Parallel Programming in Chapel: The Cascade High-Productivity Language

Brad Chamberlain, Chapel Team, Cray Inc.

PRACE Autumn School: October 25<sup>th</sup>, 2010
What is Chapel?

• A new parallel programming language
  • Design and development led by Cray Inc.
  • Initiated under the DARPA HPCS program

• **Overall goal:** Improve programmer productivity
  • Improve the programmability of parallel computers
  • Match or beat the performance of current programming models
  • Support better portability than current programming models
  • Improve the robustness of parallel codes

• A work-in-progress
Chapel's Implementation

- Being developed as open-source at SourceForge
- Licensed as BSD software

- **Target Architectures:**
  - multicore desktops and laptops
  - clusters of commodity processors
  - Cray architectures
  - systems from other vendors
  - CPU+GPU hybrids (ongoing work)
Today's Goals

- Introduce you to the Chapel language in-depth
- Give you experience...
  ...using the Chapel compiler
  ...writing Chapel code
- Get your feedback on Chapel
- Point you toward resources for future reference
Who Are You?

- Name
- Institution
- Role (student, postdoc, professor, researcher, ...)
- Favorite Programming Languages
- Parallel Programming Models (MPI, OpenMP, ...)

PRACE Autumn School 2010: Parallel Programming in Chapel
Approximate Schedule

11:00 – Welcome
11:05 – Background
11:30 – Base Language
12:00 – Data Parallelism
12:30 – Hands-On I
13:00 – Lunch
14:30 – Task Parallelism
15:00 – Locality Control
15:30 – Project Summary
16:00 – Hands-On II
17:00 – Done
Chapel: Background
Chapel's Origins

- **HPCS**: High Productivity Computing Systems
  - Overall goal: Raise high-end user productivity by 10x
    \[\text{Productivity} = \text{Performance} + \text{Programmability} + \text{Portability} + \text{Robustness}\]

- **Phase II**: Cray, IBM, Sun (July 2003 – June 2006)
  - Goal: Propose new productive system architectures
  - Each vendor created a new programming language
    - **Cray**: Chapel
    - **IBM**: X10
    - **Sun**: Fortress

- **Phase III**: Cray, IBM (July 2006 – )
  - Goal: Develop the systems proposed in phase II
  - Each vendor implemented a compiler for their language
    - Sun also continued their Fortress effort without HPCS funding
Chapel's Productivity Goals

- Vastly improve **programmability** over current languages
  - Writing parallel programs
  - Reading, modifying, porting, tuning, maintaining them

- Support **performance** at least as good as MPI
  - Competitive with MPI on generic clusters
  - Better than MPI on more capable architectures

- Improve **portability** over current languages
  - As ubiquitous as MPI but more abstract
  - More portable than OpenMP, UPC, and CAF are thought to be

- Improve **robustness** via improved semantics
  - Eliminate common error cases
  - Provide better abstractions to help avoid other errors
Chapel’s Context

Chapel’s Motivating Themes

1. General parallel programming
2. *Global-view* abstractions
3. *Multiresolution* design
4. Control over locality/affinity
5. Reduce gap between mainstream & HPC languages
With a unified set of concepts...

...express any parallelism desired in a user’s program

- **Styles:** data-parallel, task-parallel, concurrency, nested, ...
- **Levels:** model, function, loop, statement, expression

...target all parallelism available in the hardware

- **Systems:** multicore desktops, clusters, HPC systems, ...
- **Levels:** machines, nodes, cores, instructions
In pictures: “Apply a 3-Point Stencil to a vector”

\[
\begin{align*}
\text{Global-View} & \quad (\begin{array}{cccc}
\text{pink} & \text{pink} & \text{pink} & \text{red}
\end{array})/2 \\
+ & \quad (\begin{array}{cccc}
\text{red} & \text{red} & \text{red} & \text{red}
\end{array})/2 \\
= & \quad (\begin{array}{cccc}
\text{purple} & \text{purple} & \text{purple} & \text{yellow}
\end{array})
\end{align*}
\]

\[
\begin{align*}
\text{Local-View} & \quad (\begin{array}{cccc}
\text{pink} & \text{pink} & \text{pink} &
\end{array}) \\
& \quad (\begin{array}{cccc}
\text{red} & \text{red} & \text{red} &
\end{array}) \\
& \quad (\begin{array}{cccc}
\text{red} & \text{red} & \text{red} &
\end{array})
\end{align*}
\]
2) Global-View Abstractions

In pictures: “Apply a 3-Point Stencil to a vector”

Global-View

\[
\begin{align*}
&\left( \begin{array}{c}
\text{ mauve } \\
\text{ mauve } \\
\end{array} \right) \\
+ & \left( \begin{array}{c}
\text{ mauve } \\
\text{ mauve } \\
\end{array} \right) \\
= & \left( \begin{array}{c}
\text{ yellow } \\
\text{ yellow } \\
\end{array} \right)
\end{align*}
\]

Local-View

\[
\begin{align*}
&\left( \begin{array}{c}
\text{ mauve } \\
\end{array} \right) \\
+ & \left( \begin{array}{c}
\text{ mauve } \\
\end{array} \right) \\
= & \left( \begin{array}{c}
\text{ yellow } \\
\end{array} \right)
\end{align*}
\]
2) Global-View Abstractions

In code: “Apply a 3-Point Stencil to a vector”

**Global-View**

```chapel
def main() {
    var n = 1000;
    var A, B: [1..n] real;

    forall i in 2..n-1 do
        B[i] = (A[i-1] + A[i+1])/2;
}
```

**Local-View (SPMD)**

```chapel
def main() {
    var n = 1000;
    var A, B: [0..myN+1] real;
    var p = numProcs(),
        me = myProc(),
        myN = n/p;

    if (me < p-1) {
        send(me+1, A[myN]);
        recv(me+1, A[myN+1]);
    }
    if (me > 0) {
        send(me-1, A[1]);
        recv(me-1, A[0]);
    }

    forall i in 1..myN do
        B[i] = (A[i-1] + A[i+1])/2;
}
```

Bug: Refers to uninitialized values at ends of A
2) Global-View Abstractions

**In code:** “Apply a 3-Point Stencil to a vector”

```chapel
Global-View

```def` main() {
`var` n = 1000;
`var` A, B: [1..n] real;
forall i in 2..n-1 do
B[i] = (A[i-1] + A[i+1])/2;
}
```

```chapel
Local-View (SPMD)

```def` main() {
`var` n = 1000;
`var` p = numProcs(),
`me` = myProc(),
`myN` = n/p,
iLo = 1,
iHi = myN;
`var` A, B: [0..myN+1] real;
if (me < p-1) {
send(me+1, A[myN]);
recv(me+1, A[myN+1]);
} else
myHi = myN-1;
if (me > 0) {
send(me-1, A[1]);
recv(me-1, A[0]);
} else
myLo = 2;
forall i in iLo..iHi do
B[i] = (A[i-1] + A[i+1])/2;
}
```

Communication becomes geometrically more complex for higher-dimensional arrays

Assumes p divides n
2) rprj3 Stencil from NAS MG

```
= w0
= w1
= w2
= w3
```
2) \textit{rprj3 Stencil from NAS MG in Fortran + MPI}
2) rprj3 Stencil from NAS MG in Chapel

```chapel
def rprj3(S: [?SD], R: [?RD]) {
    const Stencil = [-1..1, -1..1, -1..1],
        W: [0..3] real = (0.5, 0.25, 0.125, 0.0625),
        W3D = [(i,j,k) in Stencil] W[(i!=0) + (j!=0) + (k!=0)];

    forall ijk in SD do
        S[ijk] = + reduce [offset in Stencil]
            (W3D[offset] * R[ijk + RD.stride*offset]);
}
```

Our previous work in ZPL demonstrated that such compact codes can result in better performance than Fortran + MPI while also supporting more flexibility at runtime*.

*e.g., the Fortran + MPI rprj3 code shown previously not only assumes $p$ divides $n$, it also assumes that $p$ and $n$ are specified at compile-time and powers of two.*
## 2) Classifying Current Programming Models

<table>
<thead>
<tr>
<th>System</th>
<th>Data Model</th>
<th>Control Model</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Communication Libraries</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI/MPI-2</td>
<td>Local-View</td>
<td>Local-View</td>
</tr>
<tr>
<td>SHMEM, ARMCI, GASNet</td>
<td>Local-View</td>
<td>SPMD</td>
</tr>
<tr>
<td><strong>Shared Memory</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenMP, Pthreads</td>
<td>Global-View (trivially)</td>
<td>Global-View (trivially)</td>
</tr>
<tr>
<td><strong>PGAS Languages</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Co-Array Fortran</td>
<td>Local-View</td>
<td>SPMD</td>
</tr>
<tr>
<td>UPC</td>
<td>Global-View</td>
<td>SPMD</td>
</tr>
<tr>
<td>Titanium</td>
<td>Local-View</td>
<td>SPMD</td>
</tr>
<tr>
<td><strong>PGAS Libraries</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Global Arrays</td>
<td>Global-View</td>
<td>SPMD</td>
</tr>
</tbody>
</table>
## 2) Classifying Current Programming Models

<table>
<thead>
<tr>
<th>Communication Libraries</th>
<th>System</th>
<th>Data Model</th>
<th>Control Model</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MPI/MPI-2</td>
<td>Local-View</td>
<td>Local-View</td>
</tr>
<tr>
<td>SHMEM, ARMCI, GASNet</td>
<td>Local-View</td>
<td>SPMD</td>
<td></td>
</tr>
<tr>
<td>Shared Memory</td>
<td>OpenMP, Pthreads</td>
<td>Global-View (trivially)</td>
<td>Global-View (trivially)</td>
</tr>
<tr>
<td>PGAS Languages</td>
<td>Co-Array Fortran</td>
<td>Local-View</td>
<td>SPMD</td>
</tr>
<tr>
<td></td>
<td>UPC</td>
<td>Global-View</td>
<td>SPMD</td>
</tr>
<tr>
<td></td>
<td>Titanium</td>
<td>Local-View</td>
<td>SPMD</td>
</tr>
<tr>
<td>PGAS Libraries</td>
<td>Global Arrays</td>
<td>Global-View</td>
<td>SPMD</td>
</tr>
<tr>
<td>HPCS Languages</td>
<td>Chapel</td>
<td>Global-View</td>
<td>Global-View</td>
</tr>
<tr>
<td></td>
<td>X10 (IBM)</td>
<td>Global-View</td>
<td>Global-View</td>
</tr>
<tr>
<td></td>
<td>Fortress (Sun)</td>
<td>Global-View</td>
<td>Global-View</td>
</tr>
</tbody>
</table>
2) Global-View Programming: A Final Note

- A language may support both global- and local-view programming — in particular, Chapel does

```chapel
def main() {
    coforall loc in Locales do
        on loc do
            MySPMDProgram(loc.id, Locales.numElements);
}

def MySPMDProgram(me, p) {
    ...
}
```
3) Multiresolution Language Design: Motivation

“Why is everything so difficult?”
“Why don’t my programs port trivially?”

“Why don’t I have more control?”

Low-Level Implementation Concepts
- MPI
- OpenMP
- Pthreads

High-Level Abstractions
- HPF
- ZPL

Target Machine
Multiresolution Language Design: Support multiple tiers of features

- higher levels for programmability, productivity
- lower levels for performance, control
- build the higher-level concepts in terms of the lower-

Chapel language concepts

- Domain Maps
- Data Parallelism
- Task Parallelism
- Base Language
- Locality Control
- Target Machine

- separate concerns appropriately for clean design
Consider:

- Scalable systems tend to store memory with processors
- Remote accesses take longer than local accesses

Therefore:

- Placement of data relative to computation affects scalability
- Programmers need control over data and task placement

Note:

- As core counts grow, locality will matter more on desktops
- GPUs and accelerators already expose node-level locality
Consider:

- Students graduate with training in Java, Matlab, Perl, Python
- Yet HPC programming is dominated by Fortran, C/C++, MPI

We’d like to narrow this gulf with Chapel:

- to leverage advances in modern language design
- to better utilize the skills of the entry-level workforce...
- ...while not ostracizing the traditional HPC programmer
  - e.g., support object-oriented programming, but make it optional

Other examples:

- function overloading, name-based argument passing
- scripting-like features: type inference, generic functions
- rich data structures with iterators (e.g., associative arrays)
Questions?

- Chapel’s Context
- Chapel’s Motivating Themes
  1. General parallel programming
  2. *Global-view* abstractions
  3. *Multiresolution* design
  4. Control over locality/affinity
  5. Reduce gap between mainstream & HPC languages
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 to be a reference and an introduction
"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

**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 (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
  
- **Whitespace matters, but not overly so**
  
  - Composed of spaces, tabs, and linefeeds
  - Separates tokens and ends //-comments
## Primitive Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Default Value</th>
<th>Default Bit Width</th>
<th>Currently-Supported Bit Widths</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>logical value</td>
<td>false</td>
<td>impl-dep</td>
<td>8, 16, 32, 64</td>
</tr>
<tr>
<td>int</td>
<td>signed integer</td>
<td>0</td>
<td>32</td>
<td>8, 16, 32, 64</td>
</tr>
<tr>
<td>uint</td>
<td>unsigned integer</td>
<td>0</td>
<td>32</td>
<td>8, 16, 32, 64</td>
</tr>
<tr>
<td>real</td>
<td>real floating point</td>
<td>0.0</td>
<td>64</td>
<td>32, 64</td>
</tr>
<tr>
<td>imag</td>
<td>imaginary floating point</td>
<td>0.0i</td>
<td>64</td>
<td>32, 64</td>
</tr>
<tr>
<td>complex</td>
<td>complex floating points</td>
<td>0.0 + 0.0i</td>
<td>128</td>
<td>64, 128</td>
</tr>
<tr>
<td>string</td>
<td>character string</td>
<td>“”</td>
<td>N/A</td>
<td>any multiple of 8</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
```
• 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**
  
  ```
  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(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**
  
  ```
  var count: int;  // initialized to 0
  const pi: real = 3.14159;
  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 type elementType = real(32);
  config param intSize = 32;
  config const epsilon = 0.01:elementType;
  config var start = 1:int(intSize);
  ```

  `% chpl -sintSize=16 -selementType=real(64) myProgram.chpl`
  `% a.out -sstart=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>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></td>
<td></td>
<td>bitwise/logical or</td>
<td>left</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>+=</td>
<td>compound assignment</td>
</tr>
<tr>
<td>-=</td>
<td>(e.g., ( x += y; ) is equivalent to ( x = x + y; ))</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>&amp;=</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=</td>
</tr>
<tr>
<td>^=</td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;=</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td></td>
</tr>
<tr>
<td>&gt;&gt;=</td>
<td></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

\[
\text{compound-stmt:} \\
\{ \text{stmt-list} \}
\]

• Semantics

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

• Example

\{
\text{writeln(“Start of compound statement”);}
\text{x += 1;}
\text{writeln(“End of compound statement”);} \\
\}
Console Input/Output

- **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

- **Output**
  - `write(expr-list)`: writes the argument expressions
  - `writeln(...)` variant: writes a linefeed after the arguments

- **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

- **File and string variants also supported**
Outline

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

- **Syntax**
  
  **homogenous-tuple-type:**
  
  \[ \text{param-int-expr} \ast \text{type} \]

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

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

- **Examples**

  ```chapel
  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);
  ```

- **Purpose**
  
  - supports lightweight grouping of values
    - (e.g., when passing or returning procedure arguments)
Range Values

- Syntax

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

- Semantics

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

- Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..6</td>
<td>1, 2, 3, 4, 5, 6</td>
</tr>
<tr>
<td>6..1</td>
<td>empty</td>
</tr>
<tr>
<td>3..</td>
<td>3, 4, 5, 6, 7, ...</td>
</tr>
</tbody>
</table>
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
  - `()` : intersects the two ranges

- **Examples**

<table>
<thead>
<tr>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..6 by 2</td>
<td>1, 3, 5</td>
</tr>
<tr>
<td>1..6 by -1</td>
<td>6, 5, 4, ... 1</td>
</tr>
<tr>
<td>1..6 # 4</td>
<td>1, 2, 3, 4</td>
</tr>
<tr>
<td>1..6(3..)</td>
<td>3, 4, 5, 6</td>
</tr>
<tr>
<td>1.. by 2</td>
<td>1, 3, 5, ...</td>
</tr>
<tr>
<td>1.. by 2 # 3</td>
<td>1, 3, 5</td>
</tr>
<tr>
<td>1.. # 3 by 2</td>
<td>1, 3</td>
</tr>
<tr>
<td>0..#n</td>
<td>0, ..., n-1</td>
</tr>
</tbody>
</table>
Array Types

• Syntax

\[
\text{array-type:}
\begin{array}{l}
\text{[ index-set-expr ] elt-type}
\end{array}
\]

• Semantics

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

• Examples

\begin{verbatim}
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
\end{verbatim}

Much more on arrays in data parallelism talk
For Loops

• Syntax

```plaintext
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

```plaintext
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

```plaintext
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

```plaintext
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**

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

- **While loops**

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

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

- **Select statements**

```c
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();
  ```

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

- **Select statements**
  ```chapel
  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
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)
```
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
  - **in**: copies actual into formal at start; permits modifications
  - **out**: copies formal into actual at procedure return
  - **inout**: does both of the above
  - **const**: disallows modification of the formal
  - **(none)**: varies with type; follows principle of least surprise
    - most types: **const**
    - arrays, domains, sync vars: passed by reference

- 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
- top-level module code is executed at program startup
- for convenience, a file with top-level code defines 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

```chapel
M1.chpl:
use M2;
writeln(“Initializing M1”);
def main() { writeln(“Running M1”); }

M2.chpl:
module M2 {
  use M1;
  writeln(“Initializing M2”);
  def 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

```chapl
writeln("Hello, world!");
```

- "Production-grade"

```chapl
module hello {
  writeln("Hello, world!");
}
```

```chapl
module HelloWorld {
  def main() {
    writeln("Hello, world!");
  }
}
```
• **Value-based objects**
  • Contain variable definitions (fields)
  • Contain procedure & iterator definitions (methods)
• **Value-based semantics**
  • *e.g.*, assignment copies field values
• Similar to C structs/C++ classes

**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
```
Classes

• Reference-based objects
  • Similar to records, but with reference semantics
    • e.g., variables store object references, assignment copies reference
  • Dynamically allocated/deallocated
  • Support dynamic method dispatch
  • Similar to Java classes

• Example

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

var c1, c2: circle;
c1 = new c1(radius=1.0);
c2 = c1; // references c1
   c1.radius = 5.0;
writeln(c2.radius);  // 5.0
delete c1;
```
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 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

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 types 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

- Unions
- Enumerated types
- Type select statements, argument type queries
- 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 lacking in some cases
Some semantic checks are incomplete
  • e.g., constness-checking for members, arrays
Error messages could often use improvement
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
- Binary I/O
- Parallel I/O
- Exceptions
- Interfaces
- Namespace control
  - private fields/methods in classes and records
  - module symbol privacy, filtering, renaming
- Interoperability with other languages
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

Chapel: Base Language