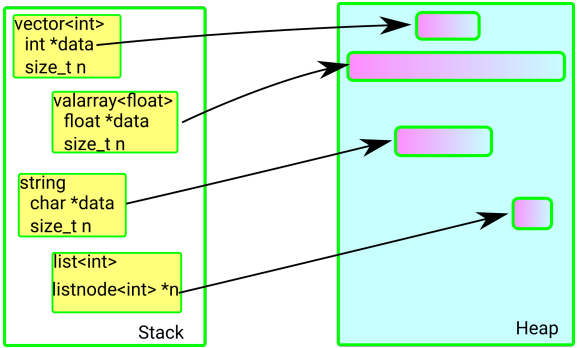
- Instead of bare heap allocation/deallocation, allocate in constructors or member functions (a)
- When the scope of the variable ends, the destructor is automatically called (b)
- Destructor should free any resources still in use (c)
- The variable can now expire (d)

The labels (a), (b), (c) and (d) refer to the figures in the following slide.

Resource handles



Resource handles



- STL containers (except `std::array`) are "resource" handles
- Memory management is done through constructors, the destructor and member functions

- No legitimate use of objects of the class should result in a memory leak
- Most data is on the heap. The objects on the stack are light-weight handles.

Resource handles

```
1  vector<int> A(32,0);
2  vector<double> B(64,0.);
3  vector<complex<double>> C(128);
4  vector<bool> D(256);
5  cout << sizeof(A) << ", "
6       << sizeof(B) << ", "
7       << sizeof(C) << ", "
8       << sizeof(D) << "\n";
```

Quiz

What will the program print ?

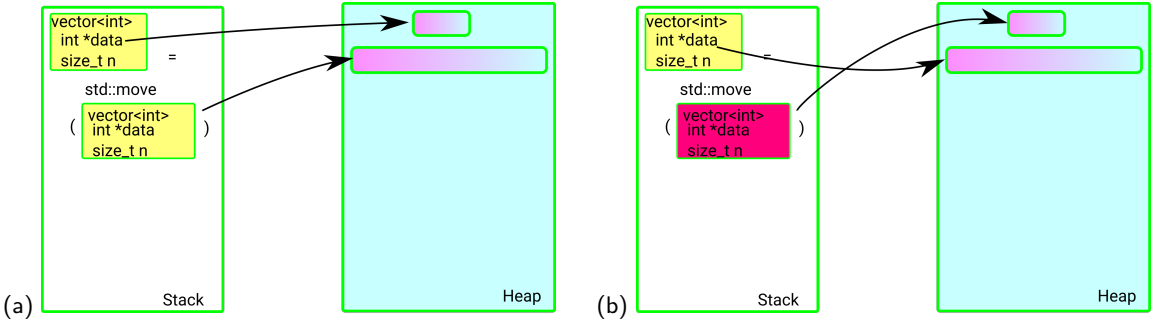
A. 32, 64, 128, 256

B. 32, 64, 256, 64

C. 24, 24, 24, 24

D. 24, 24, 24, *depends on the library*

Resource handles

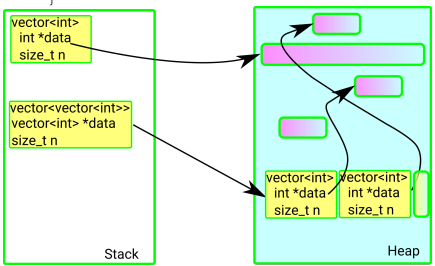


Move

- Can transfer ownership of the resources very cheaply
- Actual data on the heap need not be touched at all!
- Just some pointer re-assignments on the stack (a), (b)

Resource handles

```
vector<vector<int>> v(10, vector<int>(10,0));
...
for (int i = 0; i < 10; ++i) {
    for (int j = 0; j < 10; ++j) {
        v[i][j] = i + j;
        //v.operator[](i).operator[](j);
        //(*(*(v.dat + i)).dat + j)
    }
}
```



- In C++, objects (instances of a class) can live on the stack or on the heap
- Putting resource handles like `vector<int>` on the heap, while allowed, incurs the cost of additional indirections
- It is almost always possible to avoid cumbersome beasts like `vector<vector<int>>`, `vector<vector<vector<vector<int>>>>` or `int*****`.
- I wish I hadn't seen such "multi-dimensional arrays" in production code!

If you need your own 2D, 5D etc. arrays, ...

```
template <class T> class array2d {
    vector<T> v;
    size_t nc{0}, nr{0};
public:
    auto operator()(size_t i, size_t j) const
        -> const T& { return v[i * nc + j]; }
    auto operator()(size_t i, size_t j)
        -> T& { return v[i * nc + j]; }
};
```

- Use a wrapper class around an STL container, like `vector` or `valarray`
- Either overload the `operator()` to access a given row and column ...

Exercise 2.4:

`examples/array2d` contains the class template shown here.

If you need your own 2D, 5D etc. arrays, ...

```
1  template <class T> class array2d {
2      std::vector<T> v;
3      size_t nc{ 0 }, nr{ 0 };
4  public:
5      template <class Self> auto&& operator[](this Self&& self, size_t i, size_t j) {
6          auto&& a = std::forward<Self>(self);
7          return a.v[i * a.nc + j];
8      }
9  };
```

- Use a wrapper class around an STL container, like `vector` or `valarray`
- Either overload the `operator()` to access a given row and column ...
- ...or use C++23 and overload `operator[]` with two indexes, and deducing this...

Exercise 2.5:

`examples/array2d` contains the class template shown here.

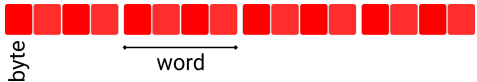
std::array

```cpp
// examples/stdarray.cc
#include <iostream>
#include <array>

auto main() -> int
{
    std::array A{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9};
    std::cout << "Size of array on stack = " << sizeof(A) << "\n";
    std::cout << "size() = " << A.size() << "\n";
}
```

- Resembles other STL containers, but this is not just a handle.
- Does not need a data element to store the size, as the size is "part of the name" of the type!
- Moving an `std::array` has order N complexity, as each individual element needs to be moved. No pointer swapping trick can do the job for this.

Data alignment



- Data is read or written with a unit size called word. On the most common architectures, word size is 4 or 8 bytes.
- Data alignment means, putting data on memory addresses which are integral multiples of the word size
- n-byte aligned address has $>= log_2(n)$ least significant zeros
- Access for aligned data is fast
- If the size of a primitive type does not exceed the word size, access to aligned data of that type is also atomic

Data alignment

- The X86 architecture is tolerant of misaligned data. Programs run, even if they can't use SSE features
- PowerPC throws a hardware exception, which may be handled by the OS. For unaligned 8 byte access, a 4,610% performance penalty has been discussed (`http://www.ibm.com/developerworks/library/pa-dalign/`)
- On other systems, crashes, data corruption, incorrect results are all possibilities

Data alignment

- Usually, primitive types are aligned by their "natural alignment": 4 byte `int` has 4 byte alignment, 8 byte double has alignment of 8 and so on
- A class has a natural alignment equal to the strictest requirement of its members
- The `alignof` operator can be used to query the alignment of a type
- The `alignas` keyword can be used to set a stricter alignment requirement

Exercise 2.6:

Verify the above using the example program `examples/alignof.cc`.

Data structure padding

```
class D { // alignment : 8, because of d
    int nm; // alignment requirement 4.
    double d; // Must have alignment 8.
public:
    void val(double x) { d=x; }
    auto val() const -> double { return d; }
    auto operator+(double x1) const -> double;
auto D::operator+(double x) const -> double
{
    return d + x * x;
D::operator+(double) const:
    vfmadd213sd     xmm0, xmm0, QWORD PTR [rdi+8]
    ret
```

- Alignment requirement of members can necessitate introduction of padding between members

Data structure padding

```
class D { // alignment : 8, because of d
    int nm; // alignment requirement 4.
    double d; // Must have alignment 8.
public:
    void val(double x) { d=x; }
    auto val() const -> double { return d; }
    auto operator+(double x1) const -> double;
auto D::operator+(double x) const -> double
{
    return d + x * x;
D::operator+(double) const:
    vfmadd213sd     xmm0, xmm0, QWORD PTR [rdi+8]
    ret
```

- Alignment requirement of members can necessitate introduction of padding between members
- What happens to the assembler here, if we put a comma between n and m in the name `nm` in class `D`?

Data structure padding

```
class D { // alignment : 8, because of d
    int nm; // alignment requirement 4.
    double d; // Must have alignment 8.
public:
    void val(double x) { d=x; }
    auto val() const -> double { return d; }
    auto operator+(double x1) const -> double;
auto D::operator+(double x) const -> double
{
    return d + x * x;
D::operator+(double) const:
    vfmadd213sd     xmm0, xmm0, QWORD PTR [rdi+8]
    ret
```

- Alignment requirement of members can necessitate introduction of padding between members
- What happens to the assembler here, if we put a comma between n and m in the name `nm` in class `D`?
- What if we make it `int n, m, p;`? Test it using the compiler explorer! Click on the link or copy and paste code from `examples/assembly/class2.cc`.

Data structure padding

```
1  class A {
2      char c;
3      double x;
4      int d;
5  };
6  // Compiled as if it was ...
7  char c;
8  char pad[7];
9  double x;
10 int d;
11 char pad2[4]; // why is this here ?
12 // Overall alignment alignof(double)
13 // size of struct = 24
```

```
1  class B {
2      double x;
3      int d;
4      char c;
5  };
6  // Compiled as if it was ...
7  double x;
8  int d;
9  char c;
10 char pad[3];
11 // Overall alignment alignof(double)
12 // size of struct = 16
```

- Due to padding, size of structures can be bigger than the sum of sizes of their elements
- C++ rules do not allow the compiler to reorder elements for space
- Carefully choosing the declaration order of class members can save memory

Alignment specifiers

```
1  alignas(64) double x[4]; // ok
2
3  alignas(64) vector<double> a(4);
4  // Pointless.
5  // The above simply aligns the resource
6  // handle, not the data on the heap
7
8  alignas(64) array<double, 4> A;
9  // This is fine, as std::array has
10 // real data in its struct
11
12 template <class T, int vecsize>
13 struct alignas(vecsize) simd_t
14 {
15     array<T,vecsize/sizeof(T)> data;
16 };
17 // We have requested that all objects
18 // of type simd_t should be aligned
19 // to vecsize bytes.
```

- The `alignas` keyword can specify alignment for variables
- Can be attached to a class declaration so that all objects of that type have a specified alignment
- It is possible to attach an extended alignment specifier to the class declaration
- Be mindful about what you are aligning when you use `alignas` for a resource handle like `vector`

```
1  alignas(64) std::vector U(100UL, 3.14);
2  // Align the vector object on the stack
3  // The array managed by the vector is
4  // not aligned
5  std::vector<double,
6      tbb::cache_aligned_allocator<double>>
7          A(100UL, 3.14);
8  // Cache aligned data array on the heap
```

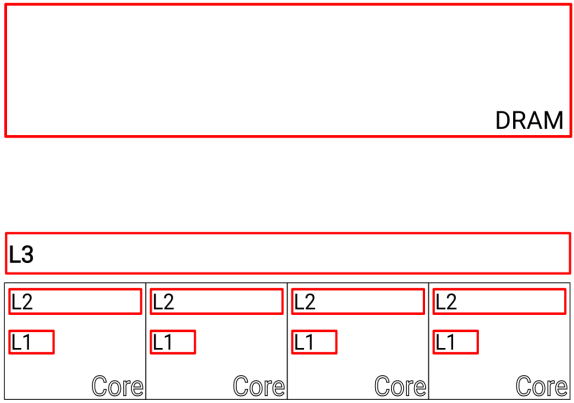
Exercise 2.7:

The `examples/align0.cc` has an example class template, which creates a data array of the right size to fill the simd vector width irrespective of the input data type. It illustrates the use of **`alignof`** and **`alignas`**, and variable templates.

Exercise 2.8:

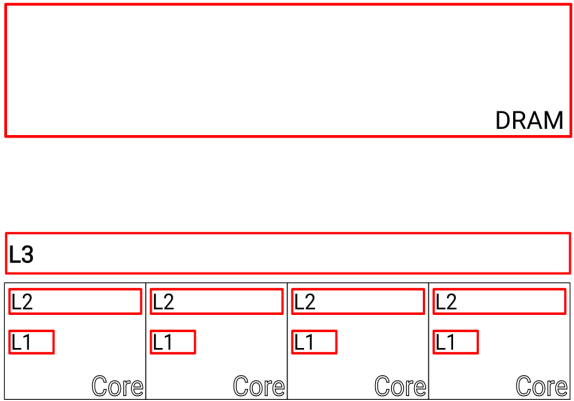
The `examples/align1.cc` shows that the usual mechanisms of dynamic allocation up to C++14 do not provide any guarantees about alignment greater than the natural alignment of the type. The behaviour changed in C++17 for types with explicitly specified extended alignment specifier like our `simd_t` class of the previous example. Finally, `examples/align2.cc` shows the use of a new version of the **`new`** operator introduced in C++17, which accepts an alignment argument.

Memory



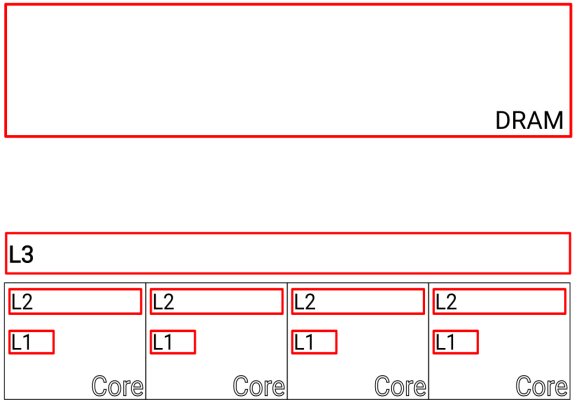
- Fun fact: in 1 clock cycle of the CPU on my laptop, a photon travels about 10 cm in vacuum!

Memory



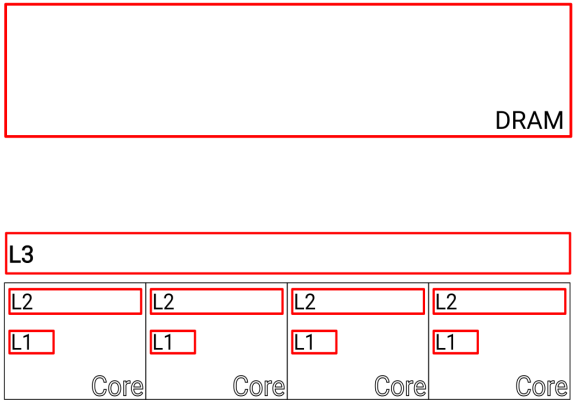
- Fun fact: in 1 clock cycle of the CPU on my laptop, a photon travels about 10 cm in vacuum!
- CPUs contain a certain amount of “cache” memory, which is faster to access, but much smaller than RAM
- Cost of fetching one integer from the main memory can be a hundred times larger than getting it from the L1 cache
- When the CPU looks for data from one address in memory, it is copied from RAM to the cache and then used.

Memory



- Fun fact: in 1 clock cycle of the CPU on my laptop, a photon travels about 10 cm in vacuum!
- CPUs contain a certain amount of “cache” memory, which is faster to access, but much smaller than RAM
- Cost of fetching one integer from the main memory can be a hundred times larger than getting it from the L1 cache
- When the CPU looks for data from one address in memory, it is copied from RAM to the cache and then used.

Memory



- Fun fact: in 1 clock cycle of the CPU on my laptop, a photon travels about 10 cm in vacuum!
- CPUs contain a certain amount of "cache" memory, which is faster to access, but much smaller than RAM
- Cost of fetching one integer from the main memory can be a hundred times larger than getting it from the L1 cache
- When the CPU looks for data from one address in memory, it is copied from RAM to the cache and then used.
- If it is immediately accessed again, it is in the cache, and can be used without the cost of fetching it from RAM
- Memory is fetched in "cache lines". Successive operations on contiguous memory locations do not incur the full cost of main memory access

Finding out cache information about your CPU

```
root > dmidecode -t cache
# dmidecode 3.1
Getting SMBIOS data from sysfs.
SMBIOS 3.0.0 present.
Handle 0x0007, DMI type 7, 19 bytes
Cache Information
        Socket Designation: L1 Cache
        Configuration: Enabled, Not Socketed, Level 1
        Operational Mode: Write Back
        Location: Internal
        Installed Size: 256 kB
        Maximum Size: 256 kB
        Supported SRAM Types:
                Synchronous
        Installed SRAM Type: Synchronous
        Speed: Unknown
        Error Correction Type: Parity
        System Type: Unified
        Associativity: 8-way Set-associative}

Handle 0x0008, DMI type 7, 19 bytes
Cache Information
        Socket Designation: L2 Cache
        Configuration: Enabled, Not Socketed, Level 2
        Operational Mode: Write Back
        Location: Internal
        Installed Size: 1024 kB
        Maximum Size: 1024 kB

        Supported SRAM Types:
                Synchronous
        Installed SRAM Type: Synchronous
        Speed: Unknown
        Error Correction Type: Single-bit ECC
        System Type: Unified
        Associativity: 4-way Set-associative
Handle 0x0009, DMI type 7, 19 bytes
Cache Information
        Socket Designation: L3 Cache
        Configuration: Enabled, Not Socketed, Level 3
        Operational Mode: Write Back
        Location: Internal
        Installed Size: 8192 kB
        Maximum Size: 8192 kB
        Supported SRAM Types:
                Synchronous
        Installed SRAM Type: Synchronous
        Speed: Unknown
        Error Correction Type: Multi-bit ECC
        System Type: Unified
        Associativity: 16-way Set-associative
```

CPU cache

```
not_root> getconf -a | grep -i cache
LEVEL1_ICACHE_SIZE                 32768
LEVEL1_ICACHE_ASSOC                8
LEVEL1_ICACHE_LINESIZE             64
LEVEL1_DCACHE_SIZE                 32768
LEVEL1_DCACHE_ASSOC                8
LEVEL1_DCACHE_LINESIZE             64
LEVEL2_CACHE_SIZE                  262144
LEVEL2_CACHE_ASSOC                 4
LEVEL2_CACHE_LINESIZE              64
LEVEL3_CACHE_SIZE                  8388608
LEVEL3_CACHE_ASSOC                 16
LEVEL3_CACHE_LINESIZE              64
LEVEL4_CACHE_SIZE                  0
LEVEL4_CACHE_ASSOC                 0
LEVEL4_CACHE_LINESIZE              0
```

- Tools : `lscpu`, `dmidecode`, `lshw`, `getconf`
- Different tools may aggregate information differently (e.g., how total as opposed to per-core cache is reported)
- L1d cache is for data, L1i is for instructions (instructions must live somewhere in the cache too!)

CPU cache

```
.... .... .... .... .... .... .... ....  .... .... .... .... .... SSSS SS11 1111
```

- For a 64 byte cache line, the least 6 bits of the address refer to the location inside the cache line. Not relevant in determining parking spot in the cache
- If we have 32kb of L1d cache, with a 64 byte line, we have 512 "parking spots" (lines)
- An 8 way associative cache will then have $512/8 = 64$ sets, and use the further 6 bits of a memory address to assign a set
- If we keep accessing random places in memory, it is very easy to run out of L1 cache: in the example here, we have only 64 sets!
- Address bits higher than the least 12 are not used in determining where in the cache a value is stored: any two addresses separated by 2^{12} map to the same set in the L1 cache.
- Variables with memory addresses separated by $setcount \times linesize$ compete for the same cache line
- For better performance, one should strive to write code utilising the whole cache line before it is evicted

Memory access patterns

```
1    std::vector<int> A(N * N, 0);
2    for (size_t i = 0; i < N; ++i) {
3        for (size_t j = 0; j < N; ++j) {
4            A[i * N + j] += j + i;
5        }
6    }
```

```
1    for (size_t i = 0; i < N; ++i) {
2        for (size_t j = 0; j < N; ++j) {
3            A[j * N + i] += j + i;
4        }
5    }
```

```
1    for (size_t i = 0; i < N * N; ++i) {
2        A[ pos[i] ] += i;
3    }
```

See also: CppCon 2016: Timur Doumler "Want fast C++? Know your hardware!"

- Q: Which way of accessing the "matrix" is faster, and by how much ?
- A: For N=10000, my laptop takes about 0.037 seconds for the row major pattern (top), and about 0.26 seconds for the column major pattern (middle), and 1.86 seconds for random pattern (bottom)

Memory

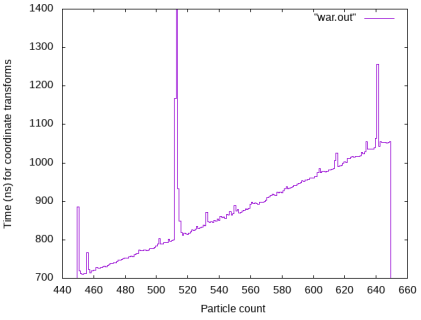
```
1  constexpr size_t size = 2 << 26;
2  std::vector< long > A(size, 0);
3  for (size_t step = 1; step <= 2048; step *= 2) {
4    for (size_t i = 0; i < size; i += step) A[i]++;
5  }
```

Step	Time
1	0.0967211
2	0.0943902
4	0.0929546
8	0.113927
16	0.137341
32	0.120449
64	0.0675447
128	0.0415029
256	0.016718
512	0.00694461
1024	0.00357155
2048	0.00178591

- For small step sizes, increasing the number of writes to the array does not change the total time.
- Multiple accesses inside a cache line has minimal extra cost.

4K aliasing

```
1   // Layout :
2   // x0, x1, x2 ... xn-1, y0, y1 ... yn-1,
3   // z0, z1 ... zn-1, wx0, wx1 ... wxn-1,
4   // wy0, wy1 ... wyn-1, wz0 ... wzn-1
5
6   for (size_t i=0;i<npart;++i) {
7     wx(i)=R(0,0)*x(i)+R(0,1)*y(i)+R(0,2)*z(i));
8     wy(i)=R(1,0)*x(i)+R(1,1)*y(i)+R(1,2)*z(i));
9     wz(i)=R(2,0)*x(i)+R(2,1)*y(i)+R(2,2)*z(i));
10  }
```



- Innocent looking code can sometimes produce weird changes in performance based on array sizes
- The spike in required time here comes for a particle count of about 512, when the different components of the data for one particle are separated by exactly 4kB.

Exercise 2.9: Memory effects

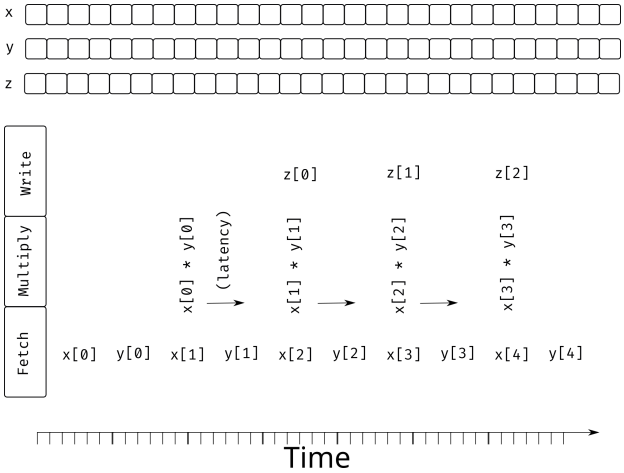
The following examples illustrate the cache effects discussed so far:

- `traverse0.cc` can be used to compare contiguous and non-contiguous access of a large array
- `every_nth.cc` compares times for accessing every n'th element, and highlights the cache line
- `transpose.cc` transpose operation on a matrix, which involves lots of non-contiguous access

Recommendations

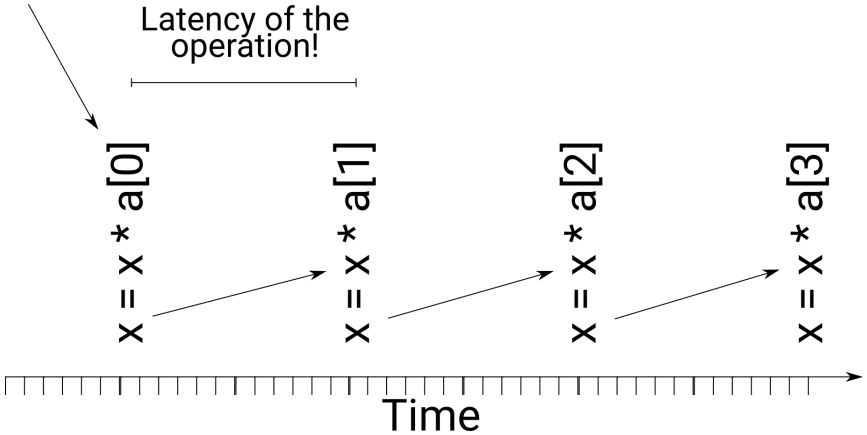
- Prefer `std::array` and `std::vector` for all your container needs as a default. Many libraries also provide other containers with contiguous storage providing advantages for specific use cases. Anything with non-contiguous storage needs to be carefully justified
- Organise code to maximise the use of any cache line that has been fetched:
  - Collate processing of nearby memory locations
  - Organise data structures so that things processed together are also stored near each other
- Keep variables as local as possible

Instruction pipeline

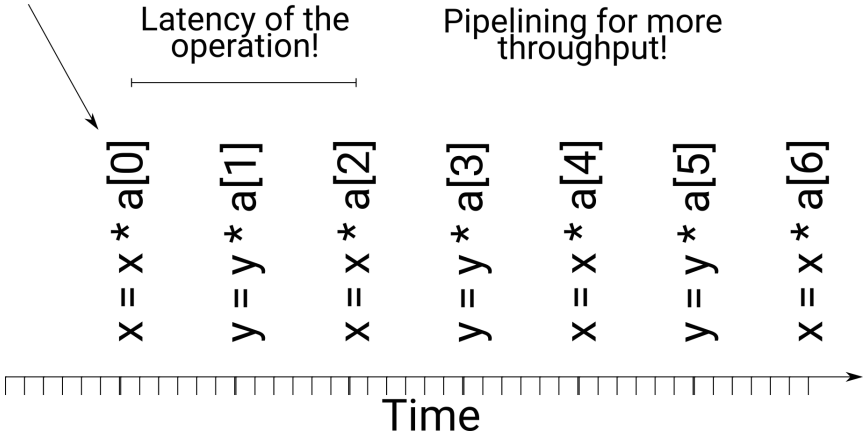


- A processor consists of many units, responsible for different actions, e.g., fetching instructions or data from memory, arithmetics, writing computed results back to memory
- When executing a program, pipelining helps keep different units busy throughout, improving throughput

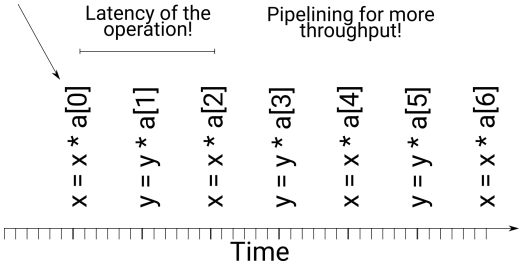
Instruction pipeline



Instruction pipeline



Instruction pipeline



- CPUs do not have to wait until one calculation is totally finished before starting another
- They "pipeline". If the data required for another instruction is available, that can start execution before the first is finished
- Because of pipelining, the processors are able to perform more operations in time Δt than $\frac{\Delta t}{t_{latency}}$
- Data dependencies create stalls in the pipeline
- Some modern processors even execute instructions "out of order", to keep the pipeline busy

Exercise 2.10:

The program `examples/ilp.cc` demonstrates the effects of data dependencies. Two alternative versions of a loop are given, performing the same total number of computations. One of them runs more than 5 times faster, because it avoids dependencies between successive calculations.

Pipeline and branched code

- Program execution flows through different units responsible for different work
- Branching can introduce holes in the pipeline without some workaround

- Instruction fetch
- Instruction decode
- Instruction execute
- Memory access
- Register write back

```
1  if (x+y>5) f();
2  else g();
```

- request mem x
- request mem y
- calc x+y
- calc res $>$ 5
- ?

The "next instruction" depends on the outcome of an instruction.

Branch prediction

```
1   for (int i = 0; i < N; ++i) {
2       if (p[i] > gen()) {
3           b[i] = a[i] + c[i];
4           ++fwd;
5       } else {
6           a[i] = b[i] + c[i];
7           ++rev;
8       }
9   }
10  nngb = 0;
11  while (a) {
12      dist[nngb++] = distf(a,i);
13  }
```

- For efficient execution, different units in the pipeline must be kept busy as much as possible

- When branches are encountered, the CPU simply guesses which way it will go, and fetches instructions accordingly
- If the guess is right, no pipeline stall
- If it is wrong, all operations done with that guess must be purged

Branch mis-prediction penalty

```
1   for (int i = 0; i < N; ++i) {
2       if (p[i] > gen()) {
3           a[i] = (b[i] > r0 && b[i] < r1
4                               && c[i] < b[i]);
5       } else {
6           a[i] = b[i] + c[i];
7           ++rev;
8       }
9   }
10  nngb=0;
11  while (a) {
12      dist[nngb++] = distf(a,i);
13  }
```

- If statements, switches, loops contain obvious branches
- The ternary operator `a = cond ? v1 : v2` is (sometimes) a branch

- Not so obvious branches include boolean `||` and `&&` operators:
  - In a sequence of operations like `a || b || c || ...`, the operands are evaluated left to right until the first true value is obtained
  - In a sequence of operations like `a && b && c && ...`, the operands are evaluated left to right until the first false value is obtained

Not branches

```
1  auto f(int i) -> int
2  {
3      static const int a[4]={4,3,2,1};
4      int ans=0;
5      ans += (a[1]<i)?1:2;
6      return ans;
7  }
```

```
1  0000000000000000 <_Z1fi>:
2    cmp    edi,0x4
3    setl   al
4    movzx  eax,al
5    inc    eax
6    ret
```

- Conditional assignments are often reorganised as simple sequential instructions by compilers using special assembler instructions when available
- Loops with small loop counts may be automatically unrolled at compile time leaving simple linear code

```
1  0000000000000000 <_Z1fdPd>:
2    subsd  xmm0,QWORD PTR [rdi]
3    subsd  xmm0,QWORD PTR [rdi+0x8]
4    subsd  xmm0,QWORD PTR [rdi+0x10]
5    subsd  xmm0,QWORD PTR [rdi+0x18]
6    ret
```

Not branches

```
1  auto f(double x, double A[4]) -> double
2  {
3      double a=x;
4      for (int i=0;i<4;++i) a-=A[i];
5      return a;
6  }
```

- Conditional assignments are often reorganised as simple sequential instructions by compilers using special assembler instructions when available
- Loops with small loop counts may be automatically unrolled at compile time leaving simple linear code

```
1  0000000000000000 <_Z1fdPd>:
2    subsd  xmm0,QWORD PTR [rdi]
3    subsd  xmm0,QWORD PTR [rdi+0x8]
4    subsd  xmm0,QWORD PTR [rdi+0x10]
5    subsd  xmm0,QWORD PTR [rdi+0x18]
6    ret
```

Exercise 2.11:

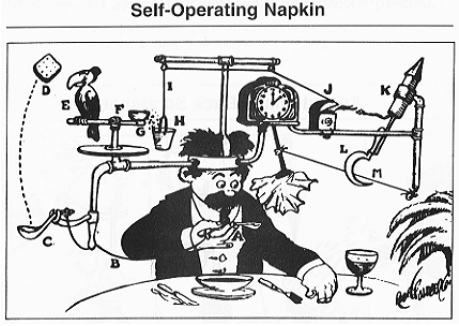
Branch prediction effectiveness using the example program `examples/branch_prediction.cc`, compare the processor on your own computer with the processors on JUSUF login nodes or compute nodes. The program partitions an array of integers into 3 ranges. Running it with a command line argument (value ignored) causes it to first sort the array and then perform the same partitioning actions. In the sorted array, the branches are easier to predict. What do you observe ? How do different compilers compare ?

Exercise 2.12:

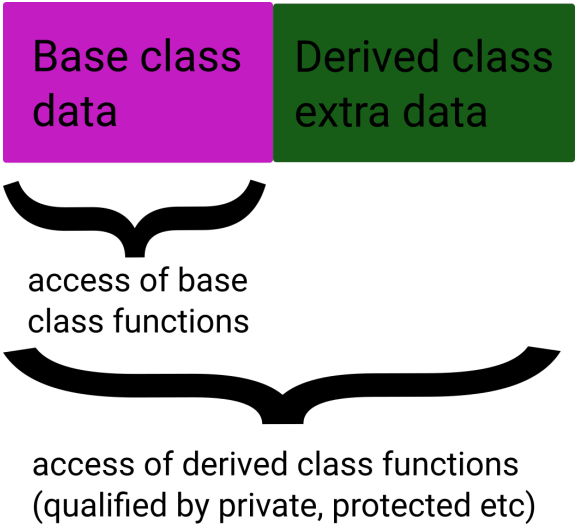
The program `examples/branch_prediction1.cc` further illustrates hardware branch prediction. Here, two alternative kinds of calculations need to be done and accumulated separately. Depending on the value inside a random array of numbers, we decide between the two calculations. It is impossible for the compiler to pre-determine the branches. Adjust the threshold to shift the probability of the two branches and observe the performance. Again, compare the 3 compilers!

Class inheritance, virtual functions and performance

- Class hierarchies constitute a flexible and beginner friendly tool kit
- In a fairly wide variety of applications, such as graphics, and many simulations, they may form the backbone of a robust, flexible code base
- Because of their success in some areas, they have been massively overused, leading to elaborate Rube Goldberg machines, which are neither easy to read nor particularly fast
- In modern C++, we should explore alternative ways to solve our problems
- Understanding how it works can help us more easily identify situations where a deep class hierarchy will be a bad idea.

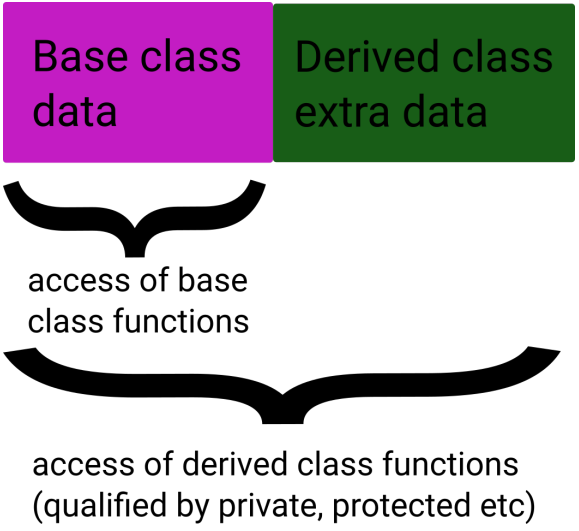


Inheritance



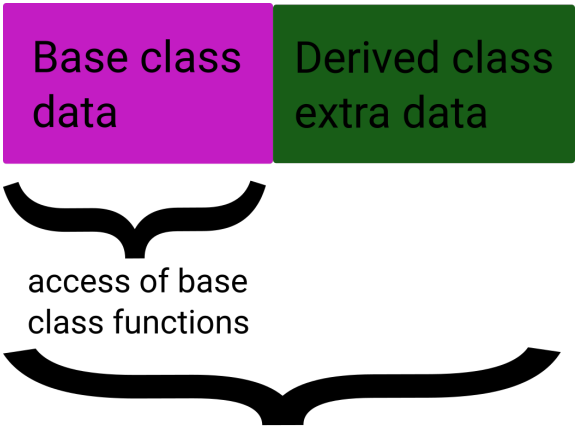
- Inheriting class may add more data, but it retains all the data of the base
- The base class functions, if invoked, will see a base class object
- The derived class object *is a* base class object, but with additional properties

Inheritance



- A pointer to a derived class always points to an address which also contains a valid base class object.
- `baseptr=derivedptr` is called "upcasting". Always allowed.
- Implicit downcasting is not allowed. Explicit downcasting is possible with `static_cast` and `dynamic_cast`

Inheritance



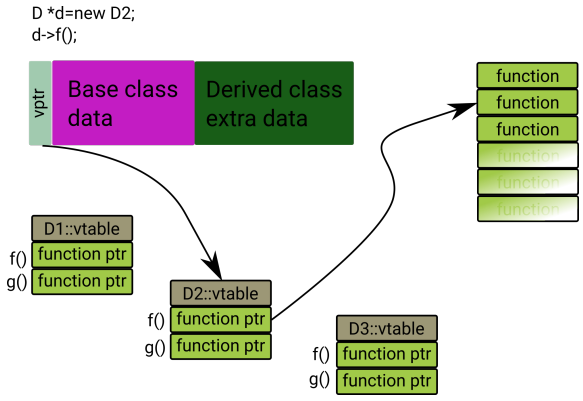

```
1  class Base {
2  public:
3    void f(){std::cout<<"Base::f()\n";}
4  protected:
5    int i{4};
6  };
7  class Derived : public Base {
8    int k{0};
9  public:
10   void g(){std::cout<<"Derived::g()\n";}
11 };
12 auto main() -> int
13 {
14   Derived b;
15   Base *ptr=&b;
16   ptr->g(); // Error!
17   static_cast<Derived *>(ptr)->g(); //OK
18 }
```

Class inheritance with virtual functions

```
1  auto main() -> int
2  {
3    vector<unique_ptr<Shape>> shapes;
4    shapes.push_back(make_unique<Circle>(0.5, Point(3,7)));
5    shapes.push_back(make_unique<Triangle>(Point(1,2),Point(3,3),Point(2.5,0)));
6  ...
7    for (auto&& shape : shapes) {
8      std::cout << shape->area() << '\n';
9    }
10 }
```

- A [smart] pointer to a base class is allowed to point to an object of a derived class
- Here, `shape[0]->area()` will call `Circle::area()`, `shape[1]->area()` will call `Triangle::area()`
- But, how does it work ?

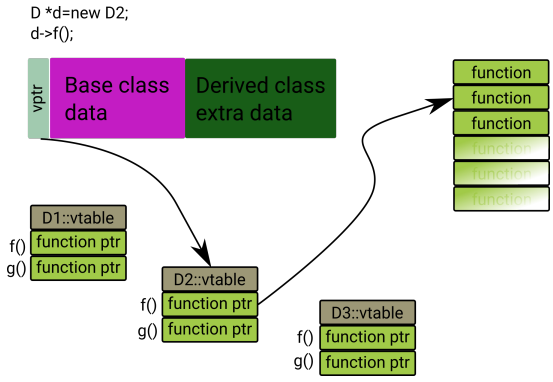
Calling virtual functions: how it works



- For classes with virtual functions, the compiler inserts an invisible pointer member to the data and additional book keeping code
- There is a table of virtual functions for each derived class, with entries pointing to function code somewhere
- The `vptr` pointer points to the *vtable* of that particular class

Calling virtual functions: how it works

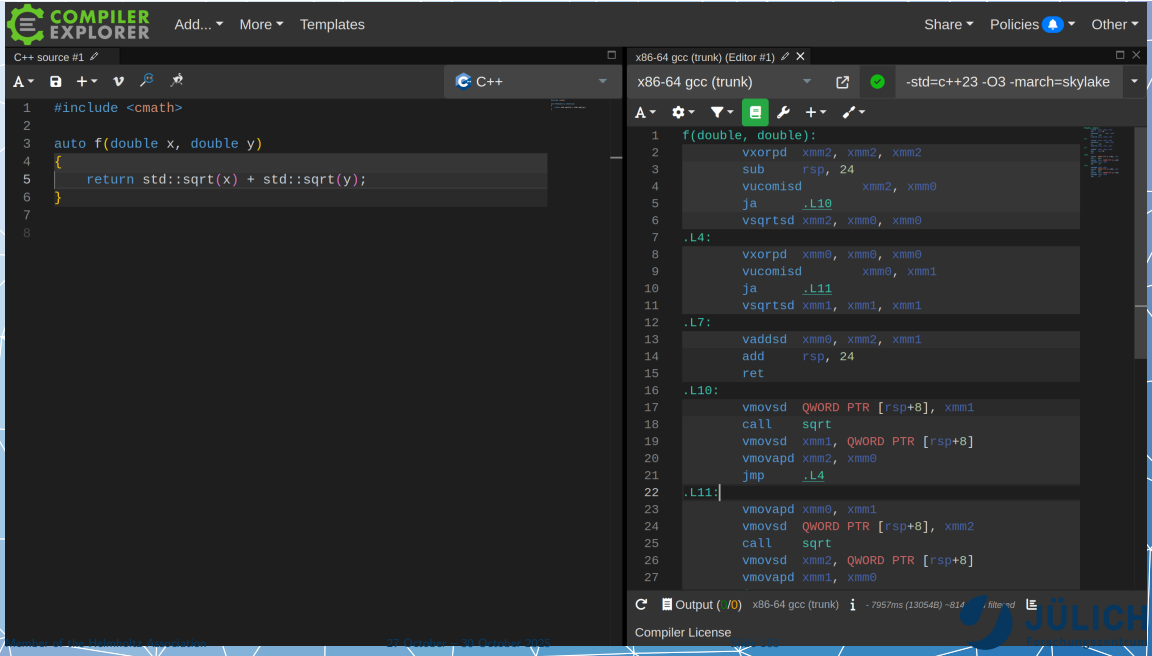
- Virtual function call proceeds by first finding the right *vtable*, then the correct entry for the called function, dereferencing that function pointer and then executing the correct function body
- Branch mispredictions, cache misses ...
- For HPC applications, use of virtual functions in hot sections will hurt performance

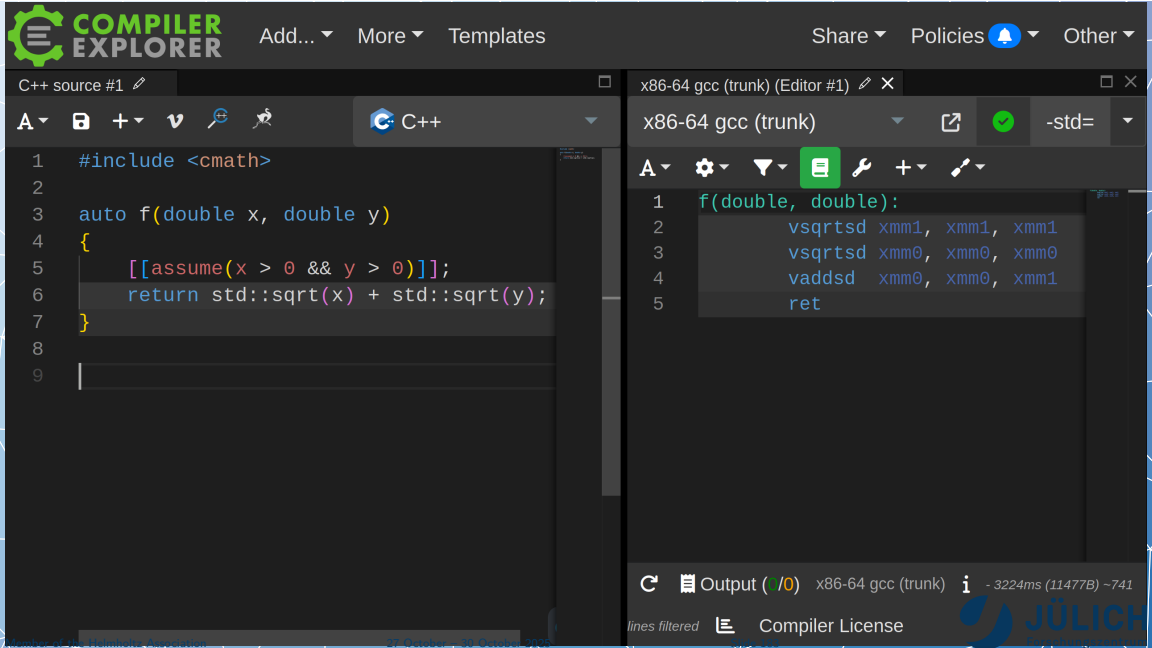


- Often, the polymorphic behaviour sought after using virtual functions can be implemented with CRTP without the virtual function overhead

Expressing assumptions

- Sometimes, relationships between function inputs can not be expressed through their types. The application developer might know that
  - the floating point input to a function is always between 0 and 1.
  - two integer inputs are always ordered smaller, greater
  - an array is never empty
  - ...
- When the compiler translates our code, such information is usually not available.
- To ensure correct results, code is generated to handle all kinds of corner cases, which we are certain can not ever happen
- C++23 introduced one such way to express such relations in code: `[[assume(expr)]]`
- `[[assume()]]` expressions may be placed anywhere in the function body and allow the compiler to make those assumptions at that point in code
- This gives a license to the compiler to make those assumptions and hence possibly generate some faster code. But faster code is not guaranteed.
- If the explicitly expressed assumptions are violated at the runtime, the result is undefined behaviour.
- It is usually better to use `[[assume(expr)]]` along with `assert` so that violations are detected during debugging


COMPILER EXPLORER
Add...
More
Templates
Share
Policies
Other
C++ source #1
C++
#include <cmath>
auto f(double x, double y)
{
return std::sqrt(x) + std::sqrt(y);
}
x86-64 gcc (trunk) (Editor #1)
x86-64 gcc (trunk)
-std=c++23 -O3 -march=skylake
f(double, double):
vxorpd xmm2, xmm2, xmm2
sub rsp, 24
vucomisd xmm2, xmm0
ja .L10
vsqrtsd xmm2, xmm0, xmm0
.L4:
vxorpd xmm0, xmm0, xmm0
vucomisd xmm0, xmm1
ja .L11
vsqrtsd xmm1, xmm1, xmm1
.L7:
vaddsd xmm0, xmm2, xmm1
add rsp, 24
ret
.L10:
vmovsd QWORD PTR [rsp+8], xmm1
call sqrt
vmovsd xmm1, QWORD PTR [rsp+8]
vmovapd xmm2, xmm0
jmp .L4
.L11:
vmovapd xmm0, xmm1
vmovsd QWORD PTR [rsp+8], xmm2
call sqrt
vmovsd xmm2, QWORD PTR [rsp+8]
vmovapd xmm1, xmm0
Output (./0)
x86-64 gcc (trunk)
Compiler License


COMPILER EXPLORER
Add...
More
Templates
Share
Policies
Other
C++ source #1
C++
x86-64 gcc (trunk) (Editor #1)
x86-64 gcc (trunk)
-std=
#include <cmath>
auto f(double x, double y)
{
[[assume(x > 0 && y > 0)]];
return std::sqrt(x) + std::sqrt(y);
}
f(double, double):
vsqrtsd xmm1, xmm1, xmm1
vsqrtsd xmm0, xmm0, xmm0
vaddsd xmm0, xmm0, xmm1
ret
Output (0/0)
x86-64 gcc (trunk)
- 3224ms (11477B) ~741
lines filtered
Compiler License

Using notifying special functions to learn

Exercise 2.13:

The course material includes a class called `Vbose` where the special member functions like constructors and the destructor emit messages when they are used. Such a class can be used to develop a better understanding of many run time effects. Three notebooks are provided in the folder `examples/`. They are meant for self study and experimentation. Open them by browsing in the left panel of your Jupyter session and double clicking on the notebook name. Go through them in the following order:

- `grow_vector.ipynb`
- `ref_qual_members.ipynb`
- `perfect_forwarding.ipynb`

The ideas introduced in these notebooks will be used later.

Reference qualified member functions

```
1  struct Box {
2      value_type r{};
3      auto value() const -> const value_type&
4      {
5          return r;
6      }
7      auto value() -> value_type& { return r; }
8  };
```

- If `b` is of the type `Box`, `b.value()` is a `const value_type&` or just `value_type&` depending on whether `b` is `const` or not

Reference qualified member functions

```cpp
// Since C++11
struct Box {
    value_type r{};
    auto value() const & -> const value_type&
    {
        return r;
    }
    auto value() & -> value_type& { return r; }
    auto value() const && -> const value_type&&
    {
        return r;
    }
    auto value() && -> value_type&& { return r; }
};
```

- If `b` is of the type `Box`, `b.value()` is a `const value_type&` or just `value_type&` depending on whether `b` is `const` or not
- Since C++11, more overloads are possible: one can have different versions of member functions depending on whether the calling instance is an L-value or an R-value reference.

Reference qualified member functions

```cpp
// Since C++11
struct Box {
    value_type r{};
    auto value() const & -> const value_type&
    {
        return r;
    }
    auto value() & -> value_type& { return r; }
    auto value() const && -> const value_type&&
    {
        return r;
    }
    auto value() && -> value_type&& { return r; }
};
```

- If `b` is of the type `Box`, `b.value()` is a `const value_type&` or just `value_type&` depending on whether `b` is `const` or not
- Since C++11, more overloads are possible: one can have different versions of member functions depending on whether the calling instance is an L-value or an R-value reference.
- Potentially quadruples the number of variations of a member function depending on the calling instance

Reference qualified member functions

```
1  struct FileData {
2      std::string header_text{};
3      std::vector<std::byte> bulk{};
4
5      auto header() const & -> const std::string&
6      {
7          return header_text;
8      }
9      auto header() & -> std::string&
10     {
11         return header_text;
12     }
13     auto header() const && -> const std::string&&
14     {
15         return std::move(header_text);
16     }
17     auto header() && -> std::string&&
18     {
19         return std::move(header_text);
20     }
21 };
22
23 auto readfile(std::filesystem::path fn) -> FileData;
```

- If `b` is of the type `Box`, `b.value()` is a `const value_type&` or just `value_type&` depending on whether `b` is `const` or not
- Since C++11, more overloads are possible: one can have different versions of member functions depending on whether the calling instance is an L-value or an R-value reference.
- Potentially quadruples the number of variations of a member function depending on the calling instance
- Sometimes, it is possible to provide some optimisations in situations where the calling instance is an R-value. An example demonstrating this can be explored in the notebook `ref_qual_members.ipynb`

Using templates for deduplication

```
struct Entity {
  Entity(const Vbose& x, const Vbose& y)
    : l{x}, r{y} {}
  Entity(const Vbose& x, Vbose&& y)
    : l{x}, r{std::move(y)} {}
  Entity(Vbose&& x, const Vbose& y)
    : l{std::move(x)}, r{y} {}
  Entity(Vbose&& x, Vbose&& y)
    : l{std::move(x)}, r{std::move(y)} {}

  Vbose l, r;
};
```

```
template <class T>
struct Entity {
  template <class U, class V>
  Entity(U&& x, V&& y)
      : l{std::forward<U>(x)},
      r{std::forward<V>(y)} {}

  T l, r;
};
```

- In the notebook `perfect_forwarding.ipynb` we explored a vaguely similar situation
- Instead of the 4 constructors in the first example, we could write a single function template, using forwarding references and `std::forward`
- The forwarding references, `U&&` and `V&&` capture the constantness L/R-value reference characteristics of the inputs
- The `std::forward` wrapping the uses of the respective variables casts them into their fully CVR qualified typenames.
- Can we do something like that to reduce the clutter in the previous examples?

Using templates for deduplication

```
1  struct FileData {
2      std::string header_text{};
3      std::vector<std::byte> bulk{};
4
5      auto header() const & -> const std::string&
6      {
7          return header_text;
8      }
9      auto header() & -> std::string&
10     {
11         return header_text;
12     }
13     auto header() const && -> const std::string&&
14     {
15         return std::move(header_text);
16     }
17     auto header() && -> std::string&&
18     {
19         return std::move(header_text);
20     }
21 };
```

- Imagine that, instead of these class member functions...

Using templates for deduplication

```
1  struct FileData {
2      std::string header_text{};
3      std::vector<std::byte> bulk{};
4  };
5  auto header(const FileData& fd) -> const std::string&
6  {
7      return fd.header_text;
8  }
9  auto header(FileData& fd) -> std::string&
10 {
11     return fd.header_text;
12 }
13 auto header(const FileData&& fd) -> const std::string&&
14 {
15     return std::move(fd.header_text);
16 }
17 auto header(FileData && fd) -> std::string&&
18 {
19     return std::move(fd.header_text);
20 }
21 };
```

- Imagine that, instead of these class member functions...
- we had a set of free standing functions doing the same thing

Using templates for deduplication

```
1  struct FileData {
2      std::string header_text{};
3      std::vector<std::byte> bulk{};
4  };
5  template <class C>
6  requires std::same_as<FileData,
7               std::remove_cvref_t<C>>
8  auto&& header(C&& fd)
9  {
10     return std::forward<C>(fd).header_text;
11 }
```

- Imagine that, instead of these class member functions...
- we had a set of free standing functions doing the same thing
- We could easily write a single template version for all this. This is doable, because the `FileData` object is explicitly available, and therefore can be made into a template parameter and so on.

Using templates for deduplication

```
1  struct FileData {
2      std::string header_text{};
3      std::vector<std::byte> bulk{};
4
5      auto header() const & -> const std::string&
6      {
7          return header_text;
8      }
9      auto header() & -> std::string&
10     {
11         return header_text;
12     }
13     auto header() const && -> const std::string&&
14     {
15         return std::move(header_text);
16     }
17     auto header() && -> std::string&&
18     {
19         return std::move(header_text);
20     }
21 };
```

- Imagine that, instead of these class member functions...
- we had a set of free standing functions doing the same thing
- We could easily write a single template version for all this. This is doable, because the `FileData` object is explicitly available, and therefore can be made into a template parameter and so on.
- Member functions don't expose the calling instance in the same way, since it is passed implicitly.

Using templates for deduplication

```
1  struct FileData {
2      std::string header_text{};
3      std::vector<std::byte> bulk{};
4      template <class Self>
5      auto&& header(this Self&& self)
6      {
7          return std::forward<Self>(self)
8              .header_text;
9      }
10 };
11
```

- Imagine that, instead of these class member functions...
- we had a set of free standing functions doing the same thing
- We could easily write a single template version for all this. This is doable, because the `FileData` object is explicitly available, and therefore can be made into a template parameter and so on.
- Member functions don't expose the calling instance in the same way, since it is passed implicitly.
- Good news! Since C++23, they can!

Using templates for deduplication

```
1  struct FileData {
2      std::string header_text{};
3      std::vector<std::byte> bulk{};
4      template <class Self>
5      auto&& header(this Self&& self)
6      {
7          return std::forward<Self>(self)
8              .header_text;
9      }
10 };
11
```

- Imagine that, instead of these class member functions...
- we had a set of free standing functions doing the same thing
- We could easily write a single template version for all this. This is doable, because the `FileData` object is explicitly available, and therefore can be made into a template parameter and so on.
- Member functions don't expose the calling instance in the same way, since it is passed implicitly.
- Good news! Since C++23, they can!
- The names `Self` etc are not special. You choose.

Using templates for deduplication

```
1  struct FileData {
2      std::string header_text{};
3      std::vector<std::byte> bulk{};
4      template <class Self>
5      auto&& header(this Self&& self)
6      {
7          return std::forward<Self>(self)
8              .header_text;
9      }
10 };
11
```

- Imagine that, instead of these class member functions...
- we had a set of free standing functions doing the same thing
- We could easily write a single template version for all this. This is doable, because the `FileData` object is explicitly available, and therefore can be made into a template parameter and so on.
- Member functions don't expose the calling instance in the same way, since it is passed implicitly.
- Good news! Since C++23, they can!
- The names `Self` etc are not special. You choose.
- The special syntax to explicitly name the calling instance is shown here

Using templates for deduplication

```
1  struct FileData {
2      std::string header_text{};
3      std::vector<std::byte> bulk{};
4      template <class Self>
5      auto&& header(this Self&& self)
6      {
7          return std::forward<Self>(self)
8              .header_text;
9      }
10 };
11
```

- Imagine that, instead of these class member functions...
- we had a set of free standing functions doing the same thing
- We could easily write a single template version for all this. This is doable, because the `FileData` object is explicitly available, and therefore can be made into a template parameter and so on.
- Member functions don't expose the calling instance in the same way, since it is passed implicitly.
- Good news! Since C++23, they can!
- The names `Self` etc are not special. You choose.
- The special syntax to explicitly name the calling instance is shown here
- Can't use **`this`** in such member functions

Polymorphism without virtual functions

- We have already seen how function overloading gives us a polymorphic unit : the overload set
- Different variation of a function is picked based on the type of the input parameters, or the constraints satisfied by the input parameters
- This is one kind of *static polymorphism*

Tag dispatching

```
1  struct flying {};
2  struct swimming {};
3  template <class T>
4  void do_something(T && t, flying)
5  {
6    t.fly(a,b);
7  }
8  template <class T>
9  void do_something(T && t, swimming) {...}
10 //...
11 template <class T>
12 void do_something(T t)
13 {
14   do_something(t, typename T::preferred_mode{});
15 }
```

```
1  class Buzzard {
2  public:
3    using preferred_mode = typename flying;
4  };
5  class Whale {
6  public:
7    using preferred_mode = typename swimming;
8  };
9  //...
10 Buzzard b;
11 do_something(b);
12 Whale w;
13 do_something(w);
```

- Logically similar operations on different types, where the operations depend on certain properties of the types

Tag dispatching

```
1  struct flying {};
2  struct swimming {};
3  template <class T>
4  void do_something(T && t, flying)
5  {
6    t.fly(a,b);
7  }
8  template <class T>
9  void do_something(T && t, swimming) {...}
10 //...
11 template <class T>
12 void do_something(T t)
13 {
14   do_something(t, typename T::preferred_mode{});
15 }
```

```
1  class Buzzard {
2  public:
3    using preferred_mode = typename flying;
4  };
5  class Whale {
6  public:
7    using preferred_mode = typename swimming;
8  };
9  //...
10 Buzzard b;
11 do_something(b);
12 Whale w;
13 do_something(w);
```

- Logically similar operations on different types, where the operations depend on certain properties of the types
- "Dispatch" functions to guide the compiler to a suitable implementation based on a "tag" in the incomming type

Tag dispatching

```cpp
struct flying {};
struct swimming {};
template <class T>
void do_something(T && t, flying)
{
  t.fly(a,b);
}
template <class T>
void do_something(T && t, swimming) {...}
//...
template <class T>
void do_something(T t)
{
  do_something(t, typename T::preferred_mode{});
}
```

```cpp
class Buzzard {
public:
  using preferred_mode = typename flying;
};
class Whale {
public:
  using preferred_mode = typename swimming;
};
//...
Buzzard b;
do_something(b);
Whale w;
do_something(w);
```

- Logically similar operations on different types, where the operations depend on certain properties of the types
- "Dispatch" functions to guide the compiler to a suitable implementation based on a "tag" in the incomming type
- Does not tie the overload to a specific type: dispatches based on some property of the input type

SFINAE : Substitution Failure is not an Error

```
1  // Examples/sfinae0.cc
2  template <class V>
3  void f(const V &v, typename V::iterator * jt=0)
4  {
5    std::cout << "Container overload\n";
6    for (auto x : v) std::cout <<  x <<" ";
7    std::cout << "\n";
8  }
9
10 void f(...)
11 {
12   std::cout << "Catch all overload\n";
13 }
14
15 auto main() -> int
16 {
17   std::list L{0.1, 0.2, 0.3, 0.4, 0.5, 0.6};
18   int A[4]{4, 3, 2, 1};
19   f(A);
20   f(L);
21 }
```

- Overload resolution of templates
- If substitution fails, overload discarded
- All parameters, expressions and the return type in declarations
- Substitution failure : ill-formed type or expression when a substitution is made
- Not in function body!

enable_if

```
1  // enable_if and enable_if_t are defined
2  // in the namespace std. We show them
3  // here to explain how they are used.
4  template <bool B, class T> struct enable_if;
5  template <class T> struct enable_if<true, T> {
6      using type=T ;
7  };
8  template <bool B, class T=void>
9  using enable_if_t=typename enable_if<B,T>::type;
10
11 template <class T>
12 enable_if_t<is_integral<T>::value, T>
13 Power(T x, T y) {
14   // Implementation suitable for
15   // integral number parameters
16 }
17 template <class T>
18 enable_if_t<is_floating_point<T>::value, T>
19 Power(T x, T y) {
20   // Implementation suitable for
21   // floating point parameters
22 }
```

- Only if the first parameter is true, the structure `enable_if` has a member type called `type` set to the second template parameter
- Using the `type` member of an `enable_if` struct in a declaration will lead to an ill-formed expression when the condition parameter is false. That version of the function will then be ignored

Let's not do such things any more. We have concepts now.

Exercise 2.14:

The tag dispatching technique is demonstrated in `examples/tag_dispatch.cc`.

Exercise 2.15:

`examples/sfinae0.cc` is a simple syntax illustration for SFINAE. Knowledge of history is imporant, but let this not be how you write your code in 2020s.

Choosing algorithm based on API

```
1  template <class C> size_t algo(C&& x)
2  {
3      if constexpr (hasAPI<C>) {
4          x.helper();
5          return x.calculateFast();
6      } else {
7          return x.calculate();
8      }
9  }
```

- We want to write a general algorithm for an operation
- In case the function argument has a certain member function, we have a neat and quick solution
- Otherwise, we have a fallback solution

Choosing algorithm based on API

```
1  template <class T> struct hasAPI_t {
2    using basetype =
3      typename remove_reference<T>::type;
4    template <class C>
5    static constexpr auto test(C * x) ->
6    decltype(x->calculateFast(),
7             x->helper(),
8             bool{})
9    {
10     return true;
11   }
12   static constexpr bool test(...) {
13     return false;
14   }
15   static constexpr auto value =
16    test(static_cast<basetype*>(nullptr));
17 };
```

- The "template function" `hasAPI_t` has a member `value` initialized via a **`constexpr`** function, which passes information about the templated type to the `test` function
- Two variants of the test function exist, one always returning false, to cover the "everything else" case

Choosing algorithm based on API

```
1  template <class T> struct hasAPI_t {
2    using basetype =
3      typename remove_reference<T>::type;
4    template <class C>
5    static constexpr auto test(C * x) ->
6    decltype(x->calculateFast(),
7             x->helper(),
8             bool{})
9    {
10     return true;
11   }
12   static constexpr bool test(...) {
13     return false;
14   }
15   static constexpr auto value =
16    test(static_cast<basetype*>(nullptr));
17 };
```

- The positive version of the `test` function defines its return type using **`decltype`**, but applying it to a comma separated list of necessary API expressions
- A comma separated list of expressions evaluates to the last value, but each value in the list is checked for syntax

Choosing algorithm based on API

```
1  template <class T> struct hasAPI_t {
2    using basetype =
3      typename remove_reference<T>::type;
4    template <class C>
5    static constexpr auto test(C * x) ->
6    decltype(x->calculateFast(),
7             x->helper(),
8             bool{})
9    {
10     return true;
11   }
12   static constexpr bool test(...) {
13     return false;
14   }
15   static constexpr auto value =
16    test(static_cast<basetype*>(nullptr));
17 };
```

- The positive version of the `test` function defines its return type using **`decltype`**, but applying it to a comma separated list of necessary API expressions
- A comma separated list of expressions evaluates to the last value, but each value in the list is checked for syntax

Choosing algorithm based on API

```
1  template <class T> struct hasAPI_t {
2    using basetype =
3      typename remove_reference<T>::type;
4    template <class C>
5    static constexpr auto test(C * x) ->
6    decltype(x->calculateFast(),
7             x->helper(),
8             bool{})
9    {
10     return true;
11   }
12   static constexpr bool test(...) {
13     return false;
14   }
15   static constexpr auto value =
16    test(static_cast<basetype*>(nullptr));
17 };
```

- The positive version of the `test` function defines its return type using **`decltype`**, but applying it to a comma separated list of necessary API expressions
- A comma separated list of expressions evaluates to the last value, but each value in the list is checked for syntax

- If the type of the argument does not have the member functions, the return type of the function can not be determined, and the overload is rejected

Choosing algorithm based on API

```
1  template <class T> constexpr bool hasAPI = hasAPI_t<T>::value;
2  template <class C> std::enable_if_t< hasAPI<C>, size_t > algo(C && x)
3  {
4      x.helper();
5      return x.calculateFast();
6  }
7  template <class C> std::enable_if_t< !hasAPI<C>, size_t > algo(C && x)
8  {
9      return x.calculate();
10 }
```

- What remains, is to make a nice wrapper template variable so that we can say `hasAPI<T>`, instead of `hasAPI_t<T>::value` when we need it.
- The dispatch functions are written using `enable_if_t`, so that we pick the `calculateFast` function over `calculate`, if it is available

Nah!

Choosing algorithm based on API

```
1  template <class T>
2  concept FastCalculator = requires (T rex) {
3    { rex.calculateFast() };
4    { rex.helper() };
5  };
6  template <FastCalculator C> auto algo(C && x)
7  {
8      x.helper();
9      return x.calculateFast();
10 }
11 template <class C> auto algo(C && x)
12 {
13     return x.calculate();
14 }
```

- Write a `concept` describing what member functions, inner types (like `value_type` for iterators) an object should have to satisfy the API
- Overload based on whether the constraints are satisfied!

Choosing algorithm based on API

```
1  auto main() -> int
2  {
3      Machinery obj;
4      auto res = algo(obj);
5      std::cout << "Result = " << res << "\n";
6  }
```

- Users of our great algorithm can simply call our `algo()` in their code
- If there is a `calculate` function, everything will work.
- If the author of the library providing `Machinery` goes on to implement `calculateFast` in the `Machinery` class, without any changes on the client side, or in the `algo` function, the compiler will make sure that the (hopefully) better, `calculateFast` function is used

Exercise 2.16:

The folder `examples/apishimming` contains the example `hasAPI` template function used in this section, with an application that uses it. By freeing the commented implementation of `calculateFast`, and recompiling, you will see that the call to `algo` automatically switches to use `calculateFast`.

```cpp
template <class T> struct hasAPI_t {
    using basetype = typename remove_reference<T>::type;
    template <class C> static constexpr auto test(C * x) ->
        decltype(x->calculateFast(),
        x->helper(),
        bool{})
    {
        return true;
    }
    static constexpr bool test(...)
    {
        return false;
    }
    static constexpr auto value =
        test(static_cast<basetype*>(nullptr));
};
template <class T>
constexpr bool hasAPI = hasAPI_t<T>::value;
template <class C>
std::enable_if_t< hasAPI<C>, size_t > algo(C && x)
{
    x.helper();
    return x.calculateFast();
}
template <class C>
std::enable_if_t< !hasAPI<C>, size_t > algo(C && x)
{
    return x.calculate();
}
```

Will get the job done.

```cpp
template <class T>
concept FastCalculator = requires (T rex) {
  { rex.calculateFast() };
  { rex.helper() };
};
template <FastCalculator C> auto algo(C && x)
{
    x.helper();
    return x.calculateFast();
}
template <class C> auto algo(C && x)
{
    return x.calculate();
}
```

Will get the job done and keep you sane.

Curiously Recurring Template Pattern

- You need types A and B which have some properties in common, which can be calculated using similar data
- There are a few polymorphic functions, but conceptually A and B are so different that you don't expect to store them in a single pointer container
- The penalty of using virtual functions seems to matter
- Option 1: implement as totally different classes, just copy and paste the common functions

Curiously Recurring Template Pattern

- You need types A and B which have some properties in common, which can be calculated using similar data
- There are a few polymorphic functions, but conceptually A and B are so different that you don't expect to store them in a single pointer container
- The penalty of using virtual functions seems to matter
- Option 1: implement as totally different classes, just copy and paste the common functions

Curiously Recurring Template Pattern

- You need types A and B which have some properties in common, which can be calculated using similar data
- There are a few polymorphic functions, but conceptually A and B are so different that you don't expect to store them in a single pointer container
- The penalty of using virtual functions seems to matter
- Option 1: implement as totally different classes, just copy and paste the common functions

Curiously Recurring Template Pattern

- You need types A and B which have some properties in common, which can be calculated using similar data
- There are a few polymorphic functions, but conceptually A and B are so different that you don't expect to store them in a single pointer container
- The penalty of using virtual functions seems to matter
- Option 1: implement as totally different classes, just copy and paste the common functions
- Option 2: try the CRTP

Curiously Recurring Template Pattern

```cpp
template <class D> struct ViewInterface {
    auto der() const -> const D * {
        return static_cast<const D *>(this);
    }
    auto begin() const {
    // Wont compile if D does not inherit from this
      return der()->begin_impl();
    }
    auto version() const -> int {
    // Non-polymorphic "common" function
      return 42;
    }
};
struct Atoi : public ViewInterface<Atoi> {
    auto begin_impl() const { return bg; }
};
struct List : public ViewInterface<List> {
    auto begin_impl() const -> string {
        return &basenode;
    }
};
```

```cpp
template <class T>
auto proc(ViewInterface<T> v) {
    auto b = v.begin();
    // ...
}
auto main() -> int {
  List H;
  proc(H);
  proc(Atoi{33});
}
```

- A function can demand that the inputs have a particular interface defined in the CRTP base
- Any input type inheriting from the CRTP base will be usable
- Polymorphism without virtual functions
- Enforces an interface at compile time
- Usually faster than virtual functions

“Mixin”

```cpp
// examples/crtp3.cc
template <class Derived> struct EnableCheckedAccess {
    auto at(std::size_t i) const {
        auto* d = static_cast<const Derived*>(this);
        if (i >= d->size())
            throw std::out_of_range(
                std::format("Index {} is out of range for container size {}", i, d->size()));
        return (*d)[i];
    }
};
struct MyVec : EnableCheckedAccess<MyVec> {
    auto operator[](std::size_t i) const { return i * i; }
    auto size() const -> std::size_t { return 5UL; }
};
auto main(int argc, char* argv[]) -> int {
    auto lim = argc > 1 ? std::stoul(argv[1]) : 5UL;
    MyVec v;
    try {
        for (auto i = 0UL; i < lim; ++i)
            std::print("Index = {}, value = {} \n", i, v.at(i));
    } catch (std::exception& err) { std::print("{}\n", err.what()); }
}
```

- Statically inject functionality into classes
- No virtual dispatch required

“Mixin”

```cpp
// examples/crtp4.cc
struct EnableCheckedAccess {
    template <class Self>
    auto at(this Self&& self, std::size_t i) {
        if (i >= self.size())
            throw std::out_of_range(
                std::format("Index {} is out of range for container size {}", i, self.size()));
        return self[i];
    }
};
struct MyVec : EnableCheckedAccess {
    auto operator[](std::size_t i) const { return i * i; }
    auto size() const -> std::size_t { return 5UL; }
};
auto main(int argc, char* argv[]) -> int {
    auto lim = argc > 1 ? std::stoul(argv[1]) : 5UL;
    MyVec v;
    try {
        for (auto i = 0UL; i < lim; ++i)
            std::print("Index = {}, value = {} \n", i, v.at(i));
    } catch (std::exception& err) { std::print("{}\n", err.what()); }
}
```

- Using the deducing this feature of C++23, we can make it much less weird!

Expression Templates

```
1  template <typename T>
2  class vec {
3    std::vector<T> dat;
4  public:
5    vec(size_t n) : dat(n) {}
6    auto operator[](size_t i) const -> T {
7      return dat[i];
8    }
9    auto operator[](size_t i) -> T & {
10     return dat[i];
11   }
12   size_t size() const{return dat.size();}
13 };
14 template <typename T>
15 auto operator+(const vec<T> & v1,
16                const vec<T> & v2) -> vec<T>
17 {
18   assert(v1.size() == v2.size());
19   auto ans = v1;
20   for (size_t i = 0; i < ans.size(); ++i)
21     ans[i] += v2[i];
22   return ans;
23 }
```

```
1  vec<double> W(N), X(N), Y(N), Z(N);
2  //..
3  W = a * X + 2 * a * Y + 3 * a * Z;
```

- Naive implementation which expresses our intent elegantly
- Each multiplication and addition creates a temporary and does a loop over elements
- Poor performance

Expression templates

If only we had a special class ...

- ... which stored references to `X`, `Y` and `Z`
- and had an `operator[]` which returns `a * X[i] + 2 * a * Y[i] + 3 * a * Z[i]`
- We could equip our `vec` class with a special assignment operator taking this special class as the right hand side

```
1  template <typename T>
2  class vec {
3    template <class XPR>
4    auto operator=(const XPR & r) -> vec &
5    {
6      for (size_t i = 0; i < size(); ++i) {
7        dat[i] = r[i]; // and r[i] returns a*X[i]+2*a*Y[i]+3*a*Z[i]
8      } // One single loop, no temporaries
9      return *this;
10   }
11 };
```

- We need a different special class for every expression we have to evaluate

Expression templates

- If we make a class like :

```
template <typename LHS, typename RHS>
class vecsum {
    const LHS & lhs;
    const RHS & rhs;
public:
    vecsum(const LHS & l, const RHS & r) : lhs{l}, rhs{r} {
        assert(l.size() == r.size());
    }
    auto operator[](size_t i) const { return lhs[i] + rhs[i]; }
    auto size() const { return lhs.size(); }
};
```

- We can define the sum of two `vec` objects to be a `vecsum` type

```
template <typename LHS, typename RHS>
auto operator+(const LHS& v1, const RHS& v2) -> vecsum<LHS, RHS>
{
    return vecsum<LHS, RHS>(v1, v2);
}
```

Expression templates

- If we try `vec1+vec2`, no evaluation happens, and we get a `vecsum<vec, vec>` object, we can call `[]` on this object and cause the calculation to happen.
- But, if we try `vec1 + 54` or `34 + "dino"`, we get nonsensical compound objects
- If we write our `operator+` like :

```
template <typename LHS, typename RHS>
auto operator+(const expr<LHS> & v1, const expr<RHS> & v2) -> vecsum<LHS, RHS> const
{
    return vecsum<LHS, RHS>(v1, v2);
}
```

  , we can restrict the template to objects which match the pattern `expr<something>`
- If we further want composability of the operations, we need `vecsum<LHS,RHS>` to also match the pattern `expr<something>`

Expression templates

Design with CRTP

- CRTP: a base template `vecxpr` to use as a base for all expressions of `vec` objects

```
template <typename X> struct vecxpr {
    X& der() noexcept { return *static_cast<X*>(this); }
    const X& der() const { return *static_cast<const X*>(this); }
};
```

Expression templates

Design with CRTP

- We make our expression classes like `vecsum` inherit from the template `vecxpr` instantiated on themselves:

```
1  template <typename T1, typename T2> class vecsum : public vecxpr<vecsum<T1,T2>> {
2    const T1 & lhs;
3    const T2 & rhs;
4  public:
5    using value_type = typename T1::value_type;
6    vecsum(const vecxpr<T1> & l, const vecxpr<T2> & r) : lhs{ l.der() }, rhs{ r.der() } {
7      assert(lhs.size() == rhs.size());
8    }
9    const auto operator[](size_t i) const { return lhs[i] + rhs[i]; }
10   size_t size() const { return lhs.size(); }
11 };
```

Expression templates

Design with CRTP

- `operator+` can now be written as:

```
1  template <typename T1, typename T2>
2  auto operator+(const vecxpr<T1> & v1, const vecxpr<T2> & v2)
3          -> const vecsum<T1, T2> {
4      return vecsum<T1, T2>{ l, r };
5  }
```

Expression templates

Design with CRTP

- We also make the original `vec` class inherit from `vecxpr`

```
1  template <typename T> class vec : public vecxpr<vec<T>> {
2    std::vector<T> dat;
3  public:
4      using value_type = T;
5      vec(size_t n) : dat(n) {}
6      auto operator[](size_t i) const -> const T& { return dat[i]; }
7      auto operator[](size_t i) -> T& { return dat[i]; }
8      size_t size() const { return dat.size(); }
9      size_t n_ops() const { return 0; }
10     template <typename X>
11     auto operator=(const vecxpr<X> & y) -> vec & {
12         dat.resize(y.der().size());
13         for (size_t i = 0; i < y.size(); ++i)
14             dat[i] = y.der()[i];
15         return *this;
16     }
17 };
```

Expression templates

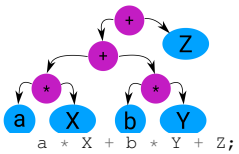
Design with CRTP

- We also make the original `vec` class inherit from `vecxpr`

```
1  template <typename T> class vec : public vecxpr<vec<T>> {
2    std::vector<T> dat;
3  public:
4      using value_type = T;
5      vec(size_t n) : dat(n) {}
6      auto operator[](size_t i) const -> const T& { return dat[i]; }
7      auto operator[](size_t i) -> T& { return dat[i]; }
8      size_t size() const { return dat.size(); }
9      size_t n_ops() const { return 0; }
10     template <typename X>
11     auto operator=(const vecxpr<X> & y) -> vec & {
12         dat.resize(y.der().size());
13         for (size_t i = 0; i < y.size(); ++i)
14             dat[i] = y.der()[i];
15         return *this;
16     }
17 };
```

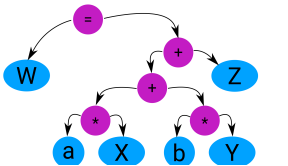
- Notice the special assignment operator from an expression!

Expression templates



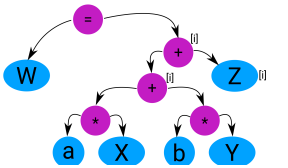

```
vecsum<
  vecsum<
    vecscl<vec<double>>,
    vecscl<vec<double>>
  >,
  vec<double>> ({{a,X},{b,Y}},Z);
// Let's call this type EXPR
```

Expression templates



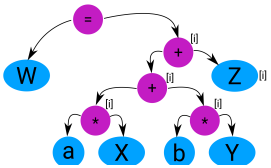

```
vec<double> &
vec<double>::operator=(const EXPR & E)
{
    dat.resize(E.size());
    for (size_t i = 0; i < E.size(); ++i)
        dat[i] = E[i];
    return *this;
}
```

Expression templates



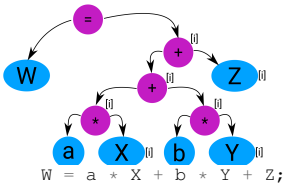

```
vec<double> &
vec<double>::operator=(const EXPR & E)
{
    dat.resize(E.size());
    for (size_t i = 0; i < E.size(); ++i)
        dat[i] = E[i];
const auto vecsum<L,R>::operator[](size_t i) const {
  return lhs[i] + rhs[i];
}
```

Expression templates




```
vec<double> &
vec<double>::operator=(const EXPR & E)
{
    dat.resize(E.size());
    for (size_t i = 0; i < E.size(); ++i)
        dat[i] = E[i];
const auto vecsum<L,R>::operator[](size_t i) const {
const auto vecscl<T>::operator[](size_t i) const {
    return lhs * rhs[i];
}
```

Expression templates




```
vec<double> &
vec<double>::operator=(const EXPR & E)
{
    dat.resize(E.size());
    for (size_t i = 0; i < E.size(); ++i)
        dat[i] = E[i];
const auto vecsum<L,R>::operator[](size_t i) const {
const auto vecscl<T>::operator[](size_t i) const {
const auto vec<T>::operator[](size_t i) const {
    return data[i];
}
```

Expression templates

- Elegant high level syntax
- Reduce temporaries
- Loop fusion
- Delayed evaluation: apply algorithmic optimizations on the entire expression, e.g.,
  - Evaluate `Matrix1 * Matrix2 * Vector` as `Matrix1 * (Matrix2 * Vector)`
  - Detect and eliminate cancelling operations, e.g., `Matrix_xpr1.transpose().transpose()`
  - Use optimized low level kernels with assembler, intrinsics, calls to vendor libraries etc to do the work
- However, can greatly increase compilation times

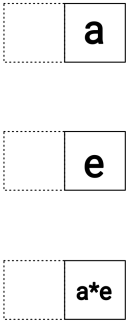
Exercise 2.17:

In `examples/xtmp0`, you will find a program which takes two numbers N and a as command line arguments, and creates 4 arrays W, X, Y, Z of size N (user defined array type `vec`). It fills X, Y and Z with random numbers and then calculates $W = a * X + 2 * a * Y + 3 * a * Z$, and times this operation by repeating the calculation 10 times. Two implementations of the user defined array type `vec` can be found: `naive_vec.hh` and `xtmp_vec0.hh`. Compile and run the program by alternating between the two headers. Study the code in `xtmp_vec0.hh`, which illustrates the ideas presented here about expression templates. The `xtmp_vec1.hh` implementation is almost the same, except using aligned allocation to store the arrays in the `vec` type. Test that as well.

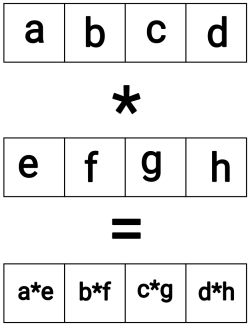
Exercise 2.18:

Introduce your own matrix class in the set up used in `examples/xtmp0`, so that matrix vector multiplications can be parts of vector expressions and `M1*M2*v` is evaluated as two matrix vector products rather than a matrix-matrix product followed by a matrix vector product.

SIMD registers and operations

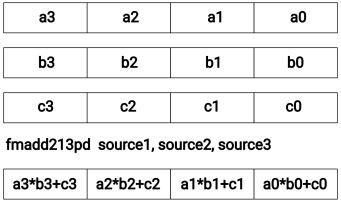


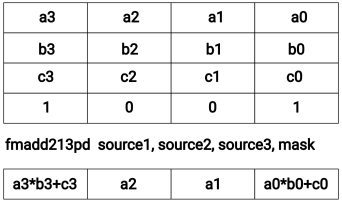
vmulsd xmm0, xmm0, xmm1

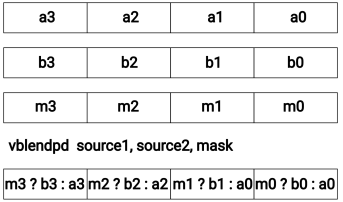


vmulpd ymm0, ymm0, ymm1

SIMD registers and operations

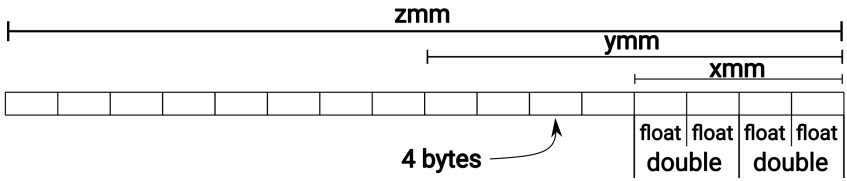






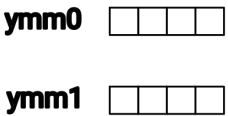
- Increasingly sophisticated instructions in newer CPUs
- Arithmatics, logical operations, shuffles, masked operations, trigonometry, cryptography ...

SIMD registers and operations



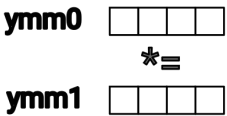
- `xmm0`, `xmm1`, ... `xmm7` (SSE)
- `xmm0` ... `xmm15`, `ymm0` ... `ymm15` (AVX, AVX2, FMA)
- `xmm0` ... `xmm31`, `ymm0` ... `ymm31`, `zmm0` ... `zmm31` (AVX512)

SIMD registers and operations



SIMD registers and operations

SIMD registers and operations



SIMD registers and operations

Automatic vectorization

- Compilers try to automatically identify opportunities to use SIMD instructions and generate appropriate code
- We write code exactly (or at least more or less) as before, and the vectorizer brings more speed
- Sometimes you may have to be careful about alignment of the arrays (`alignas()`, `std::assume_aligned())`
- Sometimes you might need to indicate to the compiler that the multiple arrays involved in a loop do not overlap, can be assumed to be independent (`#pragma ivdep`)
- You may want to allow the compiler to proceed with the assumption that floating point arithmetic is associative (`-fassociative-math`)

Automatic vectorization

```
1  void f(double x[], double y[], unsigned N)
2  {
3      for (unsigned i=0U; i<N; ++i) x[i] = 5. * x[i] + y[i];
4  }
```

- Compiler asks : Can this loop be run in blocks of 4 or 8 for all inputs x and y ? What, if `y = x+1` !
- Then it makes careful decisions so that the results are correct for *every possible input*
- Sometimes, we don't care about every possible input. Our functions are often mere cogs in a bigger machine, and their contract is more limited

OpenMP SIMD directives

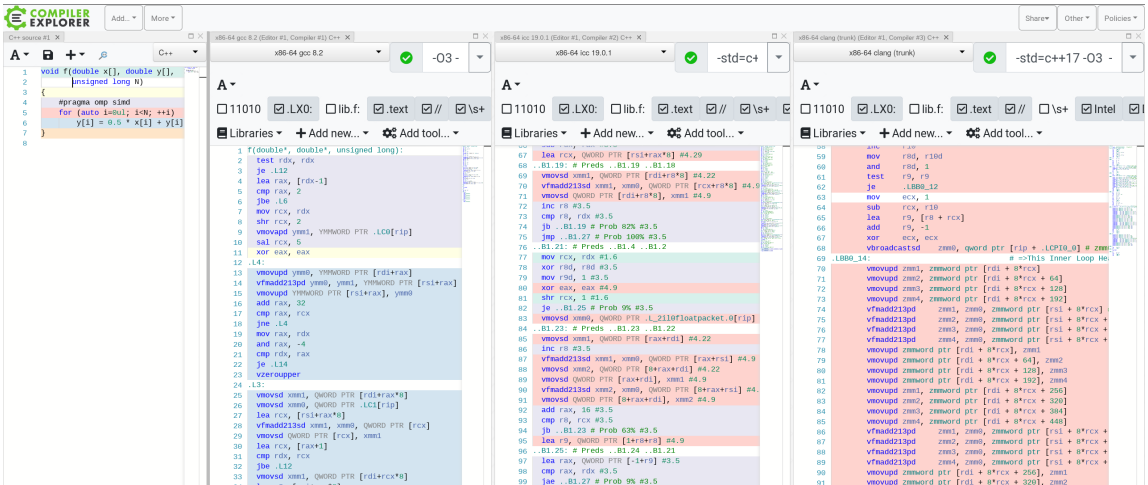
- Reorganize loop to run in chunks suitable for SIMD execution
- Syntax in C and C++ :

```
#pragma omp simd [clause [,clause ] ...]
for ( ... ) {}
```

- Often possible to call straight forward inline functions or vector enabled functions
  `#pragma omp declare simd`
- Can only be a traditional **for** loop. Loop count must be possible to determine at entry. No breaks.

```
template <typename T>
auto Vexv(T r2, T sigsa12) -> T {
    auto sg2
        = static_cast<T>(sqr(Lambda * sigsa12));
    auto a
        = static_cast<T>(icut2 * sqr(sigsa12));
    a = a * a * a;
    a = a * a;
    auto b = static_cast<T>(sixdivLLcut2 * a);
    a = 7.0 * a;
    T r6 = sg2 / r2;
    r6 = r6 * r6 * r6;
    return ksa * (r6 * r6 + a + b * r2);
}
auto addup( ___ ) -> double {
    double tot{};
    #pragma omp simd reduction(+:tot)
    for (size_t i=vec_size; i<R2.size(); ++i)
        tot += Vexv(R2[i], S12[i]);
    return tot;
}
```

For an excellent overview, search for "Michael Klemm, Intel, SIMD Vectorization with OpenMP"


COMPILER EXPLORER
Add...
More
Share
Other
Policies
C++ source #1
x86-64 gcc 8.2 (Editor #1, Compiler #1) C++
x86-64 icc 19.0.1 (Editor #1, Compiler #2) C++
x86-64 clang (trunk) (Editor #1, Compiler #3) C++
x86-64 gcc 8.2
x86-64 icc 19.0.1
x86-64 clang (trunk)
-O3 -
-std=c+
-std=c++17 -O3 -
C++
Libraries
Add new...
Add tool...
void f(double x[], double y[],
unsigned long N)
{
#pragma omp simd
for (auto i=0ul; i<N; ++i)
y[i] = 0.5 * x[i] + y[i]
}
f(double*, double*, unsigned long):
test rdx, rdx
je .L12
lea rax, [rdx-1]
cmp rax, 2
jbe .L6
mov rcx, rdx
shr rcx, 2
vmovapd ymm1, YMMWORD PTR .LC0[rip]
sal rcx, 5
xor eax, eax
.L4:
vmovupd ymm0, YMMWORD PTR [rdi+rax]
vfmadd213pd ymm0, ymm1, YMMWORD PTR [rsi+rax]
vmovupd YMMWORD PTR [rsi+rax], ymm0
add rax, 32
cmp rax, rcx
jne .L4
mov rax, rdx
and rax, -4
cmp rdx, rax
je .L14
vzeroupper
.L3:
vmovsd xmm1, QWORD PTR [rdi+rax*8]
vmovsd xmm0, QWORD PTR .LC1[rip]
lea rcx, [rsi+rax*8]
vfmadd213sd xmm1, xmm0, QWORD PTR [rcx]
vmovsd QWORD PTR [rcx], xmm1
lea rcx, [rax+1]
cmp rdx, rcx
jbe .L12
vmovsd xmm1, QWORD PTR [rdi+rcx*8]
lea r8, [rsi+rcx*8]
vfmadd213sd xmm1, xmm0, QWORD PTR [r8]
add rax, 2
vmovsd QWORD PTR [r8], xmm1
cmp rdx, rax
jbe .L12
lea rcx, QWORD PTR [rsi+rax*8] #4.29
..B1.19: # Preds ..B1.19 ..B1.18
vmovsd xmm1, QWORD PTR [rdi+r8*8] #4.22
vfmadd213sd xmm1, xmm0, QWORD PTR [rcx+r8*8] #4.9
vmovsd QWORD PTR [rdi+r8*8], xmm1 #4.9
inc r8 #3.5
cmp r8, rdx #3.5
jb ..B1.19 # Prob 82% #3.5
jmp ..B1.27 # Prob 100% #3.5
..B1.21: # Preds ..B1.4 ..B1.2
mov rcx, rdx #1.6
xor r8d, r8d #3.5
mov r9d, 1 #3.5
xor eax, eax #4.9
shr rcx, 2 #1.6
je ..B1.25 # Prob 9% #3.5
vmovsd xmm0, QWORD PTR .L_2il0floatpacket.0[rip]
..B1.23: # Preds ..B1.23 ..B1.22
vmovsd xmm1, QWORD PTR [rax+rdi] #4.22
inc r8 #3.5
vfmadd213sd xmm1, xmm0, QWORD PTR [rax+rsi] #4.9
vmovsd xmm2, QWORD PTR [8+rax+rdi] #4.22
vmovsd QWORD PTR [rax+rdi], xmm1 #4.9
vfmadd213sd xmm2, xmm0, QWORD PTR [8+rax+rsi] #4.
vmovsd QWORD PTR [8+rax+rdi], xmm2 #4.9
add rax, 16 #3.5
cmp r8, rcx #3.5
jb ..B1.23 # Prob 63% #3.5
lea r9, QWORD PTR [1+r8+r8] #4.9
..B1.25: # Preds ..B1.24 ..B1.21
lea rax, QWORD PTR [-1+r9] #3.5
cmp rax, rdx #3.5
jae ..B1.27 # Prob 9% #3.5
vmovsd xmm1, QWORD PTR .L_2il0floatpacket.0[rip]
vmovsd xmm0, QWORD PTR [-8+rdi+r9*8] #4.22
vfmadd213sd xmm1, xmm0, QWORD PTR [-8+rsi+r9*8] #
vmovsd QWORD PTR [-8+rdi+r9*8], xmm1 #4.9
..B1.27: # Preds ..B1.19 ..B1.25 ..B1.1 ..B1.17 ..B
xor r10d, r10d
mov r8d, r10d
and r8d, 1
test r9, r9
je .LBB0_12
mov ecx, 1
sub rcx, r10
lea r9, [r8 + rcx]
add r9, -1
xor ecx, ecx
vbroadcastsd zmm0, qword ptr [rip + .LCPI0_0] # zmm
.LBB0_14: # =>This Inner Loop He
vmovupd zmm1, zmmword ptr [rdi + 8*rcx]
vmovupd zmm2, zmmword ptr [rdi + 8*rcx + 64]
vmovupd zmm3, zmmword ptr [rdi + 8*rcx + 128]
vmovupd zmm4, zmmword ptr [rdi + 8*rcx + 192]
vfmadd213pd zmm1, zmm0, zmmword ptr [rsi + 8*rcx]
vfmadd213pd zmm2, zmm0, zmmword ptr [rsi + 8*rcx +
vfmadd213pd zmm3, zmm0, zmmword ptr [rsi + 8*rcx +
vfmadd213pd zmm4, zmm0, zmmword ptr [rsi + 8*rcx +
vmovupd zmmword ptr [rdi + 8*rcx], zmm1
vmovupd zmmword ptr [rdi + 8*rcx + 64], zmm2
vmovupd zmmword ptr [rdi + 8*rcx + 128], zmm3
vmovupd zmmword ptr [rdi + 8*rcx + 192], zmm4
vmovupd zmm1, zmmword ptr [rdi + 8*rcx + 256]
vmovupd zmm2, zmmword ptr [rdi + 8*rcx + 320]
vmovupd zmm3, zmmword ptr [rdi + 8*rcx + 384]
vmovupd zmm4, zmmword ptr [rdi + 8*rcx + 448]
vfmadd213pd zmm1, zmm0, zmmword ptr [rsi + 8*rcx +
vfmadd213pd zmm2, zmm0, zmmword ptr [rsi + 8*rcx +
vfmadd213pd zmm3, zmm0, zmmword ptr [rsi + 8*rcx +
vfmadd213pd zmm4, zmm0, zmmword ptr [rsi + 8*rcx +
vmovupd zmmword ptr [rdi + 8*rcx + 256], zmm1
vmovupd zmmword ptr [rdi + 8*rcx + 320], zmm2
vmovupd zmmword ptr [rdi + 8*rcx + 384], zmm3
vmovupd zmmword ptr [rdi + 8*rcx + 448], zmm4
add rcx, 64
add r9, 2
jne .LBB0_14

Digging deeper

```
1  double pairwise(unsigned i, unsigned j,
2                  SOA * particle_record)
3  {
4      // very clever calculations
5  }
6  auto energy() -> double
7  {
8    double ans = 0.;
9    for (auto i = 0; i < npt; ++i) {
10     #pragma omp please vectorize
11     for (auto j = i + 1; j < npt; ++j) {
12       ans +=
13         pairwise(i, j, my_particle_record);
14     }
15   }
16   return ans;
17 }
```

Convenient. But what are we not doing ?

- Coding to load groups of 2 or 4 or 8 numbers, working with them and storing the results
- Comparing different ways to use SIMD instructions to solve the problem for our actual inputs
- Choosing to use relaxed assumptions about floating point arithmetic at specific places in the code

Deviate only for special situations!

As HPC C++ programmers, we should know how to take full control of vectorization. But automatic or OpenMP based vectorization should be your first choice for production code. Most often they provide a cleaner, easier path. Sometimes, when the easier way does not provide enough low level access, we have ways to go beyond them.


R 1 3 5
2 4 6
12V

Introduction to intrinsics

- Recognizing how numbers are stored and manipulated in the computer opens up new opportunities
- Computer arithmetic has more "fundamental" operations than normal mathematics : `+`, `-`, `*`, `/`, `%`, `&`, `|`, `<<`, `>>`

```
auto morton_plain(unsigned long x,
                  unsigned long y,
                  unsigned long z)
{
    auto ans = 0UL;
    unsigned long i=0;
    while (i<22) {
        unsigned long bx = (x & (1 << i));
        unsigned long by = (y & (1 << i));
        unsigned long bz = (z & (1 << i));
        auto j = 2*i;
        ans = ans | (bx << j)
                  | (by << (j+1))
                  | (bz << (j+2));
        ++i;
    }
    return ans;
}
```

```
auto morton(unsigned long x, unsigned long y,
            unsigned long z)
{
    constexpr unsigned long mask[]{
    0x9249249249249249,// 0b100100100...1001001
    0x2492492492492492,// 0b001001001...0010010
    0x4924924924924924 // 0b010010010...0100100
    };
    // On x86 ...
    return _pdep_u64(x, mask[0])
         | _pdep_u64(y, mask[1])
         | _pdep_u64(z, mask[2]);
}
```

Intrinsics : high(er) level interface to CPU instructions

Interface to SSE and AVX registers

- include "nmmintrin.h" (SSE 4.2) or "immintrin.h" (AVX)
- `__m128i` : integer register with 128 bits
- `__m128` : 128 bits with 4 packed floats
- `__m128d` : 128 bits with 2 doubles
- `__m256i` : 256 bit octint
- `__m256` : octfloat
- `__m256d` : quaddouble

- Intel x86 optimization manual
- Intel intrinsics guide

Interface to SSE and AVX operations

- `_mm_add_ps(__m128, __m128)`
- `_mm_sub_ps(,)`, `_mm_sqrt_ps()` ...
- `_mm256_add_pd(__m256d, __m256d)`
- Convention:
  `_(sizecode)_(operation)_(suffix)`
  - sizecode is mm for SSE, mm256 for AVX and mm512 for AVX512
  - operation is "add", "sub", "mul" etc.
  - suffix indicates data type in the register arguments. ps => float, pd => double, epi32 => 32 bit signed int, epu32 => 32 bit unsigned int

Example: direct use of intrinsics

```
float sprod_sane(size_t n, const float a[],
                 const floatb[]) {
    double res{};
    for (size_t i=0UL; i<n; ++i)
        res += a[i] * b[i];
    return res;
}
```

```
float sse_sprod(size_t n, const float a[],
                const float b[]) {
    assert(0 == n % 4);// simplifying assumption
    __m128 res, prd, ma, mb;
    res = _mm_setzero_ps();
    for (size_t i=0; i<n; i += 4) {
        ma = _mm_loadu_ps(&a[i]);
        mb = _mm_loadu_ps(&b[i]);
        prd = _mm_mul_ps(ma, mb);
        res = _mm_add_ps(prd, res);
    }
    prd = _mm_setzero_ps();
    res = _mm_hadd_ps(res, prd);
    res = _mm_hadd_ps(res, prd); // not a typo!
    float tmp;
    _mm_store_ss(&tmp, res);
    return tmp;
}
```

- (RHS) Feels *C++'ish*, but commits too much to machine level details
- This is just an example to show what bare intrinsics based code looks like. It is almost never a good idea to use raw intrinsics in application code. It's lazy and dangerous, and ends up costing you more time anyway.
- Beware of persistant superstition surrounding abstractions. Overreaching advice against compile time abstractions such as static polymorphism, template or constexpr metaprogramming is usually bad advice. Always check.

Wrapping intrinsics in zero (/low) cost abstractions

```
#include <immintrin.h>
union alignas(32) QuadDouble {
    __m256d mm;
    double d[4];
    QuadDouble(__m256d oth) : mm{oth} {}
    constexpr QuadDouble(double x, double y,double z=0., double t=0.) : d{x, y, z, t} {}

    void aligned_load(double * v) {
        assert(get_alignment(v) >= 32);
        mm = _mm256_load_pd(v);
    }
    void unaligned_load(double * v) { mm = _mm256_loadu_pd(v); }

    [[nodiscard]]auto operator[](unsigned i) const ->  double { return d[i%4]; }
    auto operator[](unsigned i) -> double & { return d[i%4]; }

    void operator=(double x) { mm = _mm256_broadcast_sd(&x); }
    [[nodiscard]] auto horizontal_add() const -> double { return d[0] + d[1] + d[2] + d[3]; }
};
```

Wrapping intrinsics in zero (/low) cost abstractions

```
1  auto get_alignment(void * var) {
2      auto n = reinterpret_cast<unsigned long>(var);
3      return (-n) & n;
4  }
5  auto operator+(QuadDouble a, QuadDouble b) -> QuadDouble {return _mm256_add_pd(a.mm, b.mm);}
6  auto operator-(QuadDouble a, QuadDouble b) -> QuadDouble {return _mm256_sub_pd(a.mm, b.mm);}
7  auto operator*(QuadDouble a, QuadDouble b) -> QuadDouble {return _mm256_mul_pd(a.mm, b.mm);}
8  auto operator/(QuadDouble a, QuadDouble b) -> QuadDouble {return _mm256_div_pd(a.mm, b.mm);}
9
10 auto main() -> int
11 {
12     QuadDouble a{3.1}, b{0.2, 5.4, 2.1, 9.8};
13     auto c = a * b - (a / b);
14     return c[2] < -1.;
15 }
```

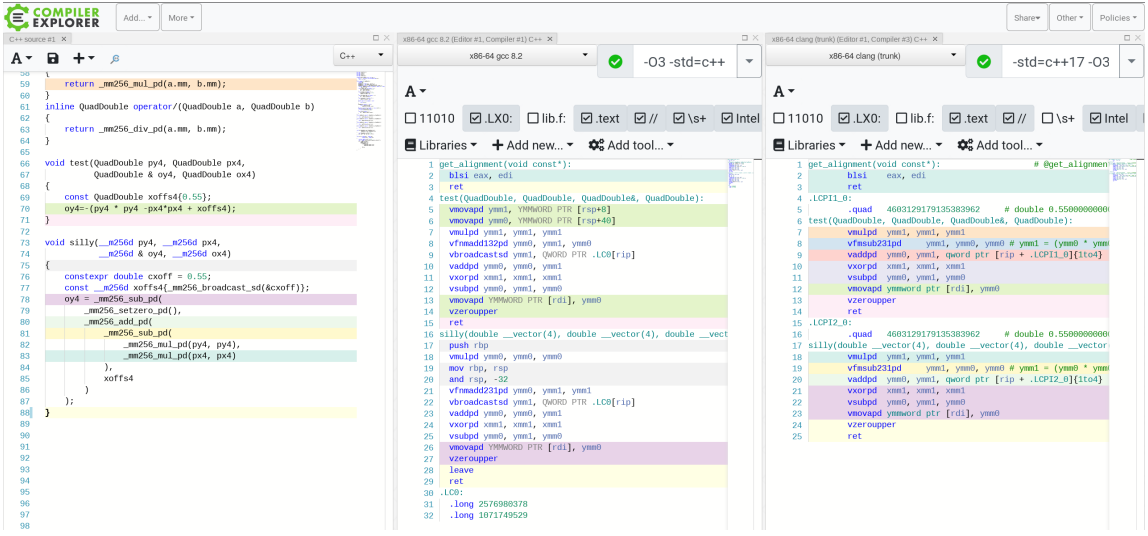
Wrapping intrinsics in zero(/low) cost abstractions

- Notational simplification, more readable and maintainable code, at no (or rather low) run time cost
- Need to wrap all operations used by your application (but only those)
- Need to hide vendor specific differences

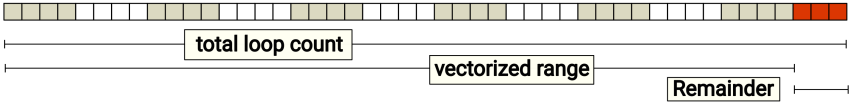
```
auto operator*(QuadDouble a, QuadDouble b) -> QuadDouble
{
    return _mm256_mul_pd(a.mm, b.mm);
}
```

```
1  # With Clang 7
2    operator*(QuadDouble, QuadDouble): # @operator*(QuadDouble, QuadDouble)
3      vmulpd  ymm0, ymm0, ymm1
4      ret
```

Note on alignment: Dynamically allocated arrays of our abstraction can cause unexpected crashes for C++98 . . . C++14, as the `new` operator could not align "over aligned" types on the heap. This was fixed in C++17, and optionally provided for C++11 and C++14 with compiler flags (GCC: `-faligned-new` Clang: `-faligned-allocation`).

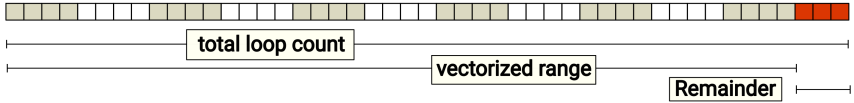

COMPILER EXPLORER
Add...
More
Share
Other
Policies
C++ source #1
x86-64 gcc 8.2 (Editor #1, Compiler #1) C++
x86-64 clang (trunk) (Editor #1, Compiler #3) C++
x86-64 gcc 8.2
-O3 -std=c++
x86-64 clang (trunk)
-std=c++17 -O3
Libraries
Add new...
Add tool...
return _mm256_mul_pd(a.mm, b.mm);
}
inline QuadDouble operator/(QuadDouble a, QuadDouble b)
{
return _mm256_div_pd(a.mm, b.mm);
}
void test(QuadDouble py4, QuadDouble px4,
QuadDouble & oy4, QuadDouble ox4)
{
const QuadDouble xoffs4(0.55);
oy4=-(py4 * py4 -px4*px4 + xoffs4);
}
void silly(__m256d py4, __m256d px4,
__m256d & oy4, __m256d ox4)
{
constexpr double cxoff = 0.55;
const __m256d xoffs4{_mm256_broadcast_sd(&cxoff)};
oy4 = _mm256_sub_pd(
_mm256_setzero_pd(),
_mm256_add_pd(
_mm256_sub_pd(
_mm256_mul_pd(py4, py4),
_mm256_mul_pd(px4, px4)
),
xoffs4
)
);
}
get_alignment(void const*):
blsi eax, edi
ret
test(QuadDouble, QuadDouble, QuadDouble&, QuadDouble):
vmovapd ymm1, YMMWORD PTR [rsp+8]
vmovapd ymm0, YMMWORD PTR [rsp+40]
vmulpd ymm1, ymm1, ymm1
vfmaddl32pd ymm0, ymm1, ymm0
vbroadcastsd ymm1, QWORD PTR .LC0[rip]
vaddpd ymm0, ymm0, ymm1
vxorpd xmm1, xmm1, xmm1
vsubpd ymm0, ymm1, ymm0
vmovapd YMMWORD PTR [rdi], ymm0
vzeroupper
ret
silly(double __vector(4), double __vector(4), double __vect
push rbp
vmulpd ymm0, ymm0, ymm0
mov rbp, rsp
and rsp, -32
vfmadd231pd ymm0, ymm1, ymm1
vbroadcastsd ymm1, QWORD PTR .LC0[rip]
vaddpd ymm0, ymm0, ymm1
vxorpd xmm1, xmm1, xmm1
vsubpd ymm0, ymm1, ymm0
vmovapd YMMWORD PTR [rdi], ymm0
vzeroupper
leave
ret
.LC0:
.long 2576980378
.long 1071749529
get_alignment(void const*): # @get_alignment
blsi eax, edi
ret
.LCPI1_0:
.quad 4603129179135383962 # double 0.5500000000
test(QuadDouble, QuadDouble, QuadDouble&, QuadDouble):
vmulpd ymm1, ymm1, ymm1
vfmsub231pd ymm1, ymm0, ymm0 # ymm1 = (ymm0 * ymm
vaddpd ymm0, ymm1, qword ptr [rip + .LCPI1_0]{1to4}
vxorpd xmm1, xmm1, xmm1
vsubpd ymm0, ymm1, ymm0
vmovapd ymmword ptr [rdi], ymm0
vzeroupper
ret
.LCPI2_0:
.quad 4603129179135383962 # double 0.5500000000
silly(double __vector(4), double __vector(4), double __vector
vmulpd ymm1, ymm1, ymm1
vfmsub231pd ymm1, ymm0, ymm0 # ymm1 = (ymm0 * ymm
vaddpd ymm0, ymm1, qword ptr [rip + .LCPI2_0]{1to4}
vxorpd xmm1, xmm1, xmm1
vsubpd ymm0, ymm1, ymm0
vmovapd ymmword ptr [rdi], ymm0
vzeroupper
ret

Using our DIY SIMD library



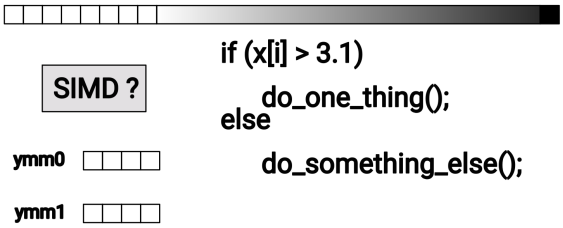

```
1    // examples/diy/daxpy.cc
2  void daxpy_explicit(const std::vector<double> & x, std::vector<double> & y, double a) {
3      QuadDouble bx{0.}, by{0.};
4      const QuadDouble ba{a};
5      unsigned long vsize = x.size() - x.size() % 4;
6      const double * xptr0 = x.data();
7      const double * xptr1 = x.data() + vsize;
8      double * yptr = y.data();
9      for (; xptr0 != xptr1; xptr0 += 4, yptr += 4) {
10         bx.unaligned_load(xptr0);
11         by.unaligned_load(yptr);
12         by = by + bx * ba;
13         by.unaligned_store(yptr);
14     }
15     for (auto i=vsize; i<x.size(); ++i) y[i] += a* x[i];
16 }
```

Using our DIY SIMD library




```
1  // examples/diy/sprod.cc
2  #include "QuadDouble.hh"
3  auto sprod_explicit(size_t n, const double x[], const double y[]) -> double {
4      QuadDouble bx{0.}, by{0.}, tot{0.};
5      unsigned long vsize = n - n % 4;
6      const double * xptr0 = x;
7      const double * xptr1 = x + vsize;
8      for (; xptr0 != xptr1; xptr0 += 4, y += 4) {
9          bx.unaligned_load(xptr0);
10         by.unaligned_load(y);
11         tot = tot + bx * by;
12     }
13     auto res = tot.horizontal_add();
14     for (auto i = vsize; i < x.size(); ++i) res += x[i] * y[i];
15     return res;
16 }
```

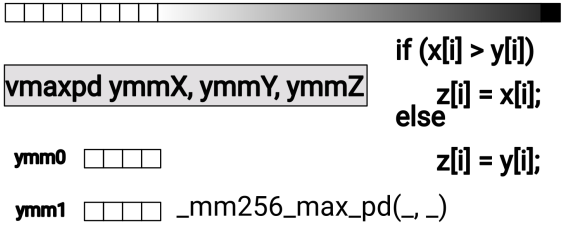
Conditional selection using masks and blend



- Different lanes in a SIMD register can not execute different instructions $\Longrightarrow$ problems with general branched code

- Various kinds of conditional selection can be executed as single instructions: picking out the larger of the corresponding elements between two arrays

Conditional selection using masks and blend



- Different lanes in a SIMD register can not execute different instructions $\Longrightarrow$ problems with general branched code

- Various kinds of conditional selection can be executed as single instructions: picking out the larger of the corresponding elements between two arrays
- Large number of "masked" instructions, e.g., `_mm256_mask_[op]_[type]` and `_mm256_maskz_[op]_[type]`. A mask is a bit field of the appropriate size storing 0 or 1. The `...maskz...` variants zero out the positions in the destination corresponding to the entries where the mask is unset. The `...mask...` variants take an additional `src` argument, and copy the result from there, if the mask is unset. Both store the result of the computation if the mask bit is in fact set.

Conditional selection using masks and blend

a3	a2	a1	a0
b3	b2	b1	b0
c3	c2	c1	c0
1	0	0	1

fmadd213pd source1, source2, source3, mask

a3*b3+c3	a2	a1	a0*b0+c0

- Different lanes in a SIMD register can not execute different instructions $\Longrightarrow$ problems with general branched code

- Various kinds of conditional selection can be executed as single instructions: picking out the larger of the corresponding elements between two arrays
- Large number of "masked" instructions, e.g., `_mm256_mask_[op]_[type]` and `_mm256_maskz_[op]_[type]`. A mask is a bit field of the appropriate size storing 0 or 1. The `...maskz...` variants zero out the positions in the destination corresponding to the entries where the mask is unset. The `...mask...` variants take an additional `src` argument, and copy the result from there, if the mask is unset. Both store the result of the computation if the mask bit is in fact set.
- Masked selection between two alternatives is also possible using "blend instructions"

Conditional selection using masks and blend

a3	a2	a1	a0

b3	b2	b1	b0

m3	m2	m1	m0

vblendpd source1, source2, mask

m3 ? b3 : a3	m2 ? b2 : a2	m1 ? b1 : a0	m0 ? b0 : a0

- Different lanes in a SIMD register can not execute different instructions $\Longrightarrow$ problems with general branched code

- Various kinds of conditional selection can be executed as single instructions: picking out the larger of the corresponding elements between two arrays
- Large number of "masked" instructions, e.g., `_mm256_mask_[op]_[type]` and `_mm256_maskz_[op]_[type]`. A mask is a bit field of the appropriate size storing 0 or 1. The `...maskz...` variants zero out the positions in the destination corresponding to the entries where the mask is unset. The `...mask...` variants take an additional `src` argument, and copy the result from there, if the mask is unset. Both store the result of the computation if the mask bit is in fact set.
- Masked selection between two alternatives is also possible using "blend instructions"

Creating and manipulating masks

- Masks are bit fields. Conceptually they are like arrays of boolean variables with the same number of elements as the corresponding SIMD register
- Many SIMD functions return mask types:
  - `_mm256_cmpeq_epi32_mask(__m256i, __m256i)` : element wise comparison. All corresponding bits of the mask set if equality comparison returns true for an element
  - `_mm256_cmplt_epi32_mask(__m256i, __m256i)` : As with cmpeq, but for “less than” comparison
- Masks can be combined with usual bit wise operations `_mm256_and_pd`, `_m256_or_pd` etc.

```
1   auto m1 = _mm256_cmpge_epi32_mask(vi, vj);
2   auto m2 = _mm256_cmpeq_epi32_mask(vk, _mm256_setzero_epi32());
3   auto mask = _mm256_and_si256(m1, m2);
4   res = _mm256_fmask_fmadd_pd(x, mask, y, z);
```

Great! Now, what about AVX512 ? Power ? ARM ?

- Application code can operate using the abstraction
- Architecture specific details can be hidden inside the SIMD library
- No run-time indirection is needed. The compiler can be made to choose one specific version (macros, template specializations . . .)
- The author(s) of the SIMD library have to deal with the available capabilities in different instruction sets
- The library can also provide additional benefits: SIMD implementation of widely used functions, e.g., trigonometric, exponential functions

XSIMD

- C++ wrappers for SIMD intrinsics from "QuantStack". Include only. BSD-3-Clause license.
  `git clone https://github.com/QuantStack/xsimd.git`
- Abstractions for batches of values for SIMD calculations, e.g.,
  `xsimd::batch<double, xsimd::avx2>`

```
using Arch = xsimd::avx2;
xsimd::batch<double, Arch> x{1.,2.,3.,4.}, y{4.,3.,2.,1.};
std::cout << x + y << "\n";
```

- The second template argument can be left out. The default value: target architecture
- Vectorized forms of commonly used mathematical functions, such as trigonometric, exponential functions, error functions, e.g., `xsimd::asin(xsimd::batch<double, Arch>)`, `xsimd::exp(xsimd::batch<double, Arch>)`
- Regular arithmatic operations along with fma functions, e.g., `xsimd::fma(a, x, y)`
- Auto-detection and parametrisation based on available instruction set, e.g., based on vector width, `xsimd::batch<double, xsimd::avx2>`
- Aligned allocator:

```
template <class T> using myvector = std::vector<T, xsimd::aligned_allocator<T>>;
myvector<double> V(1000000, 1.2); // Aligned to cache line
```

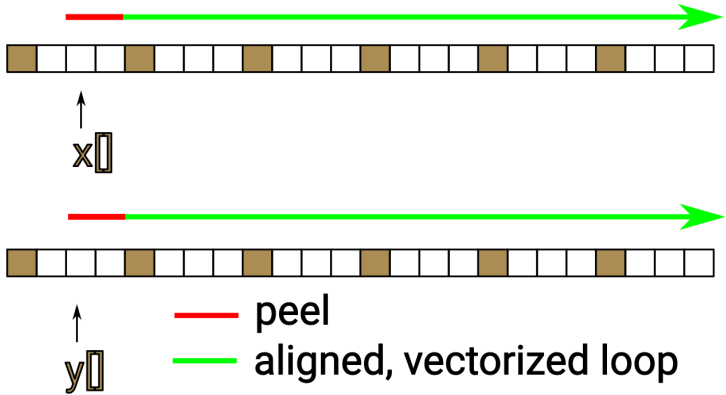
XSIMD

- Useful to write with a placeholder tag type `Arch`, to allow runtime architecture selection
- We will be using an alias `using btype = xsimd::batch<double, Arch>` in the following. Depending on the architecture, it may represent a batch of 2, 4 or 8 `double` values
- To load from an address in memory `xptr`, use `auto xb = btype::load_unaligned(xptr)`. If you know that the address is properly aligned for the batch, you can use `auto xb = btype::load_aligned(xptr)`. You can not load from an unaligned address using `load_aligned`.
- Loading can be controlled using a tag type: `auto xb = btype::load(xptr, alignment_tag)`, where `alignment_tag` is an object of one of the tag types `xsimd::aligned_mode` or `xsimd::unaligned_mode`.
- You can broadcast a scalar value to all positions in a SIMD batch like this: `auto ab = btype::broadcast(a);`.
- Batch objects can be combined using arithmetic operators, used in XSIMD mathematical functions etc to produce other batch objects
- To store the result to a location in memory, use the appropriate member function: `xb.store_unaligned(xptr)`, `xb.store_aligned(xptr)` or `xb.store(xptr, alignment_tag)`.

XSIMD

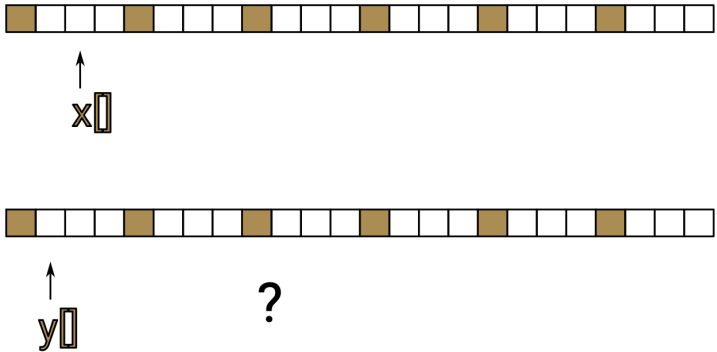
```
1  void daxpy(double a, std::span<const double> x, std::span<const double> y,
2             std::span<double> res) {
3      for (size_t i = 0UL; i < x.size(); ++i) {
4         res[i] = a * x[i] + y[i];
5      }
6  }
7  void daxpy_xsimd(double a, std::span<const double> x, std::span<const double> y,
8                   std::span<double> res) {
9      using btype = batch<double>;
10     constexpr auto vwidth = btype::size;
11     const auto ablk = btype::broadcast(a);
12     const auto vreps = x.size() - x.size() % vwidth;
13     for (size_t i = 0UL; i < vreps; i += vwidth) {
14         auto xblk = btype::load_unaligned(&x[i]);
15         auto yblk = btype::load_unaligned(&y[i]);
16         auto zblk = a * xblk + yblk;
17         zblk.store_unaligned(&res[i]);
18     }
19     for (size_t i = vreps; i < x.size(); ++i) { res[i] = a * x[i] + y[i]; }
20 }
```

Alignment and SIMD operations



Peel a few from the front and start aligned loads...

Alignment and SIMD operations



How many elements would you peel now ?

Alignment and SIMD operations

SIMD with complex numbers

- `std::complex<T>` has a fixed data layout, (*real*, *imag*) to be compatible with C
- Arrays of complex numbers have the real parts at non-adjacent, but statically predictable, locations (same applies to the imaginary parts)
- Many ways to code vectorized operations on complex numbers
- XSIMD (`batch<complex<double>, Arch>`) has abstractions for working with complex numbers
- Without such abstractions to aid us, explicit SIMD programming with complex number would be needlessly complicated

```
1  #include <xsimd/xsimd.hpp>
2  #include <complex>
3  #include <vector>
4  using namespace std;
5  void caxpy_xsimd(complex<double> a,
6          span<complex<double>> x,
7          span<const complex<double>> y)
8  {
9      using b_type =
10         xsimd::batch<complex<double>>;
11     b_type c = b_type::broadcast(a);
12     b_type xl, yl;
13     for (size_t i=0; i<x.size();
14                          i+=b_type::size) {
15         xl.load_unaligned(&x[i]);
16         yl.load_unaligned(&y[i]);
17         xl = c * xl + yl;
18         xl.store_unaligned(&x[i]);
19     }
20 }
```

XSIMD: architectures and dispatching

- It is possible to write programs for multiple architectures
- An appropriate instruction set is chosen based on architectures available at runtime
- Architecture adapted ("dispatched") functions are generated using `xsimd::dispatch()`
- Recipe:
  - Implement the function for a task as a functional with a template call operator
  - The template parameter `Arch` for the call operator serves the same purpose as our placeholder in the examples so far.
  - Generate a dispatched function using `xsimd::dispatch(functional)`
  - Use the return value of the dispatch function as a callable object with a signature without the `Arch` parameter.

```
1  struct daxpy_xsimd_t {
2    template <class Arch>
3    void operator()(Arch,
4        std::span<const double> x,
5        std::span<double> y,
6        double a) const
7    {
8      using b_type = xsimd::batch<double, Arch>;
9      b_type bx{}, by{};
10     const b_type ba{b_type::broadcast(a)};
11     // and so on...
12   }
13 };
14 inline auto daxpy_xsimd
15     = xsimd::dispatch(daxpy_xsimd_t{});
16 void elsewhere()
17 {
18   std::vector a(100UL, 4.3);
19   std::vector b(100UL, 3.2);
20   daxpy_xsimd(a, b, 8.0);
21 }
```

Exercise 2.19:

In the folder `examples/SIMD`, you will find several versions of a few short functions.

- Many examples here are not full programs and do not have `main` function.
- The DIY version does not require any libraries to compile, although it does need `immintrin.h`, which should be found in your system
- You should compile with the best available instruction set on your system (`-march=native` for GCC and Clang) and with optimization for speed
- The examples with XSIMD are in the next exercise.

Exercise 2.20:

`examples/SIMD/xsimd`: XSIMD demos

- For the examples with XSIMD, you will need to pass `-I /path/to/xsimdroot/include` to the compiler. There is nothing to link.

Example compile command :

```
$ g++ -std=c++23 -O3 -march=native -I $XSIMD_INCLUDE_DIR exvol1.cc -o exvol1.g
```

The examples `x0.cc`, `x1.cc` and `x2.cc` show the basic syntax. `x0.cc` shows an explicitly set architecture. But `x1.cc` and `x2.cc` use the default batch. Build them with and without `-march=native` and run them, to understand the role of the compiler option. The examples `exvol1.cc` and `daxpy1.cc` demonstrate architecture dispatching. The remaining two examples, `brighten.cc` and `nn_relu.cc` show two tiny applications: `brighten.cc` brightens the pixels of an image. The `nn_relu.cc` shows a single layer feed forward neural network with a reLU activation function.

Exercise 2.21:

The folder `examples/SIMD/stdx_simd` contains the corresponding implementation for the programs in the XSIMD exercise, implemented using the proposed C++ standard library SIMD functionality. This is currently part of the so called parrallelism TS-2, and not really standard. But, there are partial implementations in both GCC and Clang. The README file in the directory contains lots of comments about the programs. Learn how to use SIMD functionality from `std::experimental` namespace using the files in this example.

Summary

- Directly coding with SIMD types exposes algorithmic challenges concerning vectorization
- We are much more directly in control
- Quite often, correctly done OpenMP will bring you most of the benefits, but, knowing how to work with intrinsics gives you a fallback option when the simple approach fails. At the very least, when you try to vectorize yourself, you might see why OpenMP didn't do as you had hoped.
- If you work with C++, use its strengths: strive for zero-overhead abstractions instead of resigning to a life of verbose and error-prone misery
- Alternatively, use a SIMD library with a compatible license
  - They aleady exist, and others have already created the necessary abstractions
  - They support multiple instruction sets and CPU architectures
  - Often come with vectorized versions of common mathematical functions

Chapter 3

Lessons from matrix multiplication

Lessons from matrix multiplication

Exercise 4.1:

In the examples folder, you will find a `MatMul` subfolder, containing a written lesson called `SessionMatrix.pdf`. This file contains 8 stages organised as exercises starting with a naive implementation of a matrix type in C++, and ending with something with reasonably respectable performance (comparable to what is possible with, e.g., Eigen, or other BLAS libraries) on a single node on JUSUF. It only uses concepts introduced in this course, and does not call any linear algebra library function. Work through the exercises and test the different stages on JUSUF!

Chapter 4

Parallelisation using PSTL and TBB

Parallel computing

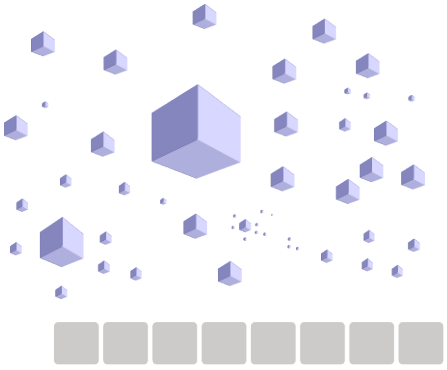
- Engineering (power consumption) challenges make processors with higher and higher clock rates impractical
- Computers in the last 20 years have instead increased processing power by adding more hardware for parallel processing

Parallel computing

```
1  auto gcd(unsigned s, unsigned l) -> unsigned
2  {
3      if (s > l)
4          std::swap(s, l);
5      while (s != 0) {
6          auto r = l % s;
7          l = s;
8          s = r;
9      }
10     return l;
11 }
```


- Engineering (power consumption) challenges make processors with higher and higher clock rates impractical
- Computers in the last 20 years have instead increased processing power by adding more hardware for parallel processing
- A sequence of dependent operations on a small set of entities is ill-suited for processing with many workers

Parallel computing



- Engineering (power consumption) challenges make processors with higher and higher clock rates impractical
- Computers in the last 20 years have instead increased processing power by adding more hardware for parallel processing
- A sequence of dependent operations on a small set of entities is ill-suited for processing with many workers
- Given a large amount of information to be processed, or a task with a large number of independent sub-tasks, it is possible to reduce the overall processing time.

Parallel computing

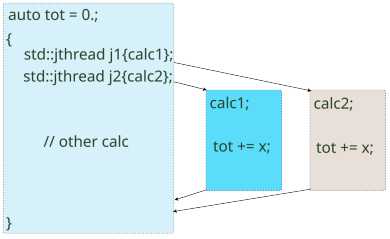
What mechanisms do we have in C++ to exploit available parallelism in hardware?

- Threads, mutexes, atomic operations
- RAII for resource management
- Libraries to partition and assign work to workers
- Templates, lambda functions, CTAD
- High-level STL style algorithms abstracting common programming building blocks
- Containers and allocators for more efficient (and corrrect) parallel processing

Threads

```
1  auto calc1 = [=]() {
2      auto tot1 = 0.;
3      for (auto i = 0UL; i < N; ++i) {
4          auto ang = 2 * i * pi / N;
5          tot1 += std::cos(ang) * std::cos(ang);
6      }
7  };
8  auto calc2 = [=]() {
9      auto tot1 = 0.;
10     for (auto i = 0UL; i < N; ++i) {
11         auto ang = 2 * i * pi / N;
12         tot1 += std::sin(ang) * std::sin(ang);
13     }
14 };
15 std::jthread j1 { calc1 };
16 std::jthread j2 { calc2 };
```

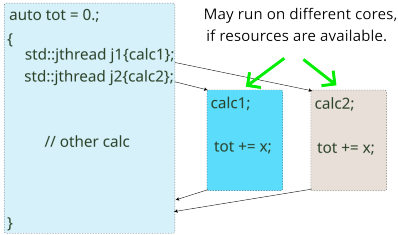
- `std::thread`, `std::async` ... since C++11
- Parallel algorithms since C++17
- `std::jthread`, `std::stop_token` since C++20
- `std::jthread` joins in the destructor



Threads

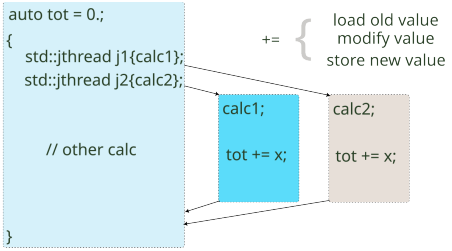
```
1  auto calc1 = [=]() {
2      auto tot1 = 0.;
3      for (auto i = 0UL; i < N; ++i) {
4          auto ang = 2 * i * pi / N;
5          tot1 += std::cos(ang) * std::cos(ang);
6      }
7  };
8  auto calc2 = [=]() {
9      auto tot1 = 0.;
10     for (auto i = 0UL; i < N; ++i) {
11         auto ang = 2 * i * pi / N;
12         tot1 += std::sin(ang) * std::sin(ang);
13     }
14 };
15 std::jthread j1 { calc1 };
16 std::jthread j2 { calc2 };
```

- `std::thread`, `std::async` ... since C++11
- Parallel algorithms since C++17
- `std::jthread`, `std::stop_token` since C++20
- `std::jthread` joins in the destructor



Threads

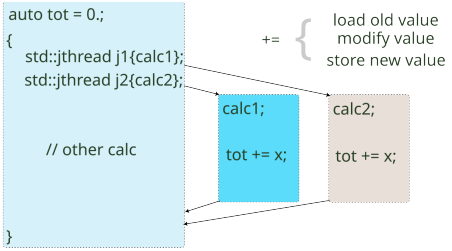
```
1  auto tot = 0.;
2  {
3      std::jthread j1 { [&]() {
4          for (auto i = 0UL; i < N; ++i) {
5              auto ang = 2 * i * pi / N;
6              tot += std::cos(ang) * std::cos(ang);
7          }
8      } };
9      std::jthread j2 { [&]() {
10         for (auto i = 0UL; i < N; ++i) {
11             auto ang = 2 * i * pi / N;
12             tot += std::sin(ang) * std::sin(ang);
13         }
14     } };
15 }
16 std::cout << "Total " << tot << "\n";
```



- Modification of data at the same address from multiple threads can lead to “data races”

Threads

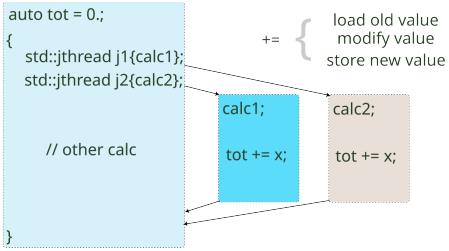
```
1  auto tot = 0.;
2  {
3      std::jthread j1 { [&]() {
4          for (auto i = 0UL; i < N; ++i) {
5              auto ang = 2 * i * pi / N;
6              tot += std::cos(ang) * std::cos(ang);
7          }
8      } };
9      std::jthread j2 { [&]() {
10         for (auto i = 0UL; i < N; ++i) {
11             auto ang = 2 * i * pi / N;
12             tot += std::sin(ang) * std::sin(ang);
13         }
14     } };
15 }
16 std::cout << "Total " << tot << "\n";
```



- The result can be incorrect, since the load-modify-commit operations from the two threads can overlap

Threads

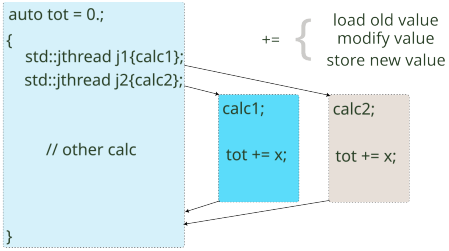
```
1   std::mutex totmutex;
2   {
3       std::jthread j1 { [&]() {
4           for (auto i = 0UL; i < N; ++i) {
5               auto ang = 2 * i * pi / N;
6               std::scoped_lock lck { totmutex };
7               tot += std::cos(ang) * std::cos(ang);
8           }
9       } };
10      std::jthread j2 { [&]() {
11          for (auto i = 0UL; i < N; ++i) {
12              auto ang = 2 * i * pi / N;
13              std::scoped_lock lck { totmutex };
14              tot += std::sin(ang) * std::sin(ang);
15          }
16      } };
17  }
18  std::cout << "Total " << tot << "\n";
```



- Fix 1: `std::mutex`: A resource which can be acquired by only one thread at a time. Must be released by the acquiring thread. `std::scoped_lock` manages mutex acquisition/release using RAII

Threads

```
1  std::atomic<double> tot {};
2  {
3      std::jthread j1 { [&]() {
4          for (auto i = 0UL; i < N; ++i) {
5              auto ang = 2 * i * pi / N;
6              tot += std::cos(ang) * std::cos(ang);
7          }
8      } };
9      std::jthread j2 { [&]() {
10         for (auto i = 0UL; i < N; ++i) {
11             auto ang = 2 * i * pi / N;
12             tot += std::sin(ang) * std::sin(ang);
13         }
14     } };
15 }
16 std::cout << "Total " << tot << "\n";
```



- `std::atomic<T>` gives us "atomic" load-modify-commit operations

Threads

```
1  struct wrapped1 {
2      int val {};
3  };
4  template <class W>
5  struct func {
6  void operator()(volatile W* var)
7  {
8      for (unsigned i = 0; i < WORKLOAD / PARALLEL; ++i) {
9          var->val = var->val + 1;
10     }
11 }
12 };
13 {
14    std::array<wrapped2, PARALLEL> arr {};
15    {
16        std::array<std::jthread, PARALLEL> threads;
17        for (unsigned i = 0U; i < PARALLEL; ++i) {
18            threads[i] =
19            std::jthread(func<wrapped2>{}, &arr[i]);
20        }
21    }
22 }
```

- Even when threads write to different addresses, there can be a significant slowdown because of “false sharing”

Threads

```
1  struct align_as(std::hardware_destructive_interference_size)
2  wrapped1 {
3      int val {};
4  };
5  template <class W>
6  struct func {
7  void operator()(volatile W* var)
8  {
9      for (unsigned i = 0; i < WORKLOAD / PARALLEL; ++i) {
10         var->val = var->val + 1;
11     }
12 }
13 };
14 {
15    std::array<wrapped2, PARALLEL> arr {};
16    {
17        std::array<std::jthread, PARALLEL> threads;
18        for (unsigned i = 0U; i < PARALLEL; ++i) {
19            threads[i] =
20            std::jthread(func<wrapped2>{}, &arr[i]);
21        }
22    }
23 }
```

- Even when threads write to different addresses, there can be a significant slowdown because of “false sharing”
- Mitigation: alignment or padding

Parallel STL

- Parallel versions of the high-level building blocks such as `std::sort`, `std::reduce` etc.
- C++17 parallel STL provides a way to express that something can be done in parallel, but does not mandate implementation strategy
- Programs already written using algorithms will offer many opportunities for exploiting parallelism
- A TBB based implementation is used since GCC 9.1. Intel and Microsoft compilers have their implementations as well.
- `std::sort` sorts.
  `std::sort(std::execution::par, ...)` sorts in parallel
- `std::reduce` adds up elements from a range.
  `std::reduce(std::execution::par, ...)` adds up elements in parallel

```
1  std::sort(std::execution::par,
2            points.begin(), points.end(),
3            [](auto p1, auto p2) {
4                return p1.x() < p2.x();
5            });
6  std::for_each(std::execution::par_unseq,
7                points.begin(), points.end(),
8                [](auto & p) {
9                    p.norm(1);
10               });
```

- As of GCC 15.2, to compile programs using parallel algorithms, we need to link with `libtbb` and `libtbbmalloc`, e.g.,
  `G par_user.cc -ltbb -ltbbmalloc`
- As of Clang 19.1, parallel STL remains an experimental feature in `libc++`, and must be enabled through `-fexperimental-library`

Execution policies

- `std::execution::sequenced_policy` : Parallel algorithm's execution may not be parallelised. Element wise operations are indeterminately sequenced in the calling thread. An instance called, `std::execution::seq` is usually used to disambiguate overload resolution
- `std::execution::parallel_policy` : May be parallelised. Element wise operations can happen in the calling thread, or on another. Relative sequencing is indeterminate. Convenience instance: `std::execution::par`
- `std::execution::parallel_unsequenced_policy` May be parallelised and vectorised. Element wise operations can run in unspecified threads, and can be unordered in each thread. `std::execution::par_unseq`
- `std::execution::unsequenced_policy` Only vectorised. `std::execution::unseq`

Parallel STL examples

Exercise 5.1:

The program `examples/pstl/inner_product.cc` demonstrates the use of the parallel STL library, performing a simple inner product calculation. Use `-ltbb -ltbbmalloc` for linking, or use the `CMake` file in the directory.

Exercise 5.2:

The program `examples/pstl/transform_reduce.cc` creates a vector of random points in 2D, and then calculates the moment of inertia using STL algorithms. Just switching the execution policy parameter, the ptogram can be parallelised and vectorised. Test!

Parallel STL examples

Exercise 5.3:

Parallelise the program `exercises/pstl/mandelbrot0.cc` using parallel STL.

Exercise 5.4:

At what size of a group of random strangers does the chance of two people sharing a birthday become greater than 0.5? The program `birthday_problem.cc` solves it using a crude, brute force Monte Carlo simulation. Parallelise it using parallel STL.

Examples in this section can be done with both GCC and Clang, with some caveats when using Clang.

```
clang++ -std=c++23 -stdlib=libc++ -fexpermental-library -O3 -march=native ___.cc
```

and

```
clang++ -std=c++23 -stdlib=libstdc++ -O3 -march=native ___.cc
```

will both will work. As of October 2025, `libc++` hasn't optimised performance when using parallel algorithms.

TBB: Threading Building Blocks I

- Provides utilities like `parallel_for`, `parallel_reduce` to simplify the most commonly used structures in parallel programs
- Provides scalable concurrent containers such as vectors, hash tables and queues for use in multi-threaded environments
- No direct support for vector parallelism. But can be combined with auto-parallelisation and `#pragma omp simd` etc or explicit SIMD with a SIMD library
- Supports complex models such as pipelines, data flow and unstructured task graphs
- Scalable memory allocation, avoidance of false sharing, thread local storage
- Low level synchronisation tools like mutexes and atomics
- Work stealing task scheduler
- `http://www.threadingbuildingblocks.org`
- Structured Parallel Programming, Michael McCool, Arch D. Robinson, James Reinders

Using TBB

- Public names are available under the namespaces `tbb` and `tbb::flow`
- You indicate "available parallelism", scheduler may run it in parallel if resources are available
- Unnecessary parallelism will be ignored

parallel invoke

```
1  void prep(Population &p);
2  void iomanage();
3  tbb::parallel_invoke(
4     [&] {
5         noise_w(0., pars.sigma, wns);
6         std::copy(wns.begin(), wns.end(), wnoisemat.begin());
7     },
8     [&] {
9         noise_phi(0., pars.sigma, phins);
10        std::copy(phins.begin(), phins.end(), phinoisemat.begin());
11    });
```

Exercise 5.5: examples/tbb/parallel_invoke.cc

Compile with

```
G parallel_invoke.cc -ltbb -ltbbmalloc
```

- A few adhoc tasks which do not depend on each other
- Runs them in parallel
- waits until all of them are finished

TBB task groups

```
1  struct Equation {
2      void solve();
3  };
4
5  std::list<Equation> equations;
6  tbb::task_group g;
7  for (auto eq : equations)
8      g.run([]{eq.solve();});
9
10 g.wait();
```

- Run an arbitrary number of callable objects in parallel
- In case an exception is thrown, the task group is cancelled

TBB task arena

```
1  auto main(int argc, char *argv[]) -> int
2  {
3      size_t nthreads=std::stoul(argv[1]);
4      tbb::task_arena main_executor;
5      main_executor.initialize(nthreads);
6      main_executor.execute([&]{
7          haha();
8      });
9  }
10 void haha()
11 {
12     ...
13     tbb::parallel_invoke(a,b,c,d,e);
14 }
15 void a()
16 {
17     tbb::parallel_for(...);
18 }
```

- Task arena to manage tasks, maps them to threads etc.
- Number of threads in an arena limited by its concurrency level
- Execute function, with a function object as argument.
- Returns the same thing as the function it is executing.

Parallel for loops

- Template function modelled after the `for` loops, like many STL algorithms
- Takes a callable object as the third argument
- Using lambda functions, you can expose parallelism in sections of your code

```
1  tbb::parallel_for(first,last,f);
2  // parallel equivalent of
3  // for (auto i=first;i<last;++i) f(i);
4
5  tbb::parallel_for(first,last,stride,f);
6  // parallel equivalent of
7  // for (auto i=first;i<last;i+=stride)
8  //     f(i);
9
10 tbb::parallel_for(first,last,
11                   [captures](anything){
12     //Code that can run in parallel
13 });
14
```

Parallel for with ranges

- Splits `range` into smaller ranges, and applies `f` to them in parallel
- Possible to optimize `f` for sub-ranges rather than a single index
- Any type satisfying a few design conditions can be used as a range
- Multidimensional ranges possible

```
1  tbb::parallel_for(0,1000000,f);
2  // One parallel invocation for each i!
3  tbb::parallel_for(range,f);
4
5  // A type R can be a range if the
6  // following are available
7  R::R(const R &);
8  R::~R();
9  bool R::is_divisible() const;
10 bool R::empty() const;
11 R::R(R & r,split);//Split constructor
```

Parallel for with ranges

```
1  tbb::blocked_range<int> r{0,30,20};
2  assert(r.is_divisible());
3  blocked_range<int> s{r};
4  //Splitting constructor
5  assert(!r.is_divisible());
6  assert(!s.is_divisible());
7
```

- `tbb::blocked_range<int>(0,4)` represents an integer range 0..4
- `tbb::blocked_range<int>(0,50,30)` represents two ranges, 0..25 and 26..50
  - So long as the size of the range is bigger than the "grain size" (third argument), the range is split

Parallel for with ranges

```
1  void daxpcy_tbb(double a, std::span<const double> x, std::span<double> y) {
2      tbb::parallel_for(tbb::blocked_range<int>(0, x.size()),
3                        [&](tbb::blocked_range<int> r) {
4          for (size_t i = r.begin(); i != r.end(); ++i) {
5              y[i] = a * sin(x[i]) + cos(y[i]);
6          }
7      });
8  }
```

- `parallel_for` with a range uses split constructor to split the range as far as possible, and then calls f(range), where `f` is the functional given to `parallel_for`
- It is unlikely that you wrote your useful functions with ranges compatible with `parallel_for` as arguments
- But with lambda functions, it is easy to fit the parts!

Exercise 5.6: TBB parallel for demo

The program `examples/dasxpcy.cc` demonstrates the use of parallel for in TBB. It is a slightly modified version of the commonly used DAXPY demos. Instead of calculating $y = a * x + y$ for scalar a and large vectors x and y, we calculate $y = a * sin(x) + cos(y)$. To compile, you need to load your compiler and TBB modules, and use them like this:

```
1  G dasxpcy.cc -ltbb -ltbbmalloc
```

2D ranges

```
1  void f(size_t i, size_t j);
2  tbb::blocked_range2d<size_t> r{0, N, 0, N};
3  tbb::parallel_for(r, [&](tbb::blocked_range2d<size_t> r) {
4      for (auto i = r.rows().begin(); i != r.rows().end(); ++i) {
5          for (auto j = r.cols().begin(); j != r.cols().end(); ++j) {
6              f(i, j);
7          }
8      }
9  });
```

- `rows()` is an object with a `begin()` and an `end()` returning just the integer row values in the range. Similarly: `cols()` ...
- 2D range can also be split
- The callable object argument should assume that the original 2D range has been split many times, and we are operating on a smaller range, whose properties can be accessed with these functions.

Parallel reductions with ranges

```
1  T result = tbb::parallel_reduce(range, identity, subrange_reduction, combine);
```

- `range` : As with parallel for
- `identity` : Identity element of type T. The type determines the type used to accumulate the result
- `subrange_reduction` : Functor taking a "subrange" and an initial value, returning reduction
- `combine` : Functor taking two arguments of type T and returning reduction over them over the subrange. Must be associative, but not necessarily commutative.

Parallel reduce with ranges

```
1  double inner_prod_tbb(std::span<const double> x, std::span<double> y) {
2      return tbb::parallel_reduce(
3            tbb::blocked_range<int>(0, n), // range
4            double{}, // identity
5            [&](tbb::blocked_range<int> &r, float in){
6                return std::inner_product(x.begin() + r.begin(), x.begin() + r.end(),
7                                          y.begin() + r.begin(), in);
8            }, // subrange reduction
9            std::plus<double>{} // combine
10     );
11 }
```

- With TBB ranges, we can use blocked implementations with hopefully vectorisable calculations in subranges
- Two functors are required, either of which could be lambda functions
- Important to add the contribution of initial value in subrange reductions

Exercise 5.7: TBB parallel reduce

The program `tbbreduce.cc` is a demo program to calculate an integral using `tbb::parallel_reduce`
What kind of speed up do you see relative to the serial version ? Does it make sense considering the number of physical cores in your computer ?

Atomic variables

- "Instantaneous" updates
- Lock-free synchronization
- For `std::atomic<T>`, T can be integral, enum or pointer type, and since C++20, also floating point, `std::shared_ptr` and `std::weak_ptr`
- If `index.load() == k` simultaneous calls to `index++` by n threads will increase `index` to `k + n`. Each thread will use a distinct value between `k` and `k + n`

```
1  std::array<double, N> A;
2  std::atomic<int> index;
3
4  void append(double val)
5  {
6      A[index++] = val;
7  }
```

Atomic variables

- "Instantaneous" updates
- Lock-free synchronization
- For `std::atomic<T>`, T can be integral, enum or pointer type, and since C++20, also floating point, `std::shared_ptr` and `std::weak_ptr`
- If `index.load() == k` simultaneous calls to `index++` by n threads will increase `index` to `k + n`. Each thread will use a distinct value between `k` and `k + n`

```
1  std::array<double, N> A;
2  std::atomic<int> index;
3
4  void append(double val)
5  {
6      A[index++] = val;
7  }
```

But it is important that we use the return value of `index++` in the threads!

Enumerable thread specific

```
1  tbb::enumerable_thread_specific<double> E;
2  double Eglob=0;
3  double f(size_t i, size_t j);
4  tbb::blocked_range2d<size_t> r{0, N, 0, N};
5  tbb::parallel_for(r, [&](tbb::blocked_range2d<size_t> r){
6      auto & eloc = E.local();
7      for (size_t i = r.rows().begin(); i != r.rows().end(); ++i) {
8          for (size_t j = r.cols().begin();j != r.cols().end(); ++j) {
9              if (j > i) eloc += f(i,j);
10         }
11     }
12 });
13 Eglob = 0;
14 for (auto& v : E) {Eglob += v; v = 0;}
```

- Thread local "views" of a variable
- behaves like an STL container of those views
- Member function `local()` gives a reference to the local view in the current thread
- Any thread can access all views by treating it as an STL container

TBB allocators

- Dynamic memory allocation in a multithreaded program must avoid conflicts from `new` calls from different threads
- Global memory lock

TBB allocators

- Interface like `std::allocator`, so that it can be used with STL containers. E.g., `std::vector<T, tbb::cache_aligned_allocator<T>>`
- `tbb::scalable_allocator<T>` : general purpose scalable allocator type, for rapid allocation from multiple threads
- `tbb::cache_aligned_allocator<T>` : Allocates with cache line alignment. As a consequence, objects allocated in different threads are guaranteed to be in different cache lines.

Concurrent containers

```
1  #include <tbb/concurrent_vector.h>
2
3  auto v = tbb::concurrent_vector<int>(N, 0);
4
5  tbb::parallel_for(v.range(), [&](tbb::concurrent_vector::range_type r) {
6      //...
7  });
```

- Random access by index
- Multiple threads can grow container and add elements concurrently
- Growing the container does not invalidate any iterators or indexes
- Has a `range()` member function for use with `parallel_for` etc.

Chapter 5

Linear algebra with Eigen

Linear algebra

- Operations on matrices, vectors, linear systems etc.
- Data parallel, simple numerical calculations
- Can be hand coded, but taking proper account of available CPU instructions, memory hierarchy etc is hard
- Libraries with standardized syntax for wide applicability
- Excellent vendor libraries are available on HPC systems

Eigen: A C++ template library for linear algebra

- Include only library. Download from `http://eigen.tuxfamily.org/`, unpack in a location of your choice, and use. Nothing to link.
- Small fixed size to large dense/sparse matrices
- Matrix operations, numerical solvers, tensors ...
- Expression templates: lazy evaluation, smart removal of temporaries

```cpp
// examples/Eigen/eigen1.cc
#include <iostream>
#include <Eigen/Dense>
using namespace Eigen;
using namespace std;
int main()
{
  MatrixXd m=MatrixXd::Random(3,3);
  m = (m + MatrixXd::Constant(3, 3, 1.2)) * 50;
  cout << "m =" << "\n" << m << "\n";
  VectorXd v(3);
  v << 1, 2, 3;
  cout << "m * v =" << "\n" << m * v << "\n";
}
```

`G eigen1.cc`

- Explicit vectorization
- Elegant API

Eigen: matrix types

- `MatirxXd` : matrix of arbitrary dimensions
- `Matrix3d` : fixed size 3×3 matrix
- `Vector3d` : fixed size 3d vector
- Element access `m(i,j)`
- Output `std::cout << m << "\n";`
- Constant : `MatrixXd::Constant(a,b,c)`
- Random : `MatrixXd::Random(n,n)`
- Products : `m * v` or `m1 * m2`
- Expressions : `3 * m * m * v1 + u * v2 + m * m * m * v3`
- Column major matrix : `Matrix<float, 3, 10, Eigen::ColMajor>`

Eigen: matrix operations

```cpp
#include <iostream>
#include <Eigen/Dense>
using namespace Eigen;
auto main() -> int {
   Matrix3f A;
   Vector3f b;
   A << 1,2,3,  4,5,6,  7,8,10;
   b << 3, 3, 4;
   std::cout << "Here is the matrix A:\n" << A << "\n";
   std::cout << "Here is the vector b:\n" << b << "\n";
   Vector3f x = A.colPivHouseholderQr().solve(b);
   std::cout << "The solution is:\n" << x << "\n";
}
```

- Blocks `m.block(start_r, start_c, nr, nc)`, or `m.block<nr,nc>(start_r, start_c)`

```cpp
   SelfAdjointEigenSolver<Matrix2f> eigensolver(A);
   if (eigensolver.info() != Success) abort();
   std::cout << "Eigenvalues " << eigensolver.eigenvalues() << "\n";
```

Eigen: examples

Exercise 6.1:

There are a few example programs using Eigen in the folder `examples/Eigen`. Read the programs `eigen0.cc` and `eigen1.cc`. To compile, use `G program.cc`.

Exercise 6.2:

The folder `examples/Eigen` contains a matrix multiplication example, `matmul.cc` using Eigen. Compare with a naive version of a matrix multiplication program, `matmul_naive.cc`, by compiling and running both programs. Try different matrix sizes. Then, you can use a parallel version of the Eigen matrix multiplication by recompiling with `-fopenmp`.

Exercise 6.3:

The file `exercises/PCA` has a data file with tabular data. Each column represents all measurements of a particular type, while each row is a different trial. In each row, the first column, x_{i0}, represents a pseudo-time variable. Write a program using Eigen to perform a Principal Component Analysis on this data set, ignoring the first column. Hint:
if $X_i = [x_{i1}, x_{i2}, ...x_{im}]$ is the data of row i, the covariance matrix is defined as,

$$C_{ab} = \frac{1}{(n-1)} \sum_k x_{ka} x_{kb}$$

The principal components of the data are obtained by right multiplying the data matrix by the matrix whose columns are the eigen vectors of the matrix C_{ab}, conventionally ordered by decreasing eigenvalues.

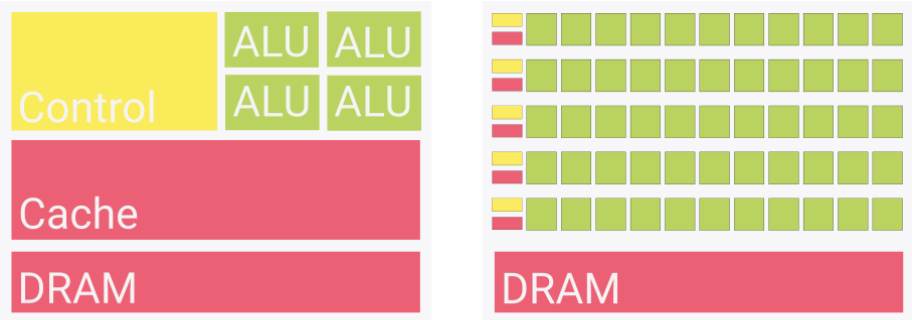
Chapter 6

GPU programming

Data parallelism

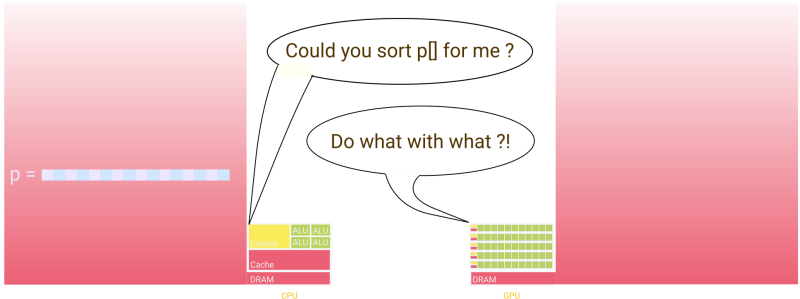
Bus/motorcycle analogy and figure stolen from GPU course/lecture slides by Andreas Herten (JSC)

Priorities



- CPU: faster clock speed, more cache, more sophisticated instructions and scheduling
- GPU: More chip area dedicated to floating point computations

A separate device



- Separate memories. GPU does not automatically know the state of any object in the memory of the CPU.
- Must transfer data.
- Must tell what to do with the data.
- Must retrieve results with another information transfer.

Can run C++ functions

- A program running on a CPU can call special functions designed to run on the GPU
- The GPU understands a different set of hardware instructions than the CPU, so any human readable function meant for the GPU must be compiled to a different kind of hardware instructions than code compiled for the CPU.
- A set of function "execution space specifiers" are provided as language extensions : `__global__`, `__device__` and `__host__`. These indicate to a CUDA aware compiler which parts to translate to the CPU language and which parts to the GPU language.
- A function running on the GPU can call other functions compiled for the GPU, leading to a call tree on the device side.

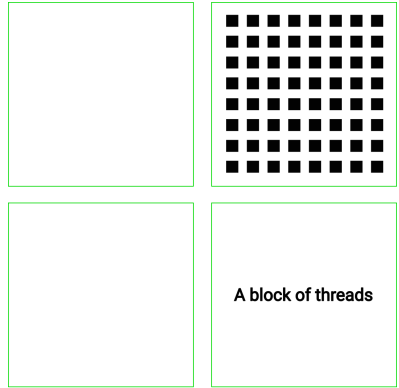
```
1  __device__ auto shuf(int id)
2  {
3      return (id + 1723) % 2000;
4  }
5  __global__
6  void gpufunc(int *ids, unsigned N)
7  {
8      // ...
9      ids[i] = shuf(ids[i]);
10     // ...
11 }
12 auto cpufunc() -> int
13 {
14     gpufunc<<<1, 100>>>(p, 3000);
15 }
16
```

Execution space specifiers

- `__device__` : the function runs on the device, and it can only be called from the device
- `__host__` : the function runs on the host, and it can only be called form the host
- `__global__`: the function is a "kernel". It runs on the device, and can be called from the host, or from device (compute capability >= 3.2)
  - Must have `void` return type
  - Can not be a member function
  - It is asynchronous : the function returns before the device performs its work
  - Must be called along with an "execution configuration" e.g., `gpufunc<<<1, 100>>>(p, 3000)`
- `__device__` and `__host__` can both be used for a function, in which case, it is compiled for both the host and the device.

Kernel call syntax

- Kernel functions are called with the `<<<GridSpec, BlockSpec>>>` notation, i.e., potentially in a large number of threads, arranged in blocks
- `BlockSpec` denotes a 3 dimensional object, 3 integers, specifying the arrangement of threads in a thread block
- `GridSpec` denotes a 3 dimensional object, 3 integers, specifying how blocks are arranged in a grid
- Each thread running a kernel function has a built in variable, `threadIdx`, specifying the position of the thread in its block, and another variable `blockIdx` to identify the block in the grid, and `blockDim` = number of threads in a block
- Overall x index: `blockIdx.x * blockDim.x + threadIdx.x` etc.



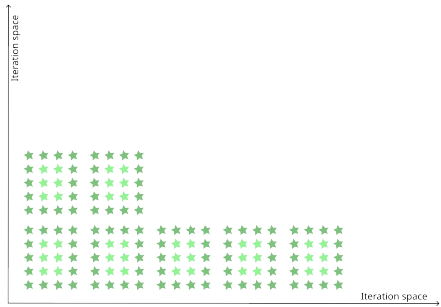
Kernel call syntax

```
1  __global__ void MatAdd(float A[N][N], float B[N][N],
2                         float C[N][N])
3  {
4      int i = blockIdx.x * blockDim.x + threadIdx.x;
5      int j = blockIdx.y * blockDim.y + threadIdx.y;
6      if (i < N && j < N)
7          C[i][j] = A[i][j] + B[i][j];
8  }
9
10 auto main() -> int
11 {
12     ...
13     // Kernel invocation
14     dim3 threadsPerBlock{16, 16};
15     dim3 numBlocks{N / threadsPerBlock.x,
16                    N / threadsPerBlock.y};
17     MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
18     ...
19 }
```

- The block and grid properties are often chosen to reflect properties of the problem being solved.
- In this example, the threads are organized in a 2D lattice: a natural fit for a matrix sum
- Each thread only needs to process one element!
- There is a maximum number of threads allowed in a block: a limit coming from hardware properties
- It is therefore necessary to arrange blocks into a grid

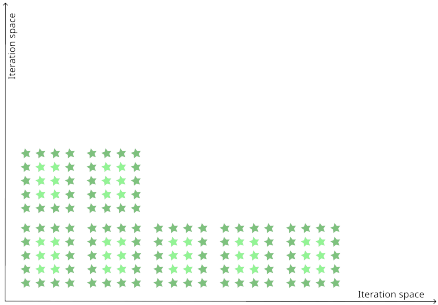
Kernel call syntax

- Remember how you had to process an array of double, 4 elements at a time and a stride of 4, when using AVX style SIMD register?



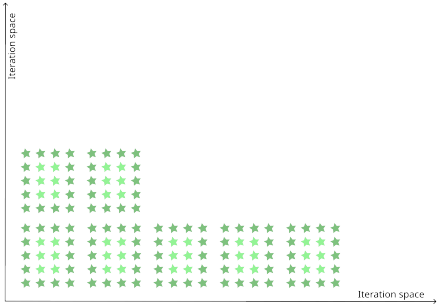
Kernel call syntax

- Remember how you had to process an array of double, 4 elements at a time and a stride of 4, when using AVX style SIMD register?
- Think of a block in CUDA as a potentially 3-dimensional stencil of tiny computations which you repeat to cover the iteration space



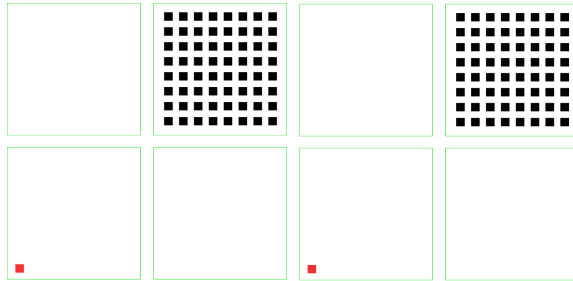
Kernel call syntax

- Remember how you had to process an array of double, 4 elements at a time and a stride of 4, when using AVX style SIMD register?
- Think of a block in CUDA as a potentially 3-dimensional stencil of tiny computations which you repeat to cover the iteration space
- We can use the grid to describe the iteration space in terms of the blocks. Depending on hardware availability, multiple blocks will run in parallel



Kernel call syntax

- Remember how you had to process an array of double, 4 elements at a time and a stride of 4, when using AVX style SIMD register?
- Think of a block in CUDA as a potentially 3-dimensional stencil of tiny computations which you repeat to cover the iteration space
- We can use the grid to describe the iteration space in terms of the blocks. Depending on hardware availability, multiple blocks will run in parallel



stride = blockDim.x * gridDim.x

- If the iteration space is much larger then the total number of GPU threads, it is sometimes helpful to do *grid stride loops* in the kernels. You have to take into account that `gridDim._ * blockDim._` GPU threads in the whole grid, which are processing lots of indexes together. That's how many indexes you would now jump over as a "stride"

Information transfer to and from the device

- Any data the kernel needs to process must be transferred using CUDA memory transfer functions
- Pointer/reference values received as input parameters in a function are interpreted on the same side of the host-device boundary

```
1  float *d_A, *d_B, *d_C;
2  auto size = N * sizeof(float);
3  cudaMalloc(&d_A, size);
4  cudaMalloc(&d_B, size);
5  cudaMalloc(&d_C, size);
6
7  cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
8  cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
```

```
1  __device__ float devData;
2  float value = 3.14f;
3  cudaMemcpyToSymbol(devData, &value, sizeof(float));
```

Information transfer to and from the device

- Any data the kernel needs to process must be transferred using CUDA memory transfer functions
- Pointer/reference values received as input parameters in a function are interpreted on the same side of the host-device boundary
- Allocations on unified memory are accessible from both the host and the device.
- Any data transfer required between the physically separate host and device memory happens automatically when using unified memory

```
1  float *u_A, *u_B, *u_C;
2  auto size = N * sizeof(float);
3  cudaMallocManaged(&u_A, size);
4  cudaMallocManaged(&u_B, size);
5  cudaMallocManaged(&u_C, size);
```

```
1  template <class T>
2  auto malloc_usm(size_t N,
3          std::optional<T> init = std::nullopt) -> T*
4  {
5      T* ans{};
6      cudaMallocManaged(&ans, N * sizeof(T));
7      if (init) {
8          for (size_t i = 0UL; i < N; ++i)
9              ans[i] = *init;
10     }
11     return ans;
12 }
```

Device side memory hierarchy and memory space specifiers

- "Local memory" -> per thread memory
- "Shared memory" -> private to a block, but shared among the threads inside a block
- "Global memory" -> visible from all threads in all blocks
- "Constant memory" -> also in the device space, and cached in the constant cache

- Memory address specifier `__device__` declares a variable which lives on the device
- `__constant__` declares a variable to be stored in constant cache
- `__shared__` : variable for the shared memory inside a block, and has the lifetime of the block
- `__managed__`: A variable declared with managed storage specifier can be accessed from both the host and the device, We can determine its address, and it can be read/written from both the host and the device. Since the host and device memories are physically separate, this behaviour is achieved by transferring memory implicitly

Example

```
1  __global__ void mul(const double *A, const double *B, double *C, size_t N) {
2      auto i = threadIdx.x + blockIdx.x * blockDim.x;
3      auto j = threadIdx.y + blockIdx.y * blockDim.y;
4      double res{};
5      if (i < N && j < N)
6          for (size_t k = 0ul; k < N; ++k)
7              res += A[N * i + k] * B[N * k +j];
8      C[N*i + j] = res;
9  }
10 auto main(int argc, char *argv[]) -> int {
11     const unsigned N = (argc > 1) ? std::stoul(argv[1]) : 2048u;
12     auto a = malloc_usm<double>(N * N);
13     auto b = malloc_usm<double>(N * N);
14     auto c = malloc_usm<double>(N * N);
15     for (size_t i = 0UL; i < N * N; ++i) { a[i] = b[i] = 1.1; }
16     auto t0 = std::chrono::high_resolution_clock::now();
17     dim3 ThreadsPerBlock{16, 16};
18     dim3 NumBlocks{N / ThreadsPerBlock.x, N / ThreadsPerBlock.y};
19     mul<<<NumBlocks, ThreadsPerBlock>>>(a, b, c, N);
20     cudaDeviceSynchronize();
```

This is simply a syntax demonstration! Not a particularly clever implementation!

Compiling CUDA code

- With `nvcc` :

```
nvcc [--extended-lambda] [-std=__] source.cu
```

- With `clang++` :

```
clang++ [-std=__] -stdlib=libstdc++ source.cc --cuda-gpu-arch=____ \
    -I /path/to/CUDA/include \
    -L /path/to/CUDA/lib64 -lcudart_static -ldl -lrt -lpthread
```

In the classroom setup on JUSUF, the GPU architecture parameter should be given as `sm_70`. The `-I` and `-L` options may be skipped.

CUDA and C++

- Except in some ancient versions, CUDA is parsed by the rules of the C++ language. Many perfectly valid code in C, e.g., using `class`, `new`, `using` etc. as variable names can not be part of CUDA programs
- Valid C++ code, can often not be used, for a variety of reasons:
  - Generally, the NVIDIA implementation of newer language features arrives a few years after standardization
  - Some language features may have to be modified for use in the context of GPUs
  - CUDA 12 supports most of C++20. Our working environment is based on CUDA 12.6.
  - No modules. No coroutines in device code. A few other smaller restrictions
  - `consteval` functions defined for the host can be invoked in the device code context: after all those are supposed to be immediately evaluated by the compiler!
- Execution space specifiers, execution configuration etc. are language extensions
- This sometimes means additional rules are necessary before a new language feature can be used with CUDA. E.g., how do we make a lambda function `__device__` ? Should `__host__` etc. be considered parts of the functions signature or not ? `nvcc` and `clang++` disagree !

NVIDIA Thrust

```
1  #include <thrust/host_vector.h>
2  #include <thrust/device_vector.h>
3  #include <thrust/generate.h>
4  #include <thrust/sort.h>
5  #include <thrust/copy.h>
6  #include <cstdlib>
7  using namespace thrust;
8  auto main() -> int
9  {
10     // generate 32 M random numbers on
11     // the host
12     host_vector<int> h_vec(32 << 20);
13     generate(h_vec.begin(), h_vec.end(), rand);
14
15     // transfer data to the device
16     device_vector<int> d_vec = h_vec;
17     sort(d_vec.begin(), d_vec.end());
18     // transfer data back to the host
19     copy(d_vec.begin(), d_vec.end(), h_vec.begin());
20 }
```

- Template library like STL or TBB for CUDA
- Elegant high level syntax (STL like iterator interface for algorithms, clever use of operator overloading ...) to clearly express the intent of the programmer
- The compiler translates the stated intents to efficient code for the GPU
- Primarily NVIDIA GPUs

NVIDIA Thrust

```
1  #include <thrust/host_vector.h>
2  #include <thrust/device_vector.h>
3  #include <thrust/generate.h>
4  #include <thrust/sort.h>
5  #include <thrust/copy.h>
6  #include <cstdlib>
7  using namespace thrust;
8  auto main() -> int
9  {
10     // generate 32 M random numbers on
11     // the host
12     host_vector<int> h_vec(32 << 20);
13     generate(h_vec.begin(), h_vec.end(), rand);
14
15     // transfer data to the device
16     device_vector<int> d_vec = h_vec;
17     sort(d_vec.begin(), d_vec.end());
18     // transfer data back to the host
19     copy(d_vec.begin(), d_vec.end(), h_vec.begin());
20 }
```

- Example: `thrust::host_vector` and `thrust::device_vector` use the assignment operator to transfer data between the CPU and the GPU

NVIDIA Thrust

```cpp
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/sort.h>
#include <thrust/copy.h>
#include <cstdlib>
using namespace thrust;
auto main() -> int
{
    // generate 32 M random numbers on
    // the host
    host_vector<int> h_vec(32 << 20);
    generate(h_vec.begin(), h_vec.end(), rand);

    // transfer data to the device
    device_vector<int> d_vec = h_vec;
    sort(d_vec.begin(), d_vec.end());
    // transfer data back to the host
    copy(d_vec.begin(), d_vec.end(), h_vec.begin());
}
```

- Example: `thrust::host_vector` and `thrust::device_vector` use the assignment operator to transfer data between the CPU and the GPU
- Thrust algorithms like `thrust::sort` have syntax like STL algorithms

NVIDIA Thrust

```
1  #include <thrust/host_vector.h>
2  #include <thrust/device_vector.h>
3  #include <thrust/generate.h>
4  #include <thrust/sort.h>
5  #include <thrust/copy.h>
6  #include <cstdlib>
7  using namespace thrust;
8  auto main() -> int
9  {
10     // generate 32 M random numbers on
11     // the host
12     host_vector<int> h_vec(32 << 20);
13     generate(h_vec.begin(), h_vec.end(), rand);
14
15     // transfer data to the device
16     device_vector<int> d_vec = h_vec;
17     sort(d_vec.begin(), d_vec.end());
18     // transfer data back to the host
19     copy(d_vec.begin(), d_vec.end(), h_vec.begin());
20 }
```

- Example: `thrust::host_vector` and `thrust::device_vector` use the assignment operator to transfer data between the CPU and the GPU
- Thrust algorithms like `thrust::sort` have syntax like STL algorithms
- Many data parallel general operations have their own algorithms: `transform`, `reduce`, `inclusive_scan`

Host and device vectors

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <iostream>
auto main() -> int
{
    thrust::host_vector<int> H(4);
    for (int i = 0; i < 4; ++i) H[i] = i;
    // resize H
    H.resize(2);
    std::cout << "H now has size "
        << H.size() << "\n";
    // Copy host_vector H to
    // device_vector D
    thrust::device_vector<int> D = H;
    // elements of D can be modified
    D[0] = 99;
    D[1] = 88;
    // print contents of D
    for(int i = 0; i < D.size(); ++i)
        std::cout << "D[" << i << "] = "
                  << D[i] << "\n";
}
```

- Containers `host_vector` and `device_vector` are designed similar to `std::vector`, but, do not have initializer list constructors or new member functions of `std::vector` like `emplace_back`

Host and device vectors

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <iostream>
auto main() -> int
{
    thrust::host_vector<int> H(4);
    for (int i = 0; i < 4; ++i) H[i] = i;
    // resize H
    H.resize(2);
    std::cout << "H now has size "
        << H.size() << "\n";
    // Copy host_vector H to
    // device_vector D
    thrust::device_vector<int> D = H;
    // elements of D can be modified
    D[0] = 99;
    D[1] = 88;
    // print contents of D
    for(int i = 0; i < D.size(); ++i)
        std::cout << "D[" << i << "] = "
                  << D[i] << "\n";
}
```

- Containers `host_vector` and `device_vector` are designed similar to `std::vector`, but, do not have initializer list constructors or new member functions of `std::vector` like `emplace_back`
- The overloaded assignment operators can copy data across devices

Other initialization options

```
1  // initialize all ten integers to 1
2  thrust::device_vector<int> D(10, 1);
3  // set the first seven elements to 9
4  thrust::fill(D.begin(), D.begin() + 7, 9);
5  // initialize a host_vector with
6  // the first five elements of D
7  thrust::host_vector<int> H(D.begin(), D.begin() + 5);
8  // set elements of H to 0, 1, 2, ...
9  thrust::sequence(H.begin(), H.end());
```

- Many algorithms to provide initial values, to serve different purposes.
- There is also `thrust::generate` which can call a functional for every element of the vector
- The type of the iterators tell the compiler which version of the respective algorithms to use. No run-time overhead

Exercise 4.1:

The example programs `examples/thrust0.cu` and `examples/thrust1.cu` contain the thrust code in the previous slides. Run them on JUSUF using the following steps:

- Load the NVidia HPC module: `ml NVHPC`
- Compile using the `nvcc` or the `nvc++` compiler: `nvcc thrust0.cu`
- Try changing the file name to `thrust0.cc` and compiling

Thrust algorithms

```
1   device_vector<int> X(10), Y(10), Z(10);
2   // initialize X to 0, 1, 2, 3, ....
3   sequence(X.begin(), X.end());
4   // compute Y = -X
5   thrust::transform(X.begin(), X.end(),
6      Y.begin(), thrust::negate<int>());
7   // fill Z with twos
8   thrust::fill(Z.begin(), Z.end(), 2);
9   // compute Y = X mod 2
10  thrust::transform(X.begin(), X.end(),
11                Z.begin(), Y.begin(),
12                thrust::modulus<int>());
13  // replace all the ones in Y with 10
14  thrust::replace(Y.begin(), Y.end(), 1, 10);
15  // print Y
16  thrust::copy(Y.begin(), Y.end(),
17  std::ostream_iterator<int>(cout, "\n"));
```

- Host and device versions
- A set of elementary functionals are available in `thrust/functional.h`
- Notice the copy from a device vector to the ostream iterator!

Universal vectors

```cpp
// examples/thrust_usm.cc
#include <thrust/universal_vector.h>
#include <thrust/sort.h>
#include <iostream>
auto main() -> int
{
    thrust::universal_vector<int> h_vec(1 << 22);
    std::cout << "Filling host vector with random numbers\n";
    thrust::generate(thrust::host, h_vec.begin(), h_vec.end(), rand);
    std::cout << "Done.\n";

    std::cout << "Sorting vector on device\n";
    thrust::sort(thrust::device, h_vec.begin(), h_vec.end());
    std::cout << "Done.\n";
}
```

- `thrust::universal_vector` is similar to `thrust::host_vector` and `thrust::device_vector`, but uses unified memory for storage

Universal vectors

```cpp
// examples/thrust_usm.cc
#include <thrust/universal_vector.h>
#include <thrust/sort.h>
#include <iostream>
auto main() -> int
{
    thrust::universal_vector<int> h_vec(1 << 22);
    std::cout << "Filling host vector with random numbers\n";
    thrust::generate(thrust::host, h_vec.begin(), h_vec.end(), rand);
    std::cout << "Done.\n";

    std::cout << "Sorting vector on device\n";
    thrust::sort(thrust::device, h_vec.begin(), h_vec.end());
    std::cout << "Done.\n";
}
```

- `thrust::universal_vector` is similar to `thrust::host_vector` and `thrust::device_vector`, but uses unified memory for storage
- Data does not need to be moved explicitly between host and device

Universal vectors

```cpp
// examples/thrust_usm.cc
#include <thrust/universal_vector.h>
#include <thrust/sort.h>
#include <iostream>
auto main() -> int
{
    thrust::universal_vector<int> h_vec(1 << 22);
    std::cout << "Filling host vector with random numbers\n";
    thrust::generate(thrust::host, h_vec.begin(), h_vec.end(), rand);
    std::cout << "Done.\n";

    std::cout << "Sorting vector on device\n";
    thrust::sort(thrust::device, h_vec.begin(), h_vec.end());
    std::cout << "Done.\n";
}
```

- `thrust::universal_vector` is similar to `thrust::host_vector` and `thrust::device_vector`, but uses unified memory for storage
- Data does not need to be moved explicitly between host and device
- Algorithms need to be told whether they are meant for host or device explicitly

Custom functionals for transforms

```
1  struct saxpy_functor {
2    const float a;
3    saxpy_functor(float _a) : a(_a) {}
4    __host__ __device__
5      auto operator()(const float& x,
6                      const float& y) const -> float {
7        return a * x + y;
8      }
9  };
10 void saxpy_fast(float A,
11       const thrust::device_vector<float>& X,
12       thrust::device_vector<float>& Y)
13 {
14   // Y <- A * X + Y
15   thrust::transform(X.begin(), X.end(),
16                     Y.begin(), Y.begin(),
17                     saxpy_functor(A));
18 }
```

- When pre-defined operations in `thrust/functional.h` do not suffice, we can write our own function objects
- The overloaded `operator()` must be marked with `__host__ __device__`

Custom functionals using placehoders

- For very simple operations, custom functionals can be generated inline using the `thrust::placehoders` namespace.

```
1  void saxpy_fast(float A,
2        thrust::device_vector<float>& X,
3        thrust::device_vector<float>& Y)
4  {
5      // Y <- A * X + Y
6      thrust::transform(X.begin(), X.end(),
7                        Y.begin(), Y.begin(),
8                        (A * _1 + _2));
9  }
```

- `_1`, `_2` ... are placehoders
- Expressions involving placeholders yield a functional mapping its arguments sequencially to `_1`, `_2` ...

Custom functionals using lambda functions

```
1  void saxpy_fast(float A,
2        thrust::device_vector<float>& X,
3        thrust::device_vector<float>& Y)
4  {
5      // Y <- A * X + Y
6      thrust::transform(X.begin(), X.end(), Y.begin(), Y.begin(),
7                        [A] __host__ __device__ (double x, double y) {
8          return A * x + y;
9      });
10 }
```

```
nvcc --extended-lambda  saxpy0.cu
```

- Note where we mark the lambda function to be for the host and device

Exercise 4.2: Placeholders and lambda functions

The example `examples/saxpy0.cu` shows how to use the placehoders with `thrust` algorithms for simple inline functionality. There is also a commented out version of the same thing done using a lambda function. The placeholder version is more compact, but the lambda version can have multiple statements, like a normal function.

Exercise 4.3: Mandelbrot set

The Mandelbrot set is the set of complex numbers c for which the function $f(z) = z^2 + c$ does not diverge when iterated from $z = 0$. An image representing the set can be created by generating the sequence $z_n = z_{n-1}^2 + c$ for each pixel in the image, by treating the x and y values of the pixel as the real and imaginary components of c. The sequence can be taken to have diverged if the magnitude of z exceeds 2. The program `exercises/mandelbrot_cpu.cc` does it, using the standard C++ library. A modified version using `thrust`, `mandelbrot_gpu.cu` is also present. Build `nvcc` and `clang++` and run. Figure out the code differences and why they are needed.

STDPAR: standard C++ for GPUs

- NVC++, the NVIDIA HPC SDK C++ compiler
- No `<<< >>>`, no `__device__` etc. Just plain C++ written with STL algorithms
- `std::execution::par` regions automatically translated into GPU code!
- There are restrictions, but they will likely be fewer and fewer in the future

```
std::transform_reduce(std::execution::par, R2.begin(),
    R2.end(), S12.begin(), 0., std::plus<double>{},
    [](auto r2, auto s12){
        return Vexv(r2, s12);
    });
```

```
nvc++ -O3 -std=c++23 -stdpar exvol.cc -o exvol.nv
```

STDPAR: standard C++ for GPUs

- Only inline functions or function templates. `nvc++` selects functions for GPU execution on its own, and that only works if it can see the definitions
- CUDA Unified Memory for all data movement between CPU and GPU: presently, only heap allocated objects in CPU code compiled by `nvc++ -stdpar` can be automatically managed. Stack and global storage not accessible. Even heap allocations from portions of CPU code not compiled by `nvc++ -stdpar` are not visible.
- Pointers dereferenced in the parallel algorithms must point to heap locations. References used must be of heap objects.
- Lambda captures by references can often entail pointer dereferencing for stack entities, which should not occur in parallel algorithm regions
- No function pointers: functions are compiled for CPU and GPU. Pointer can only point to one. Inside GPU code, there will then be the danger of accessing a pointer to a function with CPU code. Pass function objects or lambdas as arguments to the algorithms instead.
- Only random access iterators
- `catch` clauses in GPU code ignored. Fine inside CPU code.

Exercise 4.4:

The programs `stdpardemo0.cc` and `stdpardemo1.cc` are simple short programs using parallel algorithms. The second one is a slighly modified version of the `exvol.cc` program we used in connection with SIMD programming. Compile them with `nvc++` and run them on a GPU node on JUSUF.

Exercise 4.5:

The program `jacobi.cc` is in your examples folder. Identify the part which can be parallelized using STL parallel algorithms, and do the necessary code changs. It can be compiled for the GPU using `nvc++`, without any code changes. Try this new way of CPU/GPU programming where the exact same code runs on both!

SYCL

- General model for heterogeneous computing: CPU, GPU, FPGA...
- Can create queues for different devices
- Single source, portable, but not necessarily performance portable
- Open standard from Khronos
- Ref: Data Parallel C++

```
1  template <class T>
2  using usm_alloc_t = sycl::usm_allocator<T,
3      sycl::usm::alloc::shared>;
4  template <class T>
5  using my_vector = std::vector<T, usm_alloc_t<T>>;
6  auto main() -> int
7  {
8      using std::numbers::pi;
9      constexpr auto N = 1UL << 20UL;
10     sycl::queue q;
11
12     usm_alloc_t<double> usmq;
13     myvector<double> v{N, usmq};
14
15     auto* vraw = v.get();
16
17     q.submit([&](sycl::handler& h) {
18         h.parallel_for(N, [=](auto i) {
19             v[i] = 2 * pi * i / N;
20         });
21     }).wait();
22 }
```

SYCL

- Use `sycl::usm_allocator` along with `std::vector` to create USM vectors to be used in kernels
- Alternatively USM `sycl::malloc` and `sycl::free` calls in an RAII helper class, similar to a unique pointer
- Need at least one queue, e.g., `sycl::queue q;`. For more control, `sycl::queue q_gpu{gpu_selector{}};`
- Submit tasks to the queues, containing parallel algorithm calls.
- The task functional submitted should accept a `sycl::handler&`, a command group handler, as the argument

```
1  template <class T>
2  using usm_alloc_t = sycl::usm_allocator<T,
3      sycl::usm::alloc::shared>;
4  template <class T>
5  using my_vector = std::vector<T, usm_alloc_t<T>>;
```

```
ml AdaptiveCPP/git-20.1.0b
acpp -std=c++20 -O3 prog.cc -o prog.ex
```

Run it on a CPU or a GPU!

Exercise 4.6:

Example programs `examples/conv_sycl_usm.cc` and `examples/gblur_sycl.cc` demonstrate iteration over a 1D and 2D space using SYCL. They both perform a calculation similar to a convolution as a demo. The later one also demonstrates how to receive information about the hardware. Build it using the AdaptiveCPP compiler as shown and try to run the generated executable with the `batch_run` command for a CPU run and the `batch_run_gpu` command for a GPU run.