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 Hello {
    proc main() {
      writeln("Hello, world!");
    }
  }
  ```
Characteristics of Chapel

• **Design points**
  • Identifying parallelism & locality is user’s job, not compiler’s
  • No unexpected compiler-inserted array temporaries
  • No pointers and limited opportunities for aliasing
  • Intentionally not an extension of an existing language
Chapel Influences

**C, Modula**: basic syntax

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

**CRAY MTA C/Fortran**: 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

```c
/* 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>&quot;&quot;</td>
<td>any multiple of 8</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Syntax

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

### Examples

```plaintext
int(64) // 64-bit int
real(32) // 32-bit real
uint      // 32-bit uint
```
Notes:

- reals do not implicitly convert to ints as in C
- ints and uints don’t interconvert as handily as in C
Type Aliases and Casts

- **Basic Syntax**

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

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

- **Semantics**

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

- **Examples**

  ```
  type  elementType = complex(64);
  ```

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

  ```
  const 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**

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

```
% chpl myProgram.chpl -sintSize=64 -selementType=real
% a.out --start=2 --epsilon=0.00001
```
## Basic Operators and Precedence

<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><em>left</em></td>
<td>no</td>
</tr>
<tr>
<td>**</td>
<td>exponentiation</td>
<td><em>right</em></td>
<td>yes</td>
</tr>
<tr>
<td>! ~</td>
<td>logical and bitwise negation</td>
<td><em>right</em></td>
<td>yes</td>
</tr>
<tr>
<td>* / %</td>
<td>multiplication, division and modulus</td>
<td><em>left</em></td>
<td>yes</td>
</tr>
<tr>
<td>unary + -</td>
<td>positive identity and negation</td>
<td><em>right</em></td>
<td>yes</td>
</tr>
<tr>
<td>+ -</td>
<td>addition and subtraction</td>
<td><em>left</em></td>
<td>yes</td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt;</td>
<td>shift left and shift right</td>
<td><em>left</em></td>
<td>yes</td>
</tr>
<tr>
<td>&lt;= &gt;= &lt; &gt;</td>
<td>ordered comparison</td>
<td><em>left</em></td>
<td>yes</td>
</tr>
<tr>
<td>== !=</td>
<td>equality comparison</td>
<td><em>left</em></td>
<td>yes</td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise/logical and</td>
<td><em>left</em></td>
<td>yes</td>
</tr>
<tr>
<td>^</td>
<td>bitwise/logical xor</td>
<td><em>left</em></td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bitwise/logical or</td>
<td><em>left</em></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>short-circuiting logical and</td>
<td><em>left</em></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>
</tbody>
</table>
| += -= *= /= %= **= &= |= ^= &&|= |||= <<= >>= | compound assignment  
(e.g., x += y; is equivalent to x = x + y;)
| <=> | swap assignment |

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

- **Syntax**
  
  \[
  \text{compound-stmt:} \\
  \quad \{ \text{stmt-list} \} \\
  \]

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

- **Example**
  
  ```plaintext
  { 
      writeln("Starting a compound statement");
      x += 1;
      writeln("Ending the compound statement");
  }
  ```
**Console Input/Output**

- **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:**
    
    \[
    ( \text{type}, \text{type-list} )
    \]
  
  - **homogeneous-tuple-type:**
    
    \[
    \text{param-int-expr} \ast \text{type}
    \]

  - **tuple-expr:**
    
    \[
    ( \text{expr}, \text{expr-list} )
    \]

- **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);
  ```

- **Purpose**

  - supports lightweight grouping of values (e.g., when passing or returning procedure arguments)
  
  - multidimensional arrays use tuple indices
Range Values

• **Syntax**

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

• **Semantics**

  • Regular sequence of integers
    
    \text{low} \leq \text{high}: \text{low}, \text{low}+1, \text{low}+2, \ldots, \text{high}
    
    \text{low} > \text{high}: \text{degenerate (an empty range)}
    
    \text{low} \text{ or } \text{high} \text{ unspecified: unbounded in that direction}

• **Examples**

  1..6  \quad // 1, 2, 3, 4, 5, 6
  6..1  \quad // \text{empty}
  3..   \quad // 3, 4, 5, 6, 7, …
Range Operators

• Syntax

\[
\text{range-op-expr:}
\]
\[
\text{range-expr by \ by} \ \text{stride}
\]
\[
\text{range-expr \ # \ count}
\]
\[
\text{range-expr(range-expr)}
\]

• Semantics

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

• Examples

\[
1..6 \ \text{by} \ 2 \quad // \ 1, 3, 5
\]
\[
1..6 \ \text{by} \ -1 \quad // \ 6, 5, 4, \ldots, 1
\]
\[
1..6 \ #4 \quad // \ 1, 2, 3, 4
\]
\[
1..6[3..] \quad // \ 3, 4, 5, 6
\]
\[
1.. \ \text{by} \ 2 \quad // \ 1, 3, 5, \ldots
\]
\[
1.. \ \text{by} \ 2 \ #3 \quad // \ 1, 3, 5
\]
\[
1.. \ #3 \ \text{by} \ 2 \quad // \ 1, 3
\]
\[
0..\#n \quad // \ 0, \ldots, n-1
\]
Array Types

- **Syntax**
  
  \[
  \text{array-type:} \quad \begin{cases} \text{[ index-set-exp ] elt-type} \end{cases}
  \]

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

- **Examples**

  ```
  \textbf{var} \quad \begin{align*}
  \text{A: [1..3] int} & = (5, 3, 9), \quad \text{// 3-element array of ints} \\
  \text{B: [1..3, 1..5] real}, & \quad \text{// 2D array of reals} \\
  \text{C: [1..3][1..5] real;} & \quad \text{// array of arrays of reals}
  \end{align*}
  ```

  \textit{Much more on arrays in data parallelism section later...}
For Loops

• Syntax

```
for-loop:
    for index-expr in iterable-expr { stmt-list }
```

• Semantics

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

• Examples

```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 Iteration

• Syntax

zipper-for-loop:
  for index-exp in ( iterable-exp ) { stmt-list }

• Semantics

• Zipper iteration is over all yielded indices pair-wise

• Example

```chapel
var A: [0..9] real;

for (i,j,a) in (1..10, 2..20 by 2, A) do
  a = j + i/10.0;

writeln(A);
```

2.1 4.2 6.3 8.4 10.5 12.6 14.7 16.8 18.9 21.0
Other Control Flow Statements

• Conditional statements

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

• While loops

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

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

• Select statements

```plaintext
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

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

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

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

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

- **Select statements**
  ```
  select key {
    when value1 do compute1();
    when value2 do compute2();
    otherwise do compute3();
  }
  ```
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
proc area(radius: real): real {
    return 3.14 * radius**2;
}
writeLn(area(2.0)); // 12.56
```

- Example of argument default values, naming

```chapel
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 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

```plaintext
iter fibonacci(n) {
    var current = 0,
    next = 1;
    for 1..n {
        yield current;
        current += next;
        current <=> next;
    }
}
```

```plaintext
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)

- Return types can also have intents
  - (blank)/`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 }
  ```

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

- **Semantics**

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

- **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

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

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

```
% chpl M1.chpl M2.chpl --main-module M1
% ./a.out

Initializing M2
Initializing M1
Running M1
```
Revisiting "Hello World"

- Fast prototyping
  ```chapel
  hello.chpl
  writeln(“Hello, world!”);
  ```

- “Production-grade”
  ```chapel
  module HelloWorld {
    proc main() {
      writeln(“Hello, world!”);
    }
  }
  ```

Module-level code is executed during module initialization

main() executed when program begins running
Chapel’s struct/object types
- Contain variable definitions (fields)
- Contain procedure & iterator definitions (methods)
- Records: value-based (e.g., assignment copies fields)
- Classes: reference-based (e.g., assignment aliases object)

Record : Class :: C++ struct : Java class

Example

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

```chapel
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++ struct : Java class

- Example

```chapel
class circle {
  var radius: real;
  proc 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;
```
Methods are procedures associated with types

\[
\text{proc circle.circumference}
\text{ \hspace{1em} return } 2 \times \pi \times \text{radius};
\]
\[
\text{writeln(cl.area()}, “ “, cl.circumference);
\]

Methods can be defined for any type

\[
\text{proc int.square()}
\text{ \hspace{1em} return } \text{this} ** 2;
\]
\[
\text{writeln(5.square());}
\]
Generic procedures can be defined using type and param arguments:

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

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

```chapel
proc goo(x, y) { ... }
proc 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
  - generic classes: subclassing, user constructors

Memory for strings is currently leaked
Future Directions

- Error handling/Exceptions
- Fixed-length strings
- Interfaces (joint work with CU Boulder)
- Improved namespace control
  - private fields/methods in classes and records
  - module symbol privacy, filtering, renaming
- Interoperability with other languages (joint with LLNL)
Questions?

- 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