Base Language Themes

- **Style:** Block-structured, imperative
- **Syntax:**
  - Borrow heavily from C family (C, C++, Java, C#, Perl) for familiarity
  - In other cases, use something intuitive and easy to learn
- **Object-oriented programming:**
  - Support it, but don’t require it
  - Reference- and value-based objects (Java- and C++-style)
- **Type System:**
  - Statically typed for performance, safety
  - Permit types to be elided in most contexts for convenience
- **Aliasing:**
  - Minimize aliases to help with compiler analysis (e.g., no pointers)
  - Main sources: object references, array aliases
- **Compiler-inserted array temporaries:** never require them
Chapel Influences

- Intentionally not an extension to an existing language
- Instead, select attractive features from previous work:
  - **ZPL, HPF**: data parallelism, index sets, distributed arrays
    (see also APL, NESL, Fortran90)
  - **Cray MTA C/Fortran**: task parallelism, lightweight synchronization
  - **CLU**: iterators (see also Ruby, Python, C#)
  - **ML**: latent types (see also Scala, Matlab, Perl, Python, C#)
  - **Java, C#**: OOP, type safety
  - **C++**: generic programming/templates (without adopting its syntax)
  - **C, Modula, Ada**: syntax

Outline

- Starting points
- Basics
  - Lexical Structure
  - Scalar Types
  - Variable, Constant, Configuration Declarations
  - Console I/O
  - Conversions
  - Operators
- Middle Ground
- More advanced topics
Lexical Structure

- **Comments**: standard C-style comments
  
  ```plaintext
  x = 1;  // single-line comment
  x = 2;  /* multi-line
          comment */
  ```

- **Whitespace**:
  - spaces, TABs, new-lines
  - ignored, except to separate tokens and end single-line comments

- **Identifiers**:
  - made up of A–Z, a–z, 0–9, _, $
  - cannot start with 0–9

- **Case-sensitivity**: Chapel is case-sensitive

- **Statement structure**:
  - statements terminated by ;
  - compound statements enclosed by { ... }

Scalar Types

<table>
<thead>
<tr>
<th>Scalar Type</th>
<th>Description</th>
<th>Default Value</th>
<th>Default Width</th>
<th>Currently Supported Bit-widths</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>boolean value</td>
<td>false</td>
<td>impl.-dependent</td>
<td>8, 16, 32, 64</td>
</tr>
<tr>
<td>int</td>
<td>signed integer</td>
<td>0</td>
<td>32 bits</td>
<td>8, 16, 32, 64</td>
</tr>
<tr>
<td>uint</td>
<td>unsigned integer</td>
<td>0</td>
<td>32 bits</td>
<td>8, 16, 32, 64</td>
</tr>
<tr>
<td>real</td>
<td>real floating point</td>
<td>0.0</td>
<td>64 bits</td>
<td>32, 64</td>
</tr>
<tr>
<td>imag</td>
<td>imaginary floating point</td>
<td>0.0i</td>
<td>64 bits</td>
<td>32, 64</td>
</tr>
<tr>
<td>complex</td>
<td>complex value</td>
<td>0.0 + 0.0i</td>
<td>128 bits</td>
<td>64, 128</td>
</tr>
<tr>
<td>string</td>
<td>character string</td>
<td>&quot;&quot;</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

- **Syntax**

  ```plaintext
  scalar-type: scalar-type-name [(width)]
  ```

- **Examples**

  ```plaintext
  int(64) // a 64-bit int
  real(32) // a 32-bit real
  uint    // a 32-bit uint
  ```
Literals

- **Boolean:**
  - `true` // true bool
  - `false` // false bool

- **Integer:**
  - `123` // decimal int
  - `0x1fff` // hexadecimal int
  - `0b1001` // binary int

- **Floating Point:**
  - `1.2` // real
  - `3.4e-6` // real
  - `7.8i` // imag
  - `100i` // imag
  - `1.2 + 3.4i` // complex
  - `(5.6, 7.8)`: complex // complex

- **String:**
  - "hi" // string
  - 'SC08' // string

Declarations: Variables

- **Syntax**
  - `var-decl-stmt:
    var identifier [: type] [= initializer]

- **Semantics**
  - declares a new variable named `identifier`
  - `type` – if specified, indicates variable’s type
    - otherwise, type is inferred from `initializer`
  - `initializer` – if specified, used as the variable’s initial value
    - otherwise, the variable’s type determines its initial value

- **Examples**
  - `var epsilon: real = 0.01;`
  - `var count: int;` // “count” initialized to 0
  - `var name = “Brad”;` // “name” inferred to be string
  - `var x, y: int,` // comma-separated forms also
    `flag = false;` // supported
Console Output

- Syntax:
  
  ```plaintext
  write(expr-list)
  writeln(expr-list)
  ```

- Semantics:
  - `write` – print the argument list to the console in order
  - `writeln` – same as `write`, but also print a new-line at the end

- Examples:
  ```plaintext
  var n = 1000;
  writeln("n is: ", n);
  ```

- Output:
  ```plaintext
  n is 1000
  ```

Hello world: simplest version

- Program
  ```plaintext
  writeln("Hello, world!");
  ```

- Output
  ```plaintext
  Hello, world!
  ```
Console Input

- Syntax (readln versions also supported):
  
  ```chapel
  read(expr-list)
  read(type)
  read(type-list)
  ```

- Semantics:
  - `read(expr-list)` – read values into the argument list expressions
  - `read(type)` – read a value of the specified type and return it
  - `read(type-list)` – read values of the given types and return as a tuple
  - `readln` – same as read, but then read through the next new-line

- Examples:
  ```chapel
  var x, y: real,
      z: int;
  read(x, z);       // read a real into x, an int into z
  y = read(real);   // read a real into y
  (y, z) = read(real, int); // read a real into y, an int into z
  ```

Declarations: Constants

- Syntax
  ```chapel
  const-decl-stmt:
  const identifier [: type] [= initializer]
  ```

- Semantics
  - like a variable, but cannot be reassigned after initialization
  - initializer need not be a statically-known value

- Examples
  ```chapel
  const pi = 3.14159; // pi is a constant 64-bit real
  pi = 0.0;          // ILLEGAL: cannot reassign a const
  const n = computeN(); // can initialize w/ runtime value
  ```
Configuration Variables/Constants

- **Syntax**
  
  ```
  config-decl-stmt:
      config const-decl-stmt
  | config var-decl-stmt
  ```

- **Semantics**
  
  - like a standard declaration, but supports command-line overrides
  - must be declared at global scope

- **Examples**
  
  ```
  config const n = 10, 
  epsilon = 0.01, 
  verbose = false;
  ```

- **Executable Command-line**
  
  ```
  > ./a.out --n=10000 --epsilon=0.0000001 --verbose=true
  ```

---

Hello world: configurable version

- **Program**
  
  ```
  config const msg = "Hello, world!";
  writeln(msg);
  ```

- **Output**
  
  ```
  > ./a.out
  Hello, world!
  >
  > > ./a.out --msg="Hello, SC08!!"
  Hello, SC08!!
  ```
Implicit Conversions

Notes:
- reals do not implicitly convert to ints as in C
- ints and uints don’t interconvert as handily as in C
- C# has served as our model in establishing these rules

Explicit Conversions / Casts

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

Semantics
- convert `expr` to the type specified by `type`

Examples
```
const three = pi: int,
age   = "3": int;
```
Basic Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>arithmetic (plus, minus, multiply, divide, C-style modulus)</td>
</tr>
<tr>
<td>-</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td></td>
</tr>
<tr>
<td>%</td>
<td></td>
</tr>
<tr>
<td>**</td>
<td>exponentiation</td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise (and, or, xor, not, shift-left, shift-right)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^=</td>
<td>op-assignment</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td>(e.g., ( x += y; \Rightarrow x = x + y ))</td>
</tr>
<tr>
<td>&lt;= =&gt;</td>
<td>swap assignment</td>
</tr>
</tbody>
</table>

Outline

- Starting points
- Basics
- Middle Ground
  - Other Types
    - Ranges
    - Arrays
  - Loops and Control Flow
  - Program Structure
    - Functions and Iterators
    - Modules and main()
- More advanced topics
Other Types

- Covered Today:
  - Ranges: regular integer sequences
  - Domains: index sets
  - Arrays: mappings from indices to variables

- Touched on Today:
  - Tuples: lightweight mechanism for grouping variables/values
  - Records: value-based objects, like C structs or C++ classes
  - Classes: reference-based objects, like Java or C# classes

- Not covered today:
  - Unions: store multiple types in overlapping memory
    - (as in C, but type-safe)
  - Enumerated types: finite list of named values
    - e.g., enum color {red, green, blue};

Range Values

- Syntax

  \[ \text{range-expr:} \quad [lo].[hi] [by \ stride] \]

- Semantics
  - represents a regular sequence of integers
  - if \( \text{stride} > 0 \): \( lo, lo+\text{stride}, lo+2\times\text{stride}, \ldots \leq hi \)
  - if \( \text{stride} < 0 \): \( hi, hi-\text{stride}, hi-2\times\text{stride}, \ldots \geq lo \)
  - \( lo \) or \( hi \) can be omitted if \( \text{stride} \) has the appropriate sign

- Examples

  \[
  \begin{align*}
  1..6 & \quad // 1, 2, 3, 4, 5, 6 \\
  1..6 \text{ by } -1 & \quad // 6, 5, 4, 3, 2, 1 \\
  6..1 & \quad // an empty sequence \\
  1..6 \text{ by } 2 & \quad // 1, 3, 5 \\
  1..6 \text{ by } -4 & \quad // 6, 2 \\
  1.. & \quad // 1, 2, 3, 4, 5, 6, 7, 8, \ldots 
  \end{align*}
  \]
The # operator

- Syntax
  
  \[ \text{count-expr:} \]
  \[ \text{range-expr \# count-expr} \]

- Semantics
  * creates a range from the initial count-expr elements of range-expr

- Examples
  
  \[
  \begin{array}{ll}
  0..#6 & // 0, 1, 2, 3, 4, 5 \\
  0..6 \text{ by } 2 & // 0, 2, 4 \\
  0..\text{by } 2 \#6 & // 0, 2, 4, 6, 8, 10 \\
  1..6 \#3 & // 1, 2, 3 \\
  1..6 \text{ by } -1 \#3 & // 6, 5, 4
  \end{array}
  \]

Array Types

- Syntax
  
  \[ \text{array-type:} \]
  \[ [\text{index-set}] \text{ type} \]

- Semantics
  * for each index in index-set, stores an element of type

- Examples
  
  \[
  \text{var A: [1..3] int,} \quad // \text{a 3-element array of ints} \\
  \text{B: [1..3, 1..5] string,} \quad // \text{a 2D 3x5 array of strings} \\
  \text{C: [1..3] [1..5] real;} \quad // \text{a 3-element array of} \\
  \quad \text{arrays of reals} \\
  \text{D: [1..3] int = (1, 2, 3);} \quad // \text{initialized array}
  \]

Brad Chamberlain, Steve Deitz, Samuel Figueroa, David Iten; Cray Inc.
Array Indexing

- **Syntax**

  ```chapel
  index-expr:
    array-expr[index-expr]
  | array-expr(index-expr)
  ```

- **Semantics**
  - references the element in `array-expr` corresponding to `index-expr`

- **Examples**

  ```chapel
  var A: [1..3] int,
      B: [1..3, 1..5] string;
  A(1) = 2;
  B[1, 2] = "hi";
  B[2, 5] = "SC08";
  B[0, 0] = "oops"; // error: indexing out-of-bounds
  ```

For loops

- **Syntax**

  ```chapel
  for-loop:
    for identifier in iterable-expr do body-stmt
  | for identifier in iterable-expr { body }
  ```

- **Semantics**
  - executes loop body once per value yielded by `iteratable-expr`
  - stores each value in a body-local variable/const named `identifier`

- **Examples**

  ```chapel
  var A: [1..3] string = ("hi", "SC08", "!!");
  for i in 1..3 do write(A(i)); // prints: hiSC08!!
  for i in 1..3 do i += 1; // illegal, ranges yield consts
  for a in A {
    a += "-"; write(a); // prints: hi-SC08-!!
  } // A is now "hi-" "SC08-" "!!-"
  ```
Zippered/Tensor Iteration

- **Syntax**
  ```chapel
tensor-for-loop:
  for index-deci in [iter-expr1, iter-expr2, ...] loop-body
  
  zippered-for-loop:
  for index-deci in (iter-expr1, iter-expr2, ...) loop-body
  ```

- **Semantics**
  - `tensor-for-loop` – iterates over all pairs of yielded elements
  - `zippered-for-loop` – iterates over yielded elements pair-wise

- **Examples**
  ```chapel
  for i in [0..1, 0..1] ... // i = (0,0); (0,1); (1,0); (1,1)
  for i in (0..1, 0..1) ... // i = (0,0); (1,1)
  for (x,y) in (0..1, 0..1) ... // x=0, y=0; x=1, y=1
  ```

Other Control Flow

- **While loops**
  ```chapel
  while test-expr do body-stmt
  while test-expr { body-stmts }
  do { body-stmts } while test-expr;
  ```

- **Conditional Statements and Expressions**
  ```chapel
  if test-expr then true-stmt [else false-stmt]
  if test-expr { true-stmts } [else { false-stmts }]
  if test-expr then true-expr [else false-expr]
  ```

- **Also...**
  - `select`: a switch/case statement
  - `break`: break out of a loop (optionally labeled)
  - `continue`: skip to next iteration of a loop (optionally labeled)
  - `return`: return from a function
  - `exit`: exit the program
  - `halt`: exit the program due to an exceptional/error condition
Function Definitions

- **Syntax**

  \[
  \text{function-decl-stmt:} \\
  \text{def \ identifier \ [(formal-list)] \[: \ type] \ body}
  \]

- **Semantics**
  - **identifier** – name of function being defined
  - **formal-list** – list of arguments (potentially empty)
  - **type** – if specified, specifies function’s return type
    - otherwise, return type inferred from function body
  - **body** – specifies function’s definition

- **Examples**

```
def square(x: real): real { 
  return x**2;
}

const pi2 = square(pi);
```

---

Formal Arguments

- **Syntax**

  \[
  \text{formal-argument:} \\
  \text{intent \ identifier \[: \ type]\[= \ init]}
  \]

- **Semantics**
  - **identifier** – name of formal argument
  - **intent** – how to pass the actual argument
  - **type** – if specified, specifies formal type; otherwise generic (inferred)
  - **init** – if specified, permits argument to be omitted at callsite

- **Example**

```
def label(x, name: string, end = "\n") { 
  writeln(name, " is ", x, end); 
  label(n, "n", " and "); 
  label(msg "msg");
}
```

- **Output**

```
n is 1000 and msg is Hello, SC08!
```
Named Argument Passing

- Arguments may be matched by name rather than position

```python
def foo(x: int = 2, y: int = x) { ... }
foo(); // equivalent to foo(2, 2);
foo(3); // equivalent to foo(3, 3);
foo(y=3); // equivalent to foo(2, 3);
foo(y=3, x=2); // equivalent to foo(2, 3);
foo(y=3, 2); // equivalent to foo(2, 3);
```

Argument Intents

- Syntax

```
intent:
(blank) | const | in | out | inout
```

- Semantics
  - `in` – copy actual into formal at function start and permit modification
  - `out` – copy formal into actual at function return
  - `inout` – combination of “in” and “out”
  - `const` – varies with type
  - `(blank)` – varies with type; follows “principle of least surprise”
### Argument Intents and Types

<table>
<thead>
<tr>
<th>argument type</th>
<th>intent</th>
<th>scalar type</th>
<th>domain/array</th>
<th>record</th>
<th>class</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>“copy in”: copy actual into formal at function start and permit modification</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>out</td>
<td>“copy out”: copy formal into actual at function return</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inout</td>
<td>“copy in and out”: combination of in and out</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>const</td>
<td>copy in but disallow modification</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>pass by reference and disallow modification</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>copy in and disallow modification</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>copy reference in and disallow modification to reference</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(blank)</td>
<td>see const</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>pass-by-reference and permit modification</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>see const</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>see const</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Motivation for Iterators

<table>
<thead>
<tr>
<th>Given a program with a bunch of similar loops…</th>
<th>Consider the effort to convert them from RMO to CMO…</th>
<th>Or to tile the loops…</th>
</tr>
</thead>
<tbody>
<tr>
<td>for i in 0..#m do</td>
<td>for j in 0..#n do</td>
<td>for jj in 0..#n by block do</td>
</tr>
<tr>
<td>for j in 0..#n do</td>
<td>for i in 0..#m do</td>
<td>for ii in 0..#n by block do</td>
</tr>
<tr>
<td>...A(i,j)…</td>
<td>...A(i,j)…</td>
<td>for j in jj.. min(m, jj+block)-1 do</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>for i in ii..min(n, ii+block)-1 do</td>
</tr>
<tr>
<td>for i in 0..#m do</td>
<td>...</td>
<td>...A(i,j)…</td>
</tr>
<tr>
<td>for j in 0..#n do</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>...A(i,j)…</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Motivation for Iterators

Given a program with a bunch of similar loops…

```
for i in 0..#m do
    for j in 0..#n do
        ...A(i,j)...
```

Consider the effort to convert them from RMO to CMO…

```
for j in 0..#n do
    for i in 0..#m do
        ...A(i,j)...
```

Or to tile the loops…

```
for jj in 0..#n by block do
    for ii in 0..#m by block do
        for j in jj..min(m,jj+block)-1 do
            for i in ii..min(n,ii+block)-1 do
                ...A(i,j)...
```

Or to change the iteration order over the tiles…

```
for jj in 0..#n by block do
    for ii in 0..#m by block do
        for j in jj..min(m,jj+block)-1 do
            for i in ii..min(n,ii+block)-1 do
                ...A(i,j)...
```

Or to make them into fragmented loops for an MPI program…

```
for j in jj..min(m,jj+block)-1 do
    for i in ii..min(n,ii+block)-1 do
        ...A(i,j)...
```

Or to change the distribution of the work/arrays in that MPI program…

```
for j in jj..min(m,jj+block)-1 do
    for i in ii..min(n,ii+block)-1 do
        ...A(i,j)...
```

Or to label them as parallel for OpenMP or a vectorizing compiler…

```
for j in jj..min(m,jj+block)-1 do
    for i in ii..min(n,ii+block)-1 do
        ...A(i,j)...
```

Or to do anything that we do with loops all the time as a community…

```
...A(i,j)...
```

We wouldn't program straight-line code this way, so why are we so tolerant of our lack of loop abstractions?

Iterators

- like functions, but yield a number of elements one-by-one:

```python
def RMO():
    for i in 0..#m do
        for j in 0..#n do
            yield (i, j);

def tiled(block):
    for jj in 0..#n by block do
        for ii in 0..#m by block do
            for j in jj..min(m,jj+block)-1 do
                for i in ii..min(n,ii+block)-1 do
                    yield (i, j);
```

- can be used to drive for loops:

```python
for (i,j) in RMO() do
    ...A(i,j)...
```

```python
for (i,j) in tiled(block) do
    ...A(i,j)...
```

- as with functions…
  ...one iterator can be redefined to change the behavior of many loops
  ...a single invocation can be altered, or its arguments can be changed

- not necessarily any more expensive than raw, inlined loops
# Modules

**Syntax:**

- `module-def: module { code }`
- `module-use: use module-name;`

**Semantics**

- All Chapel code is stored in modules.
- Using a module causes its symbols to be available from that scope.
- Top-level code in a module is executed when the module is first used.

**Example**

```chapel
module M {
    def foo() {
        writeln("Hi from M!");
    }
    writeln("Someone used M");
}
```

**Output**

```
Someone used M
Hi from M!
```

---

# Program Entry Point

**Semantics**

- Each module can define a function “main” to serve as an entry point.
- If a module does not define “main”, its top-level code serves as main.
- If a program defines multiple “mains”, compiler flags must specify one.

**Example**

```chapel
module M1 {
    def main() {
        writeln("Running M1");
    }
}
```

**Output**

```
Running M1
```

---

Brad Chamberlain, Steve Deitz, Samuel Figueroa, David Iten; Cray Inc.
Hello world: structured version

- **Program**
  
  ```chapel
  module Hello {
      def main() {
          writeln("Hello, world!");
      }
  }
  ```

- **Output**
  
  Hello, world!

Hello world: simplest version

- **Program**
  
  ```chapel
  writeln("Hello, world!");
  ```

- **Output**
  
  Hello, world!
Outline

- Starting points
- Basics
- Middle Ground
- More advanced topics
  - Object-oriented Programming (OOP)
  - Compile-time machinery
  - Generics

Record Types

- Syntax

```
record-type-decl:
  record identifier { decl-list }
```

- Semantics
  - creates a record type named `identifier`
  - `decl-list` – defines member constants/variables, and methods
  - assignment copies members from one record to another
  - similar to C++ classes

- Example

```
record employee { var name: string, id: int; }
var e1: employee = new employee(name="Brad", id=12345),
e2: employee; // e2 defaults to name="", id=0
e1 = e2; // e2 is a distinct copy of e1
e2.name = "Joe";
writeln(e1.name); // prints "Brad"
```

Brad Chamberlain, Steve Deitz, Samuel Figueroa, David Iten; Cray Inc.
Class Types

- Syntax

```
class-type-decl:
class identifier { decl-list }
```

- Semantics
  - similar to records, but creates a reference type rather than a "struct"
  - assignment copies object reference, not members
  - similar to Java classes

- Example

```chapel
class employee { var name: string, id: int; }
var e1: employee = new employee(name="Brad", id=12345),
e2: employee;  // e2 defaults to nil
e1 = e2;       // e2 is an alias of e1
e2.name = "Joe";
writeln(e1.name);  // prints "Joe"
```

OOP Capabilities

- We won’t cover a number of standard OOP features today:
  - inheritance
  - shadowing members/fields
  - dynamic dispatch
  - point of instantiation
  - …
Standard Methods

- Classes/records support standard user-defined methods:
  - `this()` - permits indexing an instance of the class/record
  - `these()` - permits iteration over an instance of the class/record
  - `writeThis()` - overrides the default way of printing a class

Example uses:

```chapel
var myC = new C();
myC(i,j);       // legal if C supports a this() function
                // that takes i and j as arguments
for x in myC do ...   // legal if C supports a these()
                // iterator
writeln(myC);       // calls writeThis() if defined,
                // otherwise compiler supplies a default
```

Standard Methods Example

```chapel
class Pair {
  var x, y: real;
  def this(i: int) {
    if (i==0) then
      return x;
    if (i==1) then
      return y;
    halt(“out-of-bounds: ”, i);
  }
  def these() {
    yield x;
    yield y;
  }
  def writeThis(s: Writer) {
    s.write((x,y));
  }
}
```

Use

```chapel
var p = new Pair(x=1.2, y=3.4);
writeln(“p(0)=”, p(0));
writeln(“p(1)=”, p(1));
p(0)=5.6; p(1)=7.8;
for x in p {
  writeln(x);
  x = -x;
}
writeln(p);
```

Output

```
p(0)=1.2
p(1)=3.4
1.2
3.4
(1.2, 3.4)
```
Standard Methods Using var Return Types

class Pair {
  var x, y: real;
  def this(i: int) var {
    if (i==0) then
      return x;
    if (i==1) then
      return y;
    halt("out-of-bounds: ", i);
  }
  def these() var {
    yield x;
    yield y;
  }
  def writeThis(s: Writer) {
    s.write((x,y));
  }
}

var p = new Pair(x=1.2, y=3.4);
writeln("p(0)=", p(0));
writeln("p(1)=", p(1));
p(0) = 5.6;  p(1) = 7.8;
for x in p {
  writeln(x);
  x = -x;
}
writeln(p);

Compile-Time Language

- Chapel has rich compile-time capabilities
  - loop unrolling
  - conditional folding
  - user-defined functions that can be evaluated at compile-time
  - ...

- Supported via two main language concepts:
  - type – type variables and expressions
  - param – compile-time constants

- In order to support static typing and good performance...
  ...the compiler must be able to determine the static types of...
  - variables/members
  - function arguments/return types
  - parameter values are required in certain contexts
  - array ranks
  - indexing of heterogeneous tuples
### Compile-time Language Examples

```chapel
param numDims = 2; // declare a compile-time constant
type elemType = int; // declare a named type
def sqr(param x) param { // declare a param function
    return x*x; // std ops on params create params
}
param nDSq = sqr(numDims); // use it to create a param
def myInt(param big: bool) type { // declare a type fun.
    if (big) then return int(64); // param test =>
    else return int(32); // fold conditional
}
var myTuple = (1, "hi", 2.3); // heterogeneous tuple
for i in 1..3 do // illegal: types vary
    writeln(myTuple(i)); // across iterations
for param i in 1..3 do // param index =>
    writeln(myTuple(i)); // unroll loop
```

### Generic Functions, Records, Classes

- **type** and **param** are also used for generic programming
  - a copy of the function/class is stamped out for each unique signature
  - generic functions are created by accepting param/type arguments

```chapel
def x2y2(type t, x: t, y: t): t {
    return x**2 + y**2;
}
x2y2(int, 2, 3);
x2y2(real, 1.2, 3.4);
```

- generic classes are created by having param/type members

```chapel
class BoundedStack {
    type elemType;
    const bound: int = 10;
    var data: [1..bound] elemType;
}
var myStack = new BoundedStack(string, 100);
```
Other Base Language Features

- config params -- can be set on the compiler command-line
- function/operator overloading
  - where clauses to decide between overloads using type/param exprs
- argument query syntax
  ```chapel
def x2y2(x: ?t, y: t): t {
  return x**2 + y**2;
}
x2y2(2, 3);
x2y2(1.2, 3.4);
x2y2(1, 2.3);
```
- tuple types, enumerated types, type unions
- file I/O
- nested modules
- standard modules

Questions?