Base Language

Safe Harbor Statement

This presentation may contain forward-looking statements that are based on our current expectations. Forward looking statements may include statements about our financial guidance and expected operating results, our opportunities and future potential, our product development and new product introduction plans, our ability to expand and penetrate our addressable markets and other statements that are not historical facts. These statements are only predictions and actual results may materially vary from those projected. Please refer to Cray's documents filed with the SEC from time to time concerning factors that could affect the Company and these forward-looking statements.

naïve n-body computation in Chapel

n-body in Chapel (where n == 5)

- **A serial computation**
- **From the Computer Language Benchmarks Game**
  - Chapel implementation in release under examples/benchmarks/shootout/nbody.chpl
- **Computes the influence of 5 bodies on one another**
  - The Sun, Jupiter, Saturn, Uranus, Neptune
- **Executes for a user-specifiable number of timesteps**

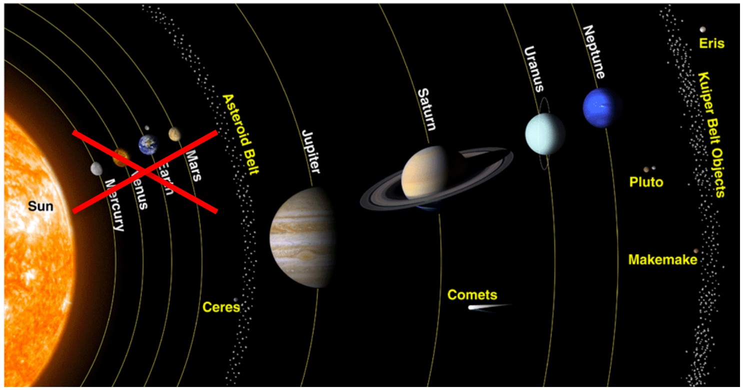

Image source: http://spaceplace.nasa.gov/review/ice-dwarf/solar-system-lrg.png

5-body in Chapel: Declarations

```
const pi = 3.141592653589793,
      solarMass = 4 * pi**2,
      daysPerYear = 365.24;

config const numsteps = 10000;

record body {
  var pos: 3*real;
  var v: 3*real;
  var mass: real;
}


…
```

5-body in Chapel: Declarations

```
const pi = 3.141592653589793,
      solarMass = 4 * pi**2,
      daysPerYear = 365.24;

config const numsteps = 10000;

record body {
  var pos: 3*real;
  var v: 3*real;
  var mass: real;
}

…
```

Variable declarations

Configuration Variable

Record declaration

Tuple type

5-body in Chapel: Declarations

```
const pi = 3.141592653589793,
      solarMass = 4 * pi**2,
      daysPerYear = 365.24;

config const numsteps = 10000;

record body {
  var pos: 3*real;
  var v: 3*real;
  var mass: real;
}

…
```

Variable declarations

Configuration Variable

Record declaration

Tuple type

Variables, Constants, and Parameters

- **Basic syntax**

```
declaration:
  var   identifier [: type] [= init-expr];
  const identifier [: type] [= init-expr];
  param identifier [: type] [= init-expr];
```

- **Meaning**
  - `var`/`const`: execution-time variable/constant
  - `param`: compile-time constant
  - No *init-expr* ⇒ initial value is the type's default
  - No *type* ⇒ type is taken from *init-expr*
- **Examples**

```
const pi: real = 3.14159;
var count: int;              // initialized to 0
param debug = true;          // inferred to be bool
```

Chapel's Static Type Inference

```
const pi = 3.14,              // pi is a real
      coord = 1.2 + 3.4i,     // coord is a complex…
      coord2 = pi*coord,      // …as is coord2
      name = "brad",          // name is a string
      verbose = false;        // verbose is boolean

proc addem(x, y) {            // addem() has generic arguments
  return x + y;               //   and an inferred return type
}

var sum = addem(1, pi),                 // sum is a real
    fullname = addem(name, "ford");     // fullname is a string

writeln((sum, fullname));
```

```
(4.14, bradford)
```

5-body in Chapel: Declarations

```
const pi = 3.141592653589793,
      solarMass = 4 * pi**2,
      daysPerYear = 365.24;

config const numsteps = 10000;

record body {
  var pos: 3*real;
  var v: 3*real;
  var mass: real;
}

…
```

Variable declarations

Configuration Variable

Record declaration

Tuple type

5-body in Chapel: Declarations

```
const pi = 3.141592653589793,
      solarMass = 4 * pi**2,
      daysPerYear = 365.24;

config const numsteps = 10000;

record body {
  var pos: 3*real;
  var v: 3*real;
  var mass: real;
}

…
```

Configuration Variable

```
$ ./nbody --numsteps=100
```

Configs

```
param intSize = 32;
type elementType = real(32);
const epsilon = 0.01:elementType;
var start = 1:int(intSize);
```

Configs

```
config param intSize = 32;
config type elementType = real(32);
config const epsilon = 0.01:elementType;
config var start = 1:int(intSize);
```

```
$ chpl myProgram.chpl -sintSize=64 -selementType=real
$ ./a.out --start=2 --epsilon=0.00001
```

5-body in Chapel: Declarations

```
const pi = 3.141592653589793,
      solarMass = 4 * pi**2,
      daysPerYear = 365.24;

config const numsteps = 10000;

record body {
  var pos: 3*real;
  var v: 3*real;
  var mass: real;
}

…
```

Variable declarations

Configuration Variable

Record declaration

Tuple type

Records and Classes

- **Chapel's struct/object types**
  - Contain variable definitions (fields)
  - Contain procedure & iterator definitions (methods)
  - Records: value-based (*e.g.*, assignment copies fields)
  - Classes: reference-based (*e.g.*, assignment aliases object)
  - Record : Class :: C++ struct : Java class

- **Example**

```
record circle {
  var radius: real;
  proc area() {
    return pi*radius**2;
  }
}
```

```
var c1, c2: circle;
c1 = new circle(radius=1.0);
c2 = c1;                // copies c1
c1.radius = 5.0;
writeln(c2.radius);   // 1.0
// records deleted by compiler
```

Records and Classes

- **Chapel's struct/object types**
  - Contain variable definitions (fields)
  - Contain procedure & iterator definitions (methods)
  - Records: value-based (*e.g.*, assignment copies fields)
  - Classes: reference-based (*e.g.*, assignment aliases object)
  - Record : Class :: C++ struct : Java class

- **Example**

```
class circle {
  var radius: real;
  proc area() {
    return pi*radius**2;
  }
}
```

```
var c1, c2: circle;
c1 = new circle(radius=1.0);
c2 = c1;  // aliases c1's circle
c1.radius = 5.0;
writeln(c2.radius);  // 5.0
delete c1; // users delete classes
```

Classes vs. Records

Classes

- **heap-allocated**
  - Pointers to fields
  - Requires 'delete'
- **'ref' semantics**
  - crucial when object identity matters
- **support dynamic dispatch**
- **support inheritance**
- **similar to Java classes**

Records

- **allocated in-place**
  - Fields in contiguous memory
  - Memory managed
- **'value' semantics**
  - compiler may introduce copies
- **no dynamic dispatch**
- **no inheritance (yet)**
- **similar to C++ structs (sans pointers)**

5-body in Chapel: Declarations

```
const pi = 3.141592653589793,
      solarMass = 4 * pi**2,
      daysPerYear = 365.24;

config const numsteps = 10000;

record body {
  var pos: 3*real;
  var v: 3*real;
  var mass: real;
}

…
```

Variable declarations

Configuration Variable

Record declaration

Tuple type

Tuples

- **Use**
  - support lightweight grouping of values
    - e.g., passing/returning procedure arguments
    - multidimensional array indices
    - short vectors
  - support heterogeneous data types
- **Examples**

```
var coord: (int, int, int) = (1, 2, 3);
var coordCopy: 3*int = coord;
var (i1, i2, i3) = coord;
var triple: (int, string, real) = (7, "eight", 9.0);
```

5-body in Chapel: Declarations

```
const pi = 3.141592653589793,
      solarMass = 4 * pi**2,
      daysPerYear = 365.24;

config const numsteps = 10000;

record body {
  var pos: 3*real;
  var v: 3*real;
  var mass: real;
}

…
```

Variable declarations

Configuration Variable

Record declaration

Tuple type

5-body in Chapel: the Bodies

```
var bodies =
    [  /* sun */
       new body(mass = solarMass),

       /* jupiter */
       new body(pos = ( 4.84143144246472090e+00,
                       -1.16032004402742839e+00,
                       -1.03622044471123109e-01),
                  v = ( 1.66007664274403694e-03 * daysPerYear,
                        7.69901118419740425e-03 * daysPerYear,
                       -6.90460016972063023e-05 * daysPerYear),
               mass =   9.54791938424326609e-04 * solarMass),

       /* saturn */
       new body(…),

       /* uranus */
       new body(…),

       /* neptune */
       new body(…)
   ]
```

5-body in Chapel: the Bodies

```
var bodies =
     [  /* sun */
        new body(mass = solarMass),

        /* jupiter */
        new body(pos = ( 4.84143144246472090e+00,
                        -1.16032004402742839e+00,
                        -1.03622044471123109e-01),
                   v = ( 1.66007664274403694e-03 * daysPerYear,
                         7.69901118419740425e-03 * daysPerYear,
                        -6.90460016972063023e-05 * daysPerYear),
                mass =   9.54791938424326609e-04 * solarMass),

        /* saturn */
        new body(…),

        /* uranus */
        new body(…),

        /* neptune */
        new body(…)
    ]
```

Creating a Record

Tuples

Array

5-body in Chapel: the Bodies

```
var bodies =
    [   /* sun */
        new body(mass = solarMass),

        /* jupiter */
        new body(pos = ( 4.84143144246472090e+00,
                        -1.16032004402742839e+00,
                        -1.03622044471123109e-01),
                   v = ( 1.66007664274403694e-03 * daysPerYear,
                         7.69901118419740425e-03 * daysPerYear,
                        -6.90460016972063023e-05 * daysPerYear),
                mass =   9.54791938424326609e-04 * solarMass),

        /* saturn */
        new body(…),

        /* uranus */
        new body(…),

        /* neptune */
        new body(…)
   ]
```

Tuples

5-body in Chapel: the Bodies

```
var bodies =
    [  /* sun */
       new body(mass = solarMass),

       /* jupiter */
       new body(pos = ( 4.84143144246472090e+00,
                       -1.16032004402742839e+00,
                       -1.03622044471123109e-01),
                  v = ( 1.66007664274403694e-03 * daysPerYear,
                        7.69901118419740425e-03 * daysPerYear,
                       -6.90460016972063023e-05 * daysPerYear),
               mass =   9.54791938424326609e-04 * solarMass),

       /* saturn */
       new body(…),

       /* uranus */
       new body(…),

       /* neptune */
       new body(…)
    ]
```

Creating a Record

Tuples

Array

5-body in Chapel: the Bodies

```
var bodies =
    [  /* sun */
       new body(mass = solarMass),

       /* jupiter */
       new body(pos = ( 4.84143144246472090e+00,
                       -1.16032004402742839e+00,
                       -1.03622044471123109e-01),
                  v = ( 1.66007664274403694e-03 * daysPerYear,
                        7.69901118419740425e-03 * daysPerYear,
                       -6.90460016972063023e-05 * daysPerYear),
               mass =   9.54791938424326609e-04 * solarMass),

       /* saturn */
       new body(…),

       /* uranus */
       new body(…),

       /* neptune */
       new body(…)
   ]
```

Creating a Record

Tuples

Array

Array Types

- **Syntax**

```
array-type:
  [ domain-expr ] elt-type
array-value:
  [elt1, elt2, elt3, … eltn]
```

- **Meaning:**
  - array-type: stores an element of *elt-type* for each index
  - array-value: represent the array with these values
- **Examples**

```
var A: [1..3] int,             // A stores 0, 0, 0
    B = [5, 3, 9],             // B stores 5, 3, 9
    C: [1..m, 1..n] real,      // 2D m by n array of reals
    D: [1..m][1..n] real;      // array of arrays of reals
```

Much more on arrays in data parallelism section later…

5-body in Chapel: the Bodies

```
var bodies =
    [  /* sun */
       new body(mass = solarMass),

       /* jupiter */
       new body(pos = ( 4.84143144246472090e+00,
                       -1.16032004402742839e+00,
                       -1.03622044471123109e-01),
                  v = ( 1.66007664274403694e-03 * daysPerYear,
                        7.69901118419740425e-03 * daysPerYear,
                       -6.90460016972063023e-05 * daysPerYear),
               mass =   9.54791938424326609e-04 * solarMass),

       /* saturn */
       new body(…),

       /* uranus */
       new body(…),

       /* neptune */
       new body(…)
    ]
```

Creating a Record

Tuples

Array

5-body in Chapel: main()

```
…

proc main() {
  initSun();

  writef("%.9r\n", energy());
  for 1..numsteps do
    advance(0.01);
  writef("%.9r\n", energy());
}

…
```

5-body in Chapel: main()

```
…

proc main() {
  initSun();

  writef("%.9r\n", energy());
  for 1..numsteps do
    advance(0.01);
  writef("%.9r\n", energy());
}

…
```

Procedure Definition

Procedure Call

Formatted I/O
*not covered here

Looping over a Range

5-body in Chapel: main()

```
…

proc main() {
  initSun();

  writef("%.9r\n", energy());
  for 1..numsteps do
    advance(0.01);
  writef("%.9r\n", energy());
}

…
```

Procedure Definition

Procedure Call

Formatted I/O
*not covered here

Looping over a Range

5-body in Chapel: main()

```
…

proc main() {
  initSun();

  writef("%.9r\n", energy());
  for 1..numsteps do
    advance(0.01);
  writef("%.9r\n", energy());
}

…
```

Range Value

Range Values

- **Syntax**

```
range-expr:
  [low] .. [high]
```

- **Semantics**
  - Regular sequence of integers

    low <= high: *low*, *low*+1, *low*+2, …, *high*

    low > *high*: degenerate (an empty range)

    low or *high* unspecified: unbounded in that direction

- **Examples**

```
1..6                // 1, 2, 3, 4, 5, 6
6..1                // empty
3..                 // 3, 4, 5, 6, 7, …
```

Range Operators

```
const r = 1..10;

printVals(r);
printVals(r # 3);
printVals(r by 2);
printVals(r by -2);
printVals(r by 2 # 3);
printVals(r # 3 by 2);
printVals(0.. #n);

proc printVals(r) {
  for i in r do
    write(i, " ");
  writeln();
}
```

```
1 2 3 4 5 6 7 8 9 10
1 2 3
1 3 5 7 9
10 8 6 4 2
1 3 5
1 3
0 1 2 3 4 … n-1
```

5-body in Chapel: main()

```
…

proc main() {
  initSun();

  writef("%.9r\n", energy());
  for 1..numsteps do
    advance(0.01);
  writef("%.9r\n", energy());
}

…
```

Procedure Definition

Procedure Call

Formatted I/O
*not covered here

Looping over a Range

For Loops

- **Syntax:**

```
for-loop:
  for [index-expr in] iteratable-expr { stmt-list }
```

- **Meaning:**
  - Executes loop body serially, once per loop iteration
  - Declares new variables for identifiers in *index-expr*
    - type and const-ness determined by *iteratable-expr*
    - *iteratable-expr* could be a range, array, or iterator

- **Examples**

```
var A: [1..3] string = [" DO", " RE", " MI"];

for i in 1..3 { write(A[i]); }          // DO RE MI
for a in A { a += "LA"; } write(A);    // DOLA RELA MILA
```

5-body in Chapel: main()

```
…

proc main() {
  initSun();

  writef("%.9r\n", energy());
  for 1..numsteps do
    advance(0.01);
  writef("%.9r\n", energy());
}

…
```

Function Declaration

Function Call

Formatted I/O
*not covered here

Looping over a Range

5-body in Chapel: advance()

```
 advance(0.01);
…
proc advance(dt) {
  for i in 1..numbodies {
    for j in i+1..numbodies {
      const dpos = bodies[i].pos - bodies[j].pos,
            mag = dt / sqrt(sumOfSquares(dpos))**3;

      bodies[i].v -= dpos * bodies[j].mass * mag;
      bodies[j].v += dpos * bodies[i].mass * mag;
    }
  }

  for b in bodies do
    b.pos += dt * b.v;
}
```

5-body in Chapel: advance()

```
 advance(0.01);
…
proc advance(dt) {
  for i in 1..numbodies {
    for j in i+1..numbodies {
      const dpos = bodies[i].pos - bodies[j].pos,
            mag = dt / sqrt(sumOfSquares(dpos))**3;

      bodies[i].v -= dpos * bodies[j].mass * mag;
      bodies[j].v += dpos * bodies[i].mass * mag;
    }
  }

  for b in bodies do
    b.pos += dt * b.v;
}
```

$$m_1 \mathbf{a}_1 = \frac{G m_1 m_2}{r_{12}^3}(\mathbf{r}_2 - \mathbf{r}_1) \quad \text{Sun-Earth}$$

$$m_2 \mathbf{a}_2 = \frac{G m_1 m_2}{r_{21}^3}(\mathbf{r}_1 - \mathbf{r}_2) \quad \text{Earth-Sun}$$

5-body in Chapel: advance()

```
 advance(0.01);
…
proc advance(dt) {
  for i in 1..numbodies {
    for j in i+1..numbodies {
      const dpos = bodies[i].pos - bodies[j].pos,
            mag = dt / sqrt(sumOfSquares(dpos))**3;

      bodies[i].v -= dpos * bodies[j].mass * mag;
      bodies[j].v += dpos * bodies[i].mass * mag;
    }
  }

  for b in bodies do
    b.pos += dt * b.v;
}
```

Procedure call

Procedure definition

Procedures, by example

- Example to compute the area of a circle

```
proc area(radius: real): real {
  return 3.14 * radius**2;
}

writeln(area(2.0));  // 12.56
```

```
proc area(radius) {
  return 3.14 * radius**2;
}
```

Argument and return types can be omitted

- Example of argument default values, naming

```
proc writeCoord(x: real = 0.0, y: real = 0.0) {
  writeln((x,y));
}

writeCoord(2.0);          // (2.0, 0.0)
writeCoord(y=2.0);        // (0.0, 2.0)
writeCoord(y=2.0, 3.0); // (3.0, 2.0)
```

5-body in Chapel: Using Iterators

```
iter triangle(n) {
  for i in 1..n do
    for j in i+1..n do
      yield (i,j);
}

proc advance(dt) {
  for (i,j) in triangle(numbodies) {
    const dpos = bodies[i].pos - bodies[j].pos,
          mag = dt / sqrt(sumOfSquares(dpos))**3;

    …
  }
…
}
…
```

Definition of iterator

Use of iterator

Additional Base Language Notes / Material

5-body in Chapel: advance() using references

```
proc advance(dt) {
  for i in 1..numbodies {
    for j in i+1..numbodies {
      ref bi = bodies[i],
          bj = bodies[j];

      const dpos = bi.pos - bj.pos,
            mag = dt / sqrt(sumOfSquares(dpos))**3;

      bi.v -= dpos * bj.mass * mag;
      bj.v += dpos * bi.mass * mag;
    }
  }

  for b in bodies do
    b.pos += dt * b.v;
}
```

Reference declarations

Reference Declarations

- **Syntax:**

```
ref-decl:
  ref ident = expr;
```

- **Meaning:**
  - Causes ‘ident’ to refer to variable specified by ‘expr’
  - Subsequent reads/writes of ‘ident’ refer to that variable
  - Not a pointer: no way to reference something else with ‘ident’
  - Similar to a C++ reference

- **Examples**

```
var A: [1..3] string = ["DO", "RE", "MI"];
ref a2 = A[2];
a2 = "YO";
for i in 1..3 { write(A[i]); }          // DO YO MI
```

Primitive Types

Type	Description	Default Value	Currently-Supported Bit Widths	Default Bit Width
bool	logical value	false	8, 16, 32, 64	impl. dep.
int	signed integer	0	8, 16, 32, 64	64
uint	unsigned integer	0	8, 16, 32, 64	64
real	real floating point	0.0	32, 64	64
imag	imaginary floating point	0.0i	32, 64	64
complex	complex floating points	0.0 + 0.0i	64, 128	128
string	character string	""	N/A	N/A

- **Syntax**

```
primitive-type:
  type-name [( bit-width )]
```

- **Examples**

```
int(16)   // 16-bit int
real(32)  // 32-bit real
uint      // 64-bit uint
```

Enum Types

- **A lot like enum types in C:**

```
enum color {red, green, blue};    // can also be assigned values
```

  - But can also be printed!

```
var myColor = color.red;
writeln(myColor);                 // prints 'red'
```

  - And support built-in iterators and queries:

```
for c in color do …
…color.size…
```

- **By default, must be fully-qualified to avoid conflicts:**

```
var myColor = red;                // error by default
```

  - But, may be 'use'd like modules to avoid qualifying

```
use color;                        // can use standard filters, renaming, etc.
var myColor = red;                // OK!
```

Type Aliases and Casts

- **Basic Syntax**

```
type-alias-declaration:
  type identifier = type-expr;

cast-expr:
  expr : type-expr
```

- **Semantics**
  - type aliases are simply symbolic names for types
  - casts are supported between any primitive types
- **Examples**

```
type elementType = complex(64);

5:int(8)   // store value as int(8) rather than int
"54":int          // convert string to an int
249:elementType   // convert int to complex(64)
```

Basic Operators and Precedence

Operator	Description	Associativity	Overloadable
`:`	cast	`left`	no
`**`	exponentiation	`right`	yes
`! ~`	logical and bitwise negation	`right`	yes
`* / %`	multiplication, division and modulus	`left`	yes
unary `+ -`	positive identity and negation	`right`	yes
`+ -`	addition and subtraction	`left`	yes
`<< >>`	shift left and shift right	`left`	yes
`<= >= < >`	ordered comparison	`left`	yes
`== !=`	equality comparison	`left`	yes
`&`	bitwise/logical and	`left`	yes
`^`	bitwise/logical xor	`left`	yes
`|`	bitwise/logical or	`left`	yes
`&&`	short-circuiting logical and	`left`	via `isTrue`
`||`	short-circuiting logical or	`left`	via `isTrue`

Control Flow: Braces vs. Keywords

Control flow statements specify bodies using curly brackets (compound statements)

- Conditional statements

```
if cond { computeA(); } else { computeB(); }
```

- While loops

```
while cond {
  compute();
}
```

- For loops

```
for indices in iteratable-expr {
  compute();
}
```

- Select statements

```
select key {
  when value1 { compute1(); }
  when value2 { compute2(); }
  otherwise   { compute3(); }
}
```

Control Flow: Braces vs. Keywords

They also support keyword-based forms for single-statement cases

- Conditional statements

```
if cond then computeA(); else computeB();
```

- While loops

```
while cond do
  compute();
```

- For loops

```
for indices in iteratable-expr do
  compute();
```

- Select statements

```
select key {
  when value1 do compute1();
  when value2 do compute2();
  otherwise   do compute3();
}
```

Control Flow: Braces vs. Keywords

Of course, since compound statements are single statements, the two forms can be mixed…

- Conditional statements

```
if cond then { computeA(); } else { computeB(); }
```

- While loops

```
while cond do {
  compute();
}
```

- For loops

```
for indices in iteratable-expr do {
  compute();
}
```

- Select statements

```
select key {
  when value1 do { compute1(); }
  when value2 do { compute2(); }
  otherwise   do { compute3(); }
}
```

Procedures and iterator features

- **pass by keyword (argument name)**

```
proc foo(name, age) { … }
foo(age=32, name="Tim");
```

- **default argument values**

```
proc foo(name, age=18) { … }
foo(name="Tim");
```

- **formal type queries**

```
proc foo(x: ?t, y: [?D] t) { … }
proc bar(x: int(?w)) { … }
```

- **overloading**
  - including where clauses to filter overloads

```
proc foo(x: int(?w), y: int(?w2)) where w = 2*w2 { … }
proc foo(x: int(?w), y: int(?w2)) { … }
proc foo(x, y) { … }
```

Methods

- **Methods are like procedures with an implicit**
  - Chapel supports both *primary methods:*

```
class circle {
  proc area() { return pi*radius**2; }
}
```

  - and *secondary methods*:

```
proc circle.circumference() {
  return 2*pi*radius;
}
var myCircle = new circle(radius=1.0);
writeln((myCircle.area(), myCircle.circumference());
```

  - Moreover, secondary methods can be defined for any type:

```
proc int.square() {
  return this**2;
}
writeln(5.square());    // prints 25
```

Paren-less procedures

Procedures without arguments don't need parenthesis

```
proc circle.diameter {
  return 2*radius;
}

writeln(c1.radius, " ", c1.diameter);
```

Support time/space tradeoffs without code changes

- Store value with variable/field?
- Or compute on-the-fly with paren-less procedure/method?
  - Like fields, such methods don't dispatch dynamically

Function Calls vs. Array Accesses

- **Chapel doesn't distinguish between call and array access**
  - An "array access" is simply a call to a special method named "this()"

```
class circle {
  proc this(x: int, y: real) {
    // do whatever we want here…
  }
}
myCircle[2, 4.2];    // calls circle.this()
```

  - Related: parens/square brackets can be used for either case:

    `A[i,j]` or `A(i,j)`   *// these are both accesses to array A*

    `foo()` or `foo[]`   *// these are both function calls to foo()*

  - By convention, we tend to use [] for arrays and () for function calls
    - but Fortran programmers may be happy to get to use () for arrays…?
  - Like paren-less methods, view this as another time-space tradeoff
    - can implement something as a function or as an array
    - since Chapel's arrays are quite rich, access is not necessarily O(1) anyway

Default object iterators

- **Objects can support default iterators**

```
class circle {
  iter these() {
    // yield whatever we want…
  }
}
for items in myCircle do …   // invokes circle.these()
```

  - Similar to the 'this()' default accessor
  - Overloads can support parallel or parallel zippered iteration
    - (true for any iterator)

Generic Procedures/Methods

Generic procedures can be defined using type and param arguments:

```
proc foo(type t, x: t) { … }
proc bar(param bitWidth, x: int(bitWidth)) { … }
```

Or by simply omitting an argument type (or type part):

```
proc goo(x, y) { … }
proc sort(A: []) { … }
```

Generic procedures are instantiated for each unique argument signature:

```
foo(int, 3);          // creates foo(x:int)
foo(string, "hi");    // creates foo(x:string)
goo(4, 2.2);          // creates goo(x:int, y:real)
```

Generic Objects

- Generic objects can be defined using type and param fields:

```
record Table { param size: int; var data: size*int; }
record Matrix { type eltType; … }
```

- Or by simply eliding a field type (or type part):

```
class Triple { var x, y, z; }
```

- Generic objects are instantiated for each unique type signature:

```
// instantiates Table, storing data as a 10-tuple
var myT: Table(10);
// instantiates Triple as x:int, y:int, z:real
var my3: Triple(int, int, real) = new Triple(1, 2, 3.0);
```

Modules

- Syntax

```
module-def:
  module identifier { code }

module-use:
  use module-identifier;
```

- Semantics
  - all Chapel code is stored in modules
  - `use`-ing a module makes its symbols visible in that scope
  - module-level statements are executed at program startup
    - typically used to initialize the module
  - for convenience, a file containing code outside of a module declaration creates a module with the file's name

Use Statement: Import Control

- **Use statements support import control**
  - 'except' keyword prevents unqualified access to symbols in list

    `use M except bar;` *// All of M's symbols other than bar can be named directly*
  - 'only' keyword limits unqualified access to symbols in list

    `use M only foo;` *// Only M's foo can be named directly*
  - Permits user to avoid importing unnecessary symbols
    - Including symbols which cause conflicts

```
module myMod {
  var bar = true;

  proc myFunc() {
    use M only foo;
    foo();
    var a = bar;    // Now finds myMod.bar, rather than M.bar
  }
}
```

```
module M {
  var bar = 13;
  proc foo() { … }
}
```

Use Statement: Symbol Renaming

- **Use'd symbols can also be renamed:**

```
use M only bar as barM;
```

  - Allows users to avoid…

    …naming conflicts between multiple used modules

    …shadowing outer variables with same name

  …while still making that symbol available for access

```
module M {
  var bar = 13;
  proc foo() { … }
}
```

```
module myMod {
  var bar = true;

  proc myFunc() {
    use M only foo, bar as barM;
    foo();
    var a = bar;     // Still finds myMod.bar, rather than M.bar
    var b = barM;    // refers to M.bar
} }
```

Modules: Public/Private Declarations

- **All module-level symbols are public by default**

```
proc foo() { … }     // public, since not decorated
```

- **module-level symbols can be declared public/private:**

```
private var bar = …;
public proc baz() { … }
```

- **Can be used in declarations of:**
  - Modules
  - Vars, consts, and params
    - including configs
  - Procedures and iterators

- **Future work: extend to other symbols**
  - particularly types / object members

Program Entry Point: main()

- **Semantics**
  - Chapel programs start by:
    - initializing all modules
    - executing main(), if it exists

```
M1.chpl:
use M2;
writeln("Initializing M1");
proc main() { writeln("Running M1"); }
```

```
M2.chpl:
module M2 {
  writeln("Initializing M2");
}
```

```
% chpl M1.chpl M2.chpl

% ./a.out

Initializing M2
Initializing M1
Running M1
```

Argument and Return Intents

- **Arguments can optionally be given intents**
  - (blank): varies with type; follows principle of least surprise
    - most types: `const`
    - arrays, domains, sync vars, atomics: passed by reference
  - `in`: copies actual into formal; permits changes
  - `out`: copies formal into actual at procedure return
  - `inout`: does both of the above
  - `ref`: pass by reference
  - `const [ref | in]`: disallows modification of the formal
  - `param`/`type`: formal must be a param/type (evaluated at compile-time)
- **Return types can also have intents**
  - (blank)/`const`: cannot be modified (without assigning to a variable)
  - `ref`: permits modification back at the callsite
  - `type`: returns a type (evaluated at compile-time)
  - `param`: returns a param value (evaluated at compile-time)

Other Base Language Features not covered here

- **Interoperability with external code**
- **Compile-time features for meta-programming**
  - type/param procedures
  - folded conditionals
  - unrolled for loops
  - user-defined compile-time warnings and errors
- **Type select statements, argument type queries**
- **Unions**

Legal Disclaimer

Information in this document is provided in connection with Cray Inc. products. No license, express or implied, to any intellectual property rights is granted by this document.

Cray Inc. may make changes to specifications and product descriptions at any time, without notice.

All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

Cray hardware and software products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Cray uses codenames internally to identify products that are in development and not yet publically announced for release. Customers and other third parties are not authorized by Cray Inc. to use codenames in advertising, promotion or marketing and any use of Cray Inc. internal codenames is at the sole risk of the user.

Performance tests and ratings are measured using specific systems and/or components and reflect the approximate performance of Cray Inc. products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, and URIKA. The following are trademarks of Cray Inc.: ACE, APPRENTICE2, CHAPEL, CLUSTER CONNECT, CRAYPAT, CRAYPORT, ECOPHLEX, LIBSCI, NODEKARE, THREADSTORM. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT, and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used in this document are the property of their respective owners.