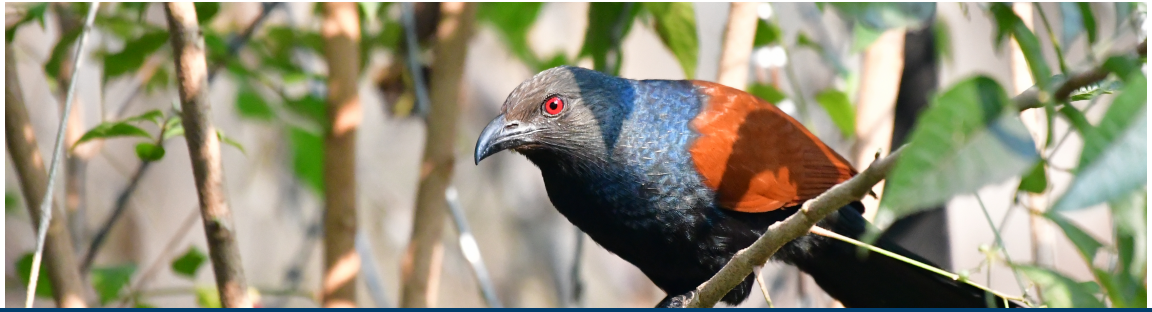


High performance scientific computing in C++

HPC C++ Course 2024

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

JÜLICH Forschungszentrum

The following code causes a SEGFAULT. There is a one character change that makes it behave correctly. Can you spot it?

```
1  auto main() -> int
2  {
3      std::vector V(10UL, 0);
4      std::iota(V.begin(), V.end(), 0);
5      for (auto i = std::size(V) - 1; i >= 0; --i) {
6          std::cout << V[i] << "\n";
7      }
8  }
```

- (A) Change `>=` to `>` in line 5
- (B) Change 0 to 0U in line 4
- (C) Change 0 to 0U in line 3
- (D) Change `size` to `ssize` in line 5

Is this code syntactically valid?

```cpp
auto three(std::span<double> S) {
    for (auto& s : S) s = 3.;
}
void print(std::span<const double> S) {
    for (auto&& s: S) std::cout << s << "\n";
}
auto main() -> int
{
    const std::vector V(10UL, 0.);
    three(V);
    print(V);
}
```

- (A) Yes
- (B) No, because `three` and `print` need spans and we are giving them vectors
- (C) No, because `three` needs a writable span and we can't make that out of a `const` qualified vector

How do I ensure that a generic function to calculate a * x + y is only considered for 3 floating point inputs of the same type?

- (A)

```
1  // Please, only use this with floating point inputs!!!
2  template <class T>
3  auto fma(T a, T x, T y) { return a * x + y; }
```

- (B)

```
1  template <class T> requires std::floating_point<T>
2  auto fma(T a, T x, T y) { return a * x + y; }
```

- (C) The two above are equivalent, because `requires` is a fancy annotation for the programmer, not actual code
- (D) It can not be done

What is wrong with this code?

```cpp
#include <concepts>

template <class T>
concept Addable = requires(T a, T b) {
    { a + b } -> std::convertible_to<T>;
};

auto aggregate(Addable auto... args)
{
    return (args + ...);
}

auto main() -> int
{
    return aggregate(1, 2, 3, 4, 5);
}
```

- (A) `Addable` should be a class
- (B) Function parameters can't be declared `auto` in C++, and why is it both `auto` and `Addable`?
- (C) This explanatory pseudo-code. You have to fill in the `...` with real code before this can work
- (D) There is nothing wrong. This is valid C++ code.

What is wrong with this code?

```cpp
#include <concepts>

template <class T>
concept Addable = requires(T a, T b) {
    { a + b } -> std::convertible_to<T>;
};

auto aggregate(Addable auto... args)
{
    return (args + ...);
}

auto main() -> int
{
    return aggregate(1, 2, 3, 4, 5.);
}
```

- (A) It is impossible to determine whether T is an `int` or a `double`
- (B) This is still OK, because each argument is independently checked for the `Addable` concept

What's wrong with this code?

```
1  #include <iostream>
2  auto main() -> int
3  {
4    auto  high = 100'000'000.0f;
5    auto low = 99'999'990.0f;
6    auto sum = 0.f;
7    do {
8      sum + = 1;
9      high -= 1;
10   } while (high > low);
11   std::cout << sum << "\n";
12 }
```

- (A) Syntax error because of weirdly placed single quotes in lines 4 and 5
- (B) High will always remain greater than low
- (C) Do-while loop will run the loop body once too often
- (D) Nothing. It will produce the obvious answer

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

```
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";
```

```
1  |
2  |
```

- 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 $2^{-23} \approx 0.000000119209289550781 3$.
- 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`
- 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 2.1:

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 nubmers are added in all possible orders. Observe the output!

Exercise 2.2: 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`**

Revisiting programming basics for performance

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);
}
```

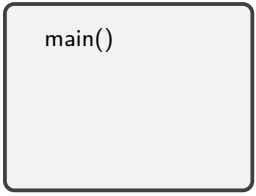
```
auto h211(int i)
    -> int
{
    return -i;
}
```

```
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;
}
```

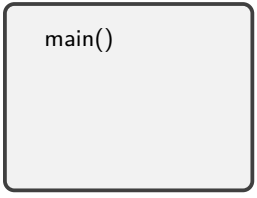


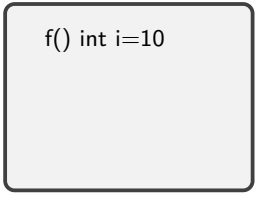
```
auto main() -> int
{
    auto N = 10;
    if (f(N) < g(N)) {
        h1(N);
    } else {
        h2(N);
    }
}
```

```
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;
}
```



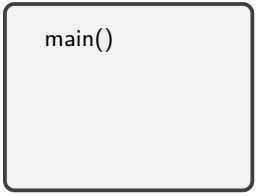


```
auto main() -> int
{
    auto N = 10;
    if (f(N) < g(N)) {
        h1(N);
    } else {
        h2(N);
    }
}
```

```
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;
}
```

main()

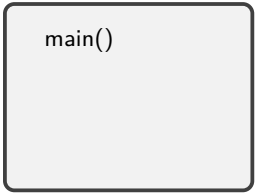
g() int i = 10

```
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;
}
```



```
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

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

```cpp
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);
}
```

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

```cpp
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()

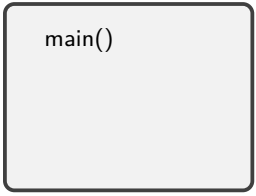
h1() int i = 10

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

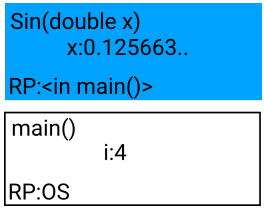
```
auto f(int i)() -> int
{
    return (i * i) %12;
}
auto g(int i)() -> int
{
    return i % 12;
}
auto h1(int i)() -> int
{
    return h11(i);
}
 auto h2(int i)() -> int
{
    return h21(i);
}
```

```
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 2.3:

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.

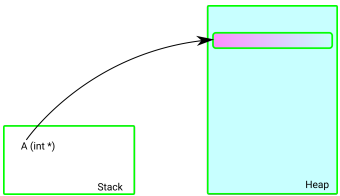
Global storage

```
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

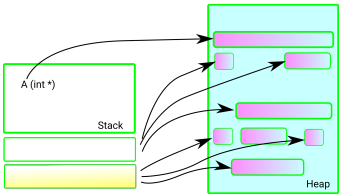
```
1  void f()
2  {
3      int *A = new int[1000000];
4      // calculations with A
5      delete [] A;
6  }
```



- 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 2.4:

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 2.5:

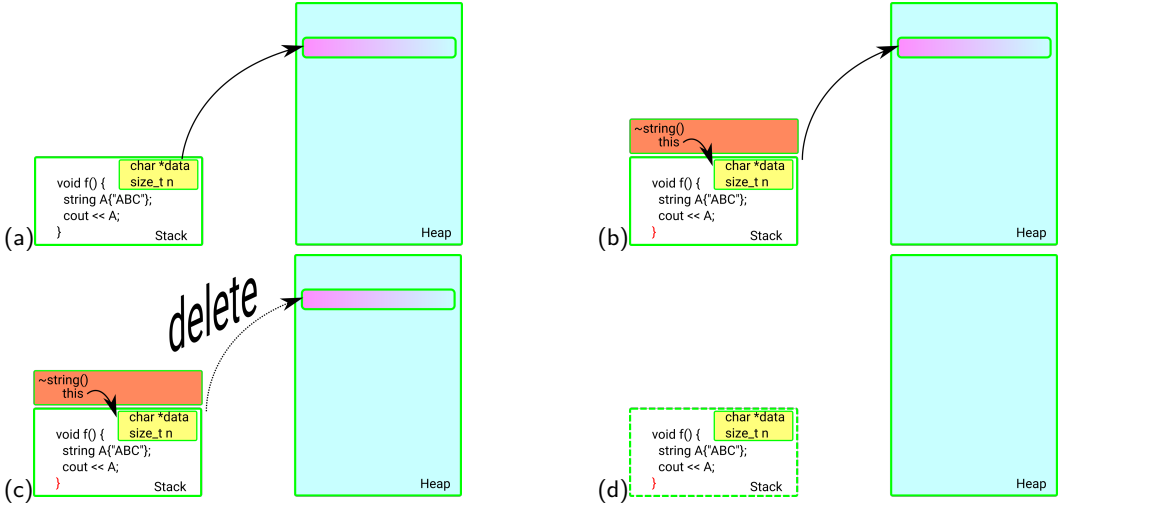
Test the cost of allocation and deallocation using the microbenchmarking site quick-bench.com! Their default example is the code given here (above), comparing string creation and copy. Note down the timings. Then add about 20 'o's at the end of the "hello" in each bench mark, i.e., "hello" $\longmapsto$ "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)

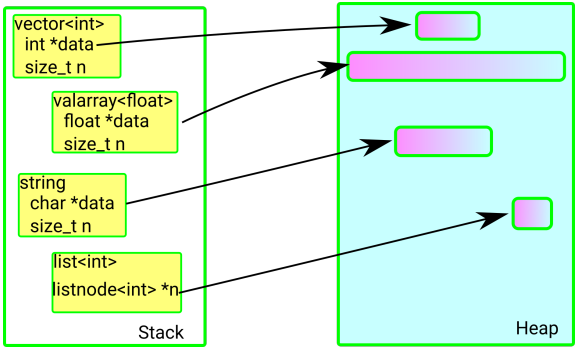
The labels (a), (b), (c) and (d) refer to the figures in the following slide.

Resource handles



JÜLICH Forschungszentrum

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 ?

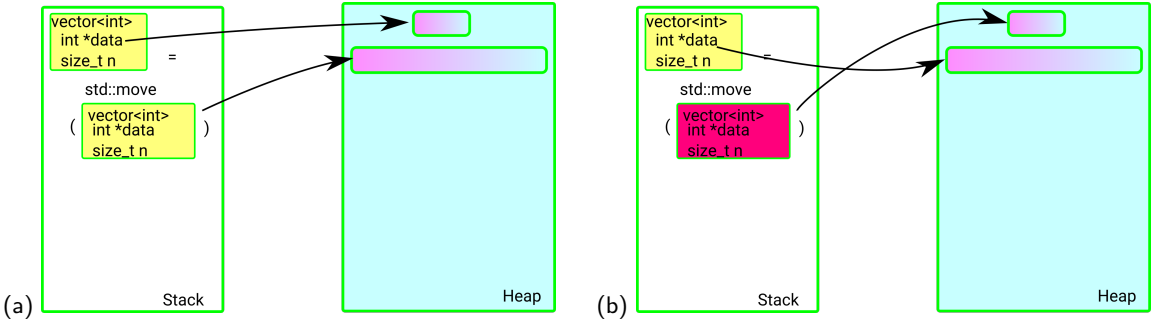
A. 32, 64, 128, 256

B. 32, 64, 256, 64

C. 24, 24, 24, 24

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

Resource handles

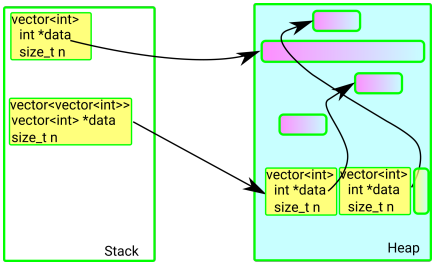


Move

- 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, ...

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

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

Exercise 2.6:

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

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

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

Exercise 2.7:

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

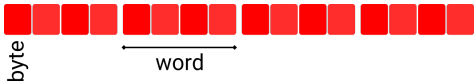
std::array

```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 2.8:

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
```

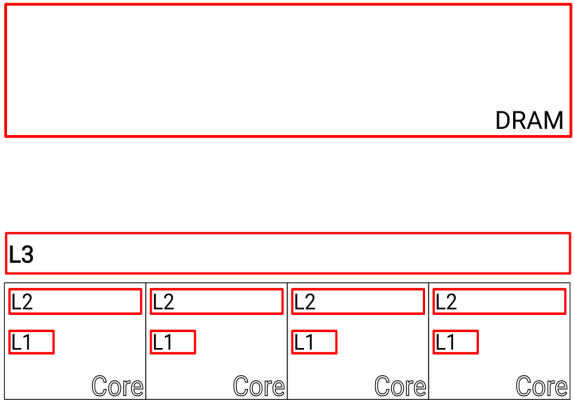
Exercise 2.9:

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 2.10:

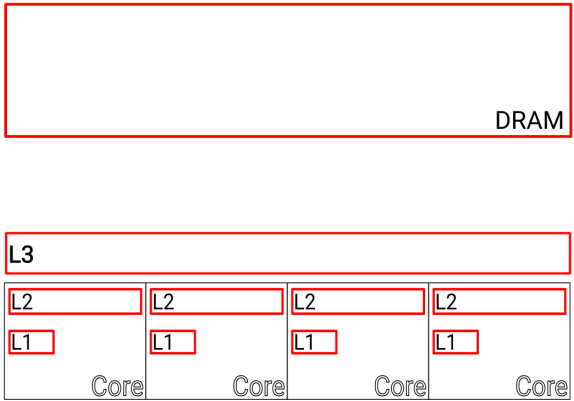
The `examples/align1.cc` shows that the usual mechanisms of dynamic allocation up to C++14 do not provide any guarantees about alignment greater than the natural alignment of the type. The behaviour changed in C++17 for types with explicitly specified extended alignment specifier like our `simd_t` class 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



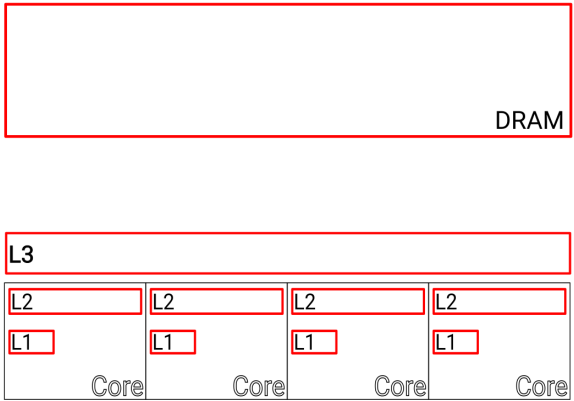
- Fun fact: in 1 clock cycle of the CPU on my laptop, a photon travels about 10 cm in vacuum!

Memory



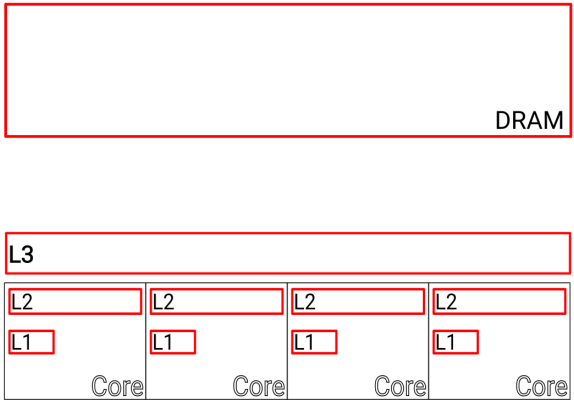
- Fun fact: in 1 clock cycle of the CPU on my laptop, a photon travels about 10 cm in vacuum!
- CPUs contain a certain amount of “cache” memory, which is faster to access, but much smaller than RAM
- Cost of fetching one integer from the main memory can be a hundred times larger than getting it from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

Analogy

- City with a 10000 cars, with a central "processing unit", which may require its citizens
- The administrative area has a small parking lot[1] with, say 1000 parking spaces
- When a person is required, he or she must drive to the parking lot to participate in the business
- Since, a car has 4 seats, every time one person is required, 3 others also arrive uninvited. There is no way to request the presence of one seat of the car.

[1]: What Every Programmer Should Know About Memory, (2007) Ulrich Drepper, Red Hat, Inc.

Analogy

- What happens when all 1000 spaces are taken and a new person is required ?
- How does the administration determine if a person is in the parking lot or must be summoned from the city?

Analogy

- What happens when all 1000 spaces are taken and a new person is required ?
- How does the administration determine if a person is in the parking lot or must be summoned from the city?
- One solution: number the parking spots 001 .. 999, and set the rule that a car with number plate ABCD can only be in the spot BCD.
  - neighbours can be in the parking lot at the same time (License plates are (obviously :-)) assigned by street address)
  - most commonly, people in the same neighbourhood are required one after the other

Analogy

- What happens when all 1000 spaces are taken and a new person is required ?
- How does the administration determine if a person is in the parking lot or must be summoned from the city?
- One solution: number the parking spots 001 .. 999, and set the rule that a car with number plate ABCD can only be in the spot BCD.
  - neighbours can be in the parking lot at the same time (License plates are (obviously :-)) assigned by street address)
  - most commonly, people in the same neighbourhood are required one after the other
- What happens when someone is called, and arrives to find the correct parking spot already occupied?
  - The previous occupant must leave, and drive all the way when required once again.

Analogy

- What happens when all 1000 spaces are taken and a new person is required ?
- How does the administration determine if a person is in the parking lot or must be summoned from the city?
- One solution: number the parking spots 001 .. 999, and set the rule that a car with number plate ABCD can only be in the spot BCD.
  - neighbours can be in the parking lot at the same time (License plates are (obviously :-)) assigned by street address)
  - most commonly, people in the same neighbourhood are required one after the other
- What happens when someone is called, and arrives to find the correct parking spot already occupied?
  - The previous occupant must leave, and drive all the way when required once again.
- What if the business involves multiple questions being asked to these two people ?

Analogy

- What happens when all 1000 spaces are taken and a new person is required ?
- How does the administration determine if a person is in the parking lot or must be summoned from the city?
- One solution: number the parking spots 001 .. 999, and set the rule that a car with number plate ABCD can only be in the spot BCD.
  - neighbours can be in the parking lot at the same time (License plates are (obviously :-)) assigned by street address)
  - most commonly, people in the same neighbourhood are required one after the other
- What happens when someone is called, and arrives to find the correct parking spot already occupied?
  - The previous occupant must leave, and drive all the way when required once again.
- What if the business involves multiple questions being asked to these two people ?
- Alternative: number parking spot sets from 00 to 99 and let there be 10 unnumbered spots in each set. A car with license plate number ABCD can go to any of the unoccupied spaces in set CD. This arrangement is somewhat more tolerant of conflicts such as the one in the question above, which is common in many programs with regular patterns in the data.

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 SSll llll
```

- 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

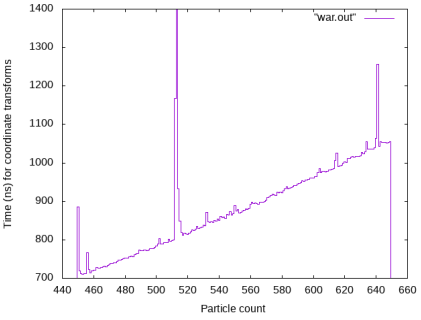
```
1  constexpr size_t size = 2 << 26;
2  std::vector< long > A(size, 0);
3  for (size_t step = 1; step <= 2048; step *= 2) {
4    for (size_t i = 0; i < size; i += step) A[i]++;
5  }
```

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 2.11: Memory effects

The following examples illustrate the cache effects discussed so far:

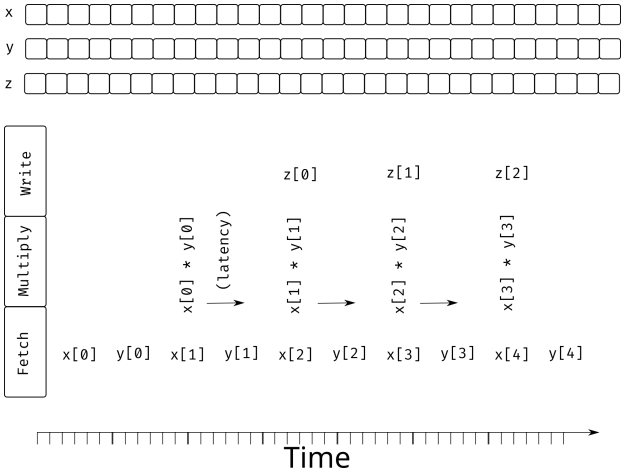
- `traverse0.cc` can be used to compare contiguous and non-contiguous access of a large array
- `every_nth.cc` compares times for accessing every n'th element, and highlights the cache line
- `transpose.cc` transpose operation on a matrix, which involves lots of non-contiguous access

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

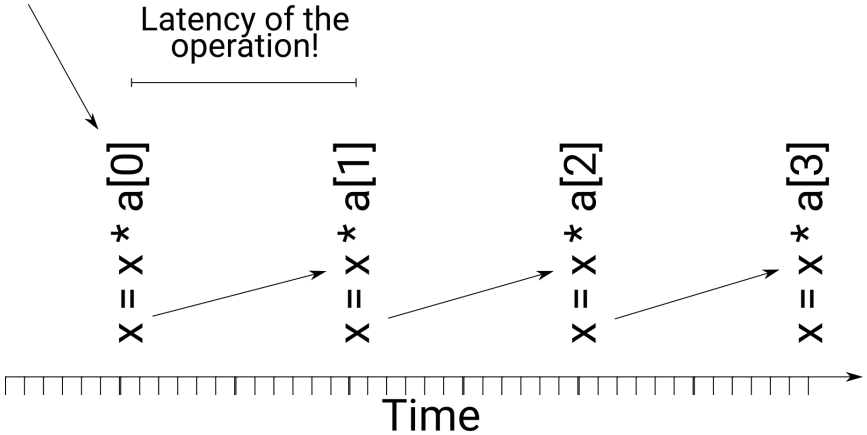
Pipeline

Instruction pipeline

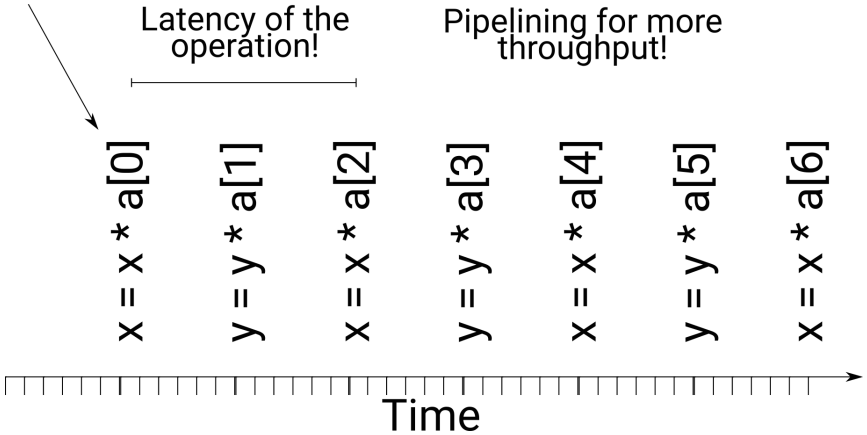


- A processor consists of many units, responsible for different actions, e.g., fetching instructions or data from memory, arithmetics, writing computed results back to memory
- When executing a program, pipelining helps keep different units busy throughout, improving throughput

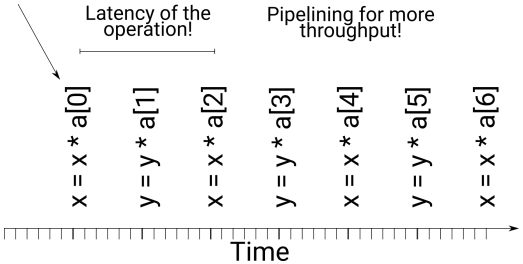
Instruction pipeline



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 2.12:

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 2.13:

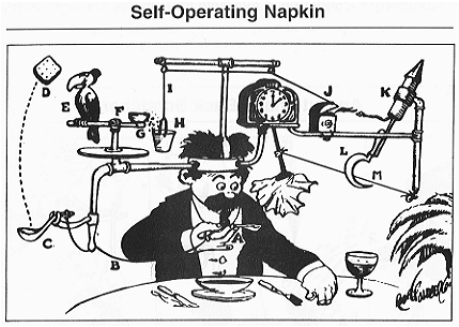
Branch prediction effectiveness using the example program `examples/branch_prediction.cc`, compare the processor on your own computer with the processors on JUSUF login nodes or compute nodes. The program partitions an array of integers into 3 ranges. Running it with a command line argument (value ignored) 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 2.14:

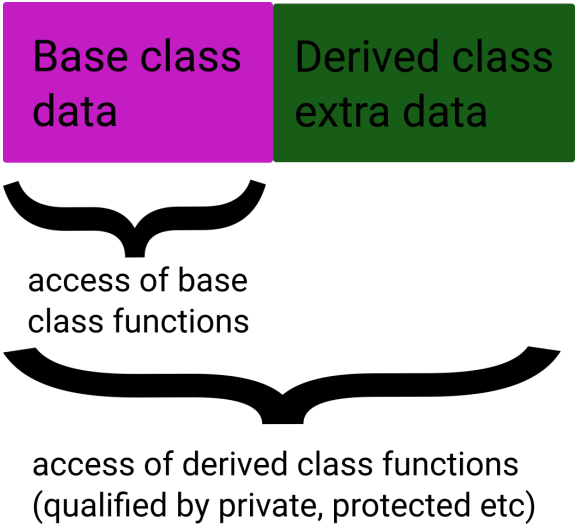
The program `examples/branch_prediction1.cc` further illustrates hardware branch prediction. Here, two alternative kinds of calculations need to be done and accumulated separately. Depending on the value inside a random array of numbers, we decide between the two calculations. It is impossible for the 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.

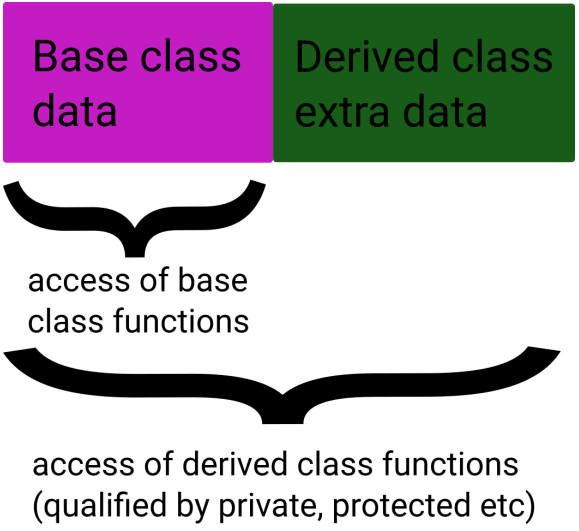


Inheritance



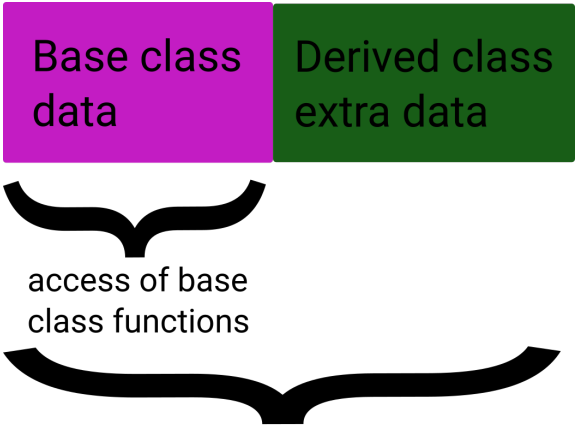
- Inheriting class may add more data, but it retains all the data of the base
- The base class functions, if invoked, will see a base class object
- The derived class object *is a* base class object, but with additional properties

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



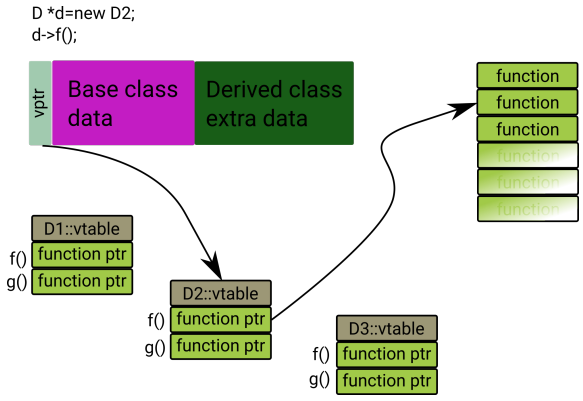

```
1  class Base {
2  public:
3    void f(){std::cout<<"Base::f()\n";}
4  protected:
5    int i{4};
6  };
7  class Derived : public Base {
8    int k{0};
9  public:
10   void g(){std::cout<<"Derived::g()\n";}
11 };
12 auto main() -> int
13 {
14   Derived b;
15   Base *ptr=&b;
16   ptr->g(); // Error!
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 ?

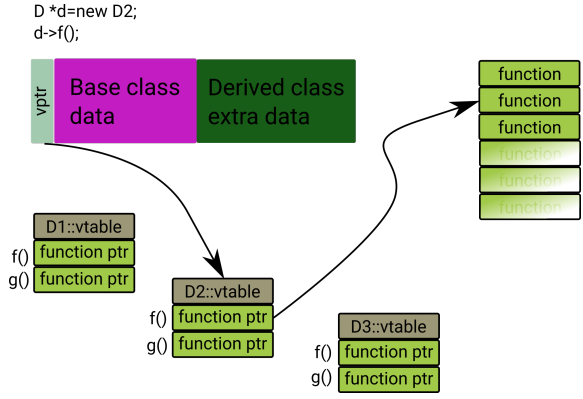
Calling virtual functions: how it works



- 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

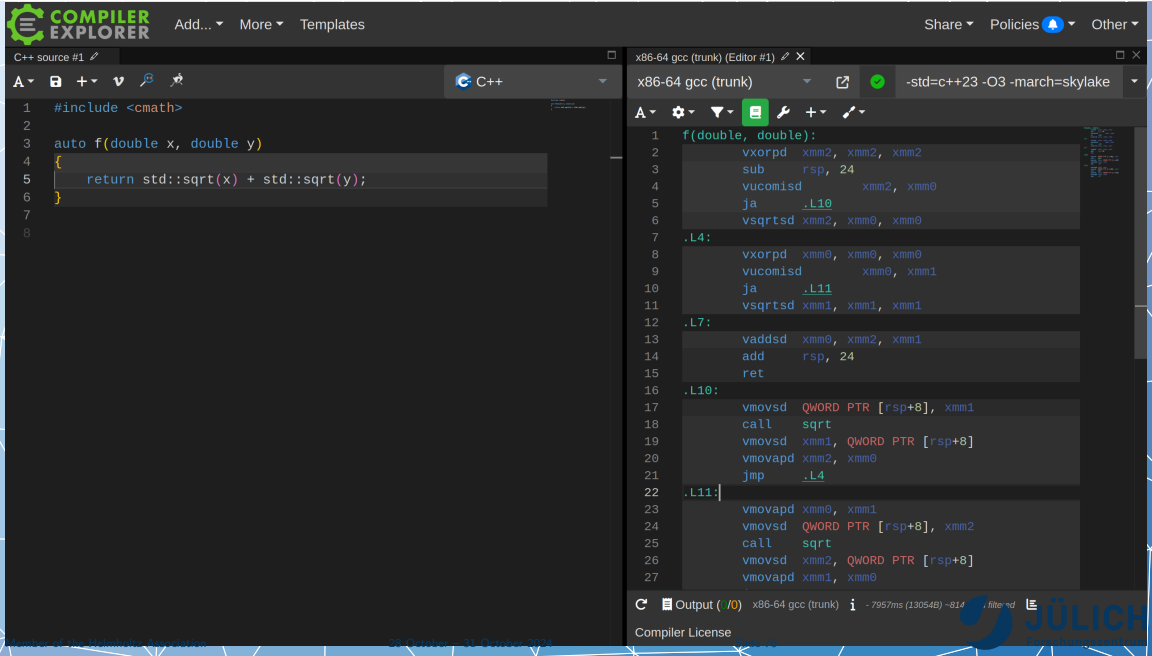


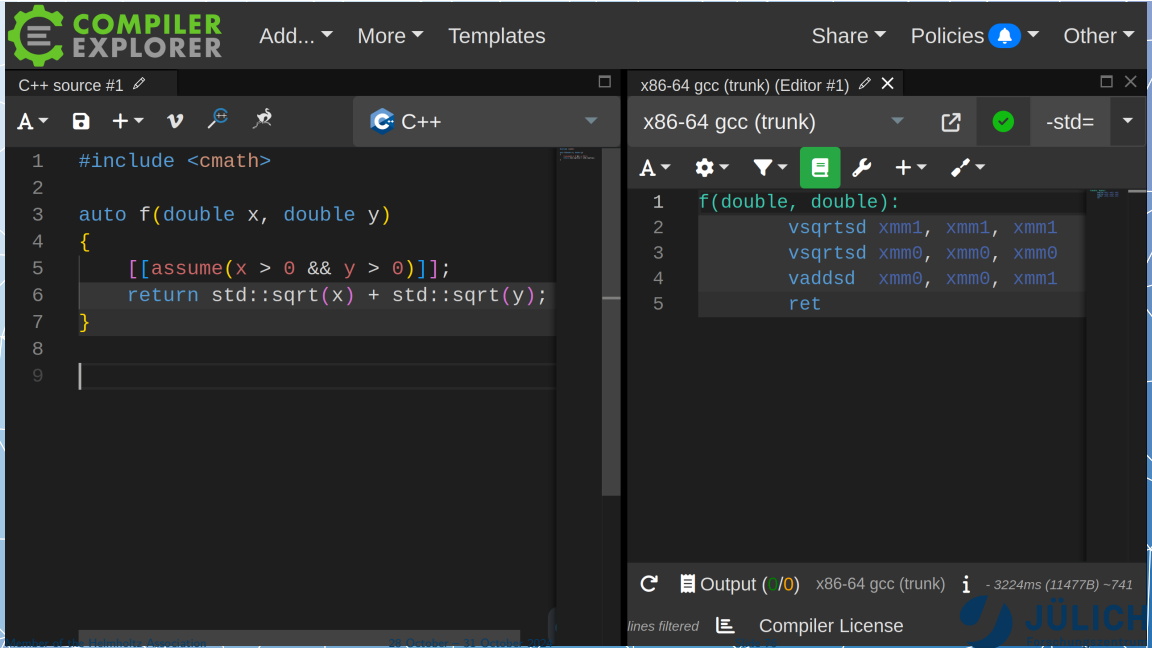
- Often, the polymorphic behaviour sought after using virtual functions can be implemented with CRTP without the virtual function overhead

Expressing assumptions

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
JÜLICH

Using notifying special functions to learn

Exercise 2.15:

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/notebooks/`. 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.

Deducing this

Using templates for deduplication

```
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

Using templates for deduplication

```
1   // Since C++11
2   struct Box {
3       value_type r{};
4       auto value() const & -> const value_type&
5       {
6           return r;
7       }
8       auto value() & -> value_type& { return r; }
9       auto value() const && -> const value_type&&
10      {
11          return r;
12      }
13      auto value() && -> value_type&& { return r; }
14  };
```

- 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.

Using templates for deduplication

```
1   // Since C++11
2   struct Box {
3       value_type r{};
4       auto value() const & -> const value_type&
5       {
6           return r;
7       }
8       auto value() & -> value_type& { return r; }
9       auto value() const && -> const value_type&&
10      {
11          return r;
12      }
13      auto value() && -> value_type&& { return r; }
14  };
```

- 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

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  };
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 do!

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 do!
- 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 do!
- 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 do!
- 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

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

```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

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

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 2.16:

The tag dispatching technique is demonstrated in `examples/tag_dispatch.cc`.

Exercise 2.17:

`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 2.18:

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

Expression Templates

```cpp
template <typename T>
class vec {
  std::vector<T> dat;
public:
  vec(size_t n) : dat(n) {}
  auto operator[](size_t i) const -> T {
    return dat[i];
  }
  auto operator[](size_t i) -> T & {
    return dat[i];
  }
  size_t size() const{return dat.size();}
};
template <typename T>
auto operator+(const vec<T> & v1,
               const vec<T> & v2) -> vec<T>
{
  assert(v1.size() == v2.size());
  auto ans = v1;
  for (size_t i = 0; i < ans.size(); ++i)
    ans[i] += v2[i];
  return ans;
}
```

```cpp
vec<double> W(N), X(N), Y(N), Z(N);
//..
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

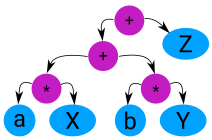
Design with CRTP

- 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!

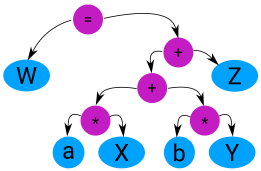
Expression templates




```
a * X + b * Y + Z;
```

```
vecsum<
  vecsum<
    vecscl<vec<double>>,
    vecscl<vec<double>>
  >,
  vec<double>> ({{a,X},{b,Y}},Z);
// Let's call this type EXPR
```

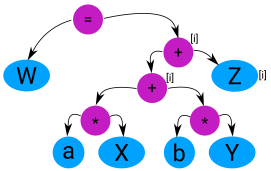
Expression templates




```
W = a * X + b * Y + Z;
```

```
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;
}
```

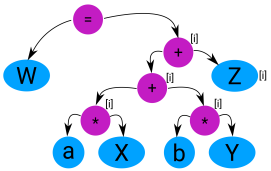
Expression templates




```
W = a * X + b * Y + Z;
```

```
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;
}
const auto vecsum<L,R>::operator[](size_t i) const {
  return lhs[i] + rhs[i];
}
```

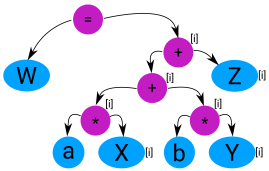
Expression templates




```
W = a * X + b * Y + Z;
```

```
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;
}
const auto vecsum<L,R>::operator[](size_t i) const {
  return lhs[i] + rhs[i];
}
const auto vecscl<T>::operator[](size_t i) const {
    return lhs * rhs[i];
}
```

Expression templates




```
W = a * X + b * Y + Z;
```

```
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;
}
const auto vecsum<L,R>::operator[](size_t i) const {
  return lhs[i] + rhs[i];
}
const auto vecscl<T>::operator[](size_t i) const {
    return lhs * rhs[i];
}
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 2.19:

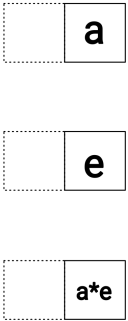
In `examples/xtmp0`, you will find a program which takes two numbers N and a as command line arguments, and creates 4 arrays W, X, Y, Z of size N (user defined array type `vec`). It fills X, Y and Z with random numbers and then calculates $W = a * X + 2 * a * Y + 3 * a * Z$, and times this operation 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 2.20:

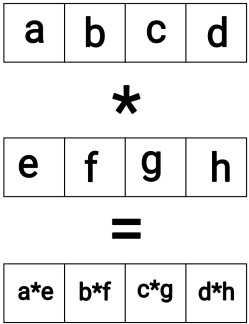
Introduce your own matrix class in the set up used in `examples/xtmp0`, so that matrix vector multiplications can be parts of vector expressions and `M1*M2*v` is evaluated as two matrix vector products rather than a matrix-matrix product followed by a matrix vector product.

Vectorization

SIMD registers and operations



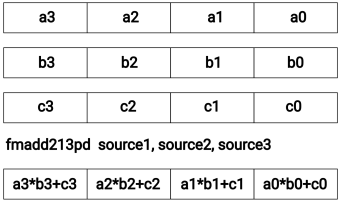
vmulsd xmm0, xmm0, xmm1

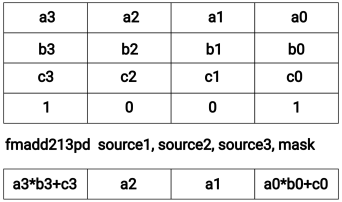


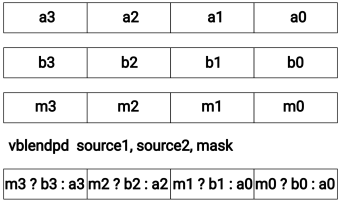
vmulpd ymm0, ymm0, ymm1

JÜLICH Forschungszentrum

SIMD registers and operations

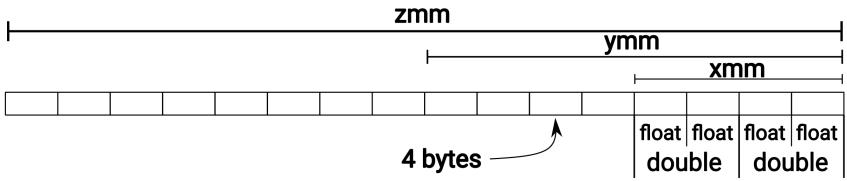






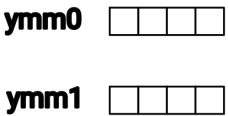
- Increasingly sophisticated instructions in newer CPUs
- Arithmatics, logical operations, shuffles, masked operations, trigonometry, cryptography ...

SIMD registers and operations



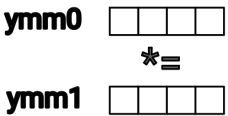
- `xmm0`, `xmm1`, ... `xmm7` (SSE)
- `xmm0` ... `xmm15`, `ymm0` ... `ymm15` (AVX, AVX2, FMA)
- `xmm0` ... `xmm31`, `ymm0` ... `ymm31`, `zmm0` ... `zmm31` (AVX512)

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

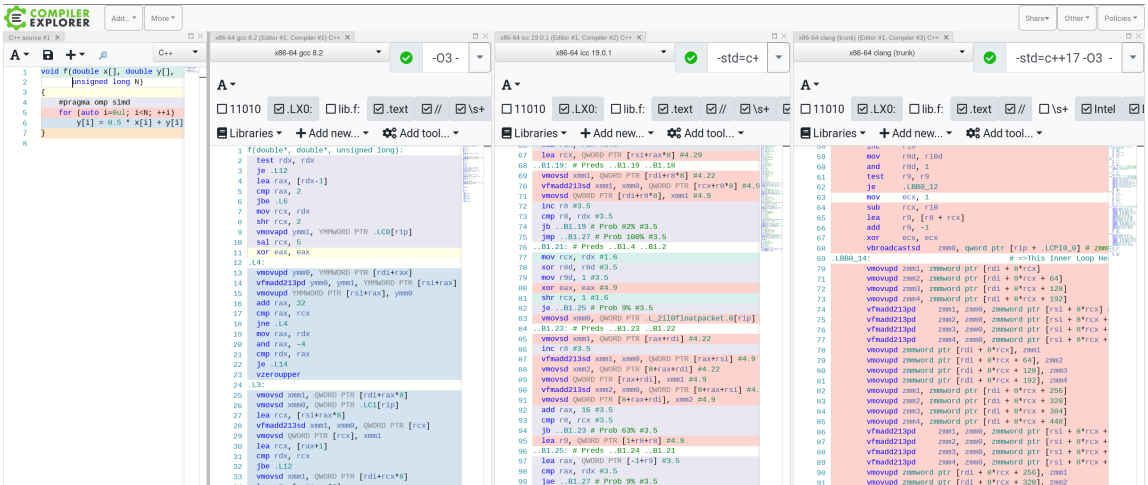
- Reorganize loop to run in chunks suitable for SIMD execution
- Syntax in C and C++ :

```
#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++
C++
x86-64 gcc 8.2
x86-64 icc 19.0.1
x86-64 clang (trunk)
-O3 -
-std=c+
-std=c++17 -O3 -
11010
.LX0:
lib.f:
.text
//
\s+
Intel
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
mov 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

```
1  float sprod_sane(size_t n, const float a[],
2                   const floatb[]) {
3      double res{};
4      for (size_t i=0UL; i<n; ++i)
5          res += a[i] * b[i];
6      return res;
7  }
```

- (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.

```
1  float sse_sprod(size_t n, const float a[],
2                  const float b[]) {
3      assert(0 == n % 4);// simplifying assumption
4      __m128 res, prd, ma, mb;
5      res = _mm_setzero_ps();
6      for (size_t i=0; i<n; i += 4) {
7          ma = _mm_loadu_ps(&a[i]);
8          mb = _mm_loadu_ps(&b[i]);
9          prd = _mm_mul_ps(ma, mb);
10         res = _mm_add_ps(prd, res);
11     }
12     prd = _mm_setzero_ps();
13     res = _mm_hadd_ps(res, prd);
14     res = _mm_hadd_ps(res, prd); // not a typo!
15     float tmp;
16     _mm_store_ss(&tmp, res);
17     return tmp;
18 }
```

- 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

```
1  #include <immintrin.h>
2  union alignas(32) QuadDouble {
3      __m256d mm;
4      double d[4];
5      QuadDouble(__m256d oth) : mm{oth} {}
6      constexpr QuadDouble(double x, double y,double z=0., double t=0.) : d{x, y, z, t} {}
7
8      void aligned_load(double * v) {
9          assert(get_alignment(v) >= 32);
10         mm = _mm256_load_pd(v);
11     }
12     void unaligned_load(double * v) { mm = _mm256_loadu_pd(v); }
13
14     [[nodiscard]]auto operator[](unsigned i) const ->  double { return d[i%4]; }
15     auto operator[](unsigned i) -> double & { return d[i%4]; }
16
17     void operator=(double x) { mm = _mm256_broadcast_sd(&x); }
18     [[nodiscard]] auto horizontal_add() const -> double { return d[0] + d[1] + d[2] + d[3]; }
19 };
```

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 }
```

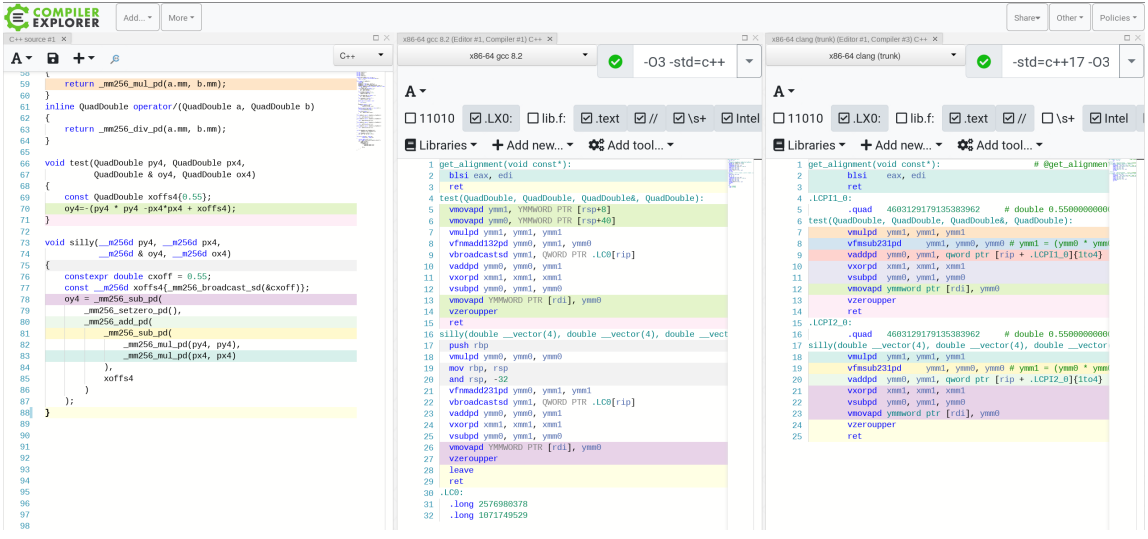
Wrapping intrinsics in zero(/low) cost abstractions

- 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

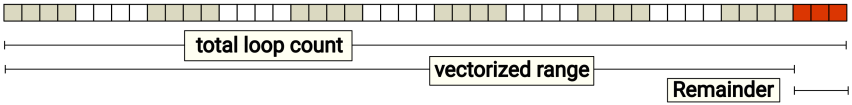
```
auto operator*(QuadDouble a, QuadDouble b) -> QuadDouble
{
    return _mm256_mul_pd(a.mm, b.mm);
}
```

```
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`).

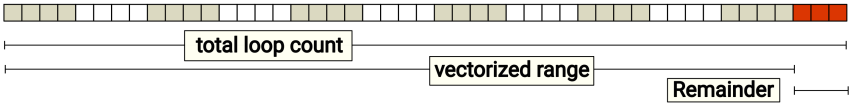

COMPILER EXPLORER
Add...
More
Share
Other
Policies
C++ source #1
x86-64 gcc 8.2 (Editor #1, Compiler #1) C++
x86-64 clang (trunk) (Editor #1, Compiler #3) C++
x86-64 gcc 8.2
-O3 -std=c++
x86-64 clang (trunk)
-std=c++17 -O3
11010
.LX0:
lib.f:
.text
//
\s+
Intel
Libraries
Add new...
Add tool...
return _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
vfnmadd231pd 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



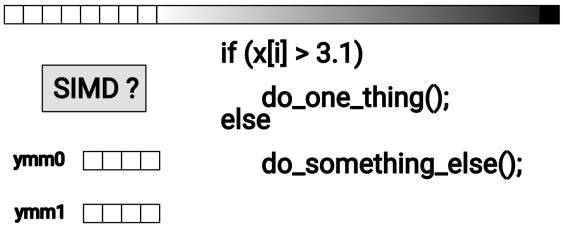

```
1    // examples/diy/daxpy.cc
2  void daxpy_explicit(const std::vector<double> & x, std::vector<double> & y, double a) {
3      QuadDouble bx{0.}, by{0.};
4      const QuadDouble ba{a};
5      unsigned long vsize = x.size() - x.size() % 4;
6      const double * xptr0 = x.data();
7      const double * 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 }
```

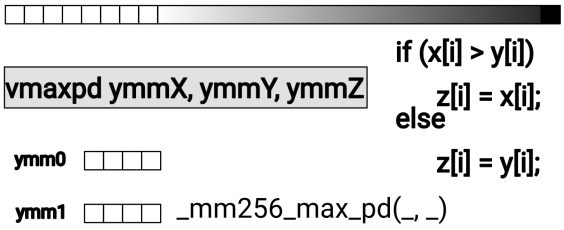
Conditional selection using masks and blend



- 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";
```

- 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 be substituted by the target architecture

XSIMD

- Useful to write with a placeholder tag type `Arch`, to be substituted by the target architecture
- 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

XSIMD

- Useful to write with a placeholder tag type `Arch`, to be substituted by the target architecture
- 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`.

XSIMD

- Useful to write with a placeholder tag type `Arch`, to be substituted by the target architecture
- 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`.

XSIMD

- Useful to write with a placeholder tag type `Arch`, to be substituted by the target architecture
- 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);`.

XSIMD

- Useful to write with a placeholder tag type `Arch`, to be substituted by the target architecture
- 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

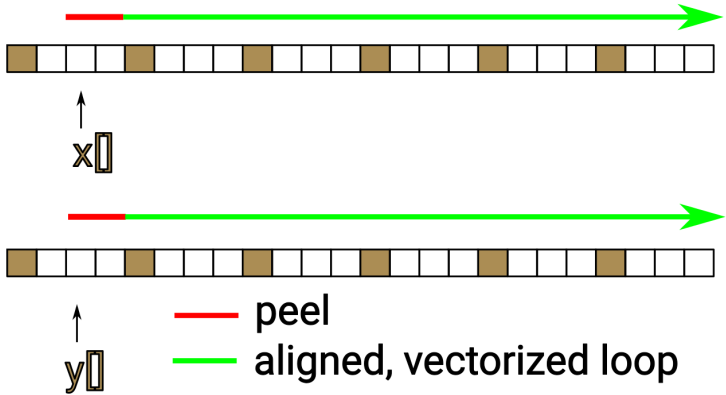
XSIMD

- Useful to write with a placeholder tag type `Arch`, to be substituted by the target architecture
- 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 batch type member function: `xb.store_unaligned(xptr)`, `xb.store_aligned(xptr)` or `xb.store(xptr, alignment_tag)`.

XSIMD

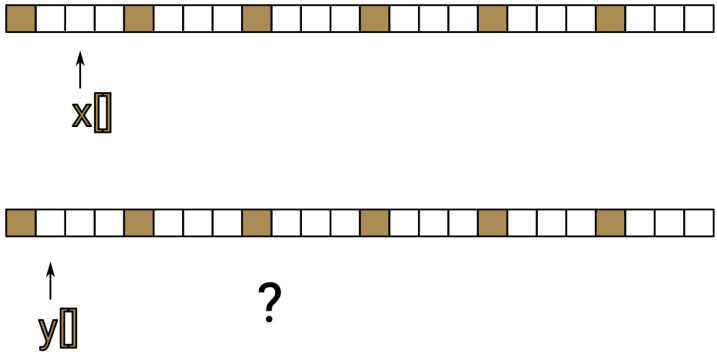
```cpp
using Arch = xsimd::avx2;
void daxpy(double a, std::span<const double> x, std::span<const double> y,
           std::span<double> res) {
    for (size_t i = 0UL; i < x.size(); ++i) {
       res[i] = a * x[i] + y[i];
    }
}
void daxpy_xsimd(double a, std::span<const double> x, std::span<const double> y,
                 std::span<double> res) {
    using btype = batch<double, Arch>;
    constexpr auto vwidth = btype::size;
    const auto ablk = btype::broadcast(a);
    const auto vreps = x.size() - x.size() % vwidth;
    for (size_t i = 0UL; i < vreps; i += vwidth) {
        auto xblk = btype::load_unaligned(&x[i]);
        auto yblk = btype::load_unaligned(&y[i]);
        auto zblk = a * xblk + yblk;
        zblk.store_unaligned(&res[i]);
    }
    for (size_t i = vreps; i < x.size(); ++i) { res[i] = a * x[i] + y[i]; }
}
```

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  using Arch = xsimd::avx2;
6  void caxpy_xsimd(complex<double> a,
7          span<complex<double>> x,
8          span<const complex<double>> y)
9  {
10     using b_type =
11         xsimd::batch<complex<double>, Arch>;
12     b_type c = b_type::broadcast(a);
13     b_type xl, yl;
14     for (size_t i=0; i<x.size();
15                             i+=b_type::size) {
16         xl.load_unaligned(&x[i]);
17         yl.load_unaligned(&y[i]);
18         xl = c * xl + yl;
19         xl.store_unaligned(&x[i]);
20     }
21 }
```

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 2.21:

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
- 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++20 -O3 -march=native -I ~/path/to/xsimd/include exvol1.cc -o exvol1.g
```

The examples `exvol1.cc` and `daxpy1.cc` demonstrate architecture dispatching.

Exercise 2.22:

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