spanny 2:
Rise of std::mdspan

GRISWALD BROOKS

Cppcon
The C++ Conference
2024
September 15 - 20

DISCLAIMER: C++23… ish

goals

- deeper understanding of std::mdspan layouts and accessors
- how to write custom layouts and accessors
- dispel common misconceptions about both

how

- motivations for std::mdspan
- review std::mdspan declaration

- layouts and their requirements
- occupancy grids and default layouts
- submaps, strides, submdspan
- rotated submaps, custom layouts, default constructability
- collision checking, lasers, statefulness

- accessors and their requirements
- pure functional accessors, hilbert, matching types
- roboticists and their lack of beer
- external state memory accessor
- improving memory access using asynchronicity

how

- motivations for std::mdspan
- review std::mdspan declaration

- layouts and their requirements
- occupancy grids and default layouts
- submaps, strides, submdspan
- rotated submaps, custom layouts, default constructability
- collision checking, lasers, statefulness

- accessors and their requirements
- pure functional accessors, hilbert, matching types
- roboticists and their lack of beer
- external state memory accessor
- improving memory access using asynchronicity

how

- motivations for std::mdspan
- review std::mdspan declaration

- layouts and their requirements
- occupancy grids and default layouts
- submaps, strides, submdspan
- rotated submaps, custom layouts, default constructability
- collision checking, lasers, statefulness

- accessors and their requirements
- pure functional accessors, hilbert, matching types
- roboticists and their lack of beer
- external state memory accessor
- improving memory access using asynchronicity


YOUMEME.COM
LET'S GO

std::mdspan

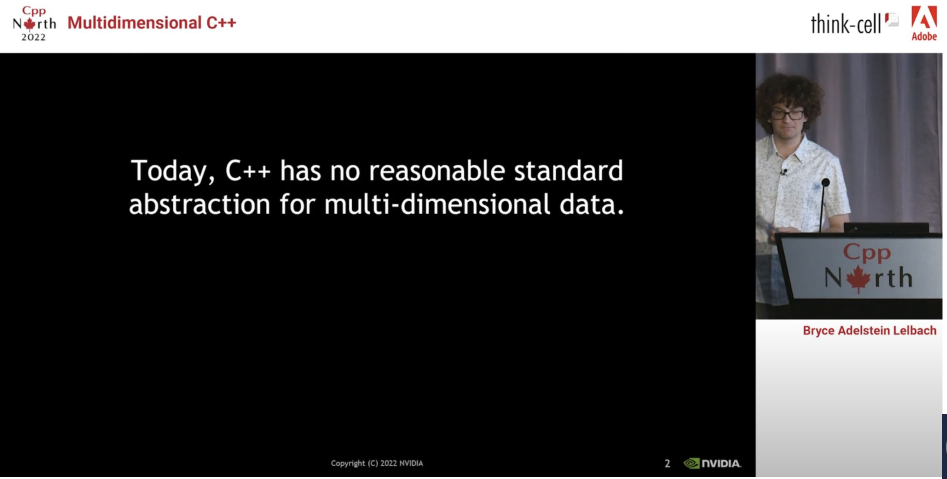
motivations: people who know more than me

I am not a historian…

… but Nevin Liber is!

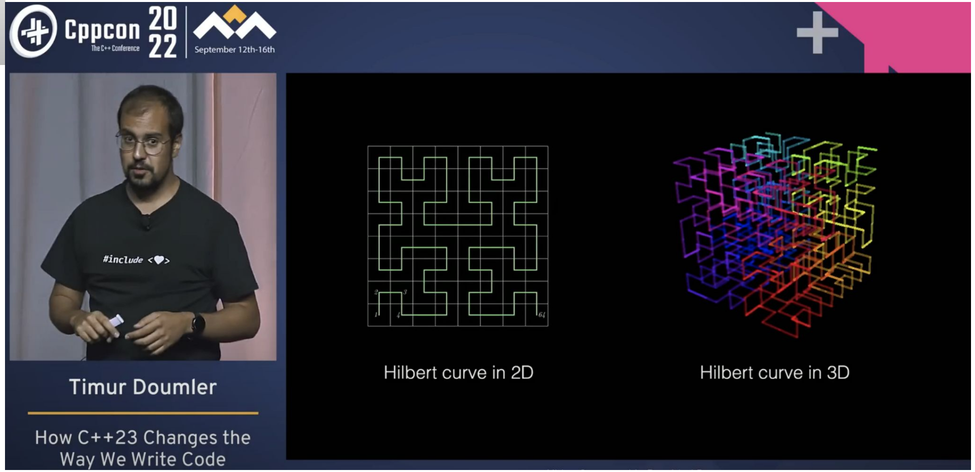
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0009r9.html

motivations: people who know more than me



Bryce Adelstein Lebach
Multidimensional C++
Cpp North

Timur Doumler
How C++23 Changes the Way We Write Code
Cppcon 2022



github.com/griswaldbrooks/spanny2

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0009r9.html

motivations: people who know more than me

Daisy Hollman
Program Chair Emeritus

(who contributed to this talk)

Damien Lebrun-Grandie
Scientific Computing Track Chair

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0009r9.html

motivations

don’t we already have multidimensional arrays?

```
int matrix[N][M];
```

contiguous elements (memory locality)

static dimensions

```
std::vector<std::vector<int>> matrix;
```

dynamic dimensions

rows are not continuous

computers already store things linearly
there is no 2x2 RAM

Let’s map!

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0009r9.html

motivations

```
template <typename T, std::size_t N, std::size_t M>
struct static_matrix {

  T& operator()(std::size_t i, std::size_t j) {
    return data_[i*M + j];
  }

 private:
  std::array<T, N*M> data_;
};
```


https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0009r9.html

motivations

```
template <typename T>
struct dynamic_matrix {
  dynamic_matrix(std::size_t n, size_t m)
  : data_(n*m, T{0}){}

  T& operator()(std::size_t i, std::size_t j) {
    return data_[i*M + j];
  }

 private:
  std::vector<T> data_;
};
```

motivations

this is great! we’ve all written this many times… time to write more!

```
auto a = dynamic_matrix(2, 3);
// view the same data with different dimensions
auto b = a.reshape(3, 2);
auto c = a.reshape(6, 1);
// dimensionality reduction
auto d = c.squeeze();
// get every other element
auto e = d.stride(0, 2, -1);
```


https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0009r9.html

motivations

Need for more flexible and multidimensional view types in C++

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0009r9.html

features of std::mdspan

multi-dimensional view type without memory restrictions

like the custom example, the view is a logical, not physical layout of the data

reshaping doesn’t require any reallocation

mdspans are typically very cheap

customization points via strategy pattern

declaration

```
template<
    class T,
    class Extents,
    class LayoutPolicy,
    class AccessorPolicy
> class mdspan;
```

declaration

```
template<
   class T,
   class Extents,
   class LayoutPolicy,
   class AccessorPolicy
> class mdspan;
```

element type, i.e.
`int, double, position_t, …`

declaration

```
template<
    class T,
    class Extents,
    class LayoutPolicy,
    class AccessorPolicy
> class mdspan;
```

describes the shape of the data

number of dimensions (rank)
length of dimension
type of dimension

ex.

std::extents<std::size_t, 2, 3>;
std::extents<int, 100, 200, 800>;
std::dextents<double, 3>;

declaration

```
template<
  class T,
  class Extents,
  class LayoutPolicy,
  class AccessorPolicy
> class mdspan;
```

maps multidimensional indices to storage locations

`data =`

0	1	2	3	4	5	6	7	8

`row major<3, 3> =`

0	1	2
3	4	5
6	7	8

`column major<3, 3> =`

0	3	6
1	4	7
2	5	8

declaration

```
template<
    class T,
    class Extents,
    class LayoutPolicy,
    class AccessorPolicy
> class mdspan;
```

retrieves values from storage locations

```
int& access(int* p, size_t i) const {
    return p[i];
}
```

layout policy

layout policy

maps multidimensional index to a storage location

```
layout(N,M,Q,R,…) → offset
```

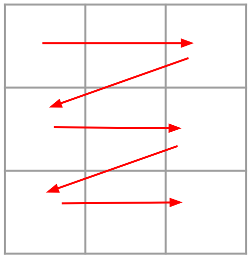
standard layout policies

layout_right = layout_left^T

row major

C++ arrays, numpy

default layout

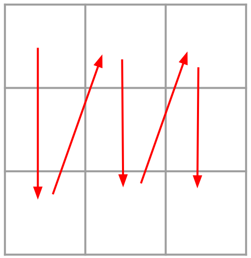


```
layout_right<N, M>(i, j):
    return i*M + j
```

layout_right = layout_left^T

column major

Eigen, Fortran, MATLAB



```
layout_left<N, M>(i, j):
    return i + j*N
```

layout_stride

```
layout_stride<N, M><S0, S1>(i, j):
        return i*S0 + j*S1
```

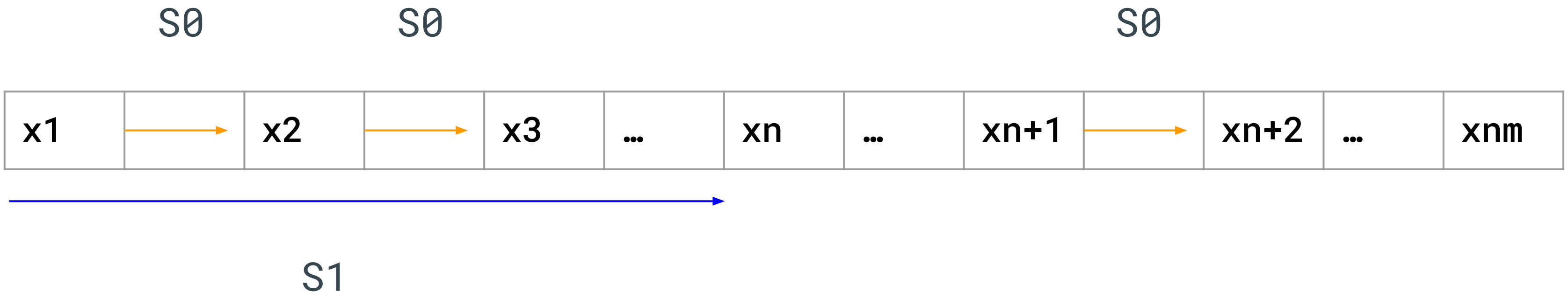
layout_stride

```
window<2, 4> = {{0, 1, 2, 3},
                 {4, 5, 6, 7}}  n
                  m
```

```
n x m window

n is the number of rows in the window

m is the number of elements in a row
```

layout_stride

```
std::layout_stride::mapping{{N, M}, {S0, S1}};
```



layout_stride: 8x4

`S0 = 1, S1 = 8`

0	1	2	3				
4	5	6	7				

`S0 = 2, S1 = 8`

0		1		2		3	
4		5		6		7	

`S0 = 1, S1 = 16`

0	1	2	3				
4	5	6	7				

`S0 = 1, S1 = 6`

0	1	2	3			4	5
6	7						

layout_stride: downsampling

0		1		2		3		4		5		6		7	
8		9		10		11		12		13		14		15	
16		17		18		19		20		21		22		23	
24		25		26		27		28		29		30		31	
32		33		34		35		36		37		38		39	

requirements and type definitions

template parameter describing the shape of the mdspan

```
struct layout_custom {
  template <typename Extents>
  struct mapping {
    Extents const& extents() const;
    ...
};
};
```

mdspan.layout.reqmts

```
size_type operator()(auto indices...)

size_type required_span_size() const

static constexpr bool is_always_unique()

bool is_unique()

static constexpr bool
is_always_exhaustive()

bool is_exhaustive()

static constexpr bool is_always_strided()

bool is_strided()
```

call operator

```
size_type operator()(auto indices...)

size_type required_span_size() const

static constexpr bool is_always_unique()

bool is_unique()

static constexpr bool
is_always_exhaustive()

bool is_exhaustive()

static constexpr bool is_always_strided()

bool is_strided()
```

maps the indices to storage locations

can be defined for any number of indices
or restricted to a single rank

mdspan.layout.reqmts

required span size

```
size_type operator()(auto indices...)

size_type required_span_size() const

static constexpr bool is_always_unique()

bool is_unique()

static constexpr bool
is_always_exhaustive()

bool is_exhaustive()

static constexpr bool is_always_strided()

bool is_strided()
```

number of contiguous elements in the referred data structure needed to contain the spanned elements

0		1		2		3	
4		5		6		7	

4x2 strided layout requires 15 elements for 8 entries

mdspan.layout.reqmts uniqueness

```
size_type operator()(auto indices...)

size_type required_span_size() const

static constexpr bool is_always_unique()

bool is_unique()

static constexpr bool
is_always_exhaustive()

bool is_exhaustive()

static constexpr bool is_always_strided()

bool is_strided()
```

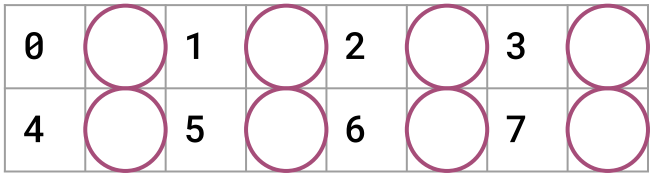
Upper triangular matrix

1	2	3	4	5
2	6	7	8	9
3	7	10	11	12
4	8	11	13	14
5	9	12	14	15

mdspan.layout.reqmts exhausted?

```
size_type operator()(auto indices...)

size_type required_span_size() const

static constexpr bool is_always_unique()

bool is_unique()

static constexpr bool
is_always_exhaustive()

bool is_exhaustive()

static constexpr bool is_always_strided()

bool is_strided()
```

no mapping to these locations



mdspan.layout.reqmts

```
size_type operator()(auto indices...)

size_type required_span_size() const

static constexpr bool is_always_unique()

bool is_unique()

static constexpr bool
is_always_exhaustive()

bool is_exhaustive()

static constexpr bool is_always_strided()

bool is_strided()
```

not exhaustive with consistent offsets

what would inconsistent offsets look like?

what’s not required?

default constructability

statelessness

“simple” computation

unique index mapping

full index mapping

robots?

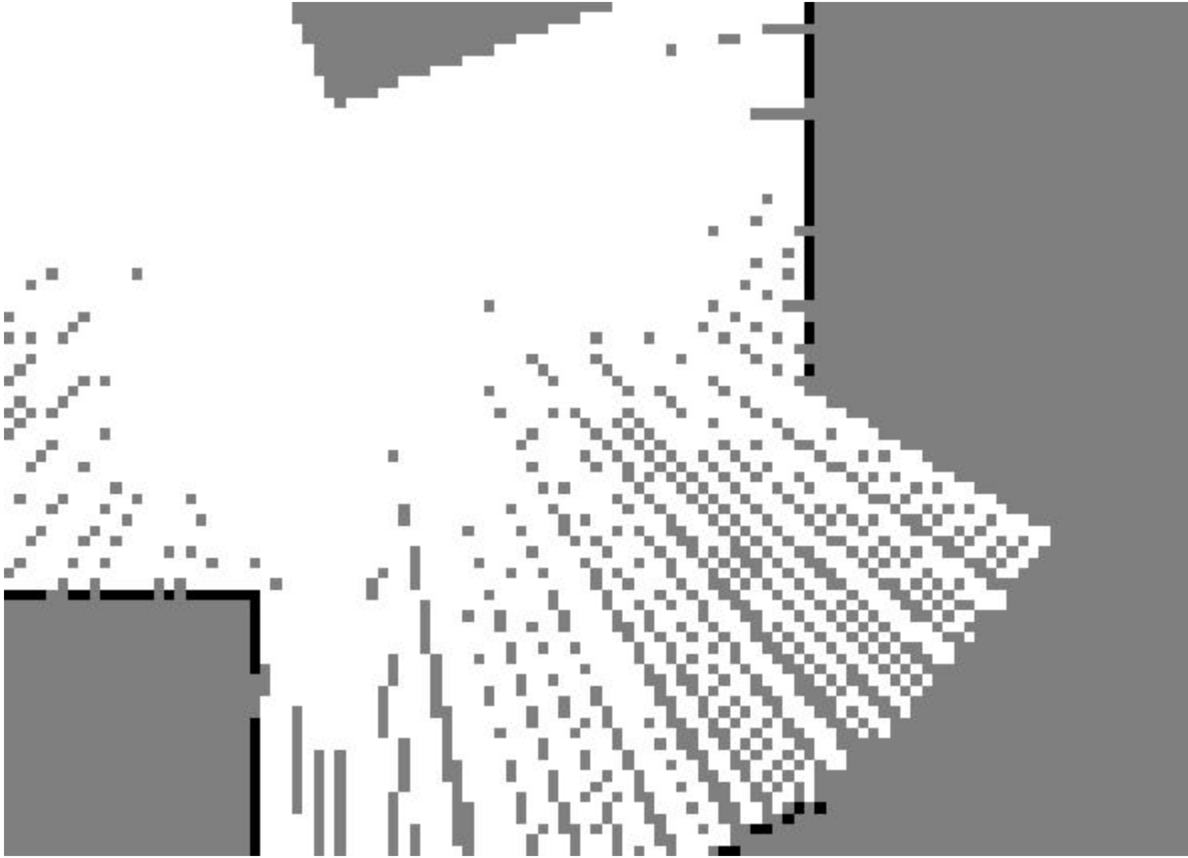
occupancy grids

represents obstacles in the environment

black pixels are occupied

white pixels are clear

grey pixels are unknown



occupancy grid

```
struct occupancy_grid_t {
occupancy_grid_t(std::dextents<std::size_t, 2> shape,
                 unsigned char value = 127)
  : data_(shape.extent(0) * shape.extent(1), value),
    grid_{data_.data(), shape} {}
 private:
  std::vector<unsigned char> data_;
  std::mdspan<unsigned char, std::dextents<std::size_t, 2>> grid_;
};
```

occupancy grid

```
struct occupancy_grid_t {
occupancy_grid_t(std::dextents<std::size_t, 2> shape,
                 unsigned char value = 127)
  : data_(shape.extent(0) * shape.extent(1), value),
    grid_{data_.data(), shape} {}
 private:
  std::vector<unsigned char> data_;
  std::mdspan<unsigned char, std::dextents<std::size_t, 2>> grid_;
};
```

occupancy grid

```
struct occupancy_grid_t {
occupancy_grid_t(std::dextents<std::size_t, 2> shape,
                 unsigned char value = 127)
  : data_(shape.extent(0) * shape.extent(1), value),
    grid_{data_.data(), shape} {}
 private:
  std::vector<unsigned char> data_;
  std::mdspan<unsigned char, std::dextents<std::size_t, 2>> grid_;
};
```

occupancy grid

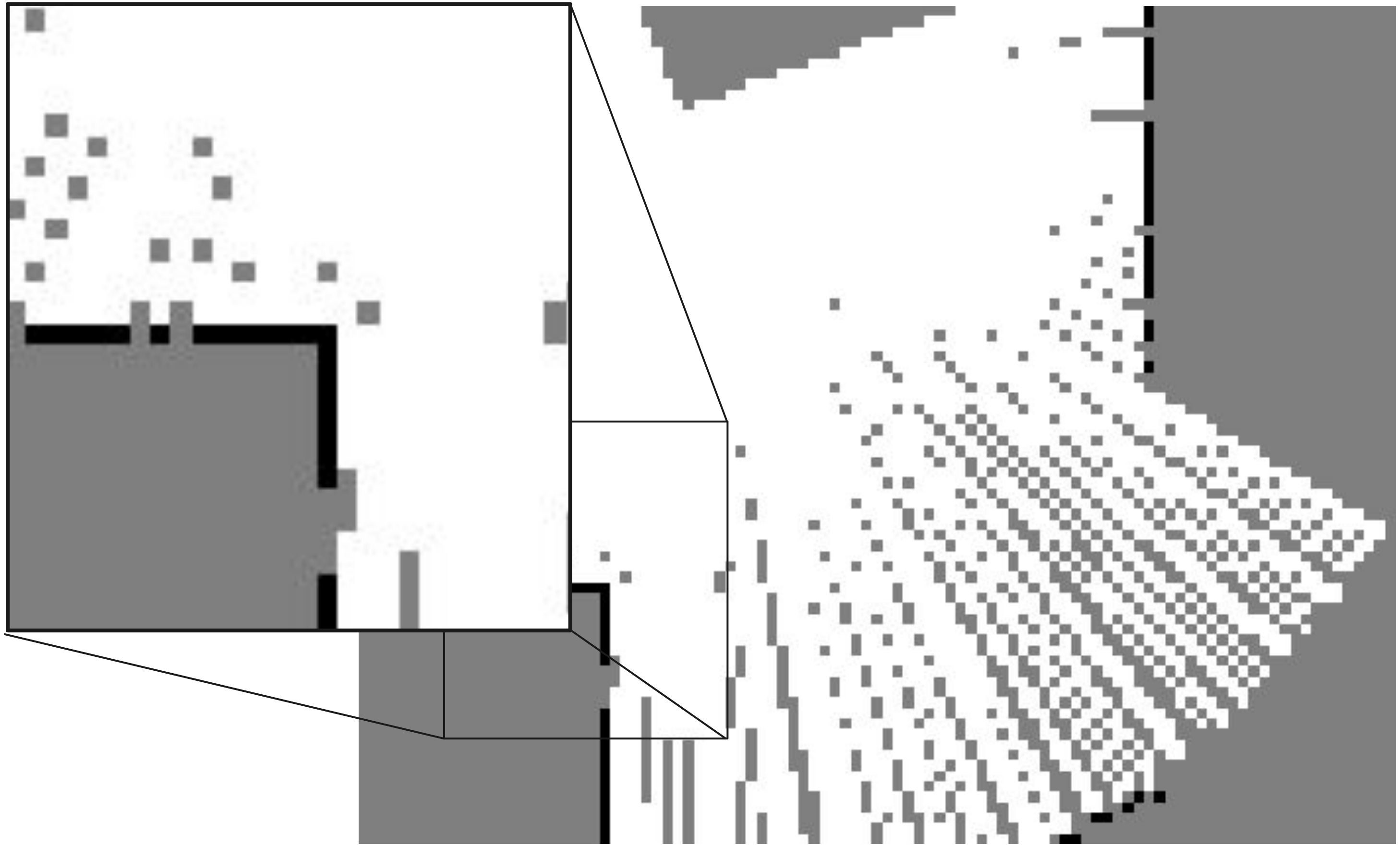
```
struct occupancy_grid_t {
occupancy_grid_t(std::dextents<std::size_t, 2> shape,
                 unsigned char value = 127)
  : data_(shape.extent(0) * shape.extent(1), value),
    grid_{data_.data(), shape} {}
 private:
  std::vector<unsigned char> data_;
  std::mdspan<unsigned char, std::dextents<std::size_t, 2>> grid_;
};
```

submaps

small region within the larger occupancy grid

typically used to constrain the planning scene for navigation

usually centered on the position of the robot



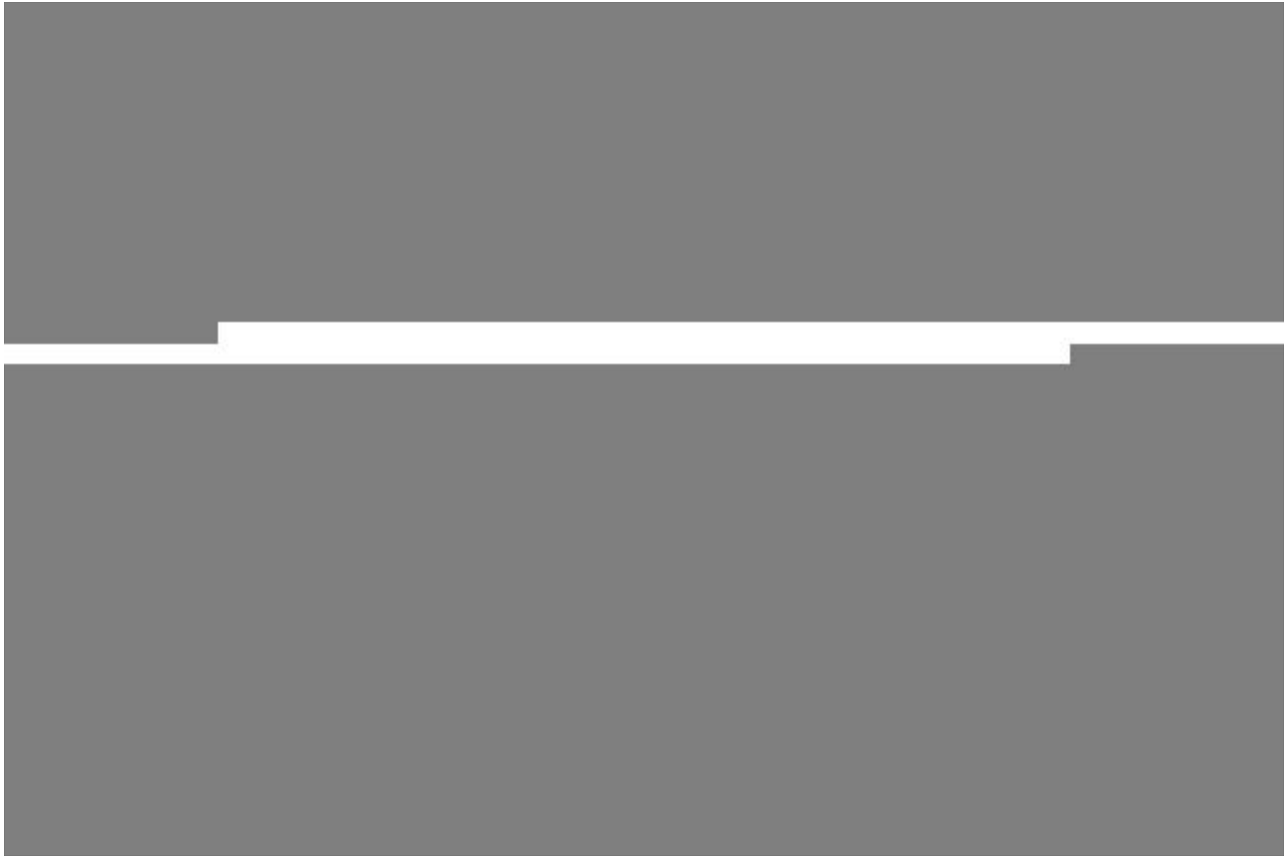
github.com/griswaldbrooks/spanny2

submaps: layout_right

```
auto from_layout_right(extents_t shape, coordinate_t upper_left) {
  auto const height = grid_.extent(0);
  // Constrain upper_left to the grid map…
  // Shrink shape if it goes out of bounds…
  auto const row = upper_left.y * width;
  auto const col = upper_left.x;
  auto first = data_.data();
  std::advance(first, row + col);
  return std::mdspan{first, std::layout_right::mapping{shape}};
}
```

Lets clear a 10x10 submap in a black occupancy grid

submaps: layout_right



submaps: layout_right

```
auto from_layout_right(extents_t shape, coordinate_t upper_left) {
  auto const height = grid_.extent(0);
  // Constrain upper_left to the grid map…
  // Shrink shape if it goes out of bounds…
  auto const row = upper_left.y * width;
  auto const col = upper_left.x;
  auto first = data_.data();
  std::advance(first, row + col);
  return std::mdspan{first, std::layout_right::mapping{shape}};
}
```

Here, shape describes the dimension of the mdspan, not how it maps to the owning `std::vector data_`

submaps: layout_stride

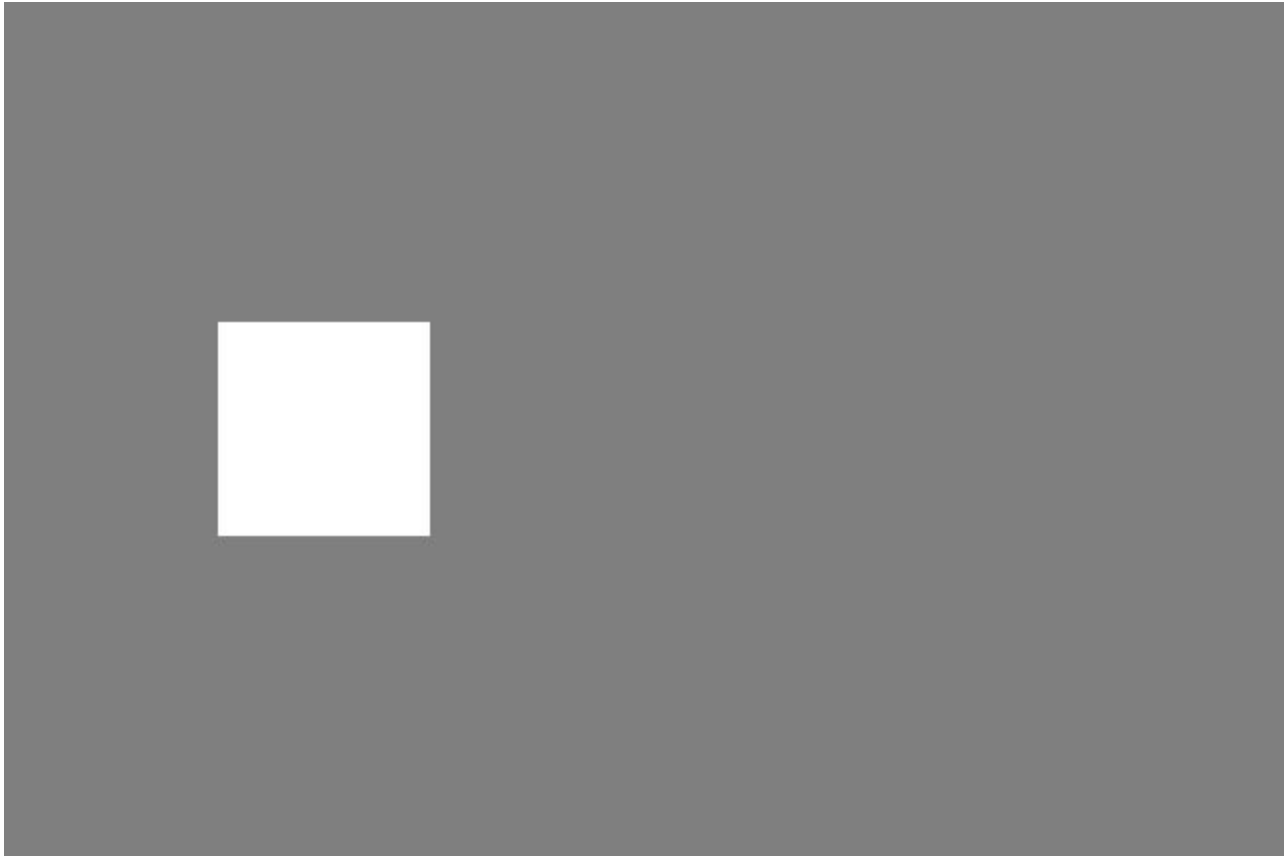
```
auto from_layout_stride(extents_t shape, coordinate_t upper_left) {
  auto const height = grid_.extent(0);
  auto const width = grid_.extent(1);
  auto const strides = std::array<std::size_t, 2>{1, width};
  auto const layout = std::layout_stride::mapping{shape, strides};
  auto const row = upper_left.y * width;
  auto const col = upper_left.x;
  auto first = data_.data();
  std::advance(first, row + col);
  return {first, layout};
}
```

We need to stride the layout!

1 means elements are next to each other

`width` is the number of elements to skip between rows

submaps: layout_stride



submaps: submdspan

```
auto from_submdspan(extents_t shape, coordinate_t upper_left) {
  auto const height = grid_.extent(0);
  auto const width = grid_.extent(1);
  // Constrain upper_left to the grid map…
  // Shrink shape if it goes out of bounds…
  auto const rows = std::tuple{upper_left.y, upper_left.y + shape.extent(1)};
  auto const cols = std::tuple{upper_left.x, upper_left.x + shape.extent(0)};
  return std::submdspan(grid_, rows, cols);
}
```

First index in the row/col

submaps: submdspan

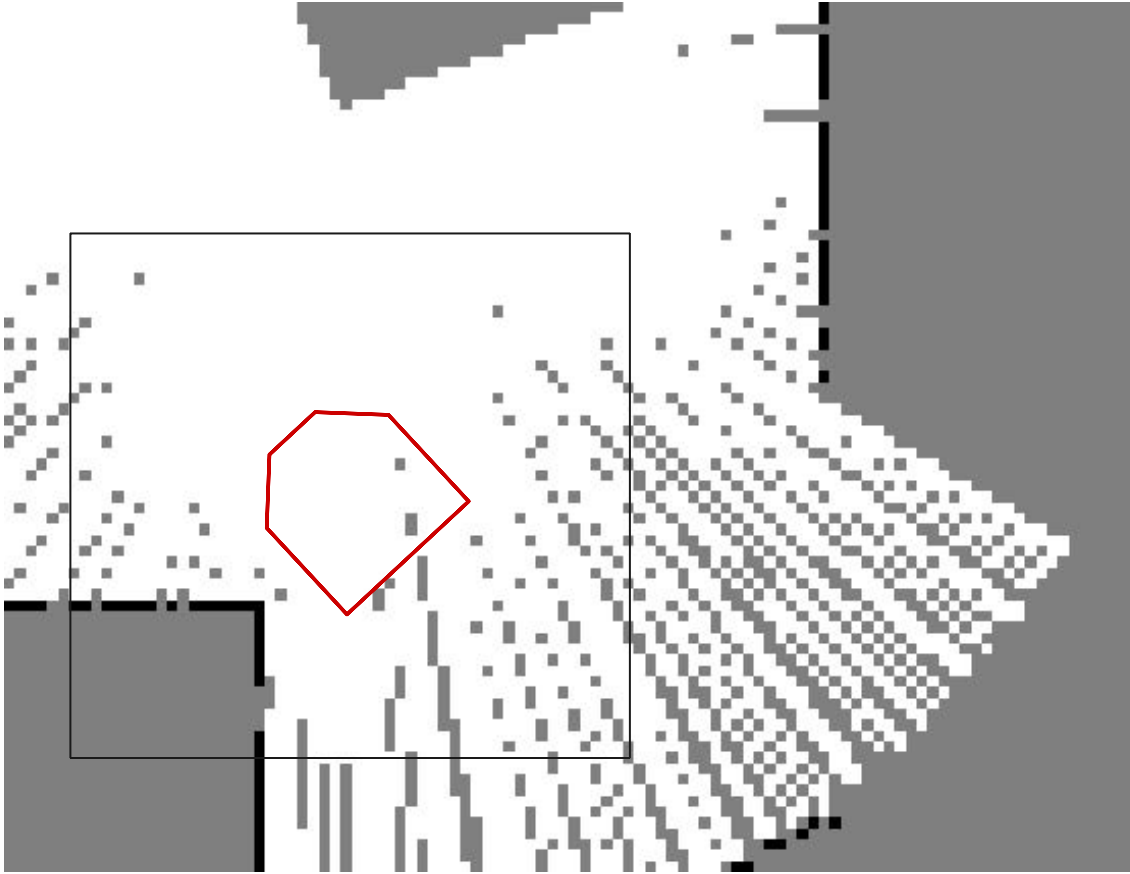
```
auto from_submdspan(extents_t shape, coordinate_t upper_left) {
  auto const height = grid_.extent(0);
  auto const width = grid_.extent(1);
  // Constrain upper_left to the grid map…
  // Shrink shape if it goes out of bounds…
  auto const rows = std::tuple{upper_left.y, upper_left.y + shape.extent(1)};
  auto const cols = std::tuple{upper_left.x, upper_left.x + shape.extent(0)};
  return std::submdspan(grid_, rows, cols);
}
```

Last index in the row/col

rotated submaps

submaps are axis aligned

robot can have a different number of pixels to plan in front of it depending on orientation, which limits the planner



rotated submaps

orientation-aware/rotated submaps have a fixed view relative to the robot meaning a consistent planning scene, increasing determinacy

rotated submaps: custom layout

```
struct layout_rotatable {
  template <typename Extents>
  requires(Extents::rank() == 2) struct mapping {
    using extents_type = Extents;
    constexpr mapping(extents_type parent_shape,
                      extents_type shape,
                      coordinate_t translation,
                      double theta) {...
```

non-default constructor

rotated submaps: custom layout

```
struct layout_rotatable {
  template <typename Extents>
  requires(Extents::rank() == 2) struct mapping {
    using extents_type = Extents;
    constexpr mapping(extents_type parent_shape,
                      extents_type shape,
                      coordinate_t translation,
                      double theta) {...
```

non-default constructor

rotated submaps: custom layout

```
struct layout_rotatable {
  struct mapping {
    constexpr mapping(...):
          ..., span_size_{[this]() {
            size_type max_index = 0;
            for (size_type y = 0; y < shape_.extent(0); ++y)
              for (size_type x = 0; x < shape_.extent(1); ++x)
                auto const index = operator()(x, y);
                if (index > max_index) max_index = index;
            return max_index + 1;
          }()}, ...
      constexpr size_type required_span_size() const noexcept {
        return span_size_;
      }
```

compute **`required_span_size`** by the maximum index created by the vertices

rotated submaps: custom layout

```
struct layout_rotatable {
  struct mapping {
    private:
     extents_type shape_;
     coordinate_t translation_;
     double theta_;
     extents_type parent_shape_;
     size_type span_size_;
  };
};
```

member variables are the layout state

rotated submaps: three skew algorithm

```
layout_rotatable::mapping {
  constexpr size_type operator()(size_type x, size_type y) const {
    double const alpha = -std::tan(theta_ / 2.);
    double const beta = std::sin(theta_);
    auto tx = static_cast<double>(x);
    auto ty = static_cast<double>(y);
    tx += std::round(alpha * ty);
    ty += std::round(beta * tx);
    tx += std::round(alpha * ty);
    x = static_cast<size_type>(std::round(tx)) + translation_.x;
    y = static_cast<size_type>(std::round(ty)) + translation_.y;
    return x + y * parent_shape_.extent(1);
  }
```

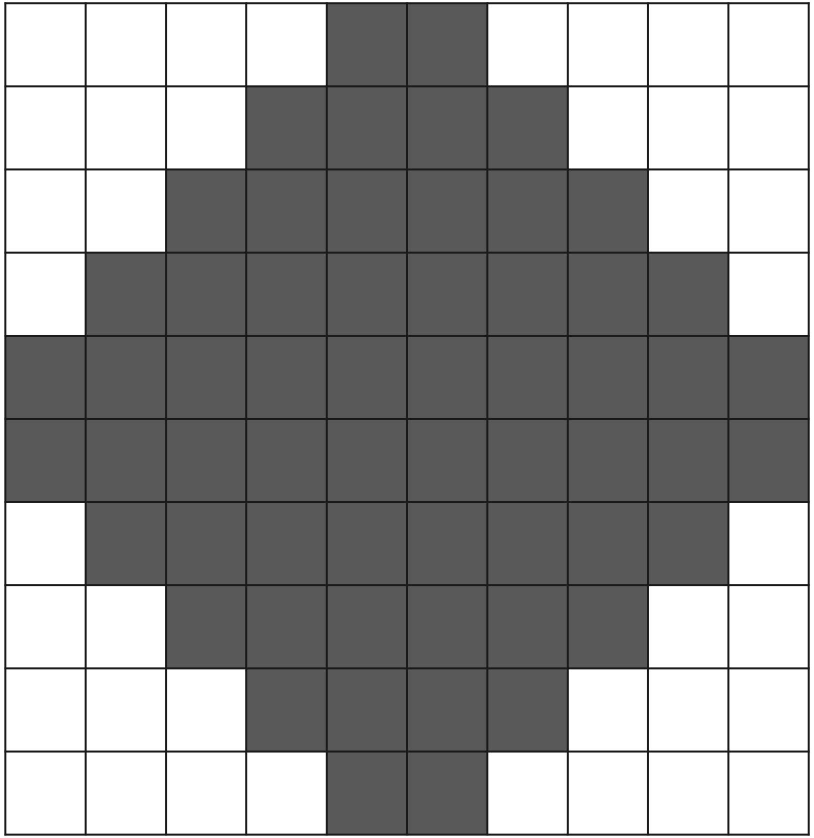
Matt Parker, Rotate without Rotating
https://youtu.be/1LCEiVDHJmc?si=qfv4M3DxXM5pLQet

rotated submaps: custom layout

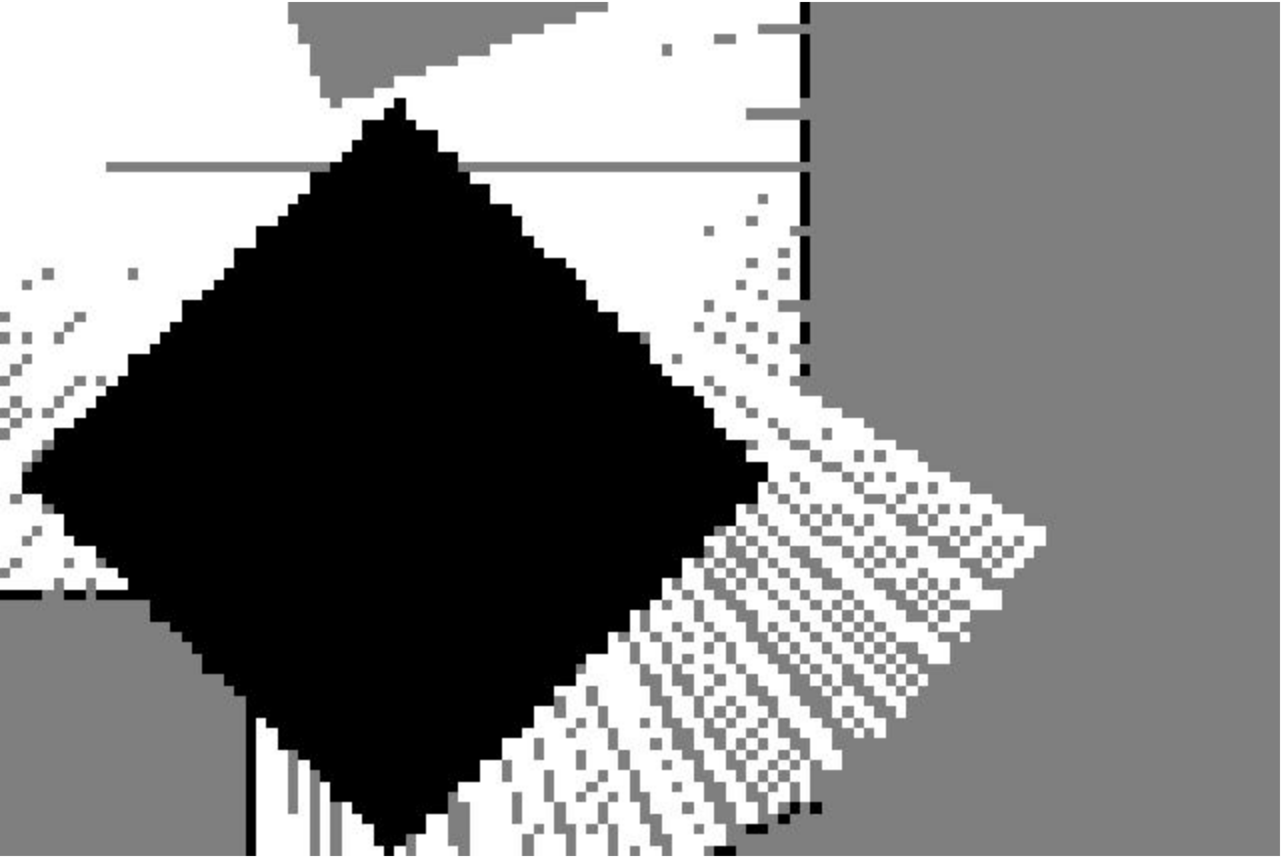
unique

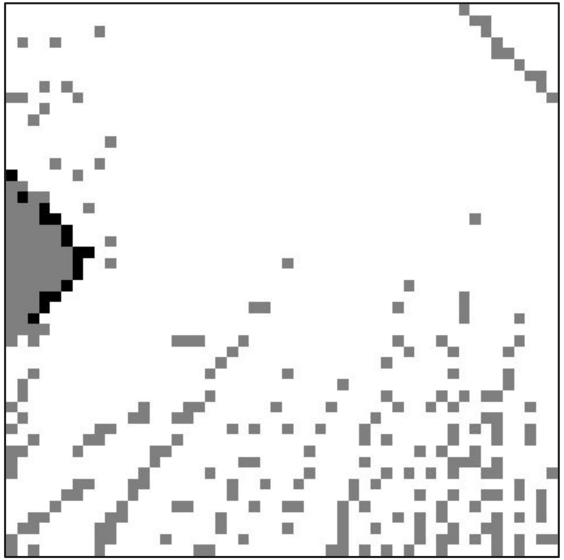
non-exhaustive

not strided



rotated submaps: custom layout





github.com/griswaldbrooks/spanny2

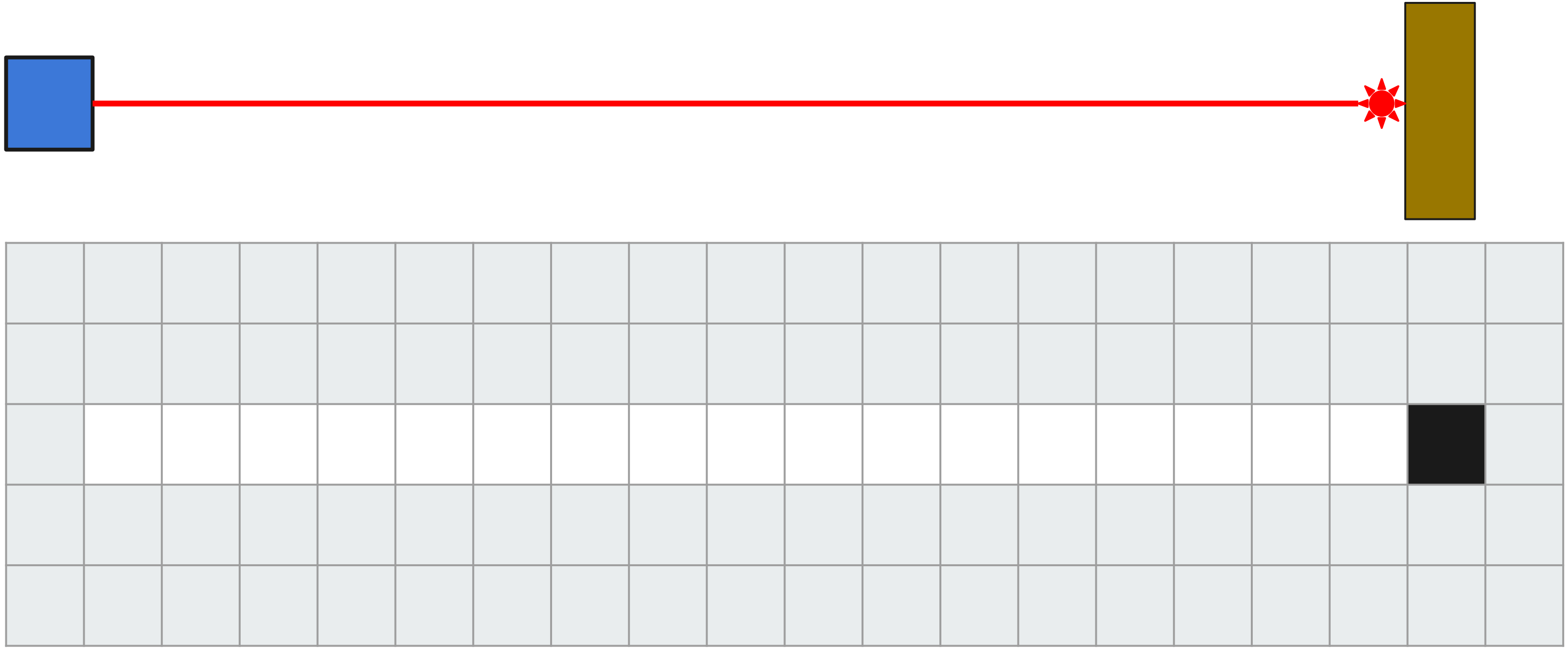
…but what about lasers?

extending layouts to non-square shapes

maintaining state within layouts

dimensionality reduction

range measurements



2d lidars take range measurements with single beams that spin a mirror
usually ~900 measurements in a 270° range at 15 Hz or more

layout_vertices: marking the range measurements

```
struct layout_vertices {
  template <typename Extents>
  requires(Extents::rank() == 1) struct mapping {
    constexpr mapping(std::dextents<std::size_t, 2> parent_shape,
                      std::vector<coordinate_t> vertices,
                      coordinate_t const& p, double theta):...
```

for non-square shapes, 2D iteration no longer makes sense
vertices come from the lidar range measurements
coordinates are in no way contiguous

layout_vertices

```
layout_vertices::mapping {
         vertices_{[&] {
           std::set<coordinate_t, coord_compare> unique_vertices;
           for (auto& v : vertices)
             // transform coordinates using
             // translation and rotation
             // ...
             if (in_bounds(v, parent_shape_))
               unique_vertices.insert(v);

           return std::vector<coordinate_t>{unique_vertices.begin(),
                                            unique_vertices.end()};

         }()
```

computing unique coordinates at construction storing them as state

layout_vertices

```
layout_vertices::mapping {
    constexpr size_type operator()(size_type i) const {
      auto const& coord = vertices_[i];
      auto const x = static_cast<size_type>(coord.x);
      auto const y = static_cast<size_type>(coord.y);
      auto const offset = x + y * parent_shape_.extent(1);
      return offset;
    }
```

1d iteration
picks coordinate from the `vertices_` and returns the value in parent occupancy grid

layout_vertices

```
std::mdspan<unsigned char, std::dextents<std::size_t, 1>, layout_vertices>
from_vertices(std::vector<coordinate_t> const& endpoints,
              coordinate_t const& p, double theta) {
  layout_vertices::mapping<std::dextents<std::size_t, 1>>
  scan{grid_.extents(), endpoints, p, theta};
  return {data_.data(), scan};
}
```

layout_vertices

```cpp
std::mdspan<unsigned char, std::dextents<std::size_t, 1>, layout_vertices>

from_vertices(std::vector<coordinate_t> const& endpoints,
              coordinate_t const& p, double theta) {

  layout_vertices::mapping<std::dextents<std::size_t, 1>>
  scan{grid_.extents(), endpoints, p, theta};

  return {data_.data(), scan};
}
```

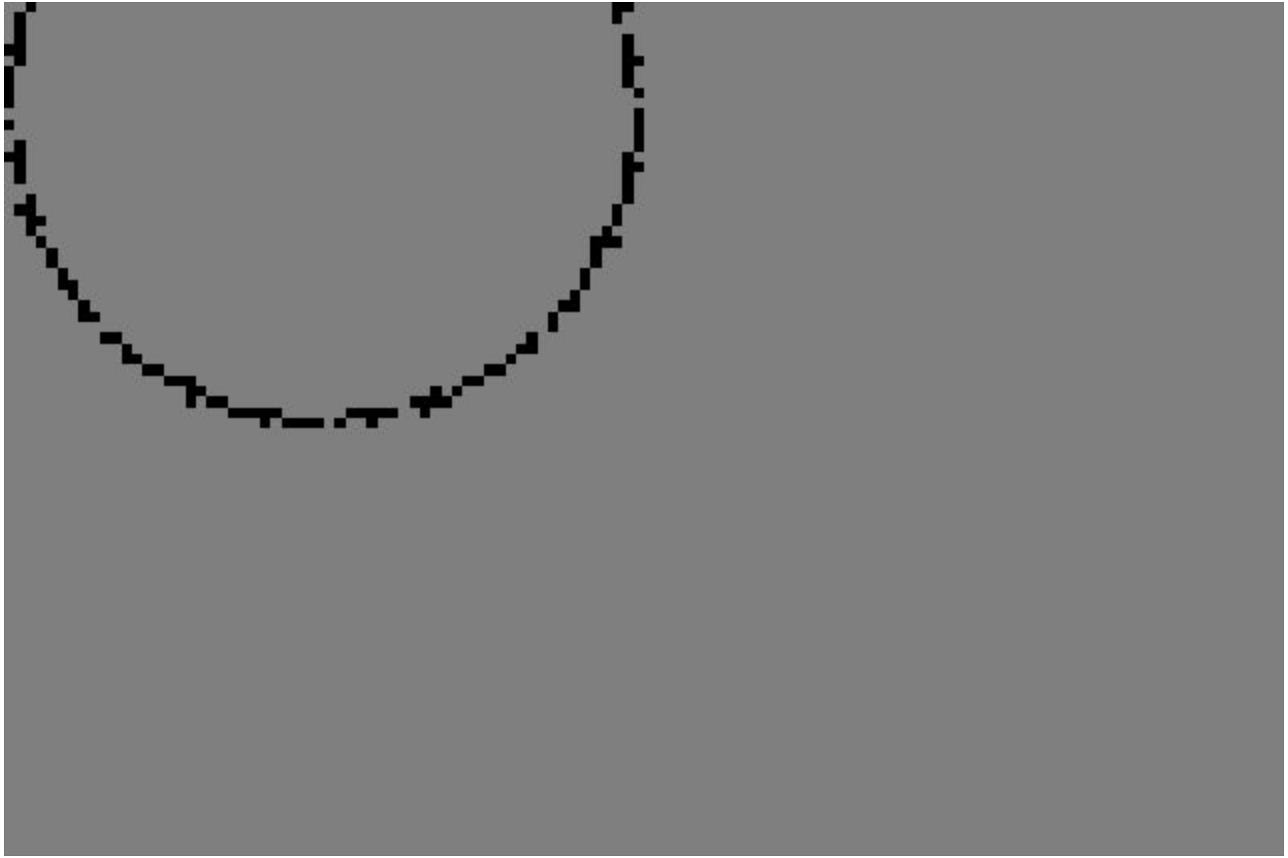
creates **`std::mdspan`** with one dimensional extent

layout_vertices

```
std::mdspan<unsigned char, std::dextents<std::size_t, 1>, layout_vertices>

from_vertices(std::vector<coordinate_t> const& endpoints,
              coordinate_t const& p, double theta) {

  layout_vertices::mapping<std::dextents<std::size_t, 1>>
  scan{grid_.extents(), endpoints, p, theta};

  return {data_.data(), scan};
}
```

used by the mapping template parameter to create the laser scan from endpoints

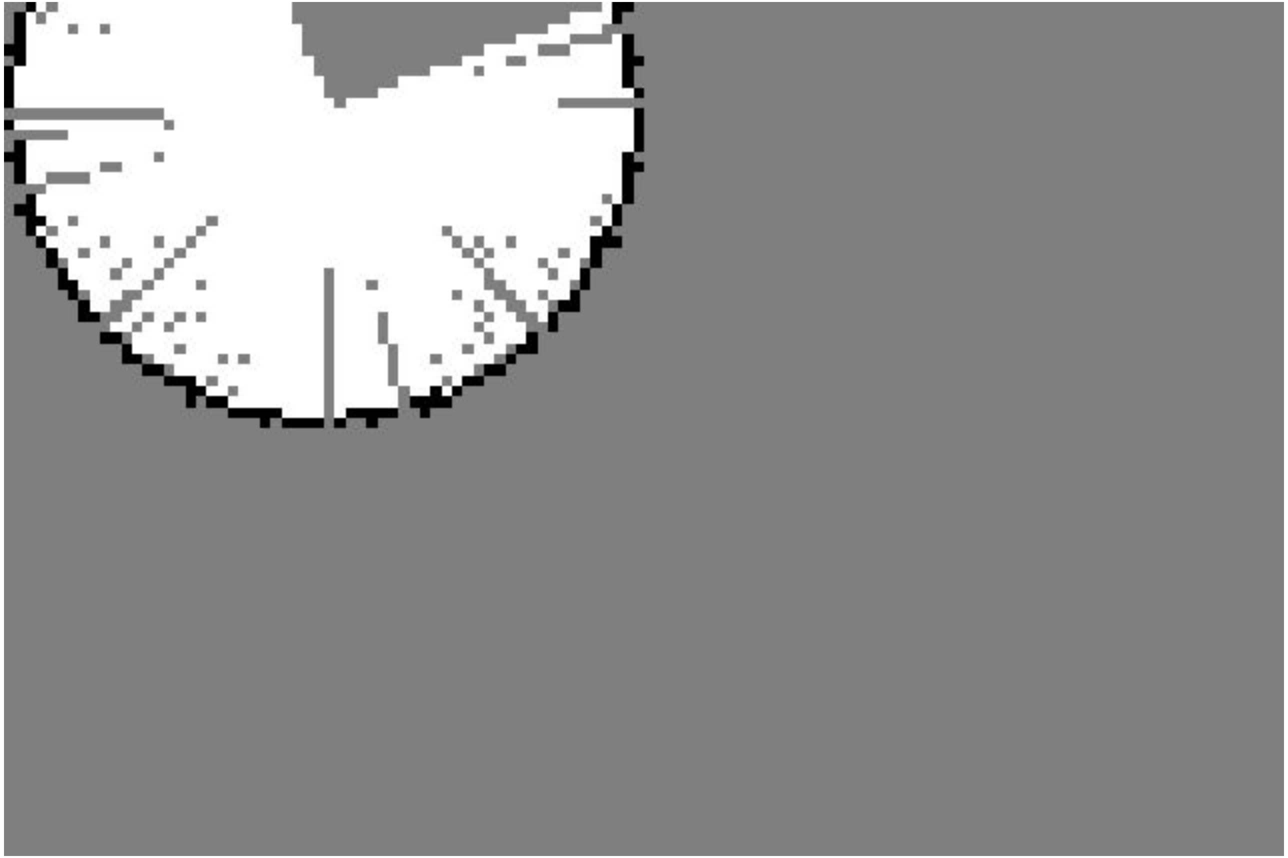
range measurements in map



layout raytrace: clearing the in the range

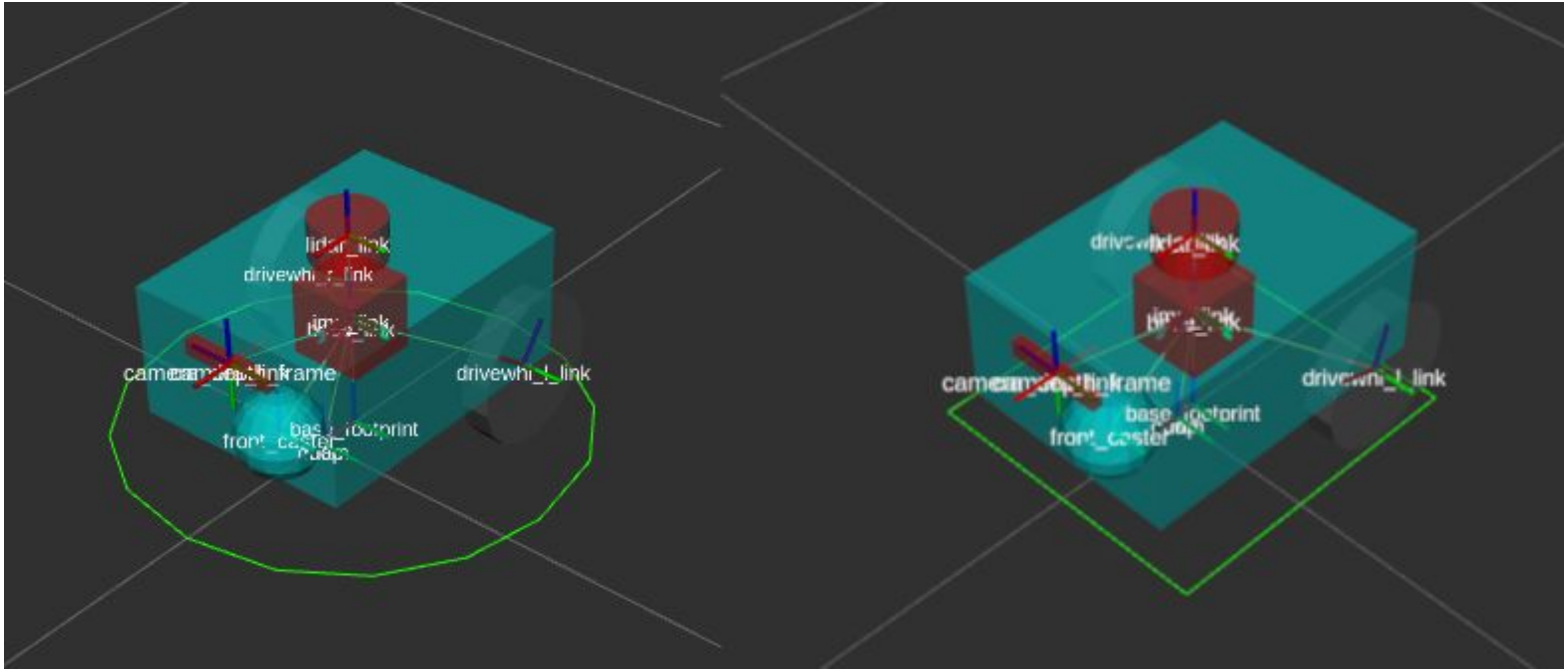
```
struct layout_raytrace::mapping {
  rays_{[&] {
    std::set<coordinate_t, coord_compare> unique_cells;
    for (auto& v : vertices)
      // transform coordinates using translation and rotation
      // ...
      auto const ray = raytrace(p, v, std::numeric_limits<int>::max());
      for (auto const& pixel : ray)
        if (in_bounds(pixel, parent_shape_))
          unique_cells.insert(pixel);

      return std::vector<geometry::Cell>{unique_cells.begin(),
                                                unique_cells.end()};

  }()}
```

laser scan in map



robot footprints and collision checking

mobile robots are usually modelled as a polygon projected onto the ground, for planning purposes



github.com/griswaldbrooks/spanny2

https://docs.nav2.org/setup_guides/footprint/setup_footprint.html

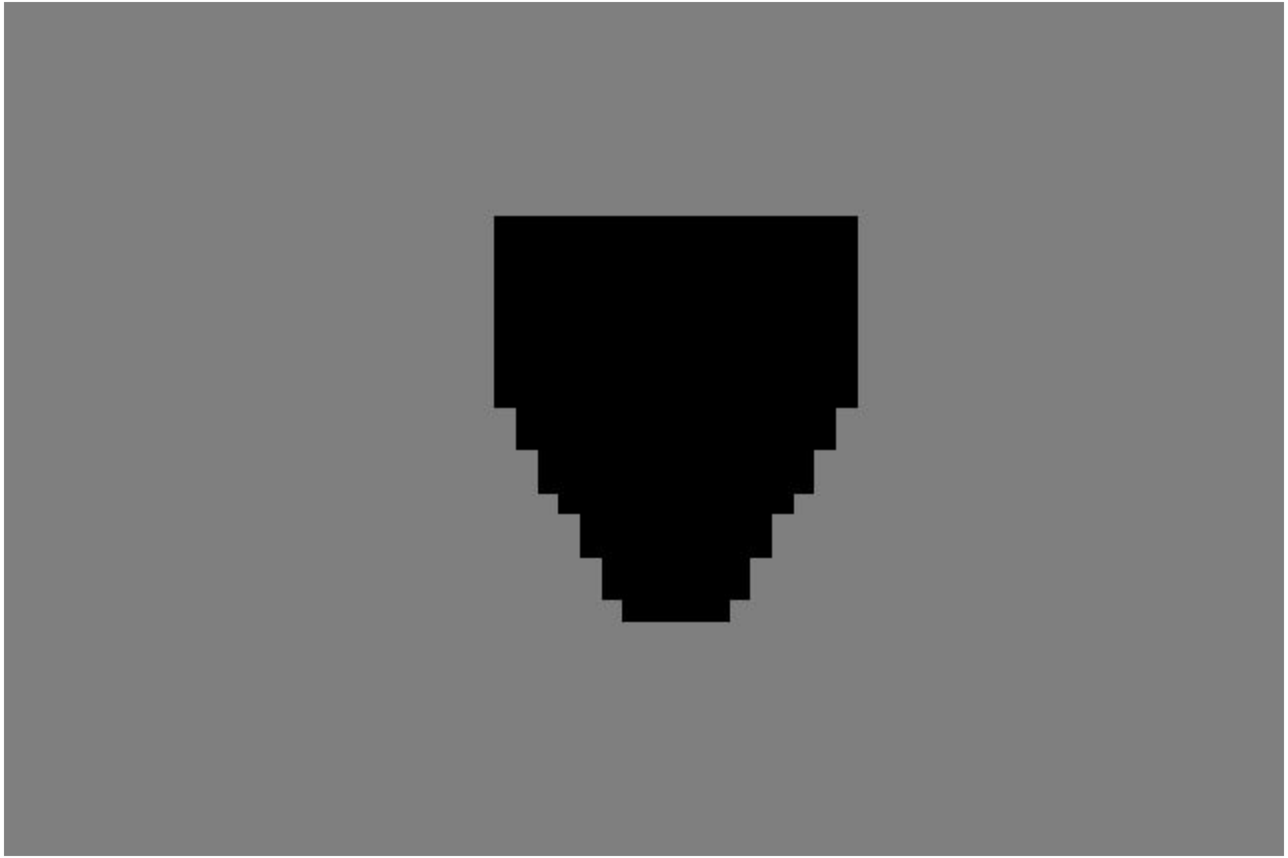
layout_polygonal

```
struct layout_polygonal::mapping {
  polygon_{[&] {
    for (auto& v : vertices) {
      auto x = v.x * std::cos(theta) - v.y * std::sin(theta) + p.x;
      auto y = v.x * std::sin(theta) + v.y * std::cos(theta) + p.y;
      v.x = x;
      v.y = y;
    }
    // create filled footprint
    return convex_fill(vertices, {parent_shape_.extent(0),
                                  parent_shape_.extent(1)});

  }()},
```

driving robot

can visualize robot in map

path planner will use this footprint to check for collisions



few! that was a lot

- learned what a layout is and its requirements
- explored built-in layouts
- created custom layouts with state and non-default constructors
- discussed dimensionality reduction

…but what if our data doesn’t live on the stack or heap?

accessor policy

accessor policy

retrieves values from storage locations

```
template <class ElementType>
struct default_accessor {
  using element_type = ElementType;
  using reference = ElementType&;
  using data_handle_type = ElementType*;
  constexpr reference access(data_handle_type p, size_t i) const noexcept {
    return p[i];
  }
};
```


https://github.com/kokkos/mdspan/blob/stable/include/experimental/__p0009_bits/default_accessor.hpp

accessor policy: element_type

retrieves values from storage locations

```
template <class ElementType>
struct default_accessor {
  using element_type = ElementType;
  using reference = ElementType&;
  using data_handle_type = ElementType*;
  constexpr reference access(data_handle_type p, size_t i) const noexcept {
    return p[i];
  }
};
```

matches

`std::mdspan::element_type`

i.e.

`std::mdspan<T, ...>`

https://github.com/kokkos/mdspan/blob/stable/include/experimental/__p0009_bits/default_accessor.hpp

accessor policy: reference

retrieves values from storage locations

```
template <class ElementType>
struct default_accessor {
  using element_type = ElementType;
  using reference = ElementType&;
  using data_handle_type = ElementType*;
  constexpr reference access(data_handle_type p, size_t i) const noexcept {
    return p[i];
  }
};
```

is what is returned by

`default_accessor::access`

and

`std::mdspan::operator[](indices...)`

https://github.com/kokkos/mdspan/blob/stable/include/experimental/__p0009_bits/default_accessor.hpp

accessor policy: data_handle_type

retrieves values from storage locations

```
template <class ElementType>
struct default_accessor {
  using element_type = ElementType;
  using reference = ElementType&;
  using data_handle_type = ElementType*;
  constexpr reference access(data_handle_type p, size_t i) const noexcept {
    return p[i];
  }
};
```

first parameter of
  `default_accessor::access`
and
  `std::mdspan(data_handle_type p, …)`

https://github.com/kokkos/mdspan/blob/stable/include/experimental/__p0009_bits/default_accessor.hpp

accessor policy: recap

`element_type`	matches	`std::mdspan::element_type` `std::mdspan<T, ...>`
`reference`	is returned by	`default_accessor::access` `std::mdspan::operator[](indices…)`
`data_handle_type`	first parameter of	`default_accessor::access` `std::mdspan(data_handle_type p, …)`

accessor policy: not a requirement

`element_type` ⇏ `reference` ⇏ `data_handle_type`

also not required

specific template parameters

default constructability

memory contiguity

referring to memory

accessor policy

accessors can call arbitrary functions

consider the Hilbert matrix...

$$H_{ij} = \frac{1}{i+j-1}.$$

$$H = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} \end{bmatrix}.$$

...?

hilbert accessor policy

```
template <std::size_t Width>
struct hilbert_accessor_t {
  using element_type = double;
  using reference = element_type;
  using size_type = std::size_t;
  using data_handle_type = std::function<double(size_type, size_type)>;

  reference access(data_handle_type p, size_type offset) const {
    return p(offset, Width);
  }
};
```

reference type here is a value type

otherwise you get a compiler error of “returned reference to temporary” 🤔

accessor policy

```
auto hilbert_function = [](std::size_t ndx, std::size_t width) {
  auto const n = static_cast<double>(ndx);
  auto const M = static_cast<double>(width);
  return 1. / (std::floor(n / M) + std::fmod(n, M) + 1);
};
```


https://en.wikipedia.org/wiki/Hilbert_matrix

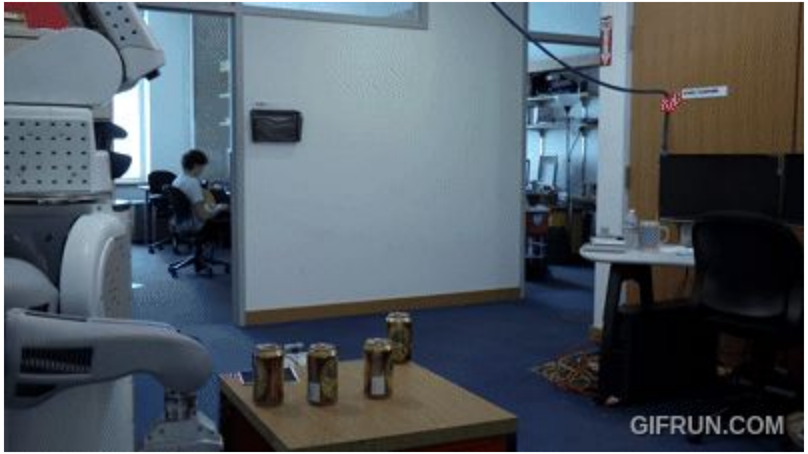
accessor policy

```
using hilbert_view_t = std::mdspan<double,
                                   std::extents<std::size_t, 3, 3>,
                                   std::layout_right,
                                   hilbert_accessor_t<3>>;

auto hilbert_view = hilbert_view_t{hilbert_function};

assert(1./5. == hilbert_view(2, 2));
```

no memory in sight 🐏🔍👀

where's the beer?

perennial problem: not enough beer



PR2 Robot Fetches Beer from the Refrigerator
Eitan Marder-Eppstein

Beerbots: Cooperative Beer Delivery Robots
Ariel Anders

https://youtu.be/c3Cq0sy4TBs?si=FWrMCQKZC-dGz0wn

https://youtu.be/Jfzun9pP74U?si=dvvz8UjZjkr4u4Iq

where's the beer?

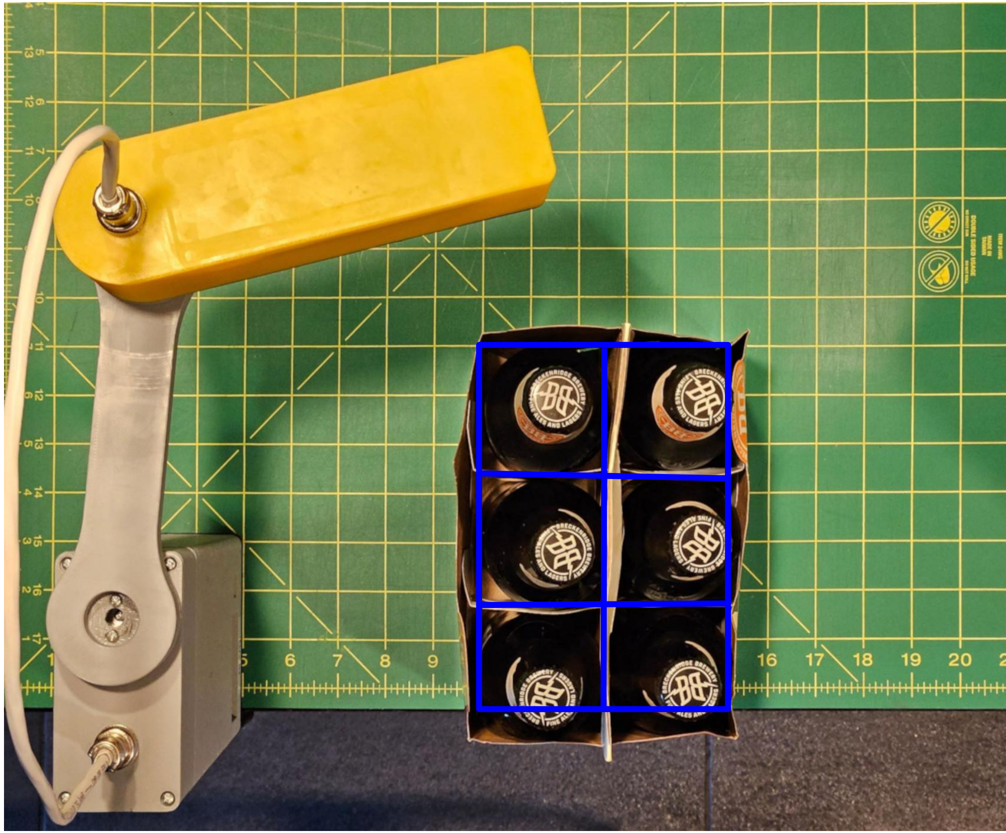
six pack has

```
std::extents<std::size_t, 2, 3>
```

```
element_type = bin_state{
bin_occupancy, bin_position};
```

robot arm has ir digital distance sensor on end effector

accessor policy commands robot arm to bin position and reads state via serial port



synchronous single arm control

```cpp
template <typename Arbiter>
struct bin_checker_t {
  using element_type = std::expected<bin_state, std::string>;
  using reference = element_type;
  using data_handle_type = position_t*;
  using size_type = std::size_t;

  explicit bin_checker_t(Arbiter* checker) : checker_{std::move(checker)} {}

  reference access(data_handle_type bin_positions, size_type offset) const {
    auto const goal = bin_positions[offset];
    return (*checker_)(goal);
  }
 private:
  Arbiter* checker_;
};
```

synchronous single arm control

```
template <typename Arbiter>
struct bin_checker_t {
  using element_type = std::expected<bin_state, std::string>;
  using reference = element_type;
  using data_handle_type = position_t*;
  using size_type = std::size_t;

  explicit bin_checker_t(Arbiter* checker) : checker_{std::move(checker)} {}

  reference access(data_handle_type bin_positions, size_type offset) const {
    auto const goal = bin_positions[offset];
    return (*checker_)(goal);
  }
 private:
  Arbiter* checker_;
};
```

not related

synchronous single arm control

```
template <typename Arbiter>
struct bin_checker_t {
  using element_type = std::expected<bin_state, std::string>;
  using reference = element_type;
  using data_handle_type = position_t*;
  using size_type = std::size_t;

  explicit bin_checker_t(Arbiter* checker) : checker_{std::move(checker)} {}

  reference access(data_handle_type bin_positions, size_type offset) const {
    auto const goal = bin_positions[offset];
    return (*checker_)(goal);
  }
 private:
  Arbiter* checker_;
};
```

not a reference type

synchronous single arm control

```
template <typename Arbiter>
using bin_view_t =
std::mdspan<typename bin_checker_t<Arbiter>::element_type,
            // We're treating our 6 bins as a 2x3 matrix
            std::extents<std::size_t, 2, 3>,
            // Our layout should do bounds-checking
            bounds_checked_layout,
            // Tell the robot to access the bin
            bin_checker_t<Arbiter>>;
```

synchronous single arm control

```
auto bin_positions = /* read from json */;
auto left_arm = /* create kinematic model of robot arm */;

auto arbiter = arbiter_single{left_arm, std::make_unique<hardware>("/dev/ttyACM0")};

auto bin_checker = bin_checker_t{&arbiter};

auto bins = bin_view_t(bin_positions.data(), {}, bin_checker);

for (auto i = 0u; i != bins.extent(0); ++i) {
  for (auto j = 0u; j != bins.extent(1); ++j) {

    std::cout << bins(i, j) << "\n";

  }
}
```

synchronous single arm control

commanding the robot arm with `arbiter_single` that reading in all of the bin positions and arm kinematics and computes the inverse kinematic linearly interpolated path from the home position of the robot arm and the goal position, commanding the sequence of joint angles through serialization command to the arduino and then reading the ir sensor…

read the code 😂

github.com/griswaldbrooks/spanny2

synchronous single arm control

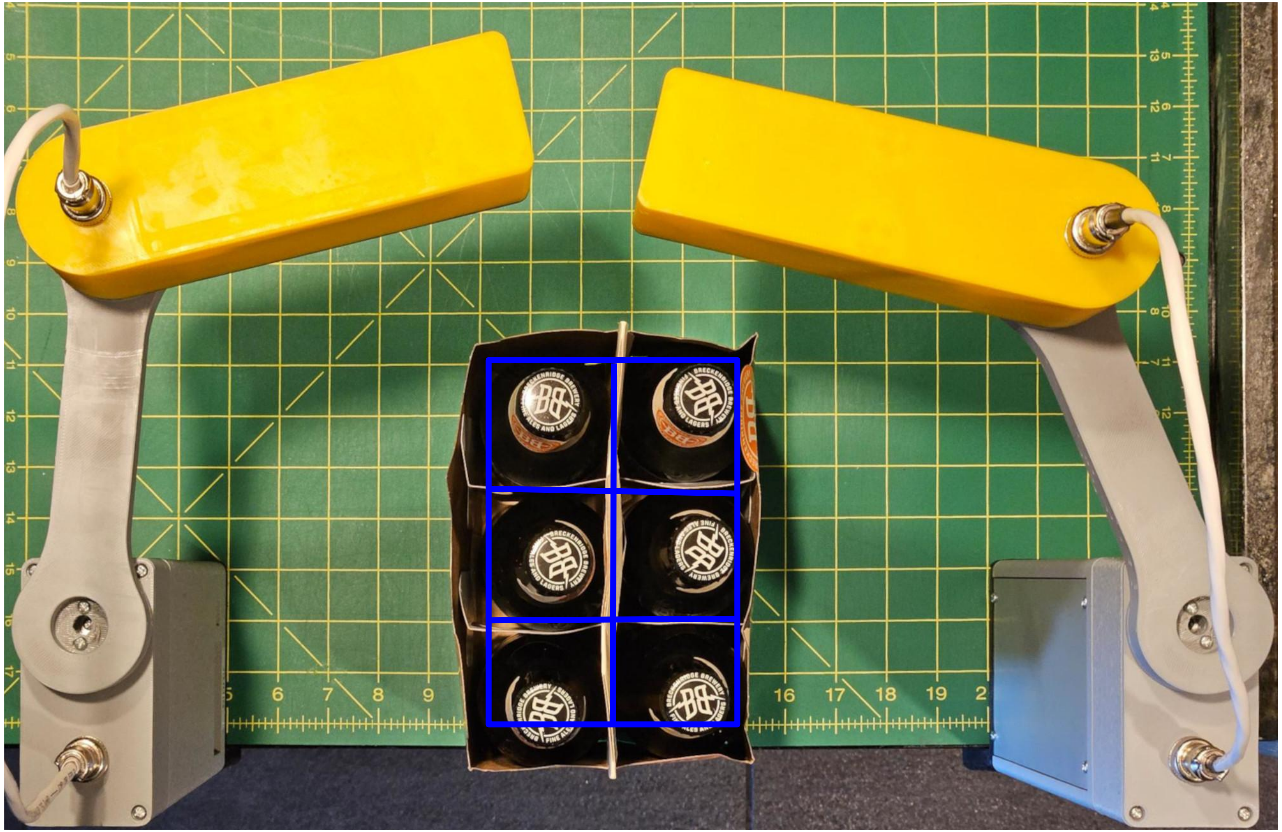
LIVE DEMO TIME!

synchronous single arm control

that was cool and worked perfectly!

maybe we could add another robot arm so we don’t have to wait as long?

synchronous dual arm control



github.com/griswaldbrooks/spanny2

synchronous dual arm control

in fact, the same accessor works in this case for the new arbiter!

```
auto arbiter = arbiter_dual{left_arm, std::make_unique<hardware>("/dev/ttyACM0"),
                            right_arm, std::make_unique<hardware>("/dev/ttyACM1")};

auto bin_checker = bin_checker_t{&arbiter};
auto bins = bin_view_t(bin_positions.data(), {}, bin_checker);

for (auto i = 0u; i != bins.extent(0); ++i) {
  for (auto j = 0u; j != bins.extent(1); ++j) {
    std::cout << bins(i, j) << "\n";
  }
}
```

synchronous dual arm control

LIVE DEMO TIME AGAIN!

synchronous dual arm control

…that didn't go so well

each call to access elements of the `std::mdspan` are blocking

the arms need to operate asynchronously

asynchronous dual arm control

`accessor::reference` can be any type
doesn't have to be a **&** or *****

```
template <typename Arbiter>
struct bin_checker_t {
  using element_type = std::expected<bin_state, std::string>;
  using reference = element_type;
  using data_handle_type = position_t*;
  using size_type = std::size_t;

  explicit bin_checker_async_t(Arbiter* checker) : checker_{std::move(checker)} {}

  reference access(data_handle_type bin_positions, size_type offset) const {
    auto const goal = bin_positions[offset];
    return (*checker_)(goal);
  }

 private:
  Arbiter* checker_;
};
```

asynchronous dual arm control

`accessor::reference` can be any type
doesn't have to be a `&` or `*`

```
template <typename Arbiter>
struct bin_checker_async_t {
  using element_type = std::expected<bin_state, std::string>;
  using reference = std::future<element_type>;
  using data_handle_type = position_t*;
  using size_type = std::size_t;

  explicit bin_checker_async_t(Arbiter* checker) : checker_{std::move(checker)} {}

  reference access(data_handle_type bin_positions, size_type offset) const {
    auto const goal = bin_positions[offset];
    return (*checker_)(goal);
  }

 private:
  Arbiter* checker_;
};
```

asynchronous dual arm control

```
struct arbiter_dual_async {
  using result_type = std::expected<bin_state, std::string>;
  using task_type = std::packaged_task</*is_bin_occupied function signature*/>;

std::future<result_type> operator()(position_t goal) {
    task_type task(is_bin_occupied);
    std::future<result_type> result = task.get_future();
    queue_.push(std::make_pair(std::move(goal), std::move(task)));
    return result;
  }


 private:
  thread_safe_queue<std::pair<position_t, task_type>> queue_;
  std::jthread left_engine_;
  std::jthread right_engine_;
};
```

asynchronous dual arm control

```
left_engine_ = std::jthread{[this](std::stop_token stoken) {
      try {
        while (!stoken.stop_requested()) {
          // get the list of available goals
          // check if any of the goals on the queue are available
          // if the goal is allowed, do it
          task(goal, model_[0], left_state_, *hw_.at(model_[0].description).get());
          // much more management code...
        } catch (std::exception const& e) {
        std::cerr << e.what() << std::endl;
      }

    }};
```

asynchronous dual arm control

```
auto arbiter = arbiter_dual_async{left_arm, std::make_unique<hw>("/dev/ttyACM0"),
                                  right_arm,std::make_unique<hw>("/dev/ttyACM1"),
                                  bin_grid};
auto bin_checker = bin_checker_async_t{&arbiter};
auto bins = bin_view_async_t(bin_positions.data(), {}, bin_checker);


std::vector<std::future</*bin element type*/> futures;
for (auto i = 0u; i != bins.extent(0); ++i) {
  for (auto j = 0u; j != bins.extent(1); ++j) {
    futures.push_back(bins(i, j));

while (!futures.empty())
  std::erase_if(futures, [](auto& future) {
    if (is_ready(future)) {
      std::cout << future.get() << "\n";
      return true;
    }
    return false;
  });
```

asynchronous dual arm control

```
auto arbiter = arbiter_dual_async{left_arm, std::make_unique<hw>("/dev/ttyACM0"),
                                  right_arm,std::make_unique<hw>("/dev/ttyACM1"),
                                  bin_grid};
auto bin_checker = bin_checker_async_t{&arbiter};
auto bins = bin_view_async_t(bin_positions.data(), {}, bin_checker);


std::vector<std::future</*bin element type*/> futures;
for (auto i = 0u; i != bins.extent(0); ++i) {
  for (auto j = 0u; j != bins.extent(1); ++j) {
    futures.push_back(bins(i, j));

while (!futures.empty())
  std::erase_if(futures, [](auto& future) {
    if (is_ready(future)) {
      std::cout << future.get() << "\n";
      return true;
    }
    return false;
  });
```

asynchronous dual arm control

```
auto arbiter = arbiter_dual_async{left_arm, std::make_unique<hw>("/dev/ttyACM0"),
                                  right_arm,std::make_unique<hw>("/dev/ttyACM1"),
                                  bin_grid};
auto bin_checker = bin_checker_async_t{&arbiter};
auto bins = bin_view_async_t(bin_positions.data(), {}, bin_checker);


std::vector<std::future</*bin element type*/> futures;
for (auto i = 0u; i != bins.extent(0); ++i) {
  for (auto j = 0u; j != bins.extent(1); ++j) {
    futures.push_back(bins(i, j));

while (!futures.empty())
  std::erase_if(futures, [](auto& future) {
    if (is_ready(future)) {
      std::cout << future.get() << "\n";
      return true;
    }
    return false;
  });
```

asynchronous dual arm control

```
auto arbiter = arbiter_dual_async{left_arm, std::make_unique<hw>("/dev/ttyACM0"),
                                  right_arm,std::make_unique<hw>("/dev/ttyACM1"),
                                  bin_grid};
auto bin_checker = bin_checker_async_t{&arbiter};
auto bins = bin_view_async_t(bin_positions.data(), {}, bin_checker);


std::vector<std::future</*bin element type*/> futures;
for (auto i = 0u; i != bins.extent(0); ++i) {
  for (auto j = 0u; j != bins.extent(1); ++j) {
    futures.push_back(bins(i, j));

while (!futures.empty())
  std::erase_if(futures, [](auto& future) {
    if (is_ready(future)) {
      std::cout << future.get() << "\n";
      return true;
    }
    return false;
  });
```

asynchronous dual arm control

WILL IT WORK THIS TIME?😬😳

let's wrap up 🍻

sum up

discussed what mdspan is and its motivations

customization with layout and accessor policies

built-in options: `layout_right`, `layout_left`, `layout_stride`, `submdspan`, `default_accessor`

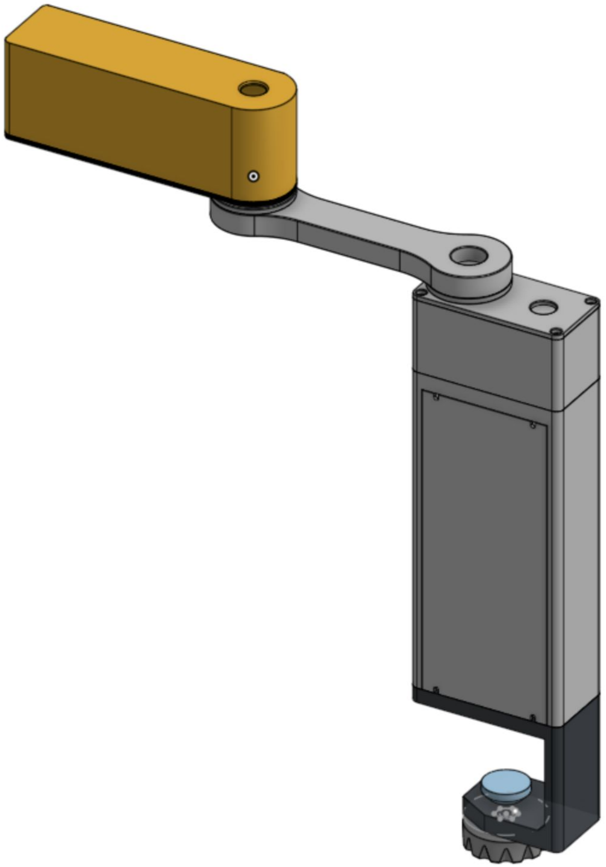
implementing custom policies with state and non-default constructors

layouts and accessors leveraging full flexibility of C++

open source

github.com/griswaldbrooks/spanny2

3d models on printables



QUESTIONS?

spanny will return...