An N-Dimensional Matrix Design

Andrew Sutton (asutton@cs.tamu.edu)
Bjarne Stroustrup (bs.cs.tamu.edu)

Texas A&M University

Language and Libraries

In isolation, language features are boring and useless

A good example shows how many language features can be used in combination to address a challenging problem

Example: an N-dimensional matrix

What Makes a Good Library?

Intuitive – should be easy to use for simple tasks

but not restricted to simple tasks

General – should be applicable to many problems

but not too general

Fast – as always

Safe – don’t easily allow obvious errors

but not necessary proof against malice

Good libraries do not just “happen”; library design is an iterative process of refinement and balancing tradeoffs

Library Design Process

Requirements:

Decide what you want the library to do

Decide how you want to use it (how usage should look)

Design and Implementation:

Try to implement what you've said

Use

Evaluate your implementation (do you like it)

Iterate

Library and Language Design

An exercise in getting the syntax that you want

and the semantics

Library design motivates language design

How can you improve the language to improve your library

Language design motivates library design

How can new features improve the design of your library

Language Features Used

- Classes
  - of course
- Parameterization with numbers and types
- Move constructors and assignments
  - to minimize copying
- RAII
  - relying on constructors and destructors
- Variadic templates
  - for specifying extents and for indexing
- Initializer lists
- Operator overloading
  - to get conventional notation
- Function objects
  - to carry information about subscripting
- Some simple template metaprogramming
  - (e.g., for checking initializer lists and for distinguishing reading and writing for Matrix_ref s)
- Implementation inheritance
  - for minimizing code replication.

Implementation

Origin C++11 Libraries

http://code.google.com/p/origin/

Overview

Basic usage and requirements

Construction and Assignment

Subscripting and Slicing

Mathematical operations

Implementation

Examples

Matrix Template

The Matrix class is a template taking an element type and number of dimensions

```
using Real_scalar = Matrix<double, 0>;
using Real_vector = Matrix<double, 1>;
using Real_matrix = Matrix<double, 2>;
```

Why not Matrix<N, T>?

Matrix Elements

Are generally assumed to be numeric, but not required

```
Matrix<int, 3> // OK
Matrix<double, 2> // OK
Matrix<string, 2> // OK, but not arithmetic
Matrix<Matrix<int,2>, 2> // OK
```

Can't use arithmetic operators with non-numeric types

Why allow non-numeric types as arguments?

Value Initialization

Initializer lists allows initialization with specific values

```
Matrix<double,0> m0 {0}; // A scalar
Matrix<double,1> m1 {0, 1, 2, 3}; // A 4-vector
Matrix<double,2> m2 { // a 3x4 matrix
  {00, 01, 02, 03},
  {10, 11, 12, 13},
  {20, 21, 22, 23},
};
```

Value Initialization

Even when there are many dimensions

```
Matrix<double,3> m3 { // a 3x4x2 matrix
  {{000,001}, {010,011}, {020,021}, {030, 031}},
  {{100,101}, {110,111}, {120,121}, {130, 131}},
  {{200,201}, {210,211}, {220,221}, {230, 231}}
};
```

Nesting of initializer lists must match number of dimensions

Complex Value Initialization

```
// 2x2 matrix of 2x2 matrices
Matrix<Matrix<int,2>, 2> mm {
  { // row 0
    {{0,1}, {2,3}}, // column 0
    {{4,5}, {6,7}}  // column 1
  },
  { // row 1
    {{a,b}, {c,d}}, // column 0
    {{e,f}, {g,h}}  // column 1
  }
};
```

Shape Initialization

Using the constructor directly creates a matrix with a specified shape (dimensions)

```
Matrix<double,1> m1(3);         // A 3-vector
Matrix<double,2> m2(3,4);       // a 3x4 matrix
Matrix<double,4> m3(4,5,2,6) // a 4x5x6x2 matrix
```

Initialization Patterns

Use {} to initialize a type over a sequence of values

Use () to construct a value with various properties

Initialization Requirements

Number of dimensions and number of elements in each dimension must match

```
using M = Matrix<int, 2>;
M m1(1, 2,3);     // error: too many dimensions
M m2{{1, 2},{2}}; // error: jagged initializers
M m3{1, 2};       // error: too few initializers
                  //        use () for extents
```

Matrix Properties

Order – number of dimensions

Extents – number of elements in each dimension

Size – total number of elements

Descriptor – an object that describes the shape of a matrix

Matrix Order

```
Matrix<double,1> m1(100);
m1.order     // Returns 1
m1.extent(0) // Returns 100
m1.extent(1) // Error: m1 is 1-dimensional
m1.size()    // Returns 100

Matrix<double,2> m2(50,6000);
m2.order     // Returns 2
m2.extent(0) // Returns 50
m2.extent(1) // Returns 6000
m2.size()    // returns 30000
```

Indexing & Subscripting

```
Matrix<double,2> m {
  {00, 01, 02, 03},
  {10, 11, 12, 13},
  {20, 21, 22, 23},
};
double d1 = m(1, 2);       // d1==12 (Fortran-style)
double d2 = m[1][2];       // d2==12 (C-style)
Matrix<double,1> r = m[1] // r is {10,11,12,13}
double d3 = r[2];          // d3==12
```

Rows and Columns

```
Matrix<double,2> m {
  {00, 01, 02, 03},
  {10, 11, 12, 13},
  {20, 21, 22, 23},
};
m.rows()         // Returns 3, same as m.extents(0)
m.columns()      // Returns 4, same as m.extents(1)
Matrix<double, 1> r = m.row(1)      // {10,11,12,13}
Matrix<double, 1> c = m.column(1)   // {01,11,21}
```

Printing

```
template <typename M>
Requires<Matrix_type<M>(), ostream&>
ostream& operator<<(ostream& os, const M& m)
{
  os << ‘{‘;
  for (size_t i = 0; i < m.rows(); ++i) {
    os << m[i];
    if (i+1 != m.rows()) os << ‘,’;
  }
  return os << ‘}’;
}
```

Matrix Requirements

N dimensions (from 0 to many)

Storage for a wide variety of types, not just numeric

Mathematical operations for stored numeric types

Fortran-style subscripting, e.g., `m(1, 2, 3)`

C-style subscripting, e.g., `m[1]` yielding a row

Reading from, writing to, and passing sub-matrices

Design Constraints

Fast

- No unnecessary indirection, allocations, or assignments
- Use move operations to avoid expensive temporaries
- Subscripting must be fast

Safe

- Bounds-check operations? In debug mode
- Absence of resource leaks (basic guarantee or stronger)

Matrix Wish List

Many more mathematical operators: full BLAS support, decompositions, algorithms

Specialized matrices: diagonal, triangular, banded

Sparse matrix support

Parallel execution of Matrix operations

Language Features

Classes (obviously)

Parameterization with types and numbers

Move constructors and assignment operators

RAII

Variadic templates

Initializer lists

Operator overloading

Function objects

Constexpr functions

Default and deleted functions

Template metaprogramming

Static asserts

Matrix Classes

Only 4 classes

```
template <typename T, size_t N> class Matrix;
template <typename T, size_t N> class Matrix_ref;
template <typename T, size_t N> class Matrix_slice;
template <typename T, size_t N> class Slice_iterator;
```

Matrix Template

```
template <typename T, size_t N>
class Matrix
{
  Matrix_slice<N> desc;
  vector<T> elems;
public:
 // Conventional aliases
  static constexpr size_t order = N;
  using value_type = T;
  using iterator = typename std::vector<T>::iterator;
  // Define const_iterator also
```

Matrix Template

```
Matrix() = default;
~Matrix() = default;

// Move construction and assignment
Matix(Matrix&&) = default;
Matrix& operator=(Matrix&&) = default;

// Copy construction and assignment
Matrix(const Matrix&) = default;
Matrix& operator=(const Matrix&) = default;
```

Matrix Template

```
// Converting construction and assignment
template <typename M, typename = Requires<Matrix<M>()>>
  Matrix(const M&);
template <typename M, typename = Requires<Matrix<M>()>>
 Matrix& operator=(const M&);

// Initialize from extents
template <typename... Args>
  Matrix(Args... Args);
```

Matrix Template

```
// Construction and assignment from nested initializers
Matrix(Matrix_initializer<T,N> init);
Matrix& operator=(Matrix_initializer<T,N> init);

// Disable use of {} for extents
template <typename U>
  Matrix(initializer_list<U>) = delete;
template <typename U>
  Matrix& operator=(initializer_list<U>) = delete;
```

Matrix Template

```
size_t extent(size_t n) const
{ return desc.extents[n]; }

size_t rows() const    { return extent(0); }
size_t columns() const { return extent(1); }

size_t size() const { return desc.size; }

const Matrix_slice<N>& descriptor() const
{ return desc; }

T*        data()       { return elems.data(); }
const T* data() const { return elems.data(); }
```

Matrix Template

```
Matrix_ref<T,N-1>         operator[](size_t i);
Matrix_ref<const T, N-1> operator[](size_t i) const;

Matrix_ref<T,N-1>        row(size_t i);
Matrix_ref<const T,N-1> row(size_t i) const;

Matrix_ref<T,N-1>        column(size_t i);
Matrix_ref<const T,N-1> column(size_t i) const;
```

Matrix Template

```
template <typename... Args>
Requires<Index_sequence<Args...>(), T&>
operator()(Args... Args);

template <typename... Args>
Requires<Slice_sequence<Args...>(), Matrix_ref<T,N>>
operator()(Args... Args);

// Define const versions of these functions also
```

Matrix Organization

A single array of contiguous memory (`elems`)

Described by N-D arrays of *extents* and *strides*

- Extent – array bounds in a dimension
- Stride – distance between elements in a dimension
- Start – the starting offset of the matrix elements

`Matrix_slice` describes `Matrix`, `Matrix_ref`

Matrix Slice

Has the following structure; defines an *addressing function* for contiguously allocated objects

```
template <size_t N>
class Matrix_slice {
  ...
  size_t size;
  size_t start;
  size_t extents[N];
  size_t strides[N];
}
```

1-D matrix

0
1
2
3
4
5

```
Matrix<T,1> m(6);

m.desc.start = 0
m.desc.size = 6;
m.desc.extents[1] {6};
m.desc.strides[1] {1};
```

m.elems

0	1	2	3	4	5

2-D matrix

0	1	2	3
4	5	6	7
8	9	10	11

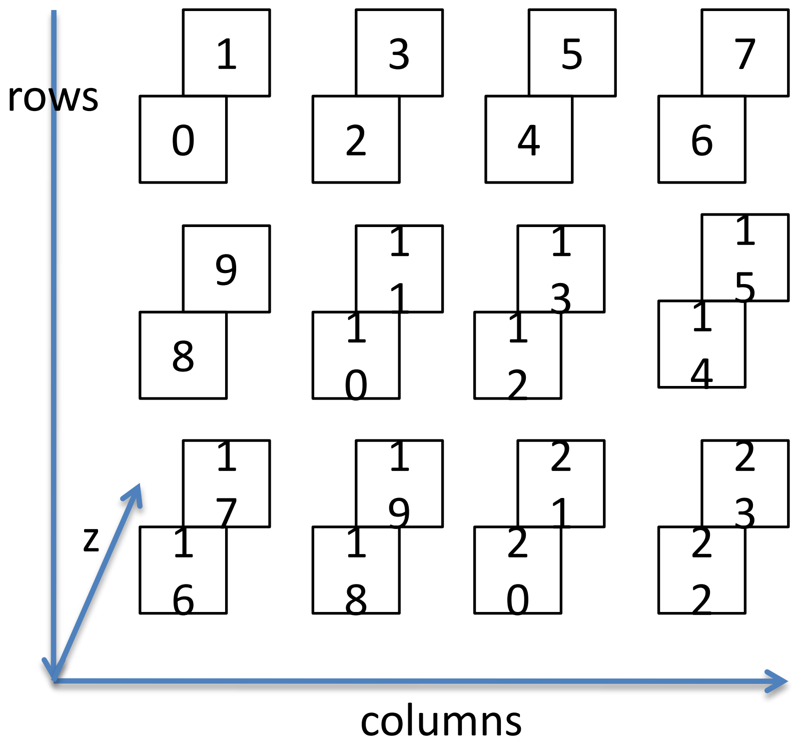
```
Matrix<T,2> m(3,4);

m.desc.start = 0
m.desc.size = 12;
m.desc.extents[2] {3,4};
m.desc.strides[2] {4,1};
```

`m.elems`

0	1	2	3	4	5	6	7	8	9	10	11

3-D matrix




```
Matrix<T,2> m(3,4,2);

m.desc.start = 0
m.desc.size = 24;
m.desc.extents[3] {3,4,2};
m.desc.strides[3] {8,2,1};
```

`m.elems`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…	23

Extent Initialization

Initialize this matrix with a sequence of arguments describing the extents in each dimension

```
template <typename T, size_t N>
class Matrix {

  template <typename... Exts>
  Matrix(Exts... exts)
    : desc(0, {exts...}), elems(desc.size)
  { }
```

Matrix Slice Initialization

Initialize this matrix slice with a sequence of arguments describing the extents in each dimension

```
template <size_t N>
class Matrix_slice {
Matrix_slice(size_t s, initializer_list<size_t> l)
    : start(s)
  {
    copy(l.begin(), l.end(), extents);
    init(); // Compute strides and size
  }
```

Row-major Order

From an array of extents, compute strides that orders elements in row-major order

```
template <std::size_t N>
  void matrix_slice<N>::init()
  {
    strides[N - 1] = 1;
    for (std::size_t i = N - 1; i != 0; --i) {
      strides[i - 1] = strides[i] * extents[i];
    }
    size = extents[0] * strides[0];
  }
```

Matrix Transpose

The transpose of a matrix exchanges rows for columns

Can you do this without resizing or swapping elements in a matrix?

Value Initialization

Initializer lists allows initialization with specific values

```
Matrix<double,0> m0 {0}; // A scalar
Matrix<double,1> m1 {0, 1, 2, 3}; // A 4-vector
Matrix<double,2> m2 { // a 3x4 matrix
  {00, 01, 02, 03},
  {10, 11, 12, 13},
  {20, 21, 22, 23},
};
```

Value Initialization

Even when there are many dimensions

```
Matrix<double,3> m3 { // a 3x4x2 matrix
  {{000,001}, {010,011}, {020,021}, {030,
  031}},
  {{100,101}, {110,111}, {120,121}, {130,
  131}},
  {{200,201}, {210,211}, {220,221}, {230, 231}}
};
```

Nesting of initializer lists must match number of dimensions

Value Initialization

```
template <typename T, size_t N>
class Matrix {
  ...
  Matrix(Matrix_initializer<T,N> init)
  {
    Matrix_impl::derive_extents(init, desc.extents);
    elems.reserve(desc.size);
    Matrix_impl::insert_flat(init, elems);
  }
```

Matrix Initializer

```
Matrix_initializer<T,1>
 // Same as: initializer_list<T>
 // Example: {a, b, c}

Matrix_initializer<T,2>
  // Same as: initializer_list<initializer_list<T>>
  // Example: {{a, b}, {c,d}}

Matrix_initializer<T,3> //
  // Same as: init_list<init_list<init_list<T>>>
  // Example:  {{{a,b,c}, {d,e,f}}, {{g,h,i}, {j,k,l}}}
```

Matrix Initializer

A template alias. Actually, defined by a *type function*, `matrix_init`.

```
template <typename T, size_t N>
  using Matrix_initializer
    = typename matrix_init<T,N>::type;
```

Sometimes a little metaprogramming is necessary

Matrix Initializer Type Function

```
template <typename T, std::size_t N>
struct matrix_init {
  using type = std::initializer_list<
    typename matrix_init<T, N - 1>::type
  >;
};

template <typename T>
struct matrix_init<T, 1> {
  using type = std::initializer_list<T>;
};
```

Value Initialization

```
template <typename T, size_t N>
class Matrix {
  ...
  Matrix(Matrix_initializer<T,N> init)
  {
    Matrix_impl::derive_extents<N>(desc.extents, init);
    elems.reserve(desc.size);
    Matrix_impl::insert_flat(init, elems);
  }
```

Deriving Extents

```
template <size_t N, typename List>
Requires<(N == 1), void>
derive_extents(size_t* p, const List& list) {
  *p = list.size();
}
```

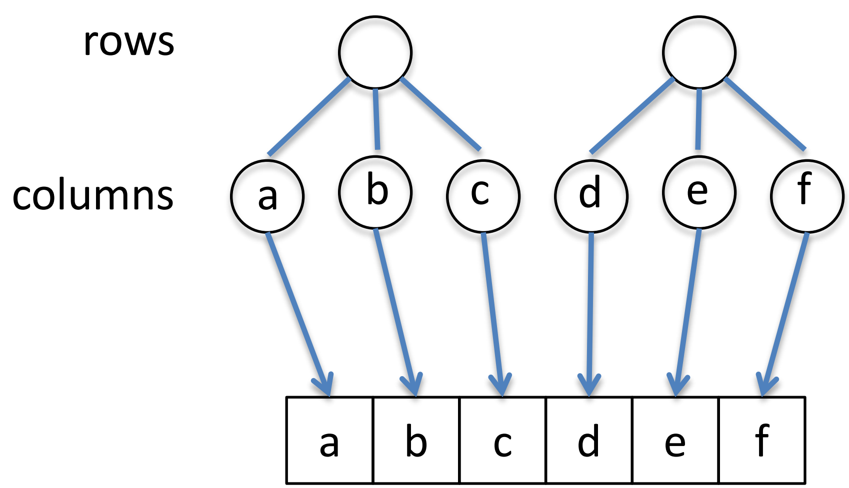
Deriving Extents

```
template <std::size_t N, typename List>
Requires<(N > 1), void>
derive_extents(size_t* p, const List& list) {
  assert(check_non_jagged(list));
  *p = list.size();
  derive_extents<N-1>(++p, *list.begin());
}
```

Insert Flat

Nested Initializer lists define a tree structure. We need to insert elements contiguously

```
{ // 2x3
  {a, b, c},
  {d, e, f}
}
```



Value vs. Extent Initialization

Disable {}'s for extent initialization

```
enum class Piece { none, cross, naught }; // or X and O
...
Matrix<Piece,2> board2(3,3); // OK
Matrix<Piece,2> board3{3,3}; // error: deleted function
```

Why delete this constructor?

Disabling Overloads

```
template <typename T, size_t N>
class Matrix {
  // Constructor values, e.g. {{...}}
  Matrix(Matrix_initializer<T,N> init);

  // Disable {}’s for extent initialization
  template <typename U>
  Matrix(initializer_list<U>) = delete;
```

Surprisingly not ambiguous when N==1. Why?

Fortran-style Subscripting

Use `m(i,j,k)` to request an element from a matrix.

Number of arguments must agree with number of dimensions

```
Matrix<int,2> m = {{01,02,03},{11,12,13}};
auto d1 = m(1);      // Error: too few subscripts
auto d2 = m(1,2,3); // Error: too many subscripts
Auto d3 = m(0, 0);  // Just right, refers to 01
```

Index Sequence

An *index sequence* is a N-vector of indexes (numbers) that refer to an element in an N-D matrix

e.g., `m(i, j, k) // Gets the element at i, j, k`

`{i, j, k}` is an index sequence

Bounds Checking

Where:

`I` is an N-element index sequence

`E` has the extents of a matrix (e.g., `m.desc.extents`)

```
for (size_t i : range(0, N)) I[i] < E[i]
```

Element Offset

Where:

`I` is an N-element index sequence

`S` has the strides of a matrix (e.g., `m.desc.strides`)

```
size_t offset = inner_product (I, S);
```

Subscript Operator

```
template <typename T, size_t N>
Class Matrix {
  // ...
  template <typename... Args>
  Requires<Index_sequence<Args...>(), T&>
  operator()(Args... Args)
  {
    assert(matrix_impl::check_bounds(desc, args...));
    return *(data() + desc(args...));
  }
```

Checking Bounds

```
template <size_t N, typename... Args>
bool
check_bounds(const matrix_slice<N>& slice, Args... args)
{
  size_t I[N] {size_t(args)...};
  less<size_t> lt;
  return equal(I, I+N, slice.extents, lt);
}
```

Element Offset

```
template <size_t N>
class Matrix_slice {
 ...
  template<typename... Args>
  Requires<Index_sequence<Args...>(), size_t>
  operator(Args... args) const
  {
    size_t I[N] {size_t(args)...};
    return inner_product(I, I+N, strides, size_t(0));
  }
```

C-style Subscripts, Rows, & Columns

How do we get rows and columns?

```
matrix<int,2> {{0, 1, 2}, {3, 4, 5}};
m[1] // {3,4,5}
m[1][1] // 4
m.row(1) // {3,4,5}
m.col(1) // {1,4}
```

C-Style Subscripting

Returns a reference to the i^{th} row.

```
template <typename T, size_t N>
class Matrix
{
  Matrix_ref<T,N-1> operator[](size_t i)
  {
    return row(i);
  }
```

Matrix References

Nearly identical to matrix, but it does not own memory

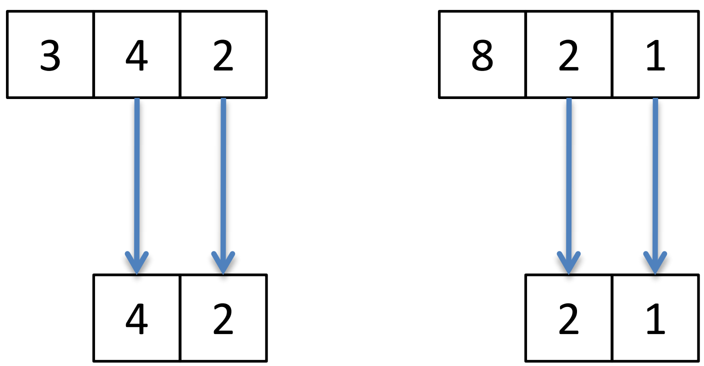
```
template <typename T, size_t N>
class Matrix_ref {
  Matrix_slice<N> desc;
  T* ptr;
Public:
  Matrix_ref(const Matrix_slice<N>& d, T* p)
    : desc(d), ptr(p)
  { }
  ...
```

Row and Column Access

```
template <typename T, size_t N>
class Matrix {
  Matrix_ref<T,N-1> row(size_t i)
  {
    assert(i < rows());
    return {get_row<0>(desc, i), data()};
  }
 Matrix_ref<T,N-1> column(size_t i)
  {
    assert(i < rows());
    return {get_row<1>(desc, i), data()};
  }
```

Describing a Row

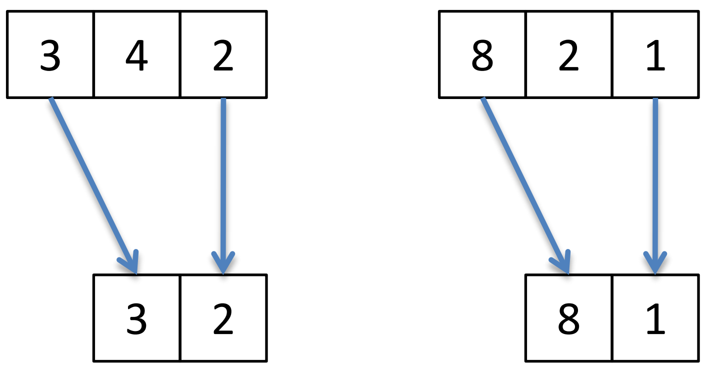
Selecting a row is just a process of making smaller extent and stride vectors



Size and start are adjusted accordingly

Describing a Column

Selecting a column is just a process of making smaller extent and stride vectors



Size and start are adjusted accordingly

Describing a Row

```
template <std::size_t D, std::size_t N>
matrix_slice<N-1>
get_row(const matrix_slice<N>& s, size_t n) {
  matrix_slice<N-1> r;
  r.size = s.size / s.extents[D];
  s.start = s.start + n * s.strides[D];
  auto i = std::copy_n(s.extents, D, r.extents);
  std::copy_n(s.extents + D+1, N-D-1, i);
  auto j = std::copy_n(s.strides, D, r.strides);
  std::copy_n(s.strides + D+1, N-D-1, j);
  return r;
}
```

Slicing

Compute arbitrary sub-matrices from a matrix

```
Matrix<int,2> = {
  {0, 1, 2},
  {3, 4, 5},
  {6, 7, 8}
};

m(slice(1,2), slice(0,3)) // {{3,4,5},{6,7,8}}
m(1, slice::all) // {3, 4, 5} -- same as m.row(0)
m(slice::all, 1) // {1, 4, 7} -- same as m.column(1)
```

Constructing Slices

A slice is a request for elements in a particular dimension of a matrix.

```
slice(i)       // Request elements from i to the end
slice(0)       // Request all elements (0 to end)
slice(i,n)     // Request elements [i,i+n)
slice(i,n,s)   // Request every sth element in [i,i+n)
slice::all     // Request all elements
slice::none    // Not implemented
```

Just a Slice

```
struct slice {
  slice() : slice(-1) {}

  explicit slice(size_t s) : slice(s, -1, -1) {}

  slice(size_t s, size_t l, size_t n=1)
    : start(s), length(l), stride(n) {}

  size_t start;
  size_t length;
  size_t stride;
};
```

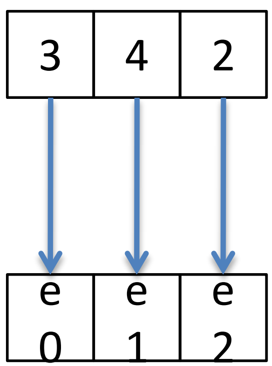
Slicing

Slicing uses the same interface as element access: the function call operator.

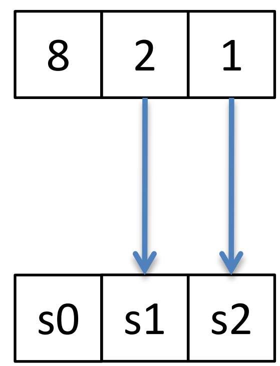
```
template <typename T, size_t N>
class Matrix {
  template <typename... Args>
  Requires<Slice_sequence<Args...>(), submatrix<T,N>>
  operator()(const Args&... args)
  {
    matrix_slice<N> d {desc, args...};
    return {d, data()};
  }
```

Describing a Slice

Selecting a row is just a process of making smaller extent and stride vectors



$E'_i = f(E,S,s.length)$



$S'_i = S_i * s.stride$

Size and start are adjusted accordingly

Slicing with Skipped Dimensions?

Left to the reader as an exercise

Gaussian Elimination

```
void eliminate_row(Matrix& A, Vector& b, size_t j)
{
  const size_t n = A.rows();
  double pivot = A(j, j);
  for (size_t i = j + 1; i < n; ++i) {
    double m = A(i, j) / pivot;
    A[i](slice(j)) =
      scale_and_add(A[j](slice(j)), -m, A[i](slice(j)));
    b(i) -= m * b(j);
  }
}
```

Arithmetic Operations

Two kinds: scalar and matrix operations

All scalar operations are “broadcast” operations

Matrix operations include addition, subtraction, and multiplication

Examples

```
Matrix<int,2> m1 {{1,2,3},{4,5,6}}; // 2x3
Matrix<int,2> m; = m1;        // copy
m1 *= 2;                      // scale: {{2,4,6},{8,10,12}}
Matrix<int,2> m3 = m1+m2;     // add: {{3,6,9},{12,15,18}}
m3 = 0;                       // assign: {{0,0,0},{0,0,0}}

Matrix<int,2> m4 {{1,2},{3,4},{5,6}}; // 3x2
Matrix<int,2> m5 = m1*m4; // {{22,28},{49,64}}
```

Assignment and Addition

Assign or add, to each element, a value x.

```
template <typename T, size_t N>
class Matrix
  Matrix<T,N>& operator=(const T& x) {
    return apply([&](T& a) { a = x; });
  }

  Matrix<T,N>& operator+=(const T& x) {
    return apply([&](T& a) { a += x; });
  }
```

Application

The apply operation allows us to transform the elements of the Matrix

```
template <typename T, size_t N>
class Matrix {
  template <typename F>
  Matrix<T,N>& apply(F f)
  {
    for (auto& x : elems) f(x);
      return *this;
  }
```

Adding Matrices

Add elements of M to the elements of this matrix

```
template <typename T, size_t N>
class Matrix
{
  template <typename M>
  Requires<Matrix_type<M>(), Matrix<T,N>&>
  Matrix<T,N>& operator+=(const M& m) {
    using U = Value_type<M>;
    return apply(m, [](T& a, const U& b) { a += b; });
  }
```

Matrix Application

We can combine this with values of other matrices

```
template <typename T, size_t N>
class Matrix
{
  Matrix<T,N>& apply(const M& m, F f) {
    auto i = begin();
    auto j = m.begin();
    while (i != end())
      f(*i++, *j++);
    return *this;
  }
```