Base Language

Outline: High-Level Feature Survey

- **Running example: naïve n-body computation**
  - Simple declarations
  - Records and Classes
  - Tuples
  - Arrays
  - Ranges: Integer Sequences
  - Basic Serial Control Flow
  - Subroutines: Procedures and Iterators
  - Reference Variables

Outline: Additional Details

- Type Aliases and Casts
- Enums
- Modules and Use Statements
- More on Procedures, Iterators, and Methods
- Generics and Compile-Time Computation
- Special Subroutine Forms
- Initializers for Records and Classes
- Class Instance Memory Management
- Error-Handling
- Error-Handling and Parallelism
- Defer Statements
- Interoperation

Running Example: naïve n-body computation

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**

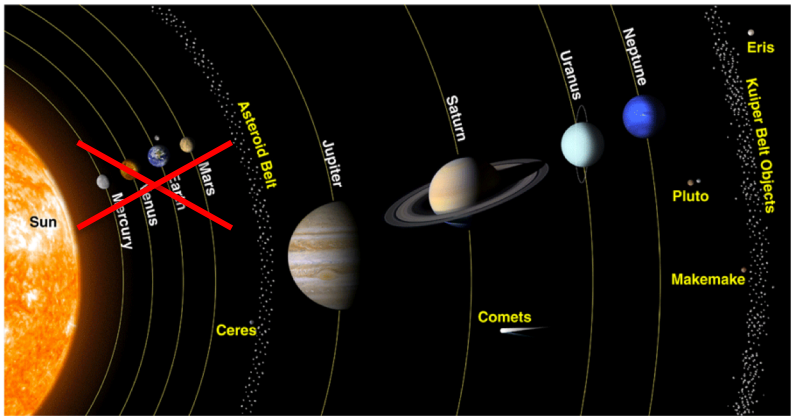


Image source: http://spaceplace.nasa.gov/review/ice-dwarf/solar-system-lrg.png

Simple Declarations

5-body in Chapel: Declarations

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

Variable declarations

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
```

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
```

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)
```

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
`<< >>`	shift left and shift right	left	yes
`&`	bitwise/logical and	left	yes
`^`	bitwise/logical xor	left	yes
`|`	bitwise/logical or	left	yes
`+ -`	addition and subtraction	left	yes
`<= >= < >`	ordered comparison	left	yes
`== !=`	equality comparison	left	yes
`&&`	short-circuiting logical and	left	via `isTrue`
`||`	short-circuiting logical or	left	via `isTrue`

5-body in Chapel: Declarations

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

Variable declarations

5-body in Chapel: Declarations

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

config const numsteps = 10000;
```

Configuration Variable

5-body in Chapel: Declarations

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

config const numsteps = 10000;
```

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
$ ./myProgram--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;
```

Configuration Variable

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;
}


…
```

Record declaration

Records and Classes

Records and Classes

- **Chapel's 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)

- **Example**

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

```
var c1: circle; // default-initialized
c1 = new circle(radius=1.0);
var c2 = c1;     // copies c1
c1.radius = 5.0;
writeln(c2.radius);   // prints 1.0
```

Records and Classes

- **Chapel's 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)

- **Example**

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

```
var c1: circle; // initially nil
c1 = new circle(radius=1.0);
var c2 = c1;     // aliases c1's circle
c1.radius = 5.0;
writeln(c2.radius);   // prints 5.0
```

Classes vs. Records

Classes

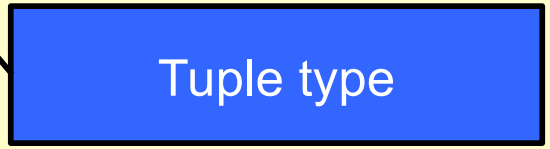
- **heap-allocated**
  - Variables point to objects
  - Objects could be anywhere
  - Support mem. mgmt. policies
- **‘reference' semantics**
  - compiler will only copy pointers
- **support inheritance**
- **support dynamic dispatch**
- **identity matters most**
- **similar to Java classes**

Records

- **allocated in-place**
  - Variables are the objects
  - Objects are “right here”
  - Always freed at end of scope
- **'value' semantics**
  - compiler may introduce copies
- **no inheritance**
- **no dynamic dispatch**
- **value matters most**
- **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;
}


…
```



Tuples

Tuples

- **Use**
  - support lightweight grouping of values
    - e.g., passing/returning multiple procedure arguments at once
    - short vectors
    - multidimensional array indices
  - 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(…)
    ]
```

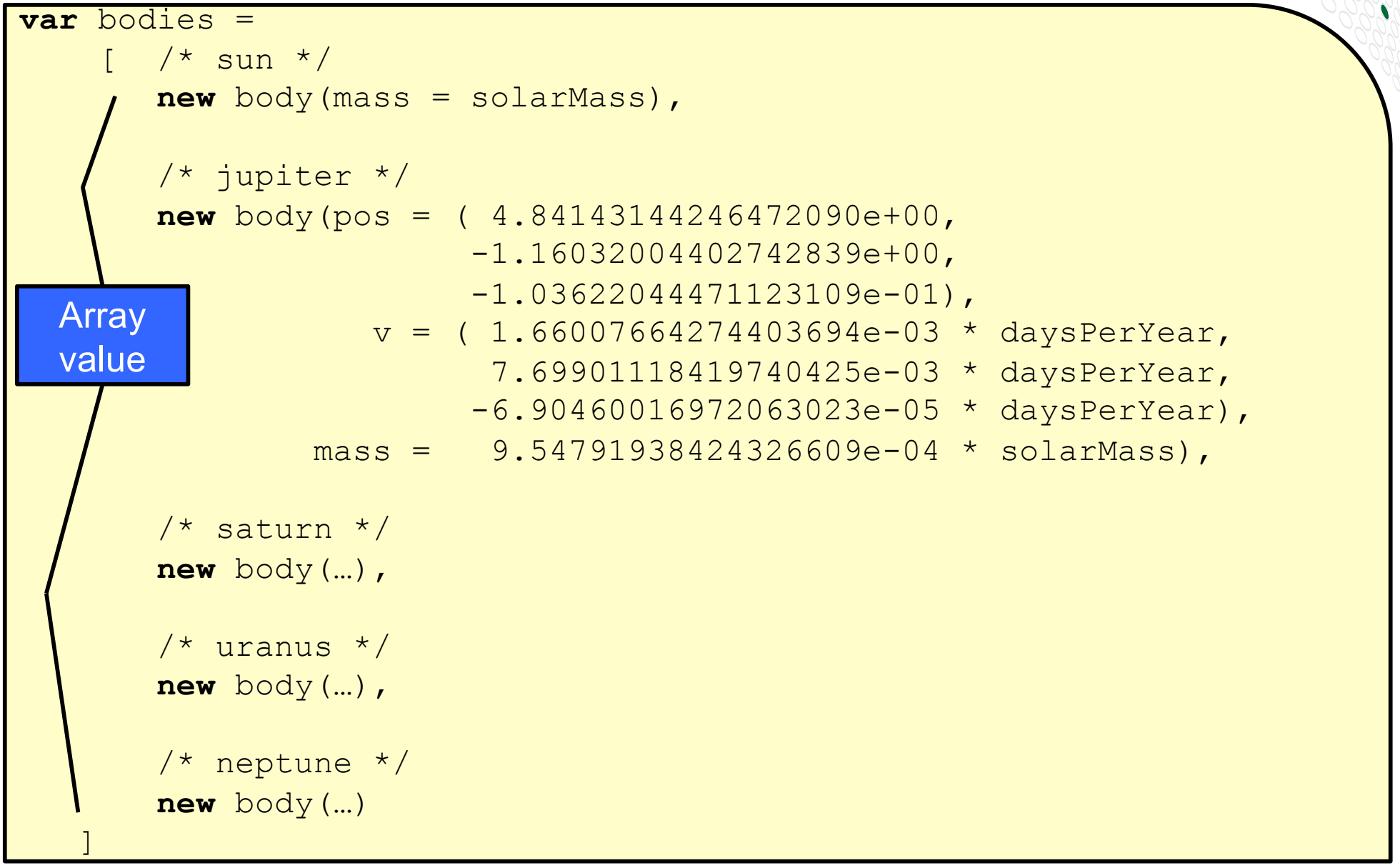
Create a record object

5-body in Chapel: the Bodies

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

       /* jupiter */
       new body(pos = ( 4.8414314424647209e+00,
                       -1.1603200440274284e+00,
                       -1.0362204447112310e-01),
                  v = ( 1.6600766427440369e-03 * daysPerYear,
                        7.6990111841974042e-03 * daysPerYear,
                       -6.9046001697206302e-05 * daysPerYear),
               mass =   9.5479193842432661e-04 * solarMass),

       /* saturn */
       new body(…),

       /* uranus */
       new body(…),

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

Tuple values

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(…)
    ]
```

Arrays

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(…)
    ]
```

Create a record object

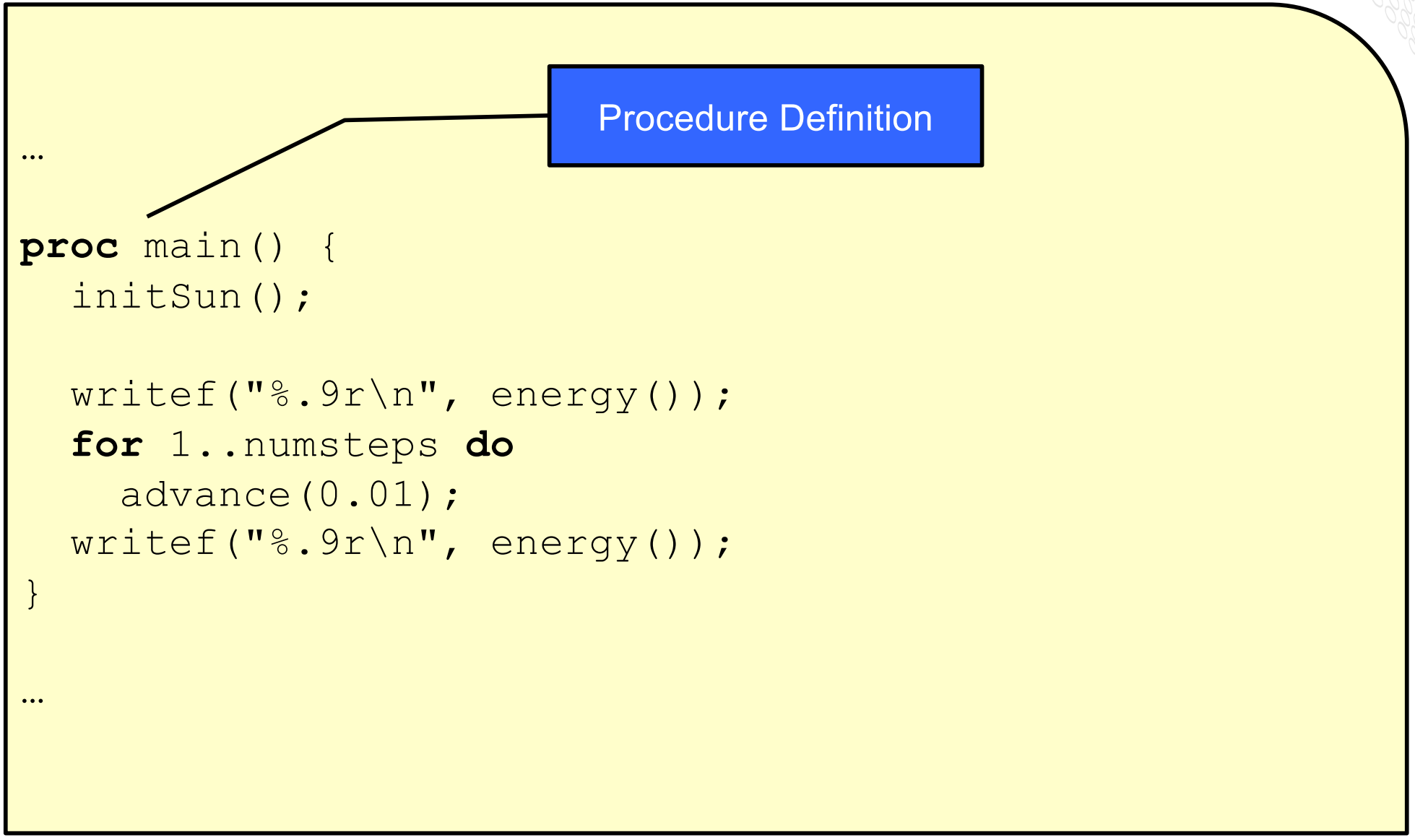
Tuple values

Array value

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()



5-body in Chapel: main()

```
…

proc main() {
  initSun();

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

…
```

Procedure Call

5-body in Chapel: main()

```
…

proc main() {
  initSun();

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

…
```

Formatted I/O

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

Ranges: Integer Sequences

Range Values

- **Syntax**

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

- **Definition**
  - 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());
}

…
```

Serial for loop

Basic Serial Control Flow

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, iterator, iterable object, …

- **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
```

Control Flow: Other Forms

- 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

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

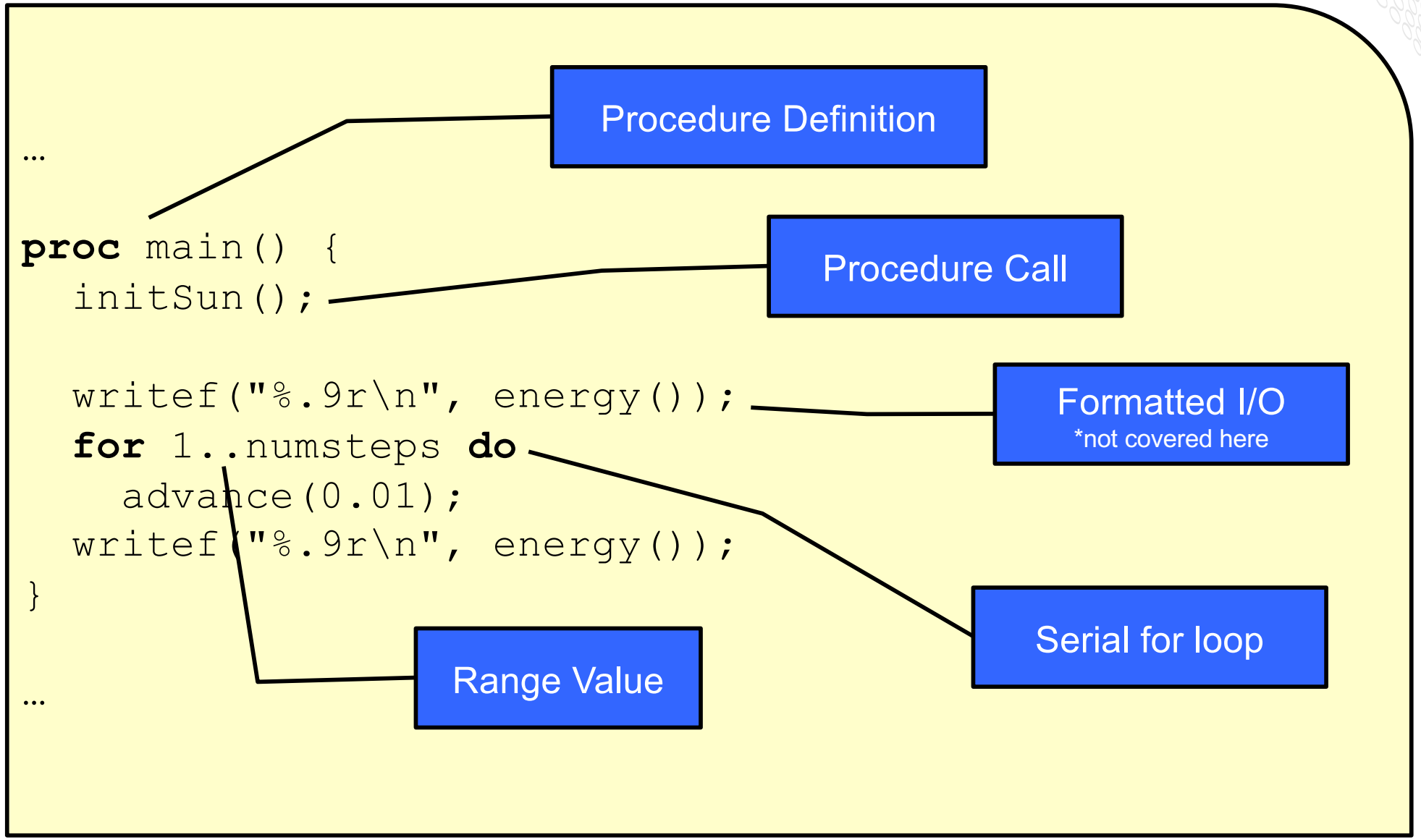
```
for indices in iteratable-expr do {
  compute();
}
```

- Select statements

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

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: 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;
}
```

$$m_1\mathbf{a}_1 = \frac{Gm_1m_2}{r_{12}^3}(\mathbf{r}_2 - \mathbf{r}_1) \quad \text{Sun-Earth}$$

$$m_2\mathbf{a}_2 = \frac{Gm_1m_2}{r_{21}^3}(\mathbf{r}_1 - \mathbf{r}_2) \quad \text{Earth-Sun}$$

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

Subroutines: Procedures and Iterators

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)
```

Argument Intents

- **Arguments can optionally be given intents**
  - (blank): varies with type; follows principle of least surprise
    - most types: `const in` or `const ref`
    - arrays, sync/single vars, atomics: `ref`
  - `in`: initializes formal using actual; permits formal to be modified
  - `out`: copies formal into actual at procedure return
  - `inout`: does both of the above
  - `ref`: formal is a reference back to the actual
  - `const` [`ref` | `in`]: disallows modification of the formal
  - `param`/`type`: actual must be a param/type

Argument Intents, by Example

- Arguments can optionally be given intents

```
proc foo(x: real, y: [] real) {
  // x = 1.2;   // illegal: scalars are passed 'const in' by default
  y = 3.4;   // OK: arrays are passed 'ref' by default
}

var r: real,
    A: [1..3] real;

foo(r, A);

writeln((r, A));   // writes (0.0, [3.4, 3.4, 3.4])
```

Argument Intents, by Example

- Arguments can optionally be given intents

```
proc foo(in x: real, in y: [] real) {
  x = 1.2;   // OK: local copy is modified
  y = 3.4;   // OK: local copy is modified
}

var r: real,
    A: [1..3] real;

foo(r, A);

writeln((r, A));   // writes (0.0, [0.0, 0.0, 0.0])
```

Argument Intents, by Example

- Arguments can optionally be given intents

```
proc foo(out x: real, out y: [] real) {
  x = 1.2;   // OK: local copy is modified
  y = 3.4;   // OK: local copy is modified
}

var r: real,
    A: [1..3] real;

foo(r, A);

writeln((r, A));   // writes (1.2, [3.4, 3.4, 3.4])
```

Argument Intents, by Example

- Arguments can optionally be given intents

```
proc foo(inout x: real, inout y: [] real) {
  x = 1.2;   // OK: local copy is modified
  y = 3.4;   // OK: local copy is modified
}

var r: real,
    A: [1..3] real;

foo(r, A);

writeln((r, A));   // writes (1.2, [3.4, 3.4, 3.4])
```

Argument Intents, by Example

- Arguments can optionally be given intents

```
proc foo(ref x: real, ref y: [] real) {
  x = 1.2;   // OK: actual is modified
  y = 3.4;   // OK: actual is modified
}

var r: real,
    A: [1..3] real;

foo(r, A);

writeln((r, A));   // writes (1.2, [3.4, 3.4, 3.4])
```

Argument Intents, by Example

- Arguments can optionally be given intents

```
proc foo(ref x: real, ref y: [] real) {
  x = 1.2;   // OK: actual is modified
  y = 3.4;   // OK: actual is modified
}

const r: real,
      A: [1..3] real;

// foo(r, A);   // illegal, can't pass references to constants

writeln((r, A));   // writes (0.0, [0.0, 0.0, 0.0])
```

Argument Intents, by Example

- Arguments can optionally be given intents

```
proc foo(const ref x: real, const ref y: [] real) {
  // x = 1.2;   // illegal: can't modify constant arguments
  // y = 3.4;   // illegal: can't modify constant arguments
}

const r: real,
      A: [1..3] real;

foo(r, A);   // OK to create constant references to constants

writeln((r, A));   // writes (0.0, [0.0, 0.0, 0.0])
```

Argument Intents, by Example

- Arguments can optionally be given intents

```
proc foo(param x: real, type t) {
  …
  …
}

const r: real,
      A: [1..3] real;

// foo(r, A);   // illegal: can't pass vars and consts to params and types

writeln((r, A));   // writes (0.0, [0.0, 0.0, 0.0])
```

Argument Intents, by Example

- Arguments can optionally be given intents

```
proc foo(param x: real, type t) {
  …
  …
}

const r: real,
      A: [1..3] real;

foo(1.2, r.type);   // OK: passing a literal/param and a type

writeln((r, A));    // writes (0.0, [0.0, 0.0, 0.0])
```

5-body in Chapel: advance()

```
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: Alternative Using Iterators

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

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

Use of iterator

Definition of iterator

5-body in Chapel: advance() Using Iterators

```
proc advance(dt) {
  for (i,j) in triangle(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: Alternative Using References

```
proc advance(dt) {
  for (i,j) in triangle(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 Variables

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 general pointer: no way to point ‘ident’ to something else
  - 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
```

This is a Good Stopping Point for Now

Type Aliases and Casts

Type Aliases and Casts

- **Basic Syntax**

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

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

- **Description**
  - 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 an int(8) rather than int
"54": int          // convert the string to an int
249: elementType  // convert the int to a complex(64)
```

Recall: Config Types

```
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
$ ./myProgram--start=2 --epsilon=0.00001
```

Enums

Enum Types

- **Somewhat like enum types in C:**

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

  - Yet purer: don't coerce to integers, don't have default int values
  - Can also be printed!

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

  - 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 to avoid qualifying (like modules)

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

Modules and Use Statements

Modules

- **Syntax**

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

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

- **Description**
  - 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 any module declaration creates a module with the file's name

Hello World in Chapel: Rapid Prototype

- **hello.chpl:**
  - rapid prototyping version:

```
writeln(”Hello, world!");
```

    - defines an implicit module “hello”
    - writeln() is its initialization

Program Entry Point: main()

- **Definition**
  - 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

% ./M1

Initializing M2
Initializing M1
Running M1
```

Hello World in Chapel: Production-Grade

- **hello.chpl:**
  - production-grade version:

```
module hello {
  proc main() {
    writeln(”Hello, world!");
  }
}
```

    - defines explicit hello module (with no module initialization code)
    - with explicit main() procedure

Module Deinitialization

- Modules also support deinit() routines to help clean up:

```
module hello {
  proc deinit() {
    writeln(”Goodbye, cruel world!");
  }
}
```

Use Statement: Basic Use

- Use statements make a module's symbols available

```
module myMod {
  var bar = true;

  proc myFunc() {
    use M;
    foo();

  }
}
```

```
module M {

 proc foo() { … }
}
```

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
    - e.g., 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 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
} }
```

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

Use Statement: Fully Qualified References

- **Module symbols can also be fully qualified:**

  …M.bar…

  - Supports explicit naming—more verbose, but more precise

```
module myMod {
  var bar = true;

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

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

Modules: Public/Private Declarations

- **All module-level symbols are (currently) 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
  - Variables, constants, and params
    - including configs
  - Procedures and iterators

More on Procedures, Iterators, and Methods

Procedure 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("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 'this' argument**
  - 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
```

Method Overrides

- **Subclasses may override superclass methods**
  - but they must say so to avoid common error cases

```
class C {
  proc foo() { … }
  proc bar(x: int) { … }
  proc baz(x: int) { … }
  proc bax(x: int) { … }
}

class D: C {
  override proc foo() { … }
  override proc bar(x: int) { … }
  proc baz(x: real) { … }   // the differing type makes this is an overload
  proc bax(y: int) { … }    // the differing name makes this an overload
}
```

Method Overrides

- **Compiler checks override errors**
  - to avoid common error cases

```
class C {
  proc foo() { … }
  proc bar(x: int) { … }
  proc baz(x: int) { … }
  proc bax(x: int) { … }
}

class D: C {
  proc foo() { … }                          // error: overrides but fails to say so
  proc bar(x: int) { … }                    // error: overrides but fails to say so
  override proc baz(x: real) { … }          // error: no matching parent method
  override proc bax(y: int) { … }           // error: no matching parent method
}
```

Generics and Compile-Time Computation

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);
```

Argument and Return Intents

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

Ref return intents

- Procedures returning refs can be used in LHS contexts:

```
var A: [1..100] real;

proc getRandElt() ref {
  const randIdx = getRandIdx(A.domain);
  return A[randIdx];
}

getRandElt() = 4.2;
```

Compile-Time Computation

Type / Param Return Intents

- **Procedures can return types and params**
  - Results in evaluation of procedure at compile-time

```
proc getNTupleSize(param n: int, type eltType) param {
  return n*numBits(eltType);
}

proc getEltType(param useReal: bool, param size: int) type {
  if useReal then
    return real(size);
  else
    return imag(size);
}

param size = getNTupleSize(3, int(8));    // returns 24
var A: [1..100] getEltType(false, 32);    // an array of imag(32)
```

Folding conditionals

- Conditionals with ‘param’ expressions are folded

```
config param debug = false;
if debug then
  writeln("[debug] x is: ", x); // folded away unless debug == true
```

Unrolling Loops

- **Loops with param index variables are unrolled**
  - Currently only supported for ranges with param bounds:

```
for param i in 1..5 do
  if (i%2 == 0) then
    foo(i-1);
  else
    foo(2*i);
```

```
// equivalent to:
foo(2);
foo(1);
foo(6);
foo(3);
foo(10);
```

```
var tup = (1, 2.0, "three");

for e in tup do …e…                  // illegal since 'i' has no well-defined type

for i in 1..tup.size do
  const e = tup(i);                  // illegal since 'e' has no well-defined type

for param i in 1..tup.size do
  const e = tup(i);                  // legal since the loop is unrolled
```

User-generated compile-time messages

- **Permit user to generate errors/warnings at compile-time**
  - Messages printed if compiler resolves routine

```
proc foo(x: int(?w)) {
  if (w > 64) then
    compilerError("foo() called with an unusually large int");
  if (w == 8) then
    compilerWarning("foo() performs poorly for int(8) args");
  …
}
```

Special Subroutine Forms

Paren-less procedures

Procedures without arguments don't need parentheses

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

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

Supports time/space tradeoffs without code changes

- Store value with variable / field?
- Or compute on-the-fly with paren-less procedure / method?

Note: Like fields, such methods don't dispatch dynamically

Subroutine Calls vs. Array Accesses

- **Chapel doesn't distinguish between calls & array accesses**
  - 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 myCircle.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 / calls to A.these()
foo() or foo[]      // these are both calls to subroutine foo()
```

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

Default object iterators

- **Objects can have default iterators**

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

  - Similar to the 'this()' default accessor
  - Invoked using for-loops

```
for i in myCircle do …   // equivalent to for i in myCircle.these()
```

  - Overloads can support forall-based parallel / parallel zippered iteration
    - (true for any iterator)

Initializers for Records and Classes

Default Initializers

- Compiler provides default initializer for classes / records:

```
record R {
  var r: real;
  var i = 42;
}

var myR1 = new R(3.14);   // r=3.14, i=42
var myR2 = new R(i=33);   // r=0.0,  i=33
```

- Effectively equivalent to:

```
proc R.init(r: real = 0.0, i: int = 42) {
  this.r = r;
  this.i = i;
}
```

User-Defined Initializers

- **Users can also write their own initializers:**
  - Doing so overrides the default initializer

```
record R {
  var r: real;
  var i = 42;

  proc init() {
    this.r = genRandomNumber();
    // compiler fills in this.i = 42;
  }
}

var myR1 = new R(),          // r = <something random>, i = 42
    myR2 = new R(i=33);      // error!  No matching init() signature
```

Post-initializers

- **Users can also / alternatively write postinit() routines**
  - A hook that's invoked after object initialization

```
record R {
  var r: real;
  var i = 42;

  proc postinit() {
    writeln("Created an R!");
  }
}

var myR1 = new R(3.14);   // r=3.14, i=42 and prints "Created an R!"
```

  - Supports leveraging the default initializer with customization

Deinitializers

- **Records and classes also support deinitializers**
  - invoked when instance is freed

```
record R {
  var x, y, z: int;
  proc deinit() {
    writeln("R is going away now!");
  }
}

{
  var myR: R;
  …
}   // myR.deinit() will be called here, printing its message
```

Initializers and Class Hierarchies

- Initializers are a bit more interesting for class hierarchies

```
class C { var a: real; proc foo(…) { … } }
class D: C { var x, y, z: int; override proc foo(…) { … } }
class E: D { var …; override proc foo(…) { … }}

proc D.init() {
  writeln("In D's initializer");
  super.init(a=3.14);   // makes 'this' into a legal C object
  this.foo();           // calls C.foo()
  // compiler notes that x was skipped, inserts this.x = 0;
  this.y = 42;
  // compiler notes that z was skipped, inserts this.z = 0;
  this.complete();      // make 'this' into a legal D object
  this.foo();           // calls D.foo()
}

proc D.postinit() { this.foo(); }   // calls E.foo() for …new E(…)…
```

Initializers and Class Hierarchies

- For simplicity, the compiler can take care of many details

```
class C { var a: real; proc foo(…) { … } }
class D: C { var x, y, z: int; override proc foo(…) { … } }
class E: D { var …; override proc foo(…) { … }}

proc D.init() {
  // compiler notes there's no super.init(), so inserts it
  // compiler notes that x was skipped, inserts this.x = 0;
  this.y = 42;
  // compiler notes that z was skipped, inserts this.z = 0;
  // compiler notes there's no this.complete(), so inserts it
}

proc D.postinit() {
  // compiler notes there's no super.postinit(), so inserts it
  this.foo();
}
```

Class Instance Memory Management

Classes vs. Records

Classes

- **heap-allocated**
  - Variables point to objects
  - Objects could be anywhere
  - Support mem. mgmt. policies
- **‘reference' semantics**
  - compiler will only copy pointers
- **support inheritance**
- **support dynamic dispatch**
- **identity matters most**
- **similar to Java classes**

Records

- **allocated in-place**
  - Variables are the objects
  - Objects are “right here”
  - Always freed at end of scope
- **'value' semantics**
  - compiler may introduce copies
- **no inheritance**
- **no dynamic dispatch**
- **value matters most**
- **similar to C++ structs**
  - (sans pointers)

Memory Management Strategies Scorecard

Garbage Collection (like Java)	Manual 'delete's (like traditional C++)
+ safety guarantees + eliminates memory leaks + eliminates double-delete + eliminates use-after-free	– more errors possible – failure to delete results in leaks – double-delete possible – use-after-free possible
+ ease-of-use + no need to write 'delete'	– more burden on programmer – think about 'delete'
– implementation challenges due to distributed memory & parallelism	+ simpler implementation
– performance challenges – stop-the-world interrupts program – concurrent collectors add overhead – scalability may prove difficult	+ predictable, scalable performance

What about Rust?

- **Rust's approach prevents memory errors at compile time**
  - programs that might have a use-after-free result in compilation error
  - its *borrow checker* is the component raising these errors
- **Rust's approach also prevents race conditions**
  - since race conditions can introduce memory errors
- **Rust programmers can also opt out and write *unsafe* code**

Motivating Question

- **Can Chapel include something Rust-like?**
  - compile-time detection of use-after-free?
- **The Big Issue: Complete Checking and Race Conditions**
  - recall that a race condition can introduce a use-after-free error
  - For example:

```
proc test() {
  var myOwned = new Owned(new MyClass());
  var b = myOwned.borrow();
  cobegin with (ref myOwned) {
    { myOwned.clear(); }    // deletes instance
    { writeln(b); }         // races to use instance before delete
  }
}
```

Complete Checking and Race Conditions

- **Should Chapel rule out race conditions at compile time?**
- **A worthy goal, but the Rust strategy doesn't fit Chapel**
  - only one mutable reference to an object can exist at a time
  - if a mutable reference exists, no const references to that object
- **Such a strategy in Chapel would make these illegal:**

```
forall a in A { a = 1; }
forall i in 1..n { A[i] = i; }
forall i in 1..n { B[permutation(i)] = A[i]; }
```

- **Could a different strategy detect these race conditions?**
  - Maybe, but it would be difficult
  - Can the compiler prove that 'permutation' is a permutation?
  - If not, how would that be communicated to the compiler?

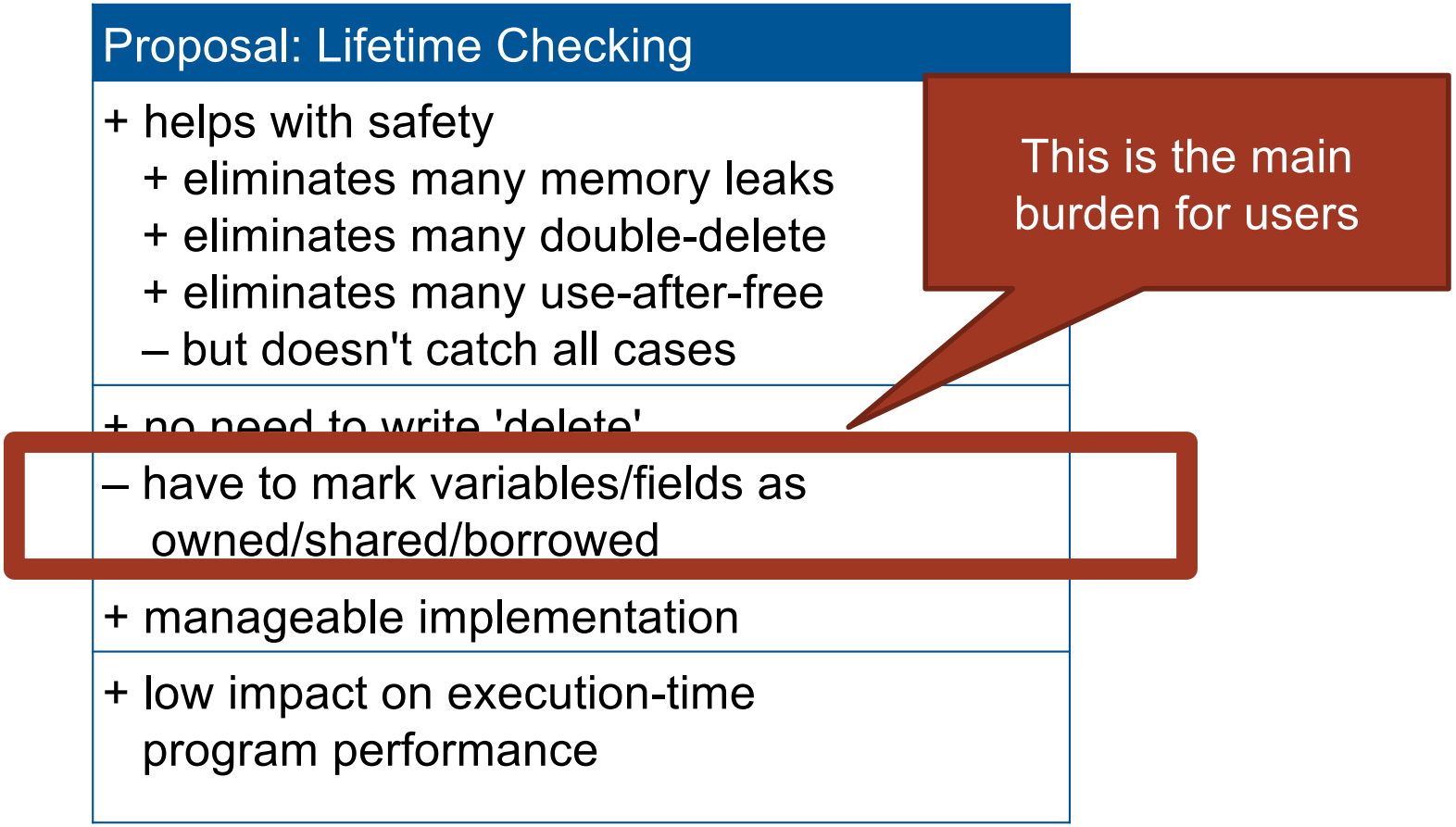
Chapel's Approach

- **Add incomplete compile-time checking to gain some of the benefits of garbage collection**

Proposal: Lifetime Checking
+ helps with safety + eliminates many memory leaks + eliminates many double-delete + eliminates many use-after-free – but doesn't catch all cases
+ no need to write 'delete' – have to mark variables/fields as owned/shared/borrowed
+ manageable implementation
+ low impact on execution-time program performance

General Goal

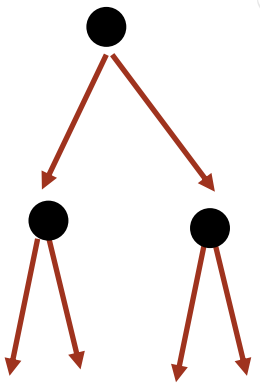
- **Add incomplete compile-time checking to gain some of the benefits of garbage collection**



Flavors of Class Instances

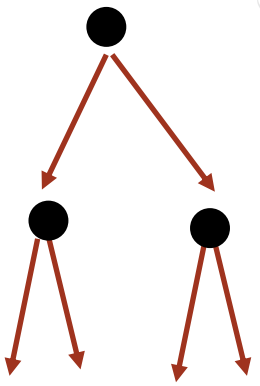
keyword	meaning
unmanaged	the instance is manually managed and needs to be deleted by the user
owned	the instance is auto-deleted at end of scope unless ownership is transferred
shared	the instance is reference counted
borrowed	the instance is managed elsewhere; this reference does not impact its lifetime

Mini Binary Trees: unmanaged version



```
class Tree {
  const left, right: unmanaged Tree;
}

proc Tree.init(const depth: int) {
  if depth >= 1 {
    this.left = new unmanaged Tree(depth-1);
    this.right = new unmanaged Tree(depth-1);
  }
}

proc Tree.deinit() {
  delete left, right; // T's subtrees must be explicitly deleted to avoid leaks
}

const T = new unmanaged Tree(2);
delete T;                    // T must be explicitly deleted or it will be leaked
```

Mini Binary Trees: shared version



```
class Tree {
  const left, right: shared Tree;
}

proc Tree.init(const depth: int) {
  if depth >= 1 {
    this.left = new shared Tree(depth-1);
    this.right = new shared Tree(depth-1);
  }
}


{
  const T = new shared Tree(2);
} // when T's scope ends, if nobody else points to it, it's deleted
  // and so are any of its subtrees that nobody else is pointing to
```

Shared and Sharing

- Multiple 'shared C' variables can point to the same instance
- Assigning or copy-initializing results in *sharing*

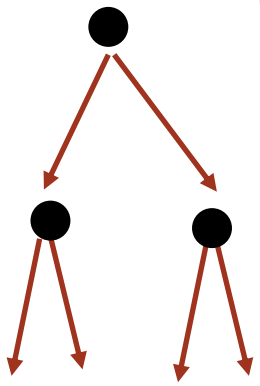
```
var otherShared = myShared;
// now otherShared and myShared point to the same instance
// the instance will be deleted when all references to the shared object go out of scope
```

- Default-intent 'shared' arguments also result in *sharing*
- 'shared' can be assigned 'nil' to release a reference

```
var x = new shared C();
x = nil;  // deletes the previous instance if no other shared variable refers to it
```

- Other methods are available, see '**shared**' docs

Mini Binary Trees: owned version



```
class Tree {
  const left, right: owned Tree;
}

proc Tree.init(const depth: int) {
  if depth >= 1 {
    this.left = new owned Tree(depth-1);
    this.right = new owned Tree(depth-1);
  }
}


{
  const T = new owned Tree(2);
} // when T's scope ends, if nobody else has taken ownership, it's deleted
  // and so are its subtrees
```

Owned and Ownership Transfer

- **Only one 'owned C' can point to a given instance**
  - thus, it can always destroy the instance when it goes out of scope
- **Assigning or copy-initializing results in *ownership transfer***
  - *ownership transfer* leaves the source variable storing 'nil'

```
var otherOwned = anotherOwned;
// anotherOwned now stores nil
```

- **'owned' can be assigned 'nil':**

```
var x = new owned C();
x = nil;  // deletes the previous value
```

- **Other methods are available, see '[owned]' docs**

Ownership Transfer on Argument Passing

- A default-intent 'owned' argument transfers ownership
- For example:

```
var global: owned C;
test();
proc test() {
  var x = new owned C();
  saveit(x); // leaves x 'nil' - instance transferred to arg & then to global
  // instance not destroyed here since x is 'nil'
}
proc saveit(arg: owned C) {
  global = arg; // OK — Transfers ownership from 'arg' to 'global'
  // now instance will be deleted at end of program
}
writeln(global);   // OK — Prints object allocated by test() as ‘x’
```

Borrowed and Borrowing

- **What is a borrow?**
  - a pointer to a class instance that does not impact its lifetime
- **Class types default to 'borrowed'**
  - 'C' is the same as 'borrowed C'
  - 'borrowed' is appropriate for the majority of class uses
- **The 'borrow' method is available to get a borrow**

```
var x = new owned C();
var b = x.borrow();
// .borrow() also available for shared, unmanaged, and borrowed objects
```

Coercions to Borrowed

- **Coercions to 'borrowed' keep code simpler:**

```
var x = new owned C();
compute(x); // Coerces to borrow to pass argument

proc compute(input: C) { ... }
// Could also be written as:
proc compute(input: borrowed C) { ... }
```

  - Coercions available from 'owned', 'shared', and 'unmanaged'
    - User can also cast these to the corresponding 'borrow' type

Borrowed Arguments Don't Impact Lifetime

- An argument with borrowed type does not impact lifetime
- For example:

```
var global: borrowed C;
test();
proc test() {
  var x = new owned C();
  saveit(x.borrow());
  // instance destroyed here
}
proc saveit(arg: borrowed C) {
  global = arg;  // Error! trying to store borrow from local 'x' into 'global'
  delete arg;    // Error! trying to delete a borrow
}
writeln(global);   // uh-oh! use-after free
```

Compile-Time Checking of Borrows

- **Lifetime checker is a new compiler component**
  - It checks that borrows do not outlive the relevant managed variable
- **For example, this will not compile:**

```
proc test() {
  var a: owned C = new owned C();
  // the instance referred to by a is deleted at end of scope
  var c: C = a.borrow();
  // c "borrows" the instance managed by a
  return c;  // lifetime checker error! returning borrow from local variable
  // a is deleted here
}

$ chpl ex.chpl
ex.chpl:1: In function 'test':
ex.chpl:6: error: Scoped variable c cannot be returned
ex.chpl:2: note: consider scope of a
```

Class Methods

- **Class methods borrow 'this'**

```
proc C.method() {
  writeln(this.type:string); // outputs the borrow type ‘C'
                             // a.k.a. 'borrowed C’
}
```

- **Coercions to borrow enable method calls on 'owned'**

```
var x = new owned C();
x.method(); // 'this' argument coerces to borrow in call
```

Class Subtyping

- **All class value kinds support subtyping**
  - Example shows 'owned', but 'shared', 'unmanaged', 'borrowed' all work

```
class ParentClass { ... }
class ChildClass: ParentClass { ... }

proc consumeParent(arg: owned ParentClass) { ... }
var x = new owned ChildClass();
consumeParent(x); // coerces 'owned ChildClass' to 'owned ParentClass'
                  // and consumes x, leaving it 'nil'

proc borrowParent(arg: ParentClass) { ... }
var y = new owned ChildClass();
borrowParent(y); // coerces 'owned ChildClass' to 'borrowed ParentClass'
                 // y still stores an object after this call
```

'new C' and 'new borrowed C'

- **What happens with an undecorated 'new'?**

```
var a = new C();
```

- **Here the type of 'a' is a 'borrowed C'**
  - the instance will be destroyed at the end the current block
    - think: “I’m borrowing this instance from this scope”
  - *ownership transfer* or *sharing* are not possible
  - returning 'a' results in a compilation error
- **The following are also equivalent to the above:**

```
var a: C = new owned C();     // coercing to borrow
var a = (new owned C()): C;   // casting to borrow
var a = (new owned C()).borrow();
```

Error-Handling

Error Handling: errors are classes

- Base 'Error' class is provided

```
class Error {
}
```

- 'Error' will typically be specialized

```
class MyStrError: Error {
  var msg: string;
}

class MyIntError: Error {
  var i: int;
}
```

Error Handling: throwing errors

- Throw errors with 'throw'

```
// throw a newly created error
throw new MyStrError("error message here");

// throwing an error stored in a variable
var e = new MyStrError("test error");
throw e;
```

- Mark procedures that can throw with 'throws'

```
proc mayThrowErrors() throws { ... }

proc mayThrowErrorsAlso(): int throws { ... }

proc mayNotThrowErrors() { ... }
```

Error Handling: try/catch

- **'try' and 'try!' are used to handle thrown errors**
  - { } blocks try to match to an associated 'catch' clause
  - Single statements will not match any 'catch' clauses

```
try {
  mayThrowErrors();
  mayNotThrowErrors();       // non-throwing calls may be included
  mayThrowErrorsAlso();
}
try! mayThrowErrors();       // halts on error
```

- **If an error is handled with no matching 'catch' clause:**
  - 'try' propagates the error
    - To an outer 'try', or out of the procedure (which must be marked 'throws')
  - 'try!' halts instead of propagating

Error Handling: try/catch

- **'catch' clause list matches against an 'Error' at run-time**
  - If a type filter matches the error, that block will be executed
  - Lack of a type filter means that all errors match

```
try {
  trickyOperation(badArg);
} catch err: IllegalArgumentError {  // IllegalArgumentError, subtypes
  writeln("illegal argument!");
} catch err: MyError {               // MyError, subtypes
  throw err;
} catch {                            // catch-all
  writeln("unknown error!");
}
```

Error Handling: ‘try’ expressions

- ‘try’/‘try!’ expressions are also available:

```
proc idiomOne() throws {
  var x = try intOrThrow(0); // throws errors upwards
  var y = try intOrThrow(1);
  return x + y;
}

proc idiomTwo() {
  return try! idiomOne(); // halts on error
}
```

Error Handling: Implicit modules

- **Implicit modules can call directly to throwing procedures**
  - program will halt if such subroutines throw
  - equivalent to wrapping all code in a try!
  - rationale: support quick sketching of code without handling every error

```
// implicitModule.chpl

thisCallMayThrow();    // equivalent to:  'try! thisCallMayThrow();'
```

Error Handling: Explicit modules

- **Explicit modules require handling / re-throwing errors**
  - rationale: now that this is a module, let's avoid surprises

```
module E {
  proc doesNotThrow() {
    // this potential error must be handled by a try! or a try with a catch all
    try! thisCallMayThrow();
  }

  proc doesThrow() throws {
    // OK because the procedure will throw errors upward
    thisCallMayThrow();
  }
}
```

Error Handling: Prototype modules

- ***Prototype* modules act like implicit modules**
  - A way to name a module without opting into handling all errors

```
prototype module NP {
  thisCallMayThrow();   // equivalent to:  'try! thisCallMayThrow();'
}
```

  - Future work: Link other behaviors to prototype modules?
    e.g., implicit / prototype module's symbols: public by default
    explicit module's symbols: private by default

Error-Handling and Parallelism

Error Handling: Parallelism, TaskErrors

- **'TaskErrors' aggregate errors across tasks**
  - 'Error' subtype that collects errors from tasks for centralized handling
  - Only thrown if there are one or more errors from the tasks
  - Can be iterated on, filtered for different kinds of errors

```
try {
  ...
} catch errors: TaskErrors {
  for e in errors {
    writeln("Caught task error e ", e.message());
  }
}
```

Error Handling: Parallelism, ‘cobegin’/‘coforall’

- ‘cobegin’ blocks and ‘coforall’/‘forall’ loops can throw
  - Errors will be stored in a ‘TaskErrors’, even if only one task is run

```
try {
  cobegin {
    canThrow(0);
    canThrow(1);
  }
} catch e: TaskErrors {
  ...
}
```

```
try {
  coforall i in 1..numTasks {
    canThrow(i);
  }
} catch e: TaskErrors {
  ...
}
```

Error Handling: Nested Parallelism

- **Nested loops or tasks do not produce nested 'TaskErrors'**
  - All errors are flattened to a single level

```
try {
  forall i in 1..m {
    forall j in 1..n {
      throw new DemoError();
    }
  }
} catch errors: TaskErrors {
  ...
}
```

Error Handling: Parallelism, 'begin'

- **Errors can be thrown from 'begin' statements as well**
  - Resulting errors can be caught from the surrounding 'sync' statement
    - Note: 'sync' statements are always considered to throw

```
try {
  sync {
    begin canThrow(0);
    begin canThrow(1);
  }
} catch errors: TaskErrors {
  ...
}
```

Defer Statements

Defer Statements

- **Support releasing general resources**
  - releasing locks, freeing memory/objects, closing files, etc.
- **Specifies a cleanup action for its enclosing block**
- **Cleanup actions run for *any* block exit**
  - through regular exit
  - subroutine return
  - error handling
  - ‘break’ and ‘continue’ within loops
- **Users could implement such patterns using records**
  - but that tends to require greater effort

Defer: Ugly Example Using Error-Handling

```
proc f(out releaseError: int) throws {
  var resource = allocateResource();
  var myInstance = new MyClass();
  if !setupResource(resource) {
    delete myInstance; // free resources if setup fails
    releaseError = releaseResource(resource);
    return;
  }
  try {
    throwingFunction();
  } catch e {
    delete myInstance; // free resources if we're throwing an error upwards
    releaseError = releaseResource(resource);
    throw e;
  }
  delete myInstance; // free resources upon normal return
  releaseError = releaseResource(resource);
}
```

Defer: Ugly Example Rewritten Using 'Defer'

```
proc f(out releaseError: int) throws {
  var resource = allocateResource();
  var myInstance = new MyClass();
  defer {
    // free resources regardless of which of the three ways we might return
    delete myInstance;
    releaseError = releaseResource(resource);
  }
  if !setupResource(resource) {
    return;
  }
  try throwingFunction();
}
```

Interoperation

Interoperation

- **Chapel supports calling between languages, primarily C**
  - …and by extension, languages that interoperate with C
    - e.g., Python and Fortran

- **Two modes:**
  - **extern**: Refer to symbols in other languages from Chapel
  - **export:** Call from other languages into Chapel subroutines
    - ultimately, would also expect to export symbols other than subroutines…

- **Chapel also supports a number of types for C interop**
  - c_char, c_int, c_string, c_size_t, c_void_ptr, c_ptr(<type>), etc.

Creating Extern Declarations

- **'extern' says that a symbol is defined outside of Chapel:**
  - tells Chapel what it needs to make use of the symbol:

```
extern type my_c_type_alias;
extern record my_c_struct {
  var x: c_int;
  var y: double;
}
extern proc sizeof(type t): size_t;
extern proc printf(format: c_string, args…);
```

  - symbols resolved by specifying C header files using 1 of 2 methods:
    1) source code: `require "c_header.h";`
    2) command-line: `$ chpl myChapelProg.chpl c_header.h`
  - supports "white lies" that the header file can resolve
    - e.g., my_c_struct has a field z that Chapel doesn't need to know about?
    - e.g., printf()'s format string is actually a const char* restrict

Two Tools for Generating Extern Declarations

1) **c2chapel:** standalone tool that converts C headers to Chapel

- https://chapel-lang.org/docs/tools/c2chapel/c2chapel.html
- Benefits: generates Chapel code that can then be edited
- Downsides: if header file evolves, needs to be re-run or re-edited

```
> c2chapel foo.h
```

C99

```
// foo.h
struct misc {
  char a;
  char* b;
  void* c;
  int* d;
};
```

Chapel

```
// Generated with c2chapel version 0.1.0

// Header given to c2chapel:
require "foo.h";

// Note: Generated with fake std headers
extern record misc {
  var a : c_char;
  var b : c_string;
  var c : c_void_ptr;
  var d : c_ptr(c_int);
}
```

Two Tools for Generating Extern Declarations

2) extern blocks: parses file-scope C code embedded in Chapel

- https://chapel-lang.org/docs/technotes/extern.html#support-for-extern-blocks
- Benefits: doesn't require a separate compilation / editing step
- Downsides: intermediate Chapel code not directly accessible to edit
  - Note: requires building Chapel with LLVM enabled due to reliance on clang

```
extern {
  int my_c_func(int x) {
    printf("C is printing this: %d\n", x);
    return x+1;
  }
  #include "gsl.h"
}
```

Creating Extern Declarations

- **'export' says a symbol may be used outside of Chapel:**
  - (it can also be called from within Chapel, of course)

```
export proc myParallelComputation() {
  var A = readDataInParallel("infile.txt");
  forall i in 1..n do
    A[i] = …
  return max reduce A;
}
```

  - current support is considered prototypical
    - compiler generates header files, Makefile stubs, Python / Cython files
    - argument types are limited to basic types, 1D arrays so far
    - see --library family of flags documented at:
      https://chapel-lang.org/docs/technotes/libraries.html
    - current status also documented in 1.18 release notes:
      - https://chapel-lang.org/releaseNotes/1.18/07-ongoing.pdf

What Else?

Other Base Language Features not covered here

- Varargs functions
- 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.