

# C++ Concepts

Walter F. Mascarenhas

USP

July 7, 2019

# Resumo

- 1 Eu
- 2 Programação genérica
- 3 Concepts
  - Tratamento de erros de compilação para templates
  - Requires clauses
  - Requires expressions
- 4 Uma aplicação: Tensores
- 5 Adicionando constraints para casos mais particulares

# Sobre mim

- Sou um matemático que programa.
- Meu foco em C++ são métodos numéricos. Por exemplo, algoritmos usados em machine learning.
- Para mim, C++ é uma ferramenta excelente porque:
  - ▶ Não parou no tempo.
  - ▶ Reflete o modo como eu penso sobre programação.
  - ▶ O código gerado tem ótima performance.
  - ▶ É orientada a objetos, o que as vezes ajuda.
  - ▶ **É a melhor linguagem para programação genérica.**

# Programação Genérica para métodos numéricos

- Vários tipos de números: `int*_t`, `uint*_t`, `float16`, `float`, `double`, `long double`, `__float128`, `mpfr`, `mpq`, intervalos.
- Muitos instruction sets e extensões (`x86`, `SSE*`, `AVX*`, `ARM`), além de OpenMP.
- Múltiplos devices: CPU, GPU, MIC, celular.
- Vários containers: arrays, vetores, matrizes, (tensores = arrays) e suas views, alocados na pilha, na heap ou GPU.

É natural portanto criar algoritmos e estruturas de dados parametrizados para os casos acima.

Como motivação para a palestra, vou considerar o seguinte código genérico, que multiplica cada elemento de um container `c` por um valor `v`:

```
template <class Container, class Value>
void mul(Container& c, Value v)
{
    for(auto& ci : c)
        ci *= v;
}
```

Quando o container é `std::array<float,N>`, com `N` é múltiplo de 8, e `Value` é `float`, o código abaixo parece ser mais eficiente:

```
template <std::size_t N>
void mul(std::array<float,N>& c, float v)
{
    __m256 vx = _mm256_set1_ps(v);
    float* p_max = c.data() + c.size();
    for(float* p = c.data(); p < p_max; p += 8)
    {
        __m256 cx = _mm256_load_ps(p);
        cx *= vx;
        _mm256_store_ps(p, cx);
    }
}
```

Será que é mesmo?

Na verdade, compilando as duas funções acima com `Container = std::array<float,16>`, `Value = float` e

```
g++-9 -S -O3 -march=native -mtune=native mul.cpp
```

obtemos essencialmente a mesma coisa:

```
vbroadcastss %xmm0, %ymm0
vmulps (%rdi), %ymm0, %ymm1
vmulps 32(%rdi), %ymm0, %ymm0
vmovups %ymm1, (%rdi)
vmovups %ymm0, 32(%rdi)
```

Ou seja, com as opções corretas, o compilador faz o que queremos fazer manualmente. Inclusive, ele usa a informação que `N` é 16.

Para fazer isso com herança seria necessário devirtualização, link time (ou whole program) optimization, se o compilador conseguir. Porém:

- 1 Com uma alternativa usando herança, a compilação seria mais rápida.
- 2 As mensagens de erro de código genérico podem ser horríveis.
- 3 Os testes também precisam ser genéricos.
- 4 A depuração é mais complicada.
- 5 Pensar sobre casos gerais é mais difícil que pensar sobre casos particulares.
- 6 O código pode ficar ininteligível (menos o meu, claro...)

Concepts ajudam no item 2 e, principalmente, no item 6.



## Jogo dos dois erros

Há dois erros no código abaixo. Quais são?

```
bool test()
{
    std::srand(10);
    float v = rand();
    alignas(32) std::array<float,64> c;
    std::generate(c.begin(), c.end(), std::rand());
    std::array<float,64> c2{c};

    mul(v, c);
    for(uint i = 0; i < c.size(); ++i)
    {
        if( c[i] != c2[i] * v)
            return false;
    }

    return true;
}
```

O próximo slide mostra o que acontece ao compilarmos o código do slide anterior: aparece uma mensagem de erro no arquivo `stl_algo.h`, que aparentemente tem pouco a ver com o que estamos compilando.

```

4396 .....@brief Assign the result of a function object to each value in a
4397 .....sequence.
4398 .....@ingroup mutating_algorithms
4399 .....@param __first A forward iterator.
4400 .....@param __last A forward iterator.
4401 .....@param __gen A function object taking no arguments and returning
4402 .....std::iterator_traits<_ForwardIterator>::value_type
4403 .....@return generate() returns no value.
4404 .....
4405 .....Performs the assignment @c *i = @p.__gen() for each @c i in the range
4406 .....@p[ __first, __last ).
4407 ...../
4408 ..template<typename _ForwardIterator, typename _Generator>
4409 .....void
4410 .....generate(_ForwardIterator __first, _ForwardIterator __last,
4411 ....._Generator __gen)
4412 .....{
4413 .....    // concept requirements
4414 .....    __glibcxx_function_requires(_ForwardIteratorConcept<_ForwardIterator>)
4415 .....    __glibcxx_function_requires(_GeneratorConcept<_Generator>,
4416 .....    typename iterator_traits<_ForwardIterator>::value_type)
4417 .....    __glibcxx_requires_valid_range(__first, __last);
4418 .....
4419 .....    for (; __first != __last; ++__first)
4420 .....        *__first = __gen();
4421 .....}

```

## Issues



In file included from /usr/include/c++/9/algorithm:62, /usr/include/c++/9/algorithm	algorithm
from ../build/f.cpp:1:	f.cpp
In instantiation of 'void std::generate(_Filter, _Filter, _Generator) [with _Filter = float*; _Generator = int]':	stl_algo.h
required from here	f.cpp
' __gen' cannot be used as a function	stl_algo.h
In instantiation of 'void mul(Container&, Value) [with Container = float; Value = std::array<float, 64>]':	f.cpp
required from here	f.cpp

# Consertando o que podemos consertar

A chamada

```
mul(v, c);
```

está errada pois o primeiro argumento tem que ser um container e o segundo um número: a ordem dos argumentos foi trocada. Podemos melhorar a mensagem de erro assim:

```
template <class Container, class Value>
requires
(std::is_integral_v<Value> || std::is_floating_point_v<Value>)
void mul(Container& c, Value v)
{
    for(auto& ci : c)
        ci *= v;
}
```

Agora a mensagem de erro aponta exatamente o problema, no lugar certo:

```
< > f.cpp* # main(): int
302 template<class Container, class Value>
303 requires(std::is_floating_point_v<Value> || std::is_integral_v<Value>)
304 void mul(Container& c, Value v)
305 {
306     for(auto& ci : c)
307         ci *= v;
308 }
309
310 int main()
311 {
312     float v{2};
313     std::array<float, 4> c{1, 2, 3, 4};
314     mul(v, c);
315 }
316
```

Issues

In function 'int main()':

- cannot call function 'void mul(Container&, Value) [with Container = float; Value = std::array<float, 4>]'  
note: constraints not satisfied
- 'is\_floating\_point\_v<Value>' evaluated to false
- 'is\_integral\_v<Value>' evaluated to false

O texto em vermelho e azul abaixo é uma “requires clause”

```
template <class Container, class Value>  
requires  
(std::is_integral_v<Value> || std::is_floating_point_v<Value>)  
void mul(Container& c, Value v)
```

A parte em azul é a constraint, que é a disjunção de duas constraints atômicas:

```
std::is_integral_v<Value>
```

e

```
std::is_floating_point_v<Value>
```

# Inteiros conhecidos em tempo de compilação

A constraint para `mul` acima ignora tipos de inteiro muito importantes em programação genérica:

- O `N` em `std::array<T,N>`, que ajudou o compilador a gerar código otimizado para `mul` acima.
- o template `std::integral_constant` na standard library.
- a `integral_constant` na `boost::Hana`.

Nessa palestra assumirei que temos um template `s_uint<std::size_t>` parecido com a integral constant da `boost::hana`, que implementa as operações aritméticas, e que há uma trait para indicar se `T` é `s_uint`:

```
template <class T>
constexpr bool is_s_uint_v = is_s_uint<T>::value;
```

Podemos então definir `mul` assim:

```
template <class Container, class Value>
requires
(std::is_integral_v<Value> || is_s_uint_v<Value> ||
 std::is_floating_point_v<Value> )
void mul(Container& c, Value v)
{
    for(auto& ci : c )
        ci *= v;
}
```



# Podemos fazer melhor usando Concepts

Primeiro definimos o concept Int:

```
template <class T>
concept bool Int =
std::is_integral_v<T> || is_s_uint<T>;
```

e depois o concept Number:

```
template <class T>
concept bool Number =
Int<T> || std::is_floating_point_v<T>;
```

Podemos então definir mul de duas formas equivalentes:

```
template <class Container, class Value>
requires
Number<Value>
void mul(Container& c, Value v)
{
    for(auto& ci : c )
        ci *= v;
}
```

ou de modo mais resumido:

```
template <class Container, Number Value>
void mul(Container& c, Value v)
{
    for(auto& ci : c )
        ci *= v;
}
```

E se o container estiver errado?

Pode acontecer algo como no próximo slide, onde passamos um container inválido para a `mul`, e o erro aparece dentro na função `mul`, e não no ponto onde ela foi chamada.

```
< > f.cpp # main(): int
304 | template<class Container, Number::Value>
305 | void mul(Container& c, Value v)
306 | {
307 |     for(auto& ci : c)
308 |         ci *= v;
309 | }
310 |
311 | int main()
312 | {
313 |     float v{2};
314 |     mul(v, v);
315 | }
```

Issues

In instantiation of 'void mul(Container&, Value) [with Container = float; Value = float]':  
required from here

**!** 'begin' was not declared in this scope; did you mean 'std::begin'?

```
307 |     for(auto& ci : c)
      |         ^~~
      |         std::begin
/home/walter/talks/cpp20/samples/f.cpp
```

In file included from /usr/include/c++/9/string:54,  
from /usr/include/c++/9/stdexcept:39,  
from /usr/include/c++/9/array:39,  
from /usr/include/c++/9/tuple:39,  
from /usr/include/c++/9/functional:54,  
from /usr/include/c++/9/pstl/glue\_algorithm\_defs.h:13,  
from /usr/include/c++/9/algorithm:71,  
from ../samples/f.cpp:1:  
'std::begin' declared here

**!** 'end' was not declared in this scope; did you mean 'std::end'?

Para lidar com esses errors, usaremos concepts e "Requires expressions" (azuis), que são diferentes de "Requires clauses" (vermelhas).

## Requires expressions

Podemos restringir os containers, a partir do concept de input iterator:

```
template <class It>
concept InputIterator =
requires(It it) {
    {*it};
    {++it};
    {It(it)};
    {it != it}->bool;
};
```

```
template <class C>
concept Container =
requires(C c) {
    typename C::value_type;
    {c.begin()}->InputIterator;
    {c.end()}->decltype(c.begin());
    {*c.begin()}->typename C::value_type&;
};
```

# Não use os concepts dessa palestra!!!

Todos eles são incompletos. Foram feitos para fins didáticos apenas, para caber nos slides.

Estude a biblioteca `ranges` para aprender como lidar com os muitos detalhes necessários para fazer bons concepts.

Uma vez que temos os concepts `Container` e `Number`, podemos definir `mul` como abaixo. Exigimos também que podemos fazer `cv *= v` para um elemento `cv` do container e um `Number v`.

```
template <Container C, Number V>
requires
requires(typename C::value_type cv, Number v)
{
    {cv *= v};
}
void mul(C& c, V v)
{
    for(auto& ci : c)
        ci *= v;
}
```



Agora, ao tentarmos compilar um código chamando a `mul` com um container inválido a mensagem de erro aparece no lugar certo (veja o próximo slide).

```

330
331 template<Container C, Number V>
332 requires requires(typename C::value_type cv, V v)
333 {
334     ..{cv /= v};
335 }
336 void mul(C& c, V v)
337 {
338     ..for(auto& ci : c)
339         ...ci *= v;
340 }
341
342 int main()
343 {
344     ..float v{2};
345     ..mul(v, v);
346 }

```

Issues



In function 'int main()':

/home/walter/talks/cpp20/samples/f.cpp

cannot call function 'void mul(C&, V) [with C = float; V = float]'

note: constraints not satisfied

within 'template<class C> concept const bool Container<C> [with C = float]'

note: with 'float c'

the required type 'typename C::value\_type' would be ill-formed

the required expression 'c.begin()' would be ill-formed

the required expression 'c.end()' would be ill-formed

the required expression '\* c.begin()' would be ill-formed

'float' is not a class, struct, or union type

note: with '<typeprefixerror>cv'

note: with 'float v'

the required expression 'cv /= v' would be ill-formed



# Aplicação a machine learning: Tensores

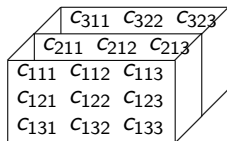
A principal estrutura de dados para as redes neurais convolucionais são os “tensores”:

$$\begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix}$$

Vetores são tensores de rank 1

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

Matrizes são tensores de rank 2



Cubos são tensores de rank 3

Veremos agora código genérico para tensores.

## Números: os elementos dos tensores

```
namespace traits {

template <class T>
struct number {
    using type = void;
};

template <class T>
requires requires() {
    typename T::number_type;
} && Scalar<typename T::number_type>
struct number<T> {
    using type = typename T::number_type;
};
} // namespace traits

template <class T>
using number_t = typename traits::number<T>::type;
```

## Rank: o número de dimensões do tensor

```
template <class T>
constexpr std::size_t rank =
(
    requires() {
        {T::rank}->std::size_t;
    }
    ?
    std::size_t{T::rank}
    :
    std::size_t{0}
);
```

## Lidando com dimensões

```
namespace traits {
template <class A, class Indexes>
struct has_dims : std::false_type {};

template <class A, std::size_t... I>
struct has_dims<A, std::index_sequence<I...> >
: std::integral_constant<bool,
  requires( A a, std::size_t s)
  {
    { ((s = a.template dim<I>()), ...) };
  } >;
} // namespace traits

template <class A>
constexpr bool has_dims_v =
traits::has_dims<A, std::make_index_sequence<rank<A>> >::value;
```

## Calculando o volume (número de elementos)

```
namespace traits {  
  
template <class A, std::size_t... I>  
constexpr auto volume(A a, std::index_sequence<I...>)  
{  
    return (s_uint<1>{} * ... * a.template dim<I>() );  
}  
} // namespace traits  
  
template <class A>  
requires has_dims<A>  
constexpr auto volume(A const& a)  
{  
    return traits::volume(a, std::make_index_sequence< rank<A> >{} );  
}
```

## Usaremos o operator() para acessar os items do tensor

(o operator[] só aceita um argumento)

A trait abaixo indica se `a(i...)` é válido.

```
namespace traits {
template <class A, class Indexes>
struct can_access_item : std::false_type ;

template <class A, std::size_t... I>
struct can_access_item<A, std::index_sequence<I...> >
: std::integral_constant<bool,
  requires(A a)
  {
    { a(I...) }->number_t<A>&&;
  } > {};
}

template <class A>
constexpr bool can_access_item_v = traits::can_access_item<T>::value
```



# Aplicação a machine learning: Views

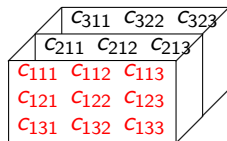
Além de tensores, em machine learning usamos “views” que são partes dos tensores. Na figura abaixo, os items das views estão em vermelho:

$$\begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix}$$

Um tensor todo é uma view dele mesmo

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

Um grupo de colunas é uma view de uma matrix



Uma face é uma view de um cubo

# Os concepts View e Span

```
template <class A>
concept bool View =
!std::is_void< number_t<A> > &&
has_dims_v<A> &&
can_access_item_v<A>
requires(A a)
{
    typename A::cursor_type;
    { a.cursor() }->typename A::cursor_type;
};
```

Um Span é uma View com mutable items:

```
template <class T>
concept bool Span = View<T> && !std::is_const_v<number_t<T>>;
```

## mul para Spans

```
template <Span S, Number V>
void mul(S& s, V v)
{
    auto c = s.cursor();
    if constexpr ( Iterator<decltype(c)> )
    {
        auto e = s.end();
        for( ; c != e; ++c)
            *c *= s;
    }
    else
    {
        for(; !c.is_done(); ++c)
            mul(*c, v);
    }
}
```

## Especializando para performance

A Intel fornece (de graça) uma biblioteca com as operações fundamentais para cálculo numérico chamada MKL, que lida com floats e doubles. Ela é otimizada para processadores Intel e suspeita em processadores AMD. Quando essa biblioteca está disponível, é bom usá-la. Isso poder feito adicionando essas constraints ao concept Span:

```
template <class T>
constexpr bool MklSpan =
Span<T> &&
linkarei_a_mkl() &&
(std::is_same_v<cursor_t<T>, double*> ||
 std::is_same_v<cursor_t<T>, float*> ) &&
std::is_integral_v<volume_t<T> >;
```

## A função mul

```
template <Span T, Number X>
void mul(T& t, X x) {
    for(auto& ti : t)
        ti *= x;
}
```

```
template <MklSpan T, Number X>
void mul(T& t, X x) {
    if constexpr ( std::is_same_v<cursor_t<T>,double*> )
        cblas_dscal(volume(t), x, x.begin(), 1);
    else
        cblas_sscal(volume(t), x, x.begin(), 1);
}
```

Como o overload com `MklSpan` é mais especializado que o com `Span`, ele será selecionado quando `T` for um `MklSpan`.

## A classe flat view: um modelo para o concept View

```
template <Number T, Int... Dims>
class flat_view {
public:
    using number_type = T;
    using cursor_type = T*;
    static constexpr std::size_t rank = sizeof...(Dims);

    template <Int... L>
    requires requires(L... l) { { std::tuple<Dims...>{l...} } }
    constexpr view(T* data, L... l)
    : data_(data), dims_(l...) {
        assert( ((l >= 0) && ... ) );
    }

    template <std::size_t I>
    requires (I < rank)
    constexpr auto dim() const {
        return std::get<I>(dims_);
    }
}
```

```
T* cursor() const {  
    return data_;  
}  
  
T* begin() const {  
    return data_;  
}  
  
T* end() const {  
    return data_ + volume(*this);  
}
```

private:

```
T* data_;  
std::tuple<Dims...> dims_;  
};
```

## Implementando flat\_view()(i0,i1,...) com pipes

```
template <class Sum, std::size_t D>
struct pipe {
    template <Int I>
    constexpr auto operator|(I i)
    {
        constexpr auto new_sum = sum_ * std::get<D>(dims_) + j;
        return pipe<decltype(new_sum), D + 1>{new_sum, dims_};
    }

    S sum_;
    std::tuple<Dims...> const& dims_;
};

template <Int IO, Int... I>
T& operator()(IO i0, I... i) const
{
    auto index = (pipe<IO,0>{i0, dims_} | ... | i).sum_;
    return data_[index];
}
```