Viewing the world through array-shaped glasses

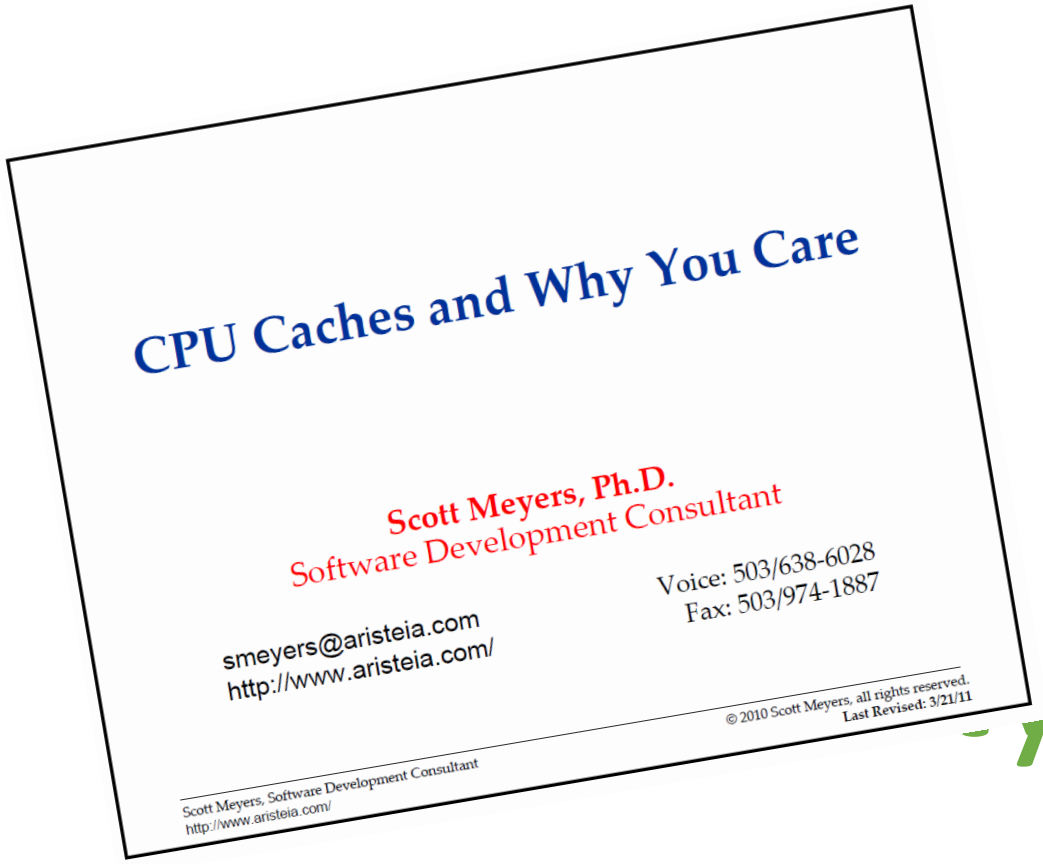
Łukasz Mendakiewicz

Software Engineer, Microsoft

CppCon 2014

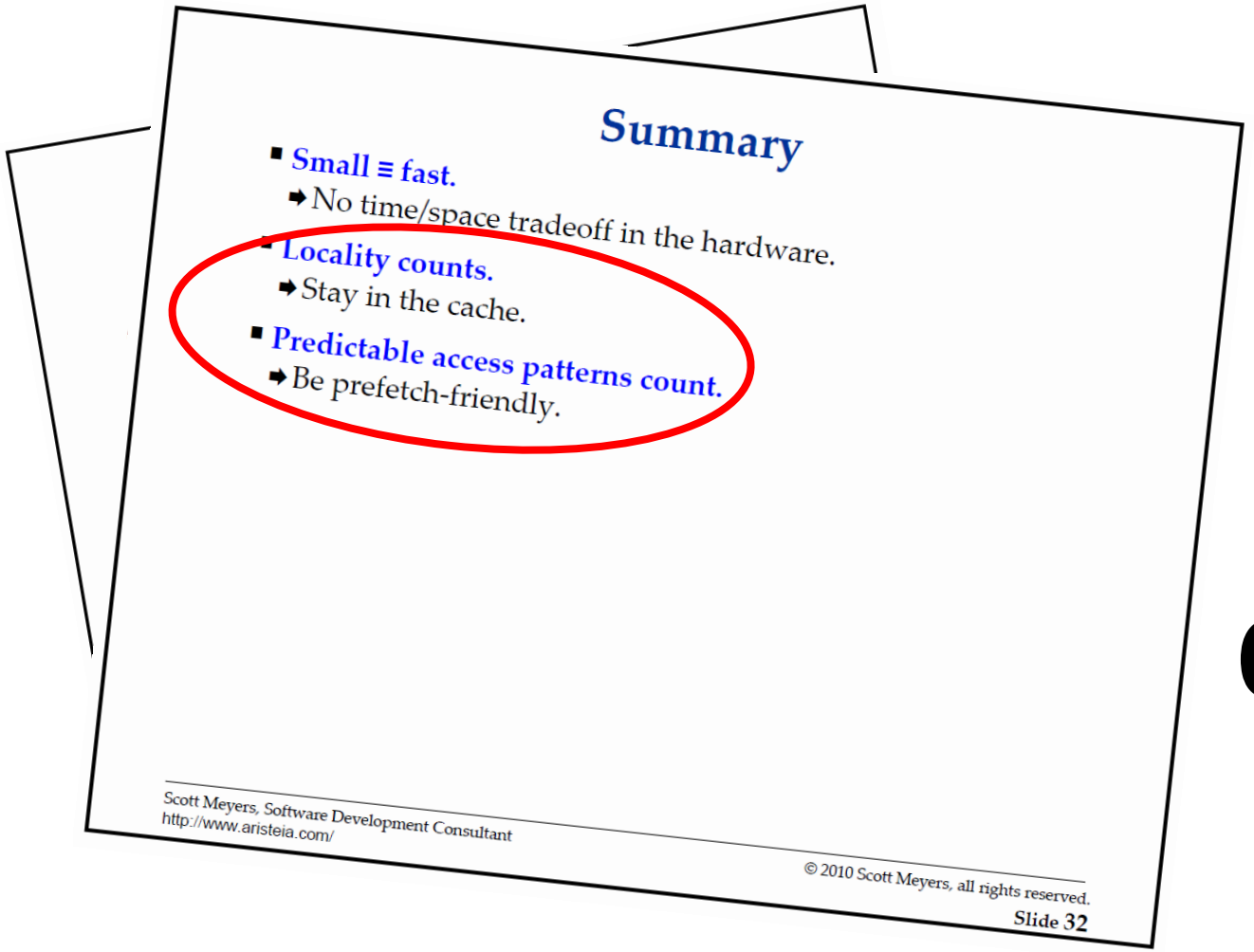
9/8/2014

Contiguity of data matters



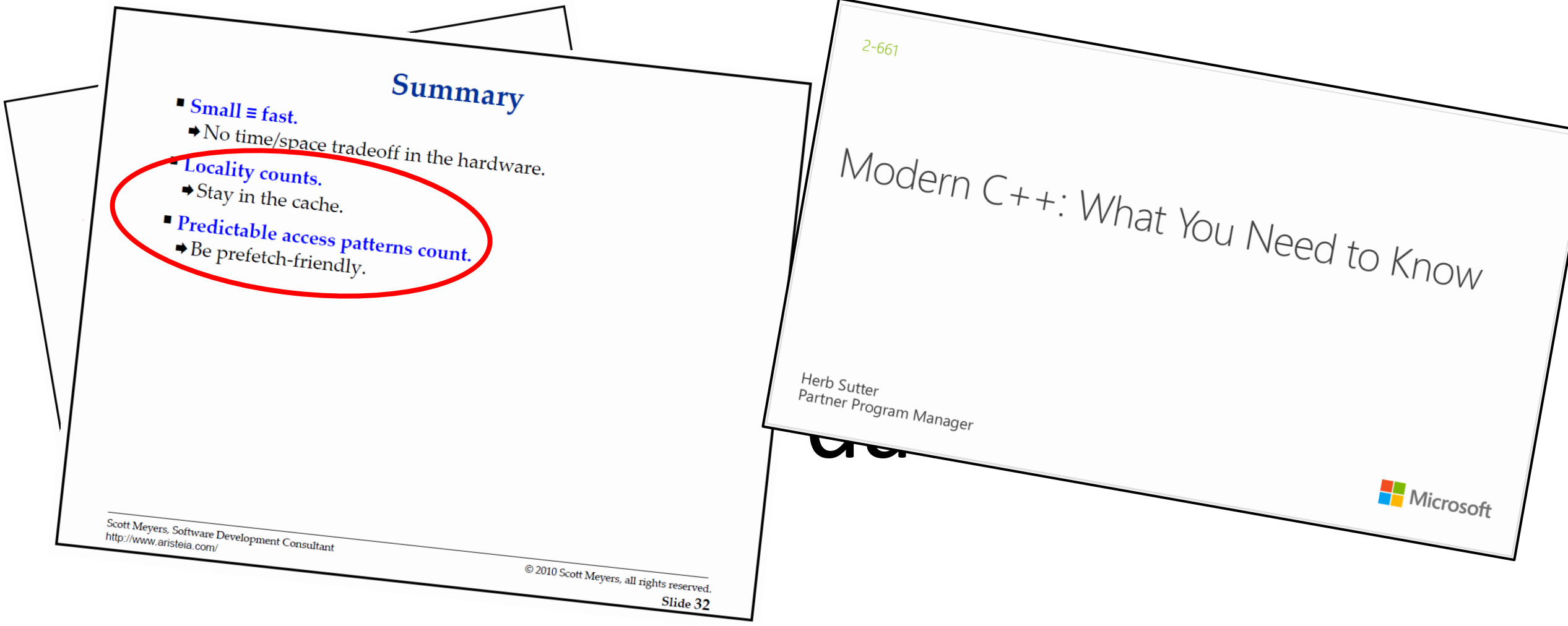
y of data matters

http://www.aristeia.com/TalkNotes/ACCU2011_CPUCaches.pdf



data matters

http://www.aristeia.com/TalkNotes/ACCU2011_CPUCaches.pdf



http://www.aristeia.com/TalkNotes/ACCU2011_CPUCaches.pdf

http://channel9.msdn.com/Events/Build/2014/2-661

Summary

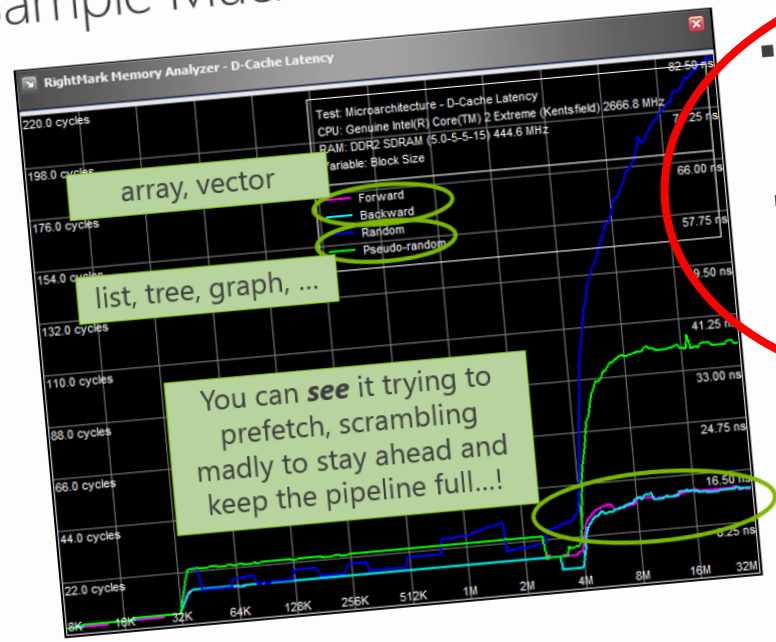
- **Small ≡ fast.**
  - ➡ No time/space tradeoff in the hardware.
- **Locality counts.**
  - ➡ Stay in the cache.
- **Predictable access patterns count.**
  - ➡ Be prefetch-friendly.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2010 Scott Meye

2-661

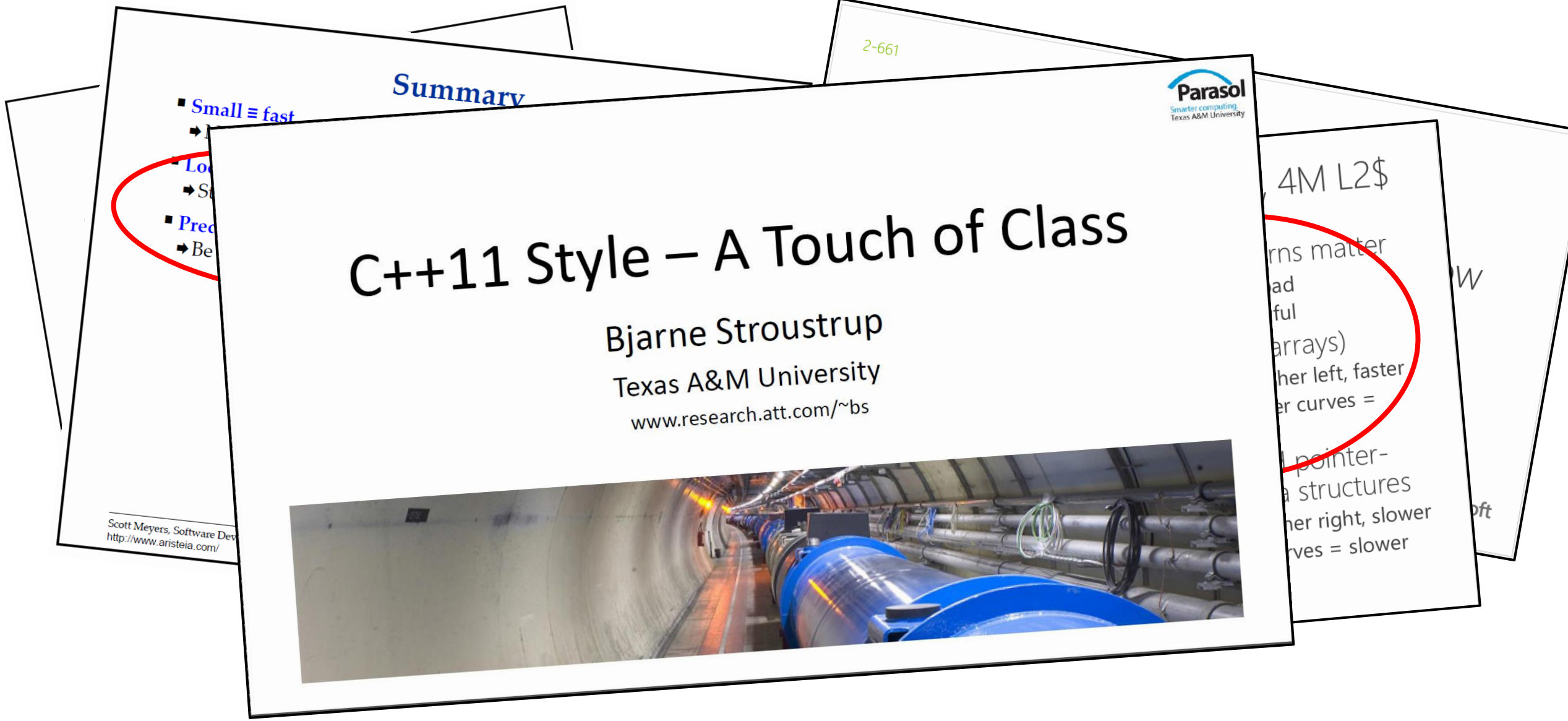
Sample Machine: Samsung tablet, 32K L1D$, 4M L2$



- Access patterns matter
  - Linear = not bad
  - Random = awful
- Arrays (real arrays)
  - Smaller = further left, faster
  - Often on lower curves = faster
- Node-based pointer-chasing data structures
  - Bigger = further right, slower
  - On higher curves = slower

http://www.aristeia.com/TalkNotes/ACCU2011_CPUCaches.pdf

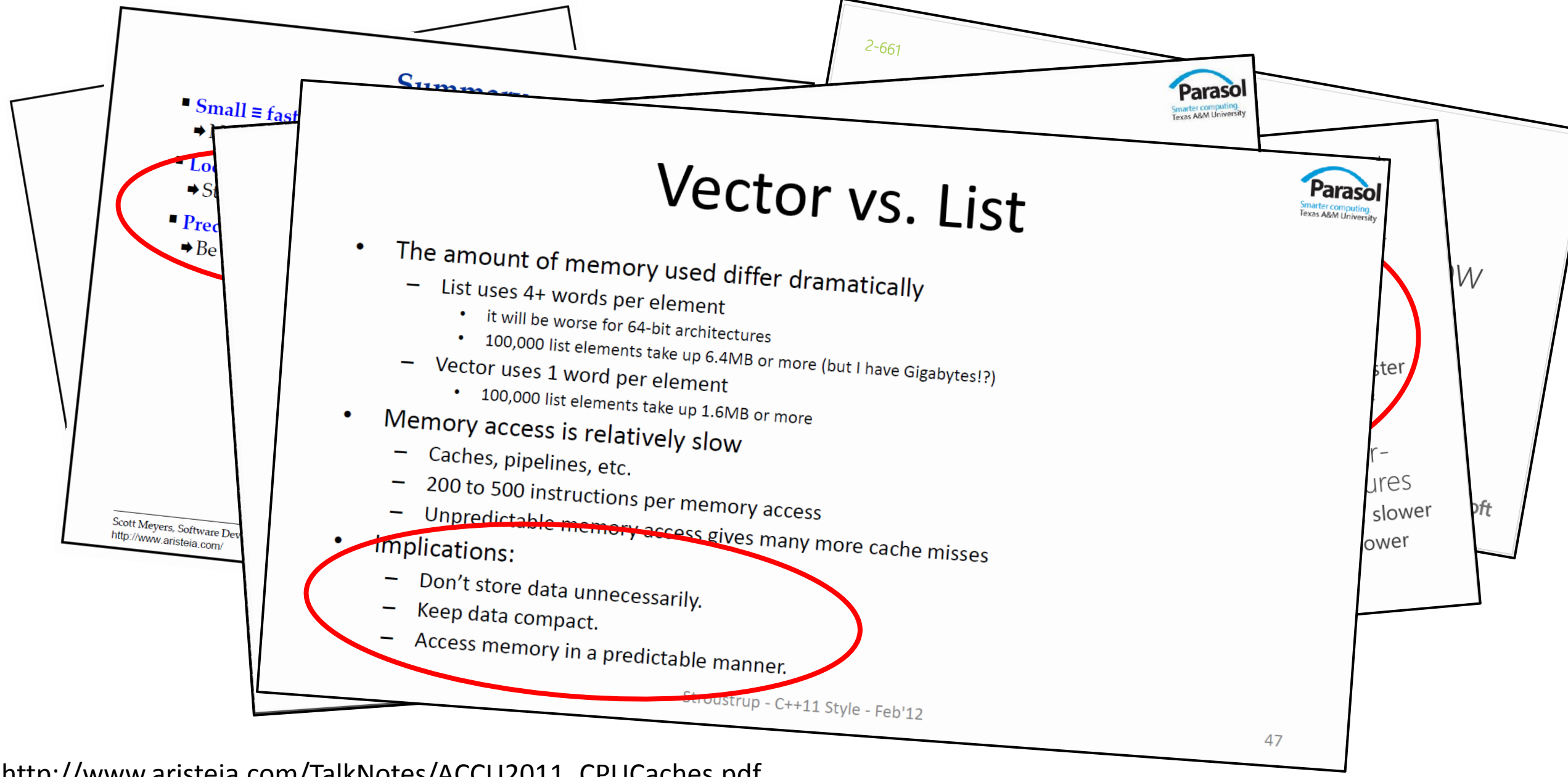
http://channel9.msdn.com/Events/Build/2014/2-661



http://www.aristeia.com/TalkNotes/ACCU2011_CPUCaches.pdf

http://channel9.msdn.com/Events/Build/2014/2-661

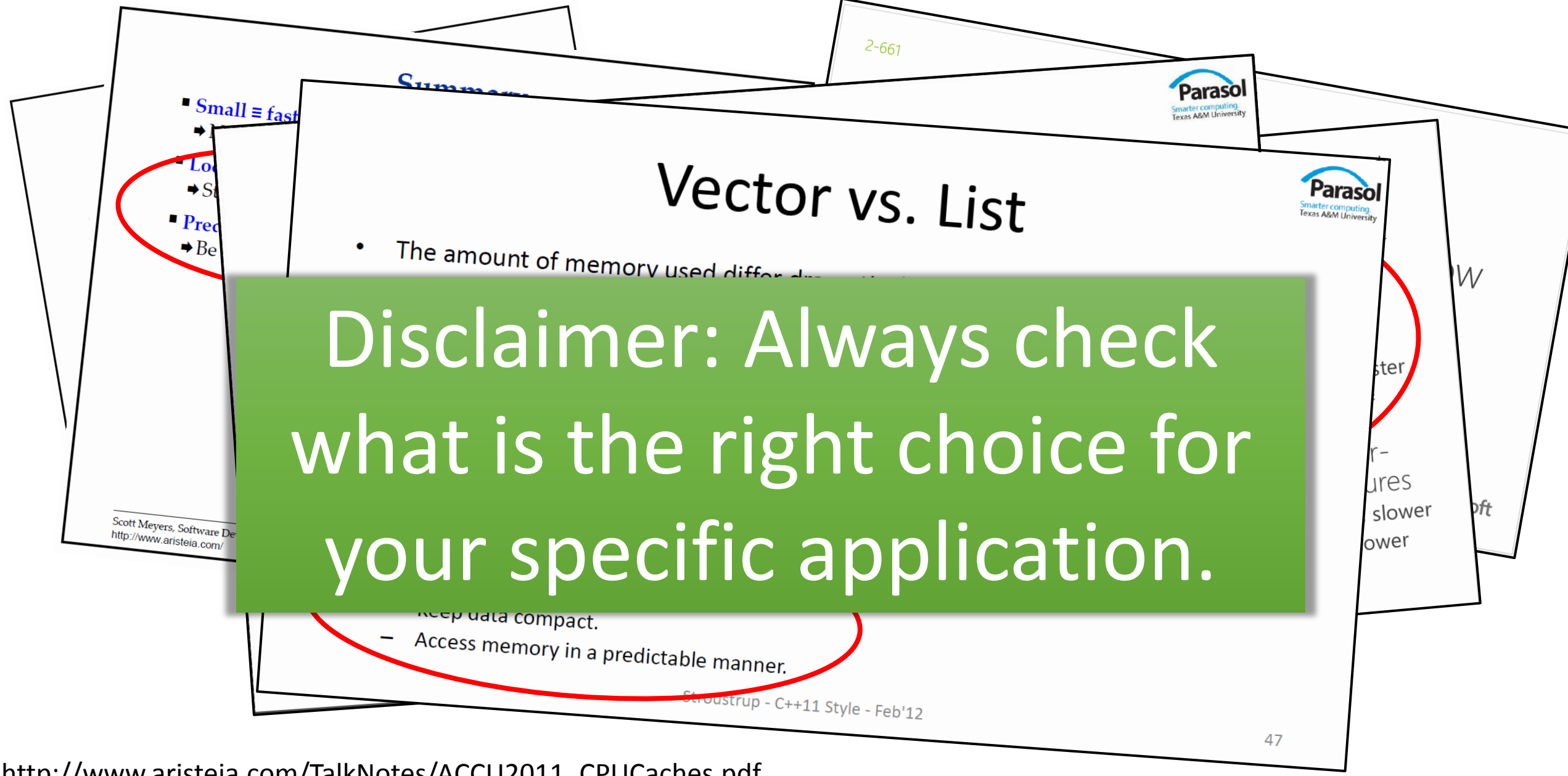
http://channel9.msdn.com/Events/GoingNative/GoingNative-2012/Keynote-Bjarne-Stroustrup-Cpp11-Style



http://www.aristeia.com/TalkNotes/ACCU2011_CPUCaches.pdf

http://channel9.msdn.com/Events/Build/2014/2-661

http://channel9.msdn.com/Events/GoingNative/GoingNative-2012/Keynote-Bjarne-Stroustrup-Cpp11-Style



http://www.aristeia.com/TalkNotes/ACCU2011_CPUCaches.pdf

http://channel9.msdn.com/Events/Build/2014/2-661

http://channel9.msdn.com/Events/GoingNative/GoingNative-2012/Keynote-Bjarne-Stroustrup-Cpp11-Style

Defining contiguous data in C++

Defining contiguous data in C++

```
int old_skool_array[1000];
```

Defining contiguous data in C++

```
int old_skool_array[1000];

std::array<int, 1000> std_array;
```

Defining contiguous data in C++

```
int old_skool_array[1000];

std::array<int, 1000> std_array;

std::vector<int> vec(1000);
```

Defining contiguous data in C++

```
int old_skool_array[1000];

std::array<int, 1000> std_array;

std::vector<int> vec(1000);

std::deque<int> deq(1000);
```

Defining contiguous data in C++

```
int old_skool_array[1000];

std::array<int, 1000> std_array;

std::vector<int> vec(1000);

~~std::deque<int> deq(1000);~~
```

Defining contiguous data in C++

```
int old_skool_array[1000];

std::array<int, 1000> std_array;

std::vector<int> vec(1000);

~~std::deque<int> deq(1000);~~

std::dynarray<int> dyn(1000); // Array Extensions TS
```

Defining contiguous data in C++

```
int old_skool_array[1000];

std::array<int, 1000> std_array;

std::vector<int> vec(1000);

~~std::deque<int> deq(1000);~~

std::dynarray<int> dyn(1000); // Array Extensions TS
```

Other libraries (BLAS? Bitmaps?)

Defining contiguous data in C++

```
int old_skool_array[1000];

std::array<int, 1000> std_array;

std::vector<int> vec(1000);

~~std::deque<int> deq(1000);~~

std::dynarray<int> dyn(1000); // Array Extensions TS
```

Other libraries (BLAS? Bitmaps?)

Homegrown types

Accepting contiguous data in C++

```
template <typename CollectionType>
int generic(CollectionType& collection) { ... }
```

Accepting contiguous data in C++

```
template <typename CollectionType>
int generic(CollectionType& collection) { ... }
```

Too generic?

Accepting contiguous data in C++

```
template <typename T>
concept bool Collection = ...;

template <Collection CollectionType>
int generic(CollectionType& collection) { ... }
```

Accepting contiguous data in C++

```
template <typename T>
concept bool Collection = ...;
```

Not there yet…

```
template <Collection CollectionType>
int generic(CollectionType& collection) { ... }
```

Accepting contiguous data in C++

```
template <typename Collection,
          typename = std::enable_if_t<std::is_convertible_v<size_t,
                                      decltype(std::declval<Collection>().size())>>
                     && ...
          >
int generic(Collection& data) { ... }
```

Accepting contiguous data in C++

```
template <typename Collection,
          typename = std::enable_if_t<std::is_convertible_v<size_t,
                                      decltype(std::declval<Collection>().size())>>
                                    && ...
          >
int generic(Collection& data) { ... }


generic(vec);
```

Accepting contiguous data in C++

```
template <typename Collection,
          typename = std::enable_if_t<std::is_convertible_v<size_t,
                                      decltype(std::declval<Collection>().size())>>
                                    && ...
          >
int generic(Collection& data) { ... }


generic(vec);
```

Accepting contiguous data in C++

```
template <typename Collection,
          typename = std::enable_if_t<std::is_convertible_v<size_t,
                                      decltype(std::declval<Collection>().size())>>
                     && ...
          >
int generic(Collection& data) { ... }


generic(vec);   generic(std_array);
```

Accepting contiguous data in C++

```
template <typename Collection,
          typename = std::enable_if_t<std::is_convertible_v<size_t,
                                      decltype(std::declval<Collection>().size())>>
                                   && ...
          >
int generic(Collection& data) { ... }


generic(vec);   generic(std_array);
```

Accepting contiguous data in C++

```
template <typename Collection,
          typename = std::enable_if_t<std::is_convertible_v<size_t,
                                      decltype(std::declval<Collection>().size())>>
                                   && ...
          >
int generic(Collection& data) { ... }
```

```
generic(vec);   generic(std_array);   generic(old_skool_array);
```

Accepting contiguous data in C++

```
template <typename Collection,
          typename = std::enable_if_t<std::is_convertible_v<size_t,
                                      decltype(std::declval<Collection>().size())>>
                                   && ...
          >
int generic(Collection& data) { ... }


generic(vec);   generic(std_array);   generic(old_skool_array);
```



Accepting contiguous data in C

```
int raw(int* ptr, size_t size) { ... }
```

Accepting contiguous data in C

```
int raw(int* ptr, size_t size) { ... }

raw(vec.data(), vec.size());
```

Accepting contiguous data in C

```
int raw(int* ptr, size_t size) { ... }
```

```
raw(vec.data(), vec.size());
```


Accepting contiguous data in C

```
int raw(int* ptr, size_t size) { ... }
```

```
raw(vec.data(), vec.size()); ✓
raw(old_skool_array, NUM_ELEMENTS(old_skool_array));
```

Accepting contiguous data in C

```
int raw(int* ptr, size_t size) { ... }
```

```
raw(vec.data(), vec.size()); ✓
raw(old_skool_array, NUM_ELEMENTS(old_skool_array)); ✓
```

Accepting contiguous data in C

```
int raw(int* ptr, size_t size) { ... }
```

```
raw(vec.data(), vec.size()); ✓
raw(old_skool_array, NUM_ELEMENTS(old_skool_array)); ✓
```

```
int raw(int* ptr, size_t width, size_t height) { ... }
```

Accepting contiguous data in C

```
int raw(int* ptr, size_t size) { ... }
```

```
raw(vec.data(), vec.size());
```
✓

```
raw(old_skool_array, NUM_ELEMENTS(old_skool_array));
```
✓

```
int raw(int* ptr, size_t width, size_t height) { ... }
```

```
int raw(int* ptr, size_t width, size_t height, size_t depth) {...}
```

The alternative: `array_view` interface

```
int compute(array_view<int> data) { ... }
```

The alternative: `array_view` interface

```
int compute(array_view<int> data) { ... }

auto data = std::vector<int>(1000);
```

The alternative: `array_view` interface

```
int compute(array_view<int> data) { ... }


auto data = std::vector<int>(1000);

auto av = array_view<int>{data};
```

The alternative: `array_view` interface

```
int compute(array_view<int> data) { ... }


auto data = std::vector<int>(1000);

auto av = array_view<int>{data};
compute(av);
```

The alternative: `array_view` interface

```
int compute(array_view<int> data) { ... }


auto data = std::vector<int>(1000);

auto av = array_view<int>{data};
compute(av);

compute(data);
```

The alternative: `array_view` interface

```
int compute(array_view<int> data) { ... }
```

```
int data[1000];

auto av = array_view<int>{data};
compute(av);

compute(data);
```

The alternative: `array_view` interface

```
int compute(array_view<int> data) { ... }
```

```
auto data = std::array<int, 1000>{};
          int data[1000];
auto av = array_view<int>{data};
compute(av);

compute(data);
```

This `_view` thing sounds familiar…

	Where x is:

`string_view{x};`	`std::string`
	`char*`
	…

`array_view<T>{x};`	`std::vector<T>`
	`T*`
	…

But there is more (dimensions)!

```
int compute(int* ptr, size_t size) { ... }

int compute(int* ptr, size_t width, size_t height) { ... }
```

But there is more (dimensions)!

```
int compute(array_view<int> data) { ... }

int compute(int* ptr, size_t width, size_t height) { ... }
```

But there is more (dimensions)!

```
int compute(array_view<int> data) { ... }

int compute(array_view<int, 2> data) { ... }
```

But there is more (dimensions)!

```
int compute(array_view<int> data) { ... }

int compute(array_view<int, 2> data) { ... }

int compute(array_view<int, 3> data) { ... }
```

But there is more (dimensions)!

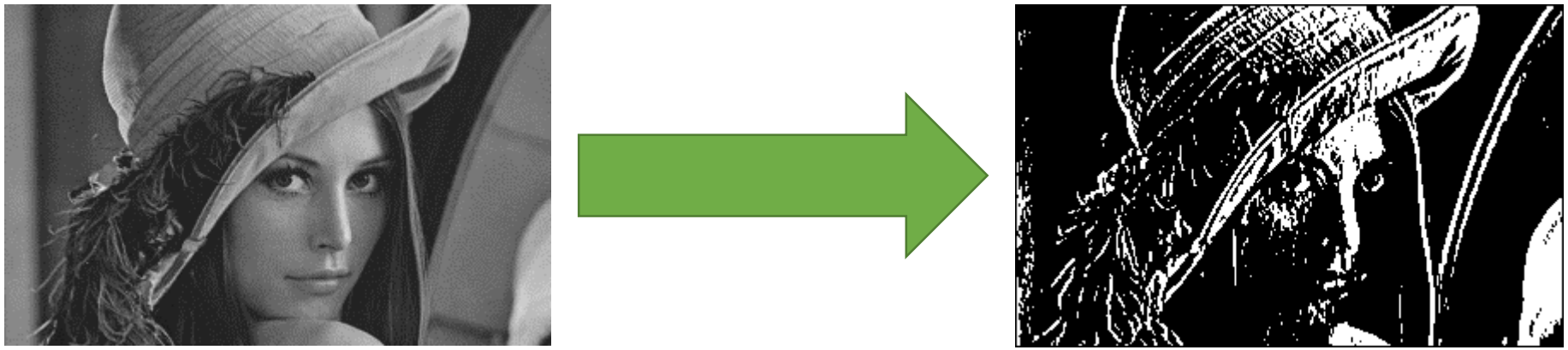
```
int compute(array_view<int> data) { ... }

int compute(array_view<int, 2> data) { ... }

int compute(array_view<int, 3> data) { ... }


auto data = std::vector<int>(1000);

auto av_1 = array_view<int>{data};
auto av_2 = array_view<int, 2>{{20, 50}, data};

compute(av_1);
compute(av_2);
```

But there is more (dimensions)!

```
int compute(array_view<int> data) { ... }

int compute(array_view<int, 2> data) { ... }

int compute(array_view<int, 3> data) { ... }


auto data = std::vector<int>(1000);

auto av_1 = array_view<int>{data};
auto av_2 = array_view<int, 2>{{20, 50}, data};

compute(av_1);
compute(av_2);
```

Lifting the linear memory into a **logically** multidimensional representation.

Using the `array_view`



in := $[uint8\_t]_{height,width}$

out := $[uint8\_t]_{height,width}$

$$out := \left(\begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * in \right) \lfloor 150$$

Using the `array_view`

```
void compute(uint8_t* in, uint8_t* out,
             size_t width, size_t height) {


  for (size_t row = 0; row < height; ++row)
  for (size_t col = 0; col < width; ++col) {
    if (row == 0 || row == height - 1
     || col == 0 || col == width - 1) {
      out[row * width + col] = 0;
    } else {
      int v =
       - 1 * in[(row - 1) * width + col - 1]
       + 1 * in[(row - 1) * width + col + 1]
       - 2 * in[ row      * width + col - 1]
       + 2 * in[ row      * width + col + 1]
       - 1 * in[(row + 1) * width + col - 1]
       + 2 * in[(row + 1) * width + col + 1];
      out[row * width + col] =
       v > 150 ? 255 : 0;
} } }
```

Using the `array_view`

```
void compute(uint8_t* in, uint8_t* out,
             size_t width, size_t height) {


  for (size_t row = 0; row < height; ++row)
  for (size_t col = 0; col < width; ++col) {
    if (row == 0 || row == height - 1
     || col == 0 || col == width - 1) {
      out[row * width + col] = 0;
    } else {
      int v =
       - 1 * in[(row - 1) * width + col - 1]
       + 1 * in[(row - 1) * width + col + 1]
       - 2 * in[ row      * width + col - 1]
       + 2 * in[ row      * width + col + 1]
       - 1 * in[(row + 1) * width + col - 1]
       + 2 * in[(row + 1) * width + col + 1];
      out[row * width + col] =
       v > 150 ? 255 : 0;
} } }
```

Using the `array_view`

```
void compute(uint8_t* in, uint8_t* out,
             size_t width, size_t height) {


  for (size_t row = 0; row < height; ++row)
  for (size_t col = 0; col < width; ++col) {
    if (row == 0 || row == height - 1
     || col == 0 || col == width - 1) {
      out[row * width + col] = 0;
    } else {
      int v =
       - 1 * in[(row - 1) * width + col - 1]
       + 1 * in[(row - 1) * width + col + 1]
       - 2 * in[ row      * width + col - 1]
       + 2 * in[ row      * width + col + 1]
       - 1 * in[(row + 1) * width + col - 1]
       + 2 * in[(row + 1) * width + col + 1];
      out[row * width + col] =
       v > 150 ? 255 : 0;
} } }
```

Nested *for* loops to iterate over a single concept (a 2D space).

Using the `array_view`

```
void compute(uint8_t* in, uint8_t* out,
             size_t width, size_t height) {


  for (size_t row = 0; row < height; ++row)
  for (size_t col = 0; col < width; ++col) {
    if (row == 0 || row == height - 1
     || col == 0 || col == width - 1) {
      out[row * width + col] = 0;
    } else {
      int v =
       - 1 * in[(row - 1) * width + col - 1]
       + 1 * in[(row - 1) * width + col + 1]
       - 2 * in[ row      * width + col - 1]
       + 2 * in[ row      * width + col + 1]
       - 1 * in[(row + 1) * width + col - 1]
       + 2 * in[(row + 1) * width + col + 1];
      out[row * width + col] =
       v > 150 ? 255 : 0;
} } }
```

Nested *for* loops to iterate over a single concept (a 2D space).

Using the `array_view`

```
void compute(uint8_t* in, uint8_t* out,
             size_t width, size_t height) {


  for (size_t row = 0; row < height; ++row)
  for (size_t col = 0; col < width; ++col) {
    if (row == 0 || row == height - 1
     || col == 0 || col == width - 1) {
      out[row * width + col] = 0;
    } else {
      int v =
       - 1 * in[(row - 1) * width + col - 1]
       + 1 * in[(row - 1) * width + col + 1]
       - 2 * in[ row      * width + col - 1]
       + 2 * in[ row      * width + col + 1]
       - 1 * in[(row + 1) * width + col - 1]
       + 2 * in[(row + 1) * width + col + 1];
      out[row * width + col] =
       v > 150 ? 255 : 0;
} } }
```

Nested *for* loops to iterate over a single concept (a 2D space).

Error-prone index calculation.

Using the `array_view`

```
void compute(uint8_t* in, uint8_t* out,
             size_t width, size_t height) {


  for (size_t row = 0; row < height; ++row)
  for (size_t col = 0; col < width; ++col) {
    if (row == 0 || row == height - 1
     || col == 0 || col == width - 1) {
      out[row * width + col] = 0;
    } else {
      int v =
       - 1 * in[(row - 1) * width + col - 1]
       + 1 * in[(row - 1) * width + col + 1]
       - 2 * in[ row      * width + col - 1]
       + 2 * in[ row      * width + col + 1]
       - 1 * in[(row + 1) * width + col - 1]
       + 2 * in[(row + 1) * width + col + 1];
      out[row * width + col] =
       v > 150 ? 255 : 0;
} } }
```

```
void compute(array_view<const uint8_t, 2> in,
             array_view<uint8_t, 2> out) {
  auto N = index<2>{-1, 0}; auto S = index<2>{1,0};
  auto W = index<2>{ 0,-1}; auto E = index<2>{0,1};

  for (index<2> idx : in.bounds()) {
    if (idx[0] == 0 || idx[0] == in.bounds()[0] - 1
     || idx[1] == 0 || idx[1] == in.bounds()[1] - 1) {
      out[idx] = 0;
    } else {
      int v =
       - 1 * in[idx + W + N]
       + 1 * in[idx + E + N]
       - 2 * in[idx + W    ]
       + 2 * in[idx + E    ]
       - 1 * in[idx + W + S]
       + 2 * in[idx + S + E];
      out[idx] =
       v > 150 ? 255 : 0;
} } }
```

Using the `array_view`

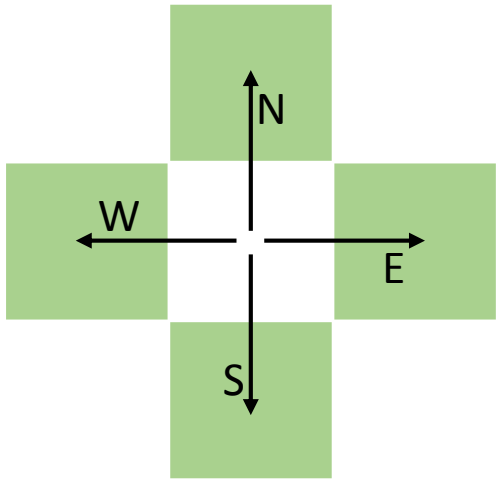
```
void compute(uint8_t* in, uint8_t* out,
             size_t width, size_t height) {


  for (size_t row = 0; row < height; ++row)
  for (size_t col = 0; col < width; ++col) {
    if (row == 0 || row == height - 1
     || col == 0 || col == width - 1) {
      out[row * width + col] = 0;
    } else {
      int v =
       - 1 * in[(row - 1) * width + col - 1]
       + 1 * in[(row - 1) * width + col + 1]
       - 2 * in[ row      * width + col - 1]
       + 2 * in[ row      * width + col + 1]
       - 1 * in[(row + 1) * width + col - 1]
       + 2 * in[(row + 1) * width + col + 1];
      out[row * width + col] =
       v > 150 ? 255 : 0;
} } }
```

```
void compute(array_view<const uint8_t, 2> in,
             array_view<uint8_t, 2> out) {
  auto N = index<2>{-1, 0}; auto S = index<2>{1,0};
  auto W = index<2>{ 0,-1}; auto E = index<2>{0,1};

  for (index<2> idx : in.bounds()) {
    if (idx[0] == 0 || idx[0] == in.bounds()[0] - 1
     || idx[1] == 0 || idx[1] == in.bounds()[1] - 1) {
      out[idx] = 0;
    } else {
      int v =
       - 1 * in[idx + W + N]
       + 1 * in[idx + E + N]
       - 2 * in[idx + W    ]
       + 2 * in[idx + E    ]
       - 1 * in[idx + W + S]
       + 2 * in[idx + S + E];
      out[idx] =
       v > 150 ? 255 : 0;
} } }
```

Using the `array_view`

```
void compute(uint8_t* in, uint8_t* out,
             size_t width, size_t height) {


  for (size_t row = 0; row < height; ++row)
  for (size_t col = 0; col < width; ++col) {
    if (row == 0 || row == height - 1
     || col == 0 || col == width - 1) {
      out[row * width + col] = 0;
    } else {
      int v =
       - 1 * in[(row - 1) * width + col - 1]
       + 1 * in[(row - 1) * width + col + 1]
       - 2 * in[ row      * width + col - 1]
       + 2 * in[ row      * width + col + 1]
       - 1 * in[(row + 1) * width + col - 1]
       + 2 * in[(row + 1) * width + col + 1];
      out[row * width + col] =
       v > 150 ? 255 : 0;
} } }
```

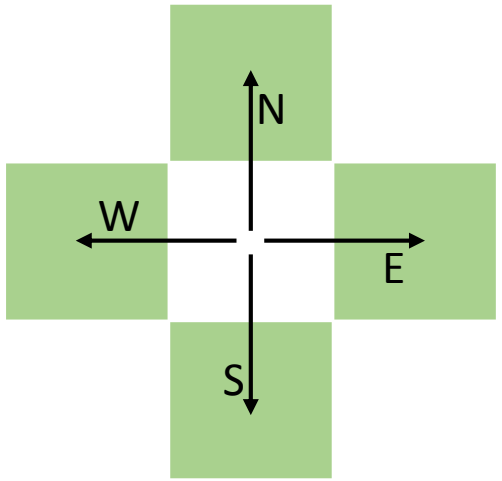
```
void compute(array_view<const uint8_t, 2> in,
             array_view<uint8_t, 2> out) {
  auto N = index<2>{-1, 0}; auto S = index<2>{1,0};
  auto W = index<2>{ 0,-1}; auto E = index<2>{0,1};

  for (index<2> idx : in.bounds()) {
    if (idx[0] == 0 || idx[0] == in.bounds()[0] - 1
     || idx[1] == 0 || idx[1] == in.bounds()[1] - 1) {
      out[idx] = 0;
    } else {
      int v =
       - 1 * in[idx + W + N]
       + 1 * in[idx + E + N]
       - 2 * in[idx + W    ]
       + 2 * in[idx + E    ]
       - 1 * in[idx + W + S]
       + 2 * in[idx + S + E];
      out[idx] =
       v > 150 ? 255 : 0;
} } }
```



Using the `array_view`

```
void compute(uint8_t* in, uint8_t* out,
             size_t width, size_t height) {


  for (size_t row = 0; row < height; ++row)
  for (size_t col = 0; col < width; ++col) {
    if (row == 0 || row == height - 1
     || col == 0 || col == width - 1) {
      out[row * width + col] = 0;
    } else {
      int v =
       - 1 * in[(row - 1) * width + col - 1]
       + 1 * in[(row - 1) * width + col + 1]
       - 2 * in[ row      * width + col - 1]
       + 2 * in[ row      * width + col + 1]
       - 1 * in[(row + 1) * width + col - 1]
       + 2 * in[(row + 1) * width + col + 1];
      out[row * width + col] =
       v > 150 ? 255 : 0;
} } }
```

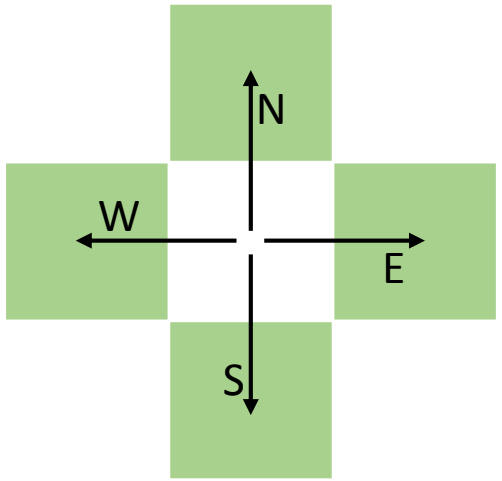
```
void compute(array_view<const uint8_t, 2> in,
             array_view<uint8_t, 2> out) {
  auto N = index<2>{-1, 0}; auto S = index<2>{1,0};
  auto W = index<2>{ 0,-1}; auto E = index<2>{0,1};

  for (index<2> idx : in.bounds()) {
    if (idx[0] == 0 || idx[0] == in.bounds()[0] - 1
     || idx[1] == 0 || idx[1] == in.bounds()[1] - 1) {
      out[idx] = 0;
    } else {
      int v =
       - 1 * in[idx + W + N]
       + 1 * in[idx + E + N]
       - 2 * in[idx + W    ]
       + 2 * in[idx + E    ]
       - 1 * in[idx + W + S]
       + 2 * in[idx + S + E];
      out[idx] =
       v > 150 ? 255 : 0;
} } }
```



Using the `array_view`

```
void compute(uint8_t* in, uint8_t* out,
             size_t width, size_t height) {


  for (size_t row = 0; row < height; ++row)
  for (size_t col = 0; col < width; ++col) {
    if (row == 0 || row == height - 1
     || col == 0 || col == width - 1) {
      out[row * width + col] = 0;
    } else {
      int v =
       - 1 * in[(row - 1) * width + col - 1]
       + 1 * in[(row - 1) * width + col + 1]
       - 2 * in[ row      * width + col - 1]
       + 2 * in[ row      * width + col + 1]
       - 1 * in[(row + 1) * width + col - 1]
       + 2 * in[(row + 1) * width + col + 1];
      out[row * width + col] =
       v > 150 ? 255 : 0;
} } }
```

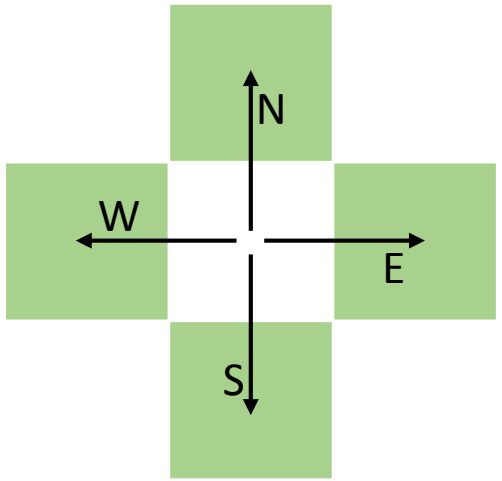
```
void compute(array_view<const uint8_t, 2> in,
             array_view<uint8_t, 2> out) {
  auto N = index<2>{-1, 0}; auto S = index<2>{1,0};
  auto W = index<2>{ 0,-1}; auto E = index<2>{0,1};

  for (index<2> idx : in.bounds()) {
    if (idx[0] == 0 || idx[0] == in.bounds()[0] - 1
     || idx[1] == 0 || idx[1] == in.bounds()[1] - 1) {
      out[idx] = 0;
    } else {
      int v =
       - 1 * in[idx + W + N]
       + 1 * in[idx + E + N]
       - 2 * in[idx + W    ]
       + 2 * in[idx + E    ]
       - 1 * in[idx + W + S]
       + 2 * in[idx + S + E];
      out[idx] =
       v > 150 ? 255 : 0;
} } }
```



Using the `array_view`

```
void compute(uint8_t* in, uint8_t* out,
             size_t width, size_t height) {


  for (size_t row = 0; row < height; ++row)
  for (size_t col = 0; col < width; ++col) {
    if (row == 0 || row == height - 1
     || col == 0 || col == width - 1) {
      out[row * width + col] = 0;
    } else {
      int v =
       - 1 * in[(row - 1) * width + col - 1]
       + 1 * in[(row - 1) * width + col + 1]
       - 2 * in[ row      * width + col - 1]
       + 2 * in[ row      * width + col + 1]
       - 1 * in[(row + 1) * width + col - 1]
       + 2 * in[(row + 1) * width + col + 1];
      out[row * width + col] =
       v > 150 ? 255 : 0;
} } }
```

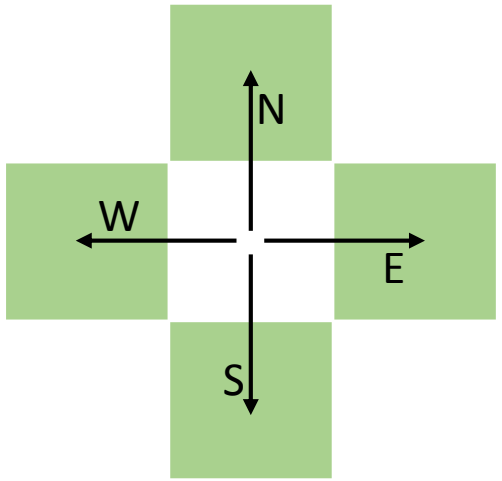
```
void compute(array_view<const uint8_t, 2> in,
             array_view<uint8_t, 2> out) {
  auto N = index<2>{-1, 0}; auto S = index<2>{1,0};
  auto W = index<2>{ 0,-1}; auto E = index<2>{0,1};

  for (index<2> idx : in.bounds()) {
    if (idx[0] == 0 || idx[0] == in.bounds()[0] - 1
     || idx[1] == 0 || idx[1] == in.bounds()[1] - 1) {
      out[idx] = 0;
    } else {
      int v =
       - 1 * in[idx + W + N]
       + 1 * in[idx + E + N]
       - 2 * in[idx + W    ]
       + 2 * in[idx + E    ]
       - 1 * in[idx + W + S]
       + 2 * in[idx + S + E];
      out[idx] =
       v > 150 ? 255 : 0;
} } }
```



Using the `array_view`

```
void compute(uint8_t* in, uint8_t* out,
             size_t width, size_t height) {


  for (size_t row = 0; row < height; ++row)
  for (size_t col = 0; col < width; ++col) {
    if (row == 0 || row == height - 1
     || col == 0 || col == width - 1) {
      out[row * width + col] = 0;
    } else {
      int v =
       - 1 * in[(row - 1) * width + col - 1]
       + 1 * in[(row - 1) * width + col + 1]
       - 2 * in[ row      * width + col - 1]
       + 2 * in[ row      * width + col + 1]
       - 1 * in[(row + 1) * width + col - 1]
       + 2 * in[(row + 1) * width + col + 1];
      out[row * width + col] =
       v > 150 ? 255 : 0;
} } }
```

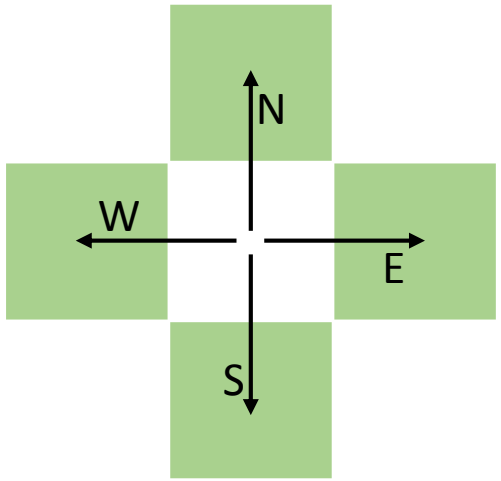
```
void compute(array_view<const uint8_t, 2> in,
             array_view<uint8_t, 2> out) {
  auto N = index<2>{-1, 0}; auto S = index<2>{1,0};
  auto W = index<2>{ 0,-1}; auto E = index<2>{0,1};

  for (index<2> idx : in.bounds()) {
    if (idx[0] == 0 || idx[0] == in.bounds()[0] - 1
     || idx[1] == 0 || idx[1] == in.bounds()[1] - 1) {
      out[idx] = 0;
    } else {
      int v =
       - 1 * in[idx + W + N]
       + 1 * in[idx + E + N]
       - 2 * in[idx + W    ]
       + 2 * in[idx + E    ]
       - 1 * in[idx + W + S]
       + 2 * in[idx + S + E];
      out[idx] =
       v > 150 ? 255 : 0;
} } }
```



Using the `array_view`

```
void compute(uint8_t* in, uint8_t* out,
             size_t width, size_t height) {


  for (size_t row = 0; row < height; ++row)
  for (size_t col = 0; col < width; ++col) {
    if (row == 0 || row == height - 1
     || col == 0 || col == width - 1) {
      out[row * width + col] = 0;
    } else {
      int v =
       - 1 * in[(row - 1) * width + col - 1]
       + 1 * in[(row - 1) * width + col + 1]
       - 2 * in[ row      * width + col - 1]
       + 2 * in[ row      * width + col + 1]
       - 1 * in[(row + 1) * width + col - 1]
       + 2 * in[(row + 1) * width + col + 1];
      out[row * width + col] =
       v > 150 ? 255 : 0;
} } }
```

```
void compute(array_view<const uint8_t, 2> in,
             array_view<uint8_t, 2> out) {
  auto N = index<2>{-1, 0}; auto S = index<2>{1,0};
  auto W = index<2>{ 0,-1}; auto E = index<2>{0,1};

  for (index<2> idx : in.bounds()) {
    if (idx[0] == 0 || idx[0] == in.bounds()[0] - 1
     || idx[1] == 0 || idx[1] == in.bounds()[1] - 1) {
      out[idx] = 0;
    } else {
      int v =
       - 1 * in[idx + W + N]
       + 1 * in[idx + E + N]
       - 2 * in[idx + W    ]
       + 2 * in[idx + E    ]
       - 1 * in[idx + W + S]
       + 2 * in[idx + S + E];
      out[idx] =
       v > 150 ? 255 : 0;
} } }
```



A different view on algorithms

Elemental algorithms	Indexable algorithms
`auto coll = std::vector<T>{N};`	`auto bnd = bounds<1>{N};`
`for(T& value : coll) { ... }`	`for(index<1> idx : bnd) { ... }`
`for_each(begin(coll), end(coll),`	`for_each(begin(bnd), end(bnd),`
`[=](T& value) {...});`	`[=](index<1> idx) {...});`

A different view on algorithms

Elemental algorithms

```
auto coll = std::vector<T>{N};
for(T& value : coll) { ... }
for_each(begin(coll), end(coll),
         [=](T& value) {...});
```

Indexable algorithms

```
auto bnd = bounds<1>{N};
for(index<1> idx : bnd) { ... }
for_each(begin(bnd), end(bnd),
         [=](index<1> idx) {...});
```

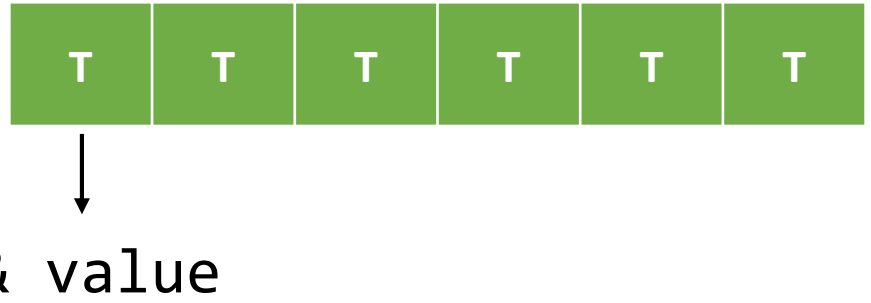
A different view on algorithms

Elemental algorithms

```
auto coll = std::vector<T>{N};
for(T& value : coll) { ... }
for_each(begin(coll), end(coll),
         [=](T& value) {...});
```

Indexable algorithms

```
auto bnd = bounds<1>{N};
for(index<1> idx : bnd) { ... }
for_each(begin(bnd), end(bnd),
         [=](index<1> idx) {...});
```



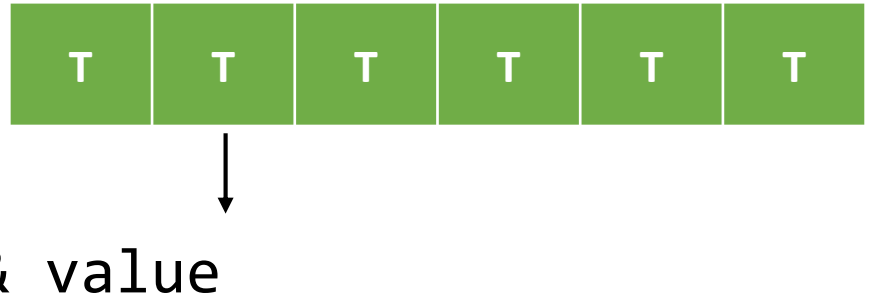
A different view on algorithms

Elemental algorithms

```
auto coll = std::vector<T>{N};
for(T& value : coll) { ... }
for_each(begin(coll), end(coll),
         [=](T& value) {...});
```

Indexable algorithms

```
auto bnd = bounds<1>{N};
for(index<1> idx : bnd) { ... }
for_each(begin(bnd), end(bnd),
         [=](index<1> idx) {...});
```



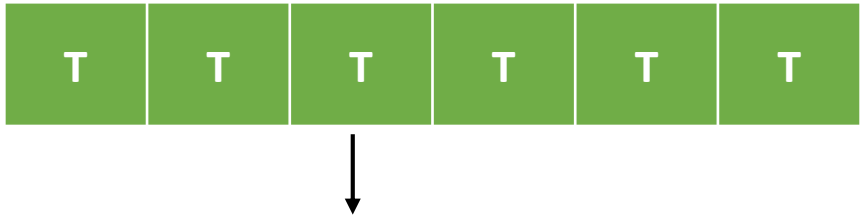
A different view on algorithms

Elemental algorithms

```
auto coll = std::vector<T>{N};
for(T& value : coll) { ... }
for_each(begin(coll), end(coll),
         [=](T& value) {...});
```

Indexable algorithms

```
auto bnd = bounds<1>{N};
for(index<1> idx : bnd) { ... }
for_each(begin(bnd), end(bnd),
         [=](index<1> idx) {...});
```



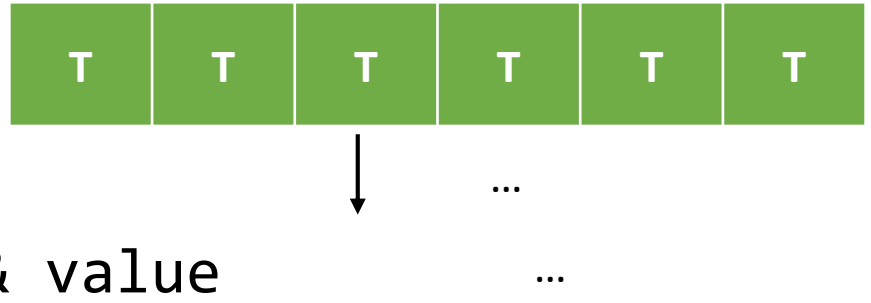
A different view on algorithms

Elemental algorithms

```
auto coll = std::vector<T>{N};
for(T& value : coll) { ... }
for_each(begin(coll), end(coll),
         [=](T& value) {...});
```

Indexable algorithms

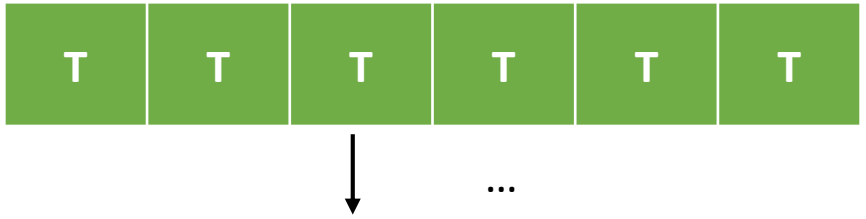
```
auto bnd = bounds<1>{N};
for(index<1> idx : bnd) { ... }
for_each(begin(bnd), end(bnd),
         [=](index<1> idx) {...});
```



A different view on algorithms

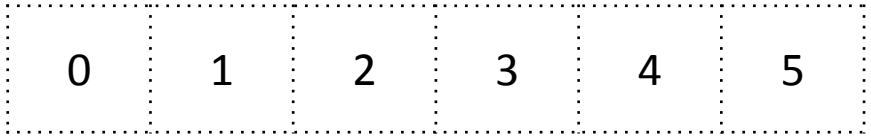
Elemental algorithms

```
auto coll = std::vector<T>{N};
for(T& value : coll) { ... }
for_each(begin(coll), end(coll),
         [=](T& value) {...});
```



Indexable algorithms

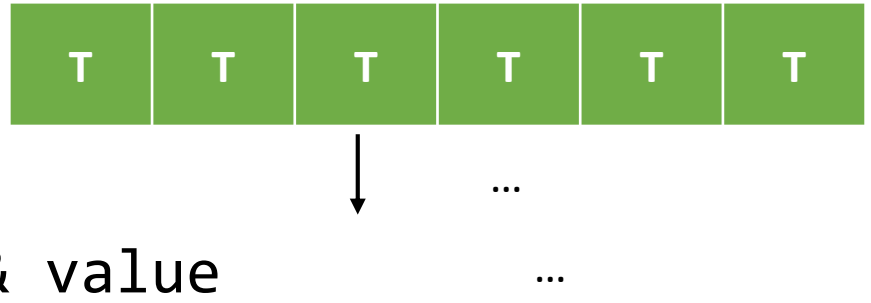
```
auto bnd = bounds<1>{N};
for(index<1> idx : bnd) { ... }
for_each(begin(bnd), end(bnd),
         [=](index<1> idx) {...});
```



A different view on algorithms

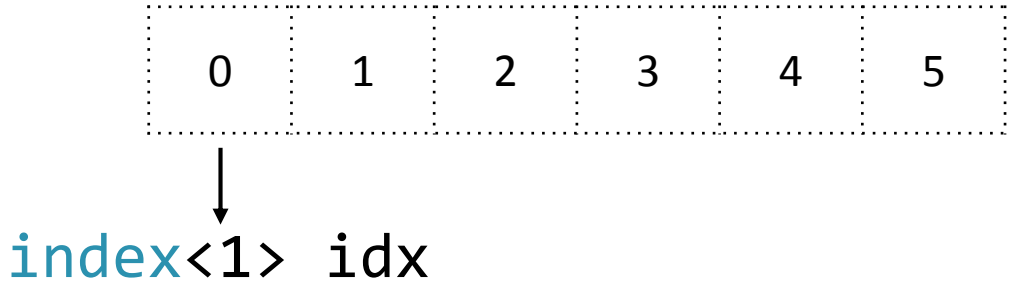
Elemental algorithms

```
auto coll = std::vector<T>{N};
for(T& value : coll) { ... }
for_each(begin(coll), end(coll),
         [=](T& value) {...});
```



Indexable algorithms

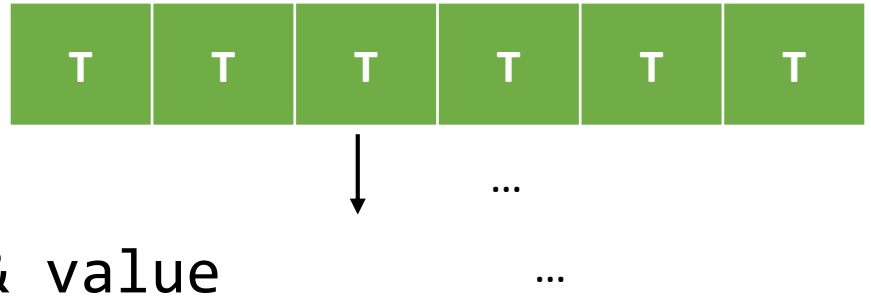
```
auto bnd = bounds<1>{N};
for(index<1> idx : bnd) { ... }
for_each(begin(bnd), end(bnd),
         [=](index<1> idx) {...});
```



A different view on algorithms

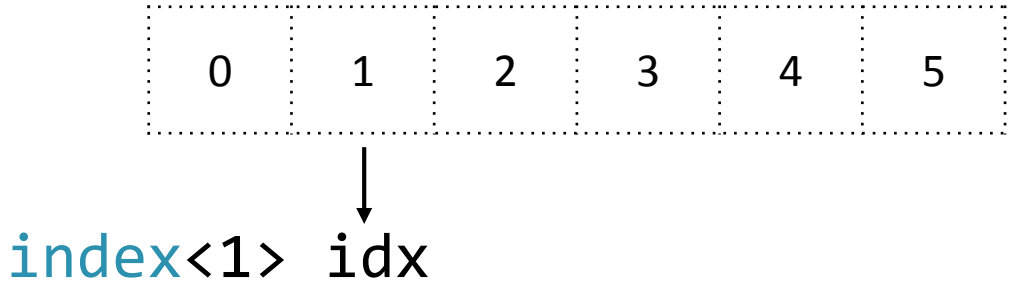
Elemental algorithms

```
auto coll = std::vector<T>{N};
for(T& value : coll) { ... }
for_each(begin(coll), end(coll),
         [=](T& value) {...});
```



Indexable algorithms

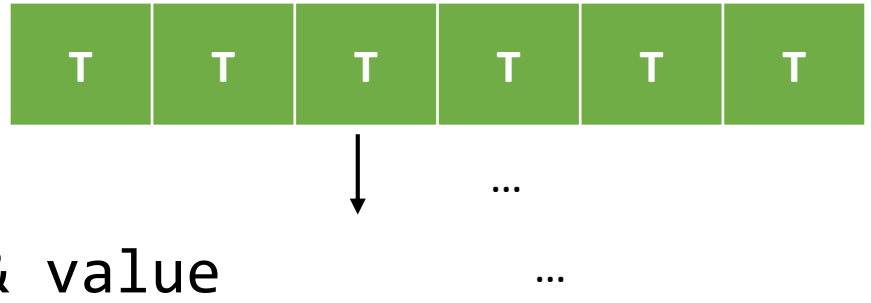
```
auto bnd = bounds<1>{N};
for(index<1> idx : bnd) { ... }
for_each(begin(bnd), end(bnd),
         [=](index<1> idx) {...});
```



A different view on algorithms

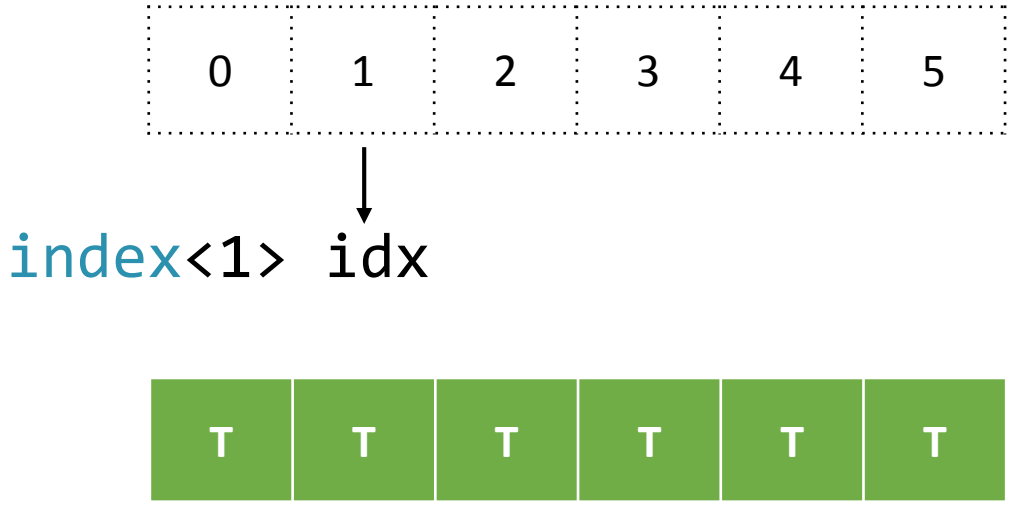
Elemental algorithms

```
auto coll = std::vector<T>{N};
for(T& value : coll) { ... }
for_each(begin(coll), end(coll),
         [=](T& value) {...});
```



Indexable algorithms

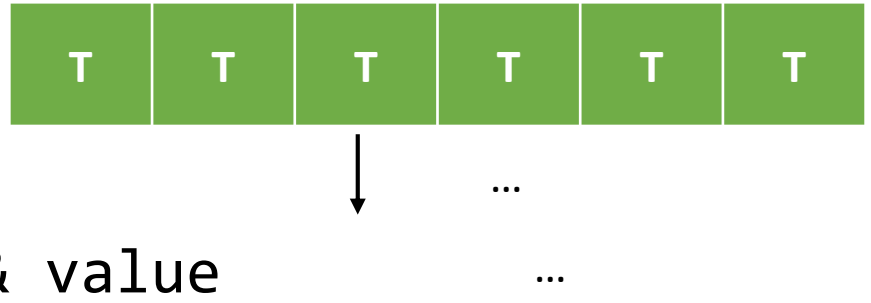
```
auto bnd = bounds<1>{N};
for(index<1> idx : bnd) { ... }
for_each(begin(bnd), end(bnd),
         [=](index<1> idx) {...});
```



A different view on algorithms

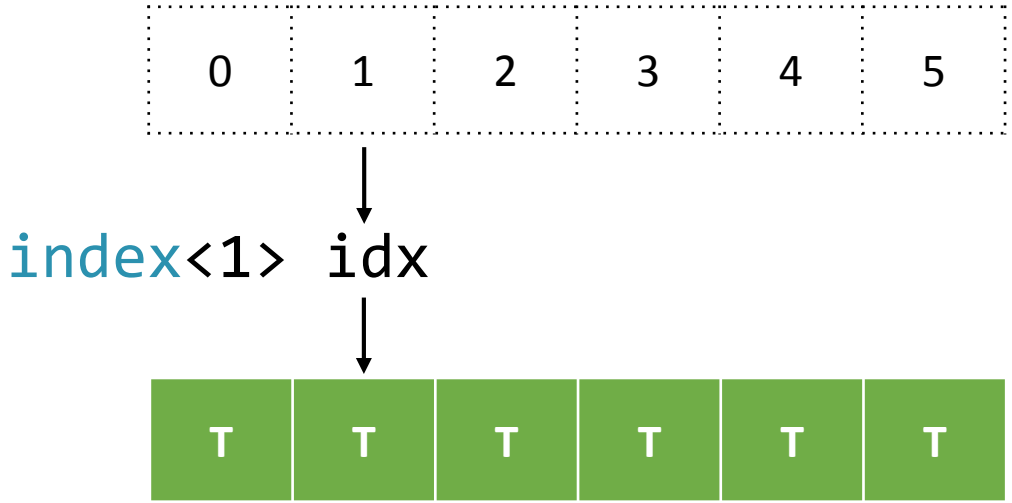
Elemental algorithms

```
auto coll = std::vector<T>{N};
for(T& value : coll) { ... }
for_each(begin(coll), end(coll),
         [=](T& value) {...});
```



Indexable algorithms

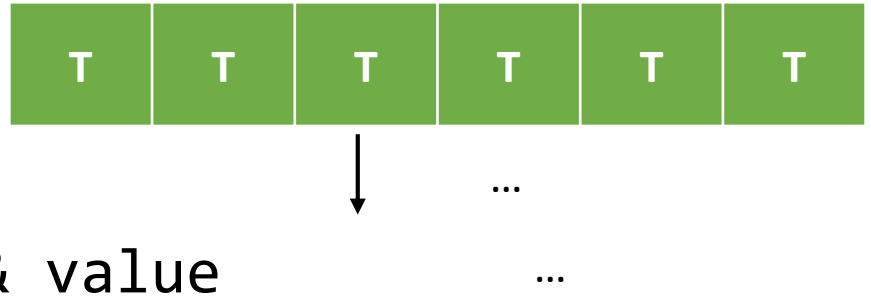
```
auto bnd = bounds<1>{N};
for(index<1> idx : bnd) { ... }
for_each(begin(bnd), end(bnd),
         [=](index<1> idx) {...});
```



A different view on algorithms

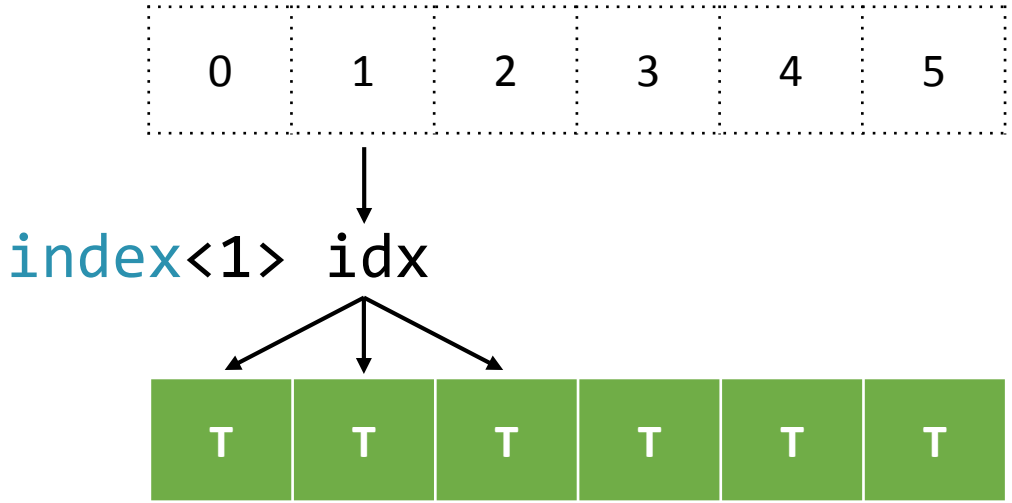
Elemental algorithms

```
auto coll = std::vector<T>{N};
for(T& value : coll) { ... }
for_each(begin(coll), end(coll),
         [=](T& value) {...});
```



Indexable algorithms

```
auto bnd = bounds<1>{N};
for(index<1> idx : bnd) { ... }
for_each(begin(bnd), end(bnd),
         [=](index<1> idx) {...});
```



Enabling index-based parallelism in C++

“Regular” STL + the multidimensional index

```
for_each(begin(bnd), end(bnd), [=](index<2> idx) {...});
```

Enabling index-based parallelism in C++

“Regular” STL + the multidimensional index

```
for_each(begin(bnd), end(bnd), [=](index<2> idx) {...});
```

C++ Extensions for Parallelism TS (“Parallel STL”) + the multidimensional index

Enabling index-based parallelism in C++

“Regular” STL + the multidimensional index

```
for_each(begin(bnd), end(bnd), [=](index<2> idx) {...});
```

C++ Extensions for Parallelism TS (“Parallel STL”) + the multidimensional index

```
for_each(experimental::parallel::par_vec,
```

Enabling index-based parallelism in C++

“Regular” STL + the multidimensional index

```
for_each(begin(bnd), end(bnd), [=](index<2> idx) {...});
```

C++ Extensions for Parallelism TS (“Parallel STL”) + the multidimensional index

```
for_each(experimental::parallel::par_vec,
         begin(bnd), end(bnd), [=](index<2> idx) {...});
```

Enabling index-based parallelism in C++

“Regular” STL + the multidimensional index

```
for_each(begin(bnd), end(bnd), [=](index<2> idx) {...});
```

C++ Extensions for Parallelism TS (“Parallel STL”) + the multidimensional index

```
for_each(experimental::parallel::par_vec,
         begin(bnd), end(bnd), [=](index<2> idx) {...});
```

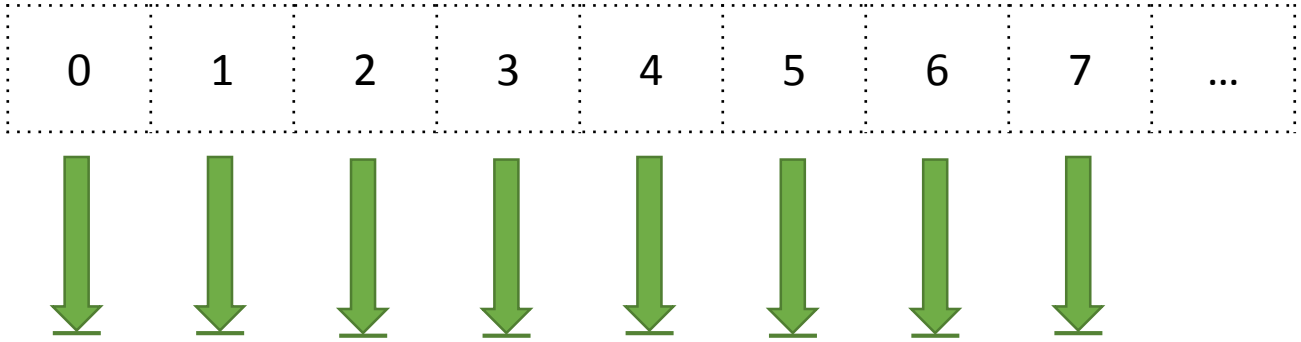
Enabling index-based parallelism in C++

“Regular” STL + the multidimensional index

```
for_each(begin(bnd), end(bnd), [=](index<2> idx) {...});
```

C++ Extensions for Parallelism TS (“Parallel STL”) + the multidimensional index

```
for_each(experimental::parallel::par_vec,
         begin(bnd), end(bnd), [=](index<2> idx) {...});
```



Enabling index-based parallelism in C++

“Regular” STL + the multidimensional index

```
for_each(begin(bnd), end(bnd), [=](index<2> idx) {...});
```

C++ Extensions for Parallelism TS (“Parallel STL”) + the multidimensional index

```
for_each(experimental::parallel::par_vec,
         begin(bnd), end(bnd), [=](index<2> idx) {...});
```

C++ AMP

```
parallel_for_each(bnd, [=](index<2> idx) restrict(amp) {...});
```

Enabling index-based parallelism in C++

"Regular" STL + the multidimensional index

```
for_each(begin(bnd), end(bnd), [=](index<2> idx) {...});
```

C++ Extensions for Parallelism TS ("Parallel STL") + the multidimensional index

```
for_each(experimental::parallel::par_vec,
         begin(bnd), end(bnd), [=](index<2> idx) {...});
```

CUDA

```
__global__ void kernel() {
  auto idx = f(blockIdx, blockDim, threadIdx); ... }
kernel<<<grid, threads>>>();
```

Enabling index-based parallelism in C++

"Regular" STL + the multidimensional index

```
for_each(begin(bnd), end(bnd), [=](index<2> idx) {...});
```

C++ Extensions for Parallelism TS ("Parallel STL") + the multidimensional index

```
for_each(experimental::parallel::par_vec,
         begin(bnd), end(bnd), [=](index<2> idx) {...});
```

OpenCL

```
__kernel__ void kernel() {
  auto row = get_global_id(0); auto col = get_global_id(1); ... }
clEnqueueNDRangeKernel(queue, kernel, 2, nullptr,
         global_work_size, local_work_size, 0, nullptr, nullptr);
```

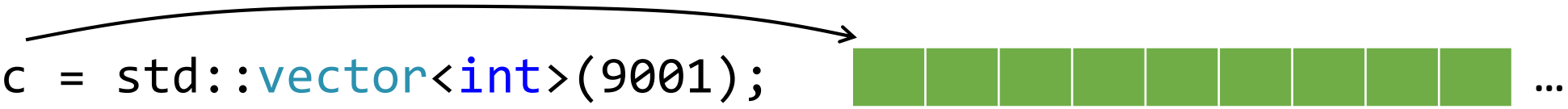
Pointer semantics – valueness

Pointer semantics – valueness

```
auto vec = std::vector<int>(9001);
```

Pointer semantics – valueness

```
auto vec = std::vector<int>(9001);
```



Pointer semantics – valueness

```
auto vec = std::vector<int>(9001);
auto vec_copy = vec;
```

Pointer semantics – valueness

```
auto vec = std::vector<int>(9001);
auto vec_copy = vec;
```

Pointer semantics – valueness

```
auto vec = std::vector<int>(9001);
auto vec_copy = vec;
```

```
auto ptr = new int[9001];
```

Pointer semantics – valueness

```
auto vec = std::vector<int>(9001);
auto vec_copy = vec;
```

```
auto ptr = new int[9001];
```

Pointer semantics – valueness

```
auto vec = std::vector<int>(9001);
auto vec_copy = vec;
```

```
auto ptr = new int[9001];
auto ptr_copy = ptr;
```

Pointer semantics – valueness

```
auto vec = std::vector<int>(9001);
auto vec_copy = vec;
```

```
auto ptr = new int[9001];
auto ptr_copy = ptr;
```

Pointer semantics – valueness

```
auto vec = std::vector<int>(9001);
auto vec_copy = vec;
```

```
auto av = array_view<int>{vec_copy};
```

```
auto ptr = new int[9001];
auto ptr_copy = ptr;
```

Pointer semantics – valueness

```
auto vec = std::vector<int>(9001);
auto vec_copy = vec;
```

```
auto av = array_view<int>{vec_copy};
auto av_copy = av;
```

```
auto ptr = new int[9001];
auto ptr_copy = ptr;
```

Pointer semantics – valueness

```
auto vec = std::vector<int>(9001);
```

```
auto av = array_view<int>{vec_copy};
auto av_copy = av;
```

```
auto ptr = new int[9001];
auto ptr_copy = ptr;
```

Pointer semantics – constness

Pointer semantics – constness

Constant view

Pointer semantics – constness

Constant view

```
const array_view<int> view  ~  int* const ptr
```

Pointer semantics – constness

Constant view

```
const array_view<int> view  ~  int* const ptr
view[0] = 42; ✓
```

Pointer semantics – constness

Constant view

```
const array_view<int> view  ~  int* const ptr
view[0] = 42; ✓
view = another_view; ✗
```

Pointer semantics – constness

Constant view

```
const array_view<int> view  ~  int* const ptr
view[0] = 42; ✓
view = another_view; ✗
```

View over constant data

Pointer semantics – constness

Constant view

```
const array_view<int> view  ~  int* const ptr
view[0] = 42; ✓
view = another_view; ✗
```

View over constant data

```
array_view<const int> view  ~  const int* ptr
```

Pointer semantics – constness

Constant view

```
const array_view<int> view  ~  int* const ptr
view[0] = 42; ✓
view = another_view; ✗
```

View over constant data

```
array_view<const int> view  ~  const int* ptr
view[0] = 42; ✗
```

Pointer semantics – constness

Constant view

```
const array_view<int> view  ~  int* const ptr
view[0] = 42; ✓
view = another_view; ✗
```

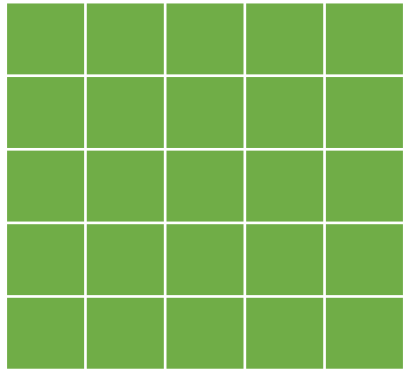
View over constant data

```
array_view<const int> view  ~  const int* ptr
view[0] = 42; ✗
view = another_view; ✓
```

Other operations – slice and section

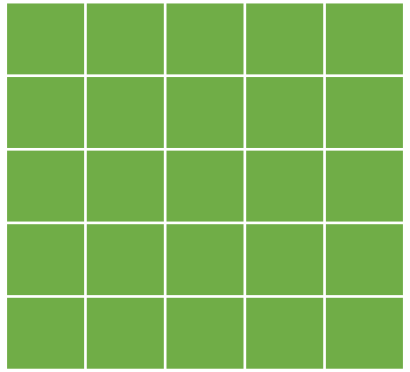
Other operations – slice and section

```
auto view = array_view<int, 2>{{5, 5}, data};
```



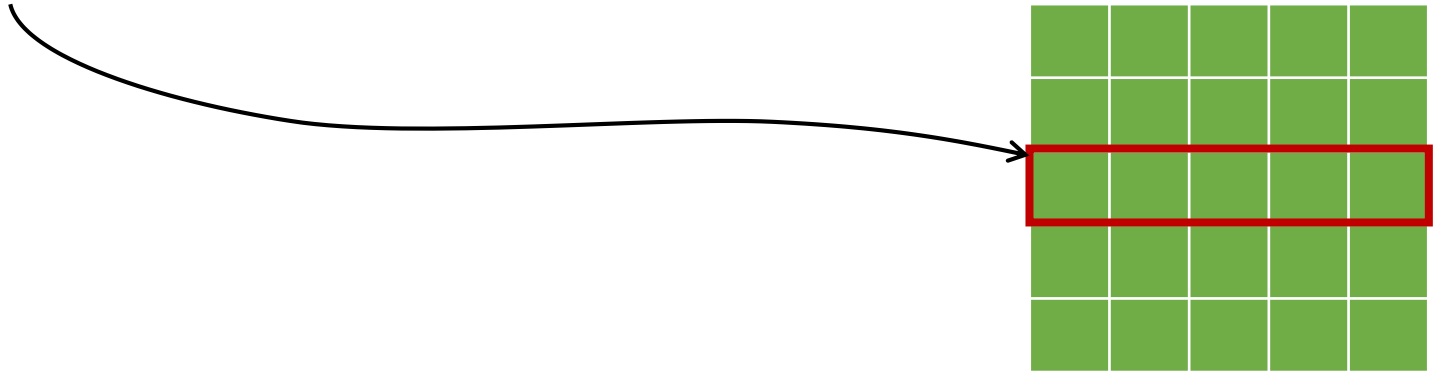
Other operations – slice and section

```
auto view = array_view<int, 2>{{5, 5}, data};
auto slice = view[2];
```



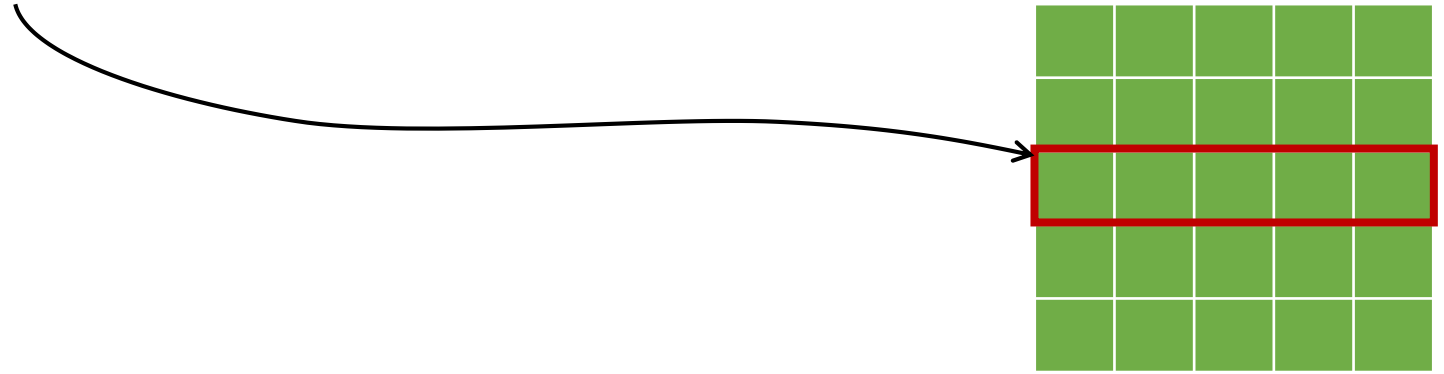
Other operations – slice and section

```
auto view = array_view<int, 2>{{5, 5}, data};
auto slice = view[2];
```



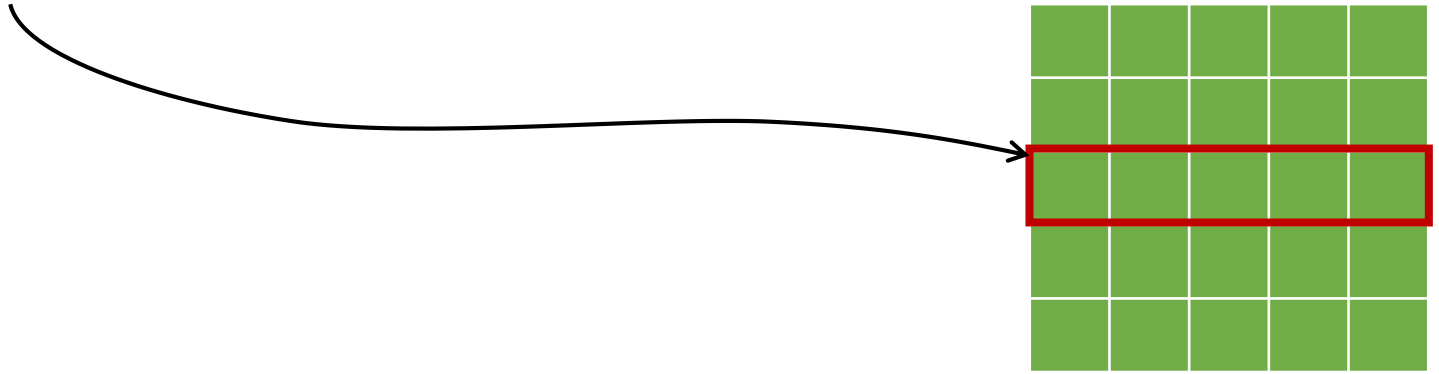
Other operations – slice and section

```
auto view = array_view<int, 2>{{5, 5}, data};
array_view<int, 1> slice = view[2];
```



Other operations – slice and section

```
auto view = array_view<int, 2>{{5, 5}, data};
array_view<int, 1> slice = view[2];
```



```
auto section = view.section({1, 2}, {3, 2});
```

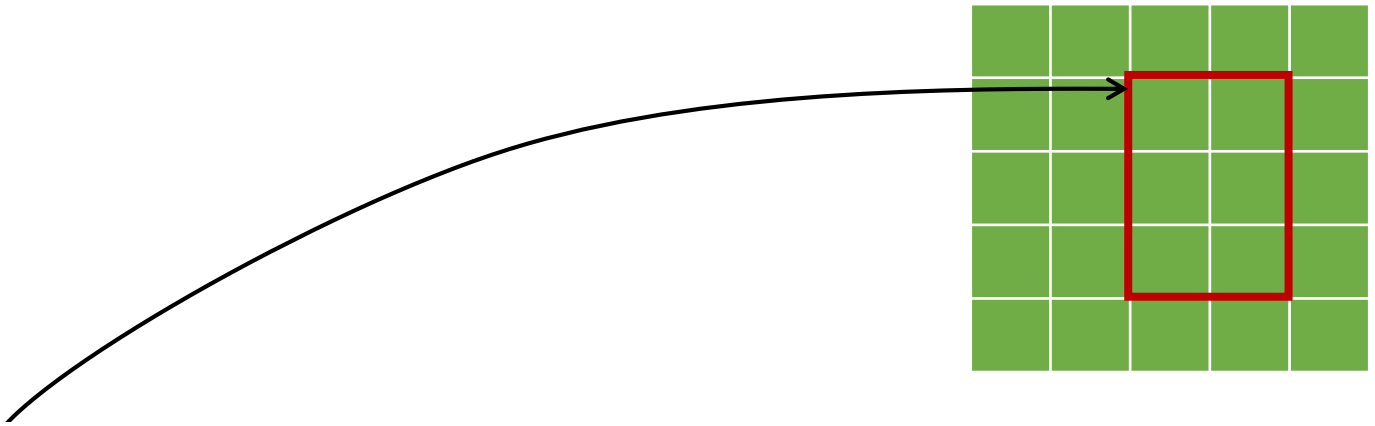
Other operations – slice and section

```
auto view = array_view<int, 2>{{5, 5}, data};
array_view<int, 1> slice = view[2];
```

```
auto section = view.section({1, 2}, {3, 2});
```

Other operations – slice and section

```
auto view = array_view<int, 2>{{5, 5}, data};
array_view<int, 1> slice = view[2];
```



```
strided_array_view<int, 2> section = view.section({1, 2}, {3, 2});
```

Novel types

- *bounds* and *index* – defining and addressing multidimensional discrete spaces.
- *array_view* and *strided_array_view* – multidimensional views on contiguous or strided memory ranges.
- *bounds_iterator* – constant random access iterator over an imaginary space imposed by a *bounds* object, with an *index* as its value type.

Towards the standardization

- N3851 – the introductory paper
  - Presented to LEWG at the Issaquah meeting (February 2014)
  - Consensus to prepare the wording for Arrays TS
- N3976 – the first formal wording paper
  - Presented to LEWG at the Rapperswil meeting (June 2014)
  - Some fixes and improvements in the wording requested
  - Consensus to forward the wording to Fundamentals v2 TS
- N4087 – the latest formal wording paper
  - To be presented to LWG at the Urbana-Champaign meeting (November 2014)
  - Hoping to have it accepted for Fundamentals v2 TS ☺

Proposed extensions

- *array_view* with a fixed size, driven by increased type safety and potential optimization opportunities:

```
fixed_array_view<int, 1, 2, 4>{ ptr }
  ≈ array_view<int, 3>{ {1, 2, 4}, ptr }
```

- Explicit column-major/row-major switch on *array_view*, driven by the desire for Fortran interop

- Parameterized traversal order for *bounds_iterator* – column-major, Morton order, Hilbert curve, …

Our proof-of-concept is available at:

http://parallelstl.codeplex.com

#include <experimental/array_view>

Our proof-of-concept is available at:

http://parallelstl.codeplex.com

```
#include <experimental/array_view>
```

include/experimental/impl/array_view.h
include/experimental/impl/coordinate.h

Q&A

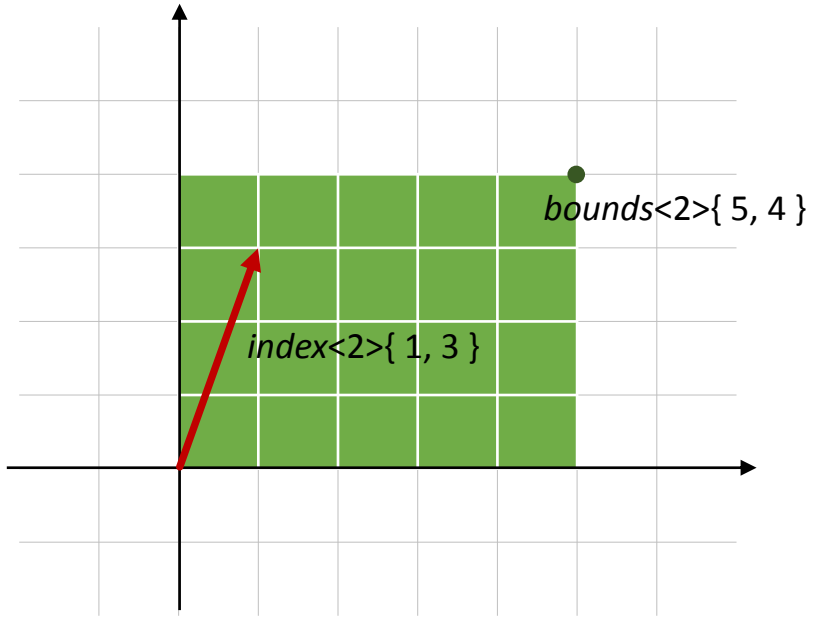
lukaszme@microsoft.com

Backup

bounds and *index*

index<N> = N-dimensional vector

bounds<N> = N-dim axis-aligned rectangle with the minimum point at **0**
≈ *maximum point of such rectangle*



bounds and *index* – basic usage

```
auto bnd = bounds<3>{ 3, 1, 4 };
auto idx = index<3>{ 2, -1, 0 };

bounds<3> bnd2 = bnd + idx;  // bnd2 is { 5, 0, 4 }
bnd2 -= idx;                 // bnd2 is { 3, 1, 4 }

auto v1 = idx[0];   // v1 is 2

bnd.contains(idx);  // -> false
bnd.size();         // -> 3 * 1 * 4 = 12
```

bounds and *index* – difference in arithmetic

bounds<N>	**index<N>**
bounds<N> ⊙ index<N> → bounds<N> + − += −= index<N> ⊙ bounds<N> → bounds<N> +	index<N> ⊙ index<N> → index<N> + += − −=
bounds<N> ⊙ arithmetic type → bounds<N> * / *= /= arithmetic type ⊙ bounds<N> → bounds<N> *	index<N> ⊙ arithmetic type → index<N> * / *= /= arithmetic type ⊙ index<N> → index<N> *
	⊙ index<N> → index<N> + − ++ −− (for N = 1, and also post- variants)

bounds and *index* – difference in functionality

Only for *bounds<N>*:

```
constexpr size_type size() const noexcept;
bool contains(const index<rank>& idx) const noexcept;

bounds_iterator<rank> begin() const noexcept;
bounds_iterator<rank> end() const noexcept;
```

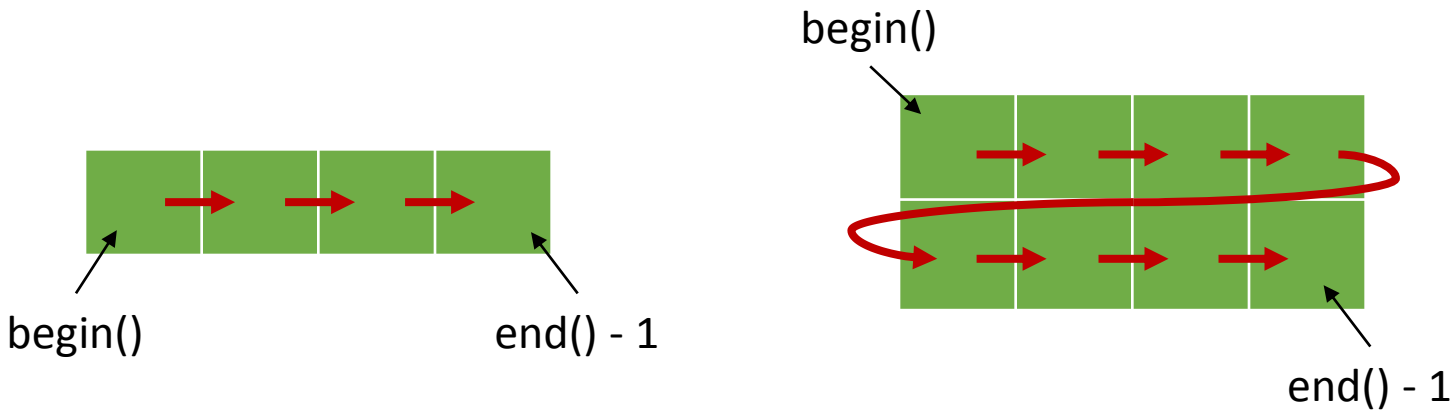
bounds_iterator

Constant iterator over *bounds<N>* returning *index<N>*

```
auto bnd = bounds<2>{4, 10};

bounds_iterator<2> it = begin(bnd);
index<2> idx = *it;  // idx is {0, 0}

++it;
idx = *it;  // idx is {0, 1}

it += 10;
idx = *it;  // idx is {1, 1}
```

bounds_iterator – linearization

Since *bounds_iterator* provides a traversal over *Rank*-dimensional discrete space defined by *bounds*, it is necessary to linearize the space.



array_view and *strided_array_view*

- *array_view* – requires contiguous regular data (e.g. int data[4][1][8]).
- *strided_array_view* – requires regular data

The only difference: **contiguity**.

- **contiguous** view allows for cache-oblivious algorithms (performance).
- **contiguous** view allows for .data() function (compatibility).

- **non-contiguous** view allows for more flexibility.

Guidance: use *array_view* when you can (reflected in constructors).

strided_array_view as a transposed view

```
int cm_array[3 * 5] = {
    1, 4, 7, 10, 13,
    2, 5, 8, 11, 14,
    3, 6, 9, 12, 15
};
auto cm_sav
 = strided_array_view<int, 2>{ { 5, 3 }, { 1, 5 }, cm_array };

assert((cm_sav[{0, 0}] == 1));
assert((cm_sav[{0, 1}] == 2));
assert((cm_sav[{1, 0}] == 4));
assert((cm_sav[{4, 2}] == 15));
```

av and *sav* implicit conversions

array_view<T, N> → *array_view<**const** T, N>*

array_view<T, N> → ***strided_**array_view<T, N>*

array_view<T, N> → ***strided_**array_view<**const** T, N>*

strided_array_view<T, N> → *strided_array_view<**const** T, N>*

Relations between (*s*)*av* and other types