Chapel: Base Language
Goals of this Talk

- Help you understand code in subsequent slide decks
- Give you the basic skills to program in Chapel today
- Provide a survey of Chapel’s base language features
- Impart an appreciation for the base language design

Note: There is more in this slide deck than we will be able to cover, so consider it a reference and overview
"Hello World" in Chapel: Two Versions

- Fast prototyping
  ```chapel
  writeln("Hello, world!");
  ```

- “Production-grade”
  ```chapel
  module HelloWorld {
    def main() {
      writeln("Hello, world!");
    }
  }
  ```
Characteristics of Chapel

- **Syntax**
  - Basics taken from C and Modula
  - Influences from several other languages

- **Semantics**
  - Imperative, block-structured execution model
  - Optional object-oriented programming
  - Type inference for convenience and generic programming
  - Static typing for performance and safety

- **Design points**
  - No pointers and limited aliases
  - No compiler-inserted array temporaries
  - Intentionally not an extension of an existing language
Chapel Influences

ZPL, HPF: data parallelism, index sets, distributed arrays

CRA Y M T A C/For tr an: task parallelism, synchronization

CLU (see also Ruby, Python, C#): iterators

Scala (see also ML, Matlab, Perl, Python, C#): type inference

Java, C#: OOP, type safety

C++: generic programming/templates
       (but with a different syntax)
Outline

• Introductory Notes
• Elementary Concepts
  • Lexical structure
  • Types, variables, and constants
  • Operators and Assignments
  • Compound Statements
  • Input and output
• Data Types and Control Flow
• Program Structure
Lexical Structure

• Comments

/* standard
   C style
   multi-line */

// standard C++ style single-line

• Identifiers:
  • Composed of A-Z, a-z, _, $, 0-9
  • Cannot start with 0-9
  • Case-sensitive
## Primitive Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Default Value</th>
<th>Currently-Supported Bit Widths</th>
<th>Default Bit Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>logical value</td>
<td>false</td>
<td>8, 16, 32, 64</td>
<td>impl. dep.</td>
</tr>
<tr>
<td>int</td>
<td>signed integer</td>
<td>0</td>
<td>8, 16, 32, 64</td>
<td>32</td>
</tr>
<tr>
<td>uint</td>
<td>unsigned integer</td>
<td>0</td>
<td>8, 16, 32, 64</td>
<td>32</td>
</tr>
<tr>
<td>real</td>
<td>real floating point</td>
<td>0.0</td>
<td>32, 64</td>
<td>64</td>
</tr>
<tr>
<td>imag</td>
<td>imaginary floating point</td>
<td>0.0i</td>
<td>32, 64</td>
<td>64</td>
</tr>
<tr>
<td>complex</td>
<td>complex floating points</td>
<td>0.0 + 0.0i</td>
<td>64, 128</td>
<td>128</td>
</tr>
<tr>
<td>string</td>
<td>character string</td>
<td>“”</td>
<td>any multiple of 8</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Syntax

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

### Examples

```
int(64) // 64-bit int
real(32) // 32-bit real
uint    // 32-bit uint
```
Implicit Type Conversions (Coercions)

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 guide in establishing these rules
Type Aliases and Casts

• **Basic Syntax**

```chapel

    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**

```chapel

type elementType = complex(64);

5:int(8)  // store value as int(8) rather than int(32)
"54":int  // convert string to an int(32)
249:elementType  // convert int to complex(64)
```
Variables, Constants, and Parameters

• Basic syntax

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

• Semantics
  • `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

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

**Syntax**

```
config-declaration:
  config type-alias-declaration
  config declaration
```

**Semantics**

- Like normal, but supports command-line overrides
- Must be declared at module/file scope

**Examples**

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

```bash
% chpl myProgram.chpl -sintSize=64 -selementType=real
% a.out --start=2 --epsilon=0.00001
```
<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Associativity</th>
<th>Overloadable</th>
</tr>
</thead>
<tbody>
<tr>
<td>:</td>
<td>cast</td>
<td>left</td>
<td>no</td>
</tr>
<tr>
<td>**</td>
<td>exponentiation</td>
<td>right</td>
<td>yes</td>
</tr>
<tr>
<td>! ~</td>
<td>logical and bitwise negation</td>
<td>right</td>
<td>yes</td>
</tr>
<tr>
<td>* / %</td>
<td>multiplication, division and modulus</td>
<td>left</td>
<td>yes</td>
</tr>
<tr>
<td>unary + −</td>
<td>positive identity and negation</td>
<td>right</td>
<td>yes</td>
</tr>
<tr>
<td>+ −</td>
<td>addition and subtraction</td>
<td>left</td>
<td>yes</td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt;</td>
<td>shift left and shift right</td>
<td>left</td>
<td>yes</td>
</tr>
<tr>
<td>&lt;= &gt;= &lt; &gt;</td>
<td>ordered comparison</td>
<td>left</td>
<td>yes</td>
</tr>
<tr>
<td>== !=</td>
<td>equality comparison</td>
<td>left</td>
<td>yes</td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise/logical and</td>
<td>left</td>
<td>yes</td>
</tr>
<tr>
<td>^</td>
<td>bitwise/logical xor</td>
<td>left</td>
<td>yes</td>
</tr>
<tr>
<td>l</td>
<td>bitwise/logical or</td>
<td>left</td>
<td>yes</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>short-circuiting logical and</td>
<td>left</td>
<td>via isTrue</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>short-circuiting logical or</td>
</tr>
</tbody>
</table>
## Assignments

<table>
<thead>
<tr>
<th>Kind</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>simple assignment</td>
</tr>
<tr>
<td>+= -= *= /= %= *= &amp;=</td>
<td>= ^= &amp;&amp;=</td>
</tr>
<tr>
<td>&lt;=&gt;</td>
<td>swap assignment</td>
</tr>
</tbody>
</table>

- **Note:** assignments are only supported at the statement level
Compound Statements

- **Syntax**

```
compound-stmt:
    { stmt-list }
```

- **Semantics**

- As in C, permits a series of statements to be used in place of a single statement

- **Example**

```
{ writeln(“Starting a compound statement”); 
x += 1; writeln(“Ending the compound statement”); }
```
Output
- `write(expr-list)`: writes the argument expressions
- `writeln(...)` variant: writes a linefeed after the arguments

Input
- `read(expr-list)`: reads values into the argument expressions
- `read(type-list)`: reads values of given types, returns as tuple
- `readln(...)` variant: same, but reads through next linefeed

Example:
```chapel
var first, last: string;
write("what is your name? ");
read(first);
last = read(string);
writeln("Hi ", first, " ", last);
```

What is your name? Chapel User
Hi Chapel User

I/O to files and strings also supported
Outline

- Introductory Notes
- Elementary Concepts
- Data Types and Control Flow
  - Tuples
  - Ranges
  - Arrays
  - For loops
  - Other control flow
- Program Structure
Tuples

• Syntax

heterogeneous-tuple-type:  
( type, type-list )

homogenous-tuple-type:  
param-int-exp * type

tuple-expr:  
( expr, expr-list )

• Purpose

• supports lightweight grouping of values  
(e.g., when passing or returning procedure arguments)

• Examples

var coord: (int, int, int) = (1, 2, 3);  
var coordCopy: 3*int = i3;  
var (i1, i2, i3) = coord;  
var triple: (int, string, real) = (7, “eight”, 9.0);
Range Values

• **Syntax**

  \[ \text{range-expr: [low] .. [high]} \]

• **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

**Syntax**

```
range-op-expr:  
  range-expr by stride  
  range-expr # count  
  range-expr(range-expr)
```

**Semantics**

- **by**: strides range; negative *stride* ⇒ start from *high*
- **#**: selects initial *count* elements of range
- **()** or **[]**: intersects the two ranges

**Examples**

```
1..6 by 2    // 1, 3, 5
1..6 by -1   // 6, 5, 4, ..., 1
1..6 #4      // 1, 2, 3, 4
1..6[3..]    // 3, 4, 5, 6
1.. by 2     // 1, 3, 5, ...
1.. by 2 #3   // 1, 3, 5
1.. #3 by 2   // 1, 3
0..#n        // 0, ..., n-1
```
Array Types

- Syntax

  \[
  \text{array-type:} \\
  \quad [ \text{index-set-expr} ] \text{ elt-type}
  \]

- Semantics
  - Stores an element of \text{elt-type} for each index
  - May be initialized using tuple expressions

- Examples

  ```chapel
  var A: [1..3] int = (5, 3, 9), // 3-element array of ints
  B: [1..3, 1..5] real,       // 2D array of reals
  C: [1..3][1..5] real;       // array of arrays of reals
  ```

*Much more on arrays in the data parallelism talk...*
**For Loops**

- **Syntax**
  
  ```chapel
  for-loop:
  for index-exp in iterable-exp { stmt-list }
  ```

- **Semantics**
  - Executes loop body serially, once per loop iteration
  - Declares new variables for identifiers in `index-exp`
    - type and const-ness determined by `iterable-exp`
  - `iterable-exp` could be a range, array, or iterator

- **Examples**
  
  ```chapel
  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
  ```
Zipper and Tensor Iteration

• Syntax

```
zipper-for-loop:
  for index-expr in ( iterable-exprs ) { stmt-list }
tensor-for-loop:
  for index-expr in [ iterable-exprs ] { stmt-list }
```

• Semantics

• Zipper iteration is over all yielded indices pair-wise
• Tensor iteration is over all pairs of yielded indices

• Examples

```
for i in (1..2, 0..1) { ... } // (1,0), (2,1)
for i in [1..2, 0..1] { ... } // (1,0), (1,1), (2,0), (2,1)
```
Other Control Flow Statements

- **Conditional statements**
  ```chapel
  if cond { computeA(); } else { computeB(); }
  ```

- **While loops**
  ```chapel
  while cond { compute(); }
  ```

- **Select statements**
  ```chapel
  select key { 
  when value1 { compute1(); } 
  when value2 { compute2(); } 
  otherwise { compute3(); } 
  }
  ```

**Note:** *Chapel also has expression-level conditionals and for loops*
Control Flow: Braces vs. Keywords

Note: Most control flow supports keyword-based forms for single-statement versions

- Conditional statements
  ```chapel
  if cond then computeA(); else computeB();
  ```

- While loops
  ```chapel
  while cond do
      compute();
  ```

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

- For loops
  ```chapel
  for indices in iterable-expr do
      compute();
  ```
Outline

- Introductory Notes
- Elementary Concepts
- Data Types and Control Flow
- Program Structure
  - Procedures and iterators
  - Modules and main()
  - Records and classes
  - Generics
  - Other basic language features
Procedures, by example

- Example to compute the area of a circle

```chapel
def area(radius: real): real {
    return 3.14 * radius**2;
}
writeln(area(2.0)); // 12.56
```

- Example of argument default values, naming

```chapel
def 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)
writeCorrd(y=2.0, 3.0); // (3.0, 2.0)
```

Argument and return types can be omitted.
Iterators

- **Iterator**: a procedure that generates values/variables
  - Used to drive loops or populate data structures
  - Like a procedure, but yields values back to invocation site
  - Control flow logically continues from that point

- **Example**

  ```chapel
  def fibonacci(n: int) {
    var current = 0,
       next = 1;
    for 1..n {
      yield current;
      current += next;
      current <=> next;
    }
  }

  for f in fibonacci(7) do writeln(f);
  ```

  0 1 1 2 3 5 8
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: passed by reference
  - `const`: disallows modification of the formal
  - `in`: copies actual into formal at start; permits modifications
  - `out`: copies formal into actual at procedure return
  - `inout`: does both of the above
  - `param/type`: formal must be a param/type (evaluated at compile-time)

- Returned values are `const` by default
  - `const`: cannot be modified (without assigning to a variable)
  - `var`: permits modification back at the callsite
  - `type`: returns a type (evaluated at compile-time)
  - `param`: returns a param value (evaluated at compile-time)
Modules

• **Syntax**

  ```chapel
  module-def:
    module identifier { code }
  
  module-use:
    use module-identifier;
  ```

• **Semantics**
  
  • all Chapel code is stored in modules
  
  • using a module makes its symbols visible in that scope
  
  • module-level statements are executed at program startup
    • useful for initializing a module
  
  • for convenience, a file with top-level code defines a module
    with the file’s name
Semantics

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

Any module may define a main() procedure
If multiple modules define main(), the user must select one

M1.chpl:
```chapel
use M2;
writeln("Init-ing M1");
def main() { writeln("Running M1"); }
```

M2.chpl:
```chapel
module M2 {
  use M1;
  writeln("Init-ing M2");
def main() { writeln("Running M2"); }
}
```

% chpl M1.chpl M2.chpl --main-module M1
% ./a.out
Init-ing M2
Init-ing M1
Running M1
Revisiting "Hello World"

- Fast prototyping
  ```chpl
  hello.chpl
  writeln("Hello, world!");
  ```

- "Production-grade"
  ```chpl
  module HelloWorld {
    def main() {
      writeln("Hello, world!");
    }
  }
  ```

Module-level code is executed during module initialization

main() executed when program begins running
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++ class:Java class

Example

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

var c1, c2: circle;
c1 = new c1(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++ class:Java class

- Example

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

var c1, c2: circle;
c1 = new c1(radius=1.0);
c2 = c1; // aliases c1’s circle
c1.radius = 5.0;
writeln(c2.radius); // 5.0
delete c1;
```
Method Examples

Methods are procedures associated with types

```chapel
def circle.circumference
    return 2* pi * radius;

writeln(c1.area(), " ", c1.circumference);
```

Methods can be defined for any type

```chapel
def int.square()
    return this**2;

writeln(5.square());
```
Generic Procedures

Generic procedures can be defined using type and param arguments:

```chapel
def foo(type t, x: t) { ... }
def bar(param bitWidth, x: int(bitWidth)) { ... }
```

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

```chapel
def goo(x, y) { ... }
def sort(A: []) { ... }
```

Generic procedures are instantiated for each unique argument signature:

```chapel
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 can be defined using type and param fields:

```chapel
class Table { 
  param size: int; 
  var data: size*int; 
} 

class Matrix { 
  type eltType; 
  ... 
}
```

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

```chapel
record Triple { 
  var x, y, z; 
}
```

Generic objects are instantiated for each unique type signature:

```chapel
// 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);
```
Other Base Language Features not covered today

- Enumerated types
- Unions
- Type select statements, argument type queries
- Parenthesis-less functions/methods
- Procedure dispatch constraints (where clauses)
- Compile-time features for meta-programming
  - type/param procedures
  - folded conditionals
  - unrolled for loops
  - user-defined compile-time warnings and errors
• Most features are in reasonably good shape
• Performance is currently lacking in some cases
• Some semantic checks are incomplete
  • e.g., constness-checking for members, arrays
• Error messages could use improvement at times
• OOP features are limited in certain respects
  • multiple inheritance
  • user constructors for generic classes, subclasses
• Memory for strings is currently leaked
Future Directions

- Fixed-length strings
- I/O improvements
  - Binary I/O
  - Parallel I/O
  - General serialization capability
- Exceptions
- Interfaces
- Namespace control
  - private fields/methods in classes and records
  - module symbol privacy, filtering, renaming
- Interoperability with other languages
- Introductory Notes
  - Characteristics
  - Influences
- Elementary Concepts
  - Lexical structure
  - Types, variables, and constants
  - Operators and assignments
  - Compound Statements
  - Input and output
- Data Types and Control Flow
  - Tuples
  - Ranges
  - Arrays
  - For loops
  - Other control flow
- Program Structure
  - Procedures and iterators
  - Modules and main()
  - Records and classes
  - Generics
  - Other basic language features

Questions?