Chapel: Language Basics
The Hello World Program

- Fast prototyping

```chapel
writeln("hello, world");
```

- Production-grade

```chapel
module HelloWorld {
    def main() {
        writeln("hello, world");
    }
}
```
Characteristics of Chapel

• Syntax
  • Basics from C and Modula
  • Influences from many other languages

• Semantics
  • Imperative, block-structured
  • Optional object-oriented programming (OOP)
  • Elided types for convenience and generic coding
  • Static typing for performance and safety

• Design points
  • No pointers and few references
  • No compiler-inserted array temporaries
ZPL, HPF: data parallelism, index sets, distributed arrays
CRAY MTA C/Fortran: task parallelism, synchronization
CLU, Ruby, Python: iterators
ML, Scala, Matlab, Perl, Python, C#: latent types
Java, C#: OOP, type safety
C++: generic programming/templates
High-Level Comments

Elementary Concepts
- Lexical structure
- Types, variables, and constants
- Input and output

Data Structures and Control

Miscellaneous
Lexical Structure

- **Comments**
  ```
  /* standard C-style */
  // standard C++ style
  ```

- **Identifiers**
  - Composed of A-Z, a-z, _, $, 0-9
  - Starting with A-Z, a-z, _, $

- **Case-sensitive**

- **Whitespace-aware**
  - 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>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>&quot;&quot;</td>
<td>N/A</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
```
Variables, Constants, and Parameters

• Syntax

```
declaration:

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

• Semantics

- Constness at runtime (`const`), at compile-time (`param`)
- Omitted `init-expr`: value is assigned default for type
- Omitted `type`: type is inferred from `init-expr`

• Examples

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

```
config-declaration:
  config declaration
```

Semantics

- Supports command-line overrides
- Must be declared at module (file) scope

Examples

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

% chpl -sintSize=16 myProgram.chpl
% a.out --start=2 --epsilon=0.001
```
Variables Examples

- examples/primers/variables.chpl
## Basic Operators and Precedence

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Associativity</th>
<th>Overloadable</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>:</code></td>
<td>cast</td>
<td>left</td>
<td>no</td>
</tr>
<tr>
<td><code>**</code></td>
<td>exponentiation</td>
<td>right</td>
<td>yes</td>
</tr>
<tr>
<td><code>!</code> <code>~</code></td>
<td>logical and bitwise negation</td>
<td>right</td>
<td>yes</td>
</tr>
<tr>
<td><code>*</code> <code>/</code> <code>%</code></td>
<td>multiplication, division and modulus</td>
<td>left</td>
<td>yes</td>
</tr>
<tr>
<td><code>unary + -</code></td>
<td>positive identity and negation</td>
<td>right</td>
<td>yes</td>
</tr>
<tr>
<td><code>+</code> <code>−</code></td>
<td>addition and subtraction</td>
<td>left</td>
<td>yes</td>
</tr>
<tr>
<td><code>&lt;&lt;</code> <code>&gt;&gt;</code></td>
<td>shift left and shift right</td>
<td>left</td>
<td>yes</td>
</tr>
<tr>
<td><code>&lt;=</code> <code>&gt;=</code> <code>&lt;</code> <code>&gt;</code></td>
<td>ordered comparison</td>
<td>left</td>
<td>yes</td>
</tr>
<tr>
<td><code>==</code> <code>!=</code></td>
<td>equality comparison</td>
<td>left</td>
<td>yes</td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td>bitwise/logical and</td>
<td>left</td>
<td>yes</td>
</tr>
<tr>
<td><code>^</code></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><code>&amp;&amp;</code></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>**= &amp;=</td>
<td>= ^= &amp;&amp;</td>
</tr>
<tr>
<td></td>
<td><em>(e.g., x += y; is equivalent to x = x + y;)</em></td>
</tr>
</tbody>
</table>

---

**Chapel: Language Basics**  
12

---
Input and Output

- **Input**
  - `read(expr-list):` reads values into the arguments
  - `read(type-list):` returns values read of given types
  - `readln(...) variant:` also reads through new line

- **Output**
  - `write(expr-list):` writes arguments
  - `writeln(...) variant:` also writes new line

- **Support for all types (including user-defined)**
- **File and string I/O via method variants of the above**
File I/O examples

- examples/primers/fileIO.chpl
Outline

- High-Level Comments
- Elementary Concepts
- Data Structures and Control
  - Tuples
  - Ranges
  - Arrays
  - For loops
  - Traditional constructs
- Miscellaneous
Tuple Values

• Syntax

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

• Semantics

• Light-weight first-class data structure

• Examples

\[
\text{var i3: (int, int, int) = (1, 2, 3); } \\
\text{var i3_2: 3*int = (4, 5, 6); } \\
\text{var triple: (int, string, real) = (7, “eight”, 9.0);}
\]
Range Values

- **Syntax**

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

- **Semantics**
  - Regular sequence of integers
    
    - \( stride > 0 \): \( low, low + \text{stride}, low + 2 \times \text{stride}, \ldots \leq high \)
    
    - \( stride < 0 \): \( high, high + \text{stride}, high + 2 \times \text{stride}, \ldots \geq low \)
  
  - Default \( stride = 1 \), default \( low \) or \( high \) is unbounded

- **Examples**

  ```plaintext
  1..6 by 2      // 1, 3, 5
  1..6 by -1     // 6, 5, 4, 3, 2, 1
  3.. by 3       // 3, 6, 9, 12, ...
  ```
Range Examples

- examples/primers/ranges.chpl
Array Types

• Syntax

\[
array-type: \\
\quad [\ index-set-exp \ ]\ elt-type
\]

• Semantics
  • Stores an element of \textit{elt-type} for each index

• Examples

\begin{verbatim}
var A: [1..3] int, // 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}

\textit{Much more on arrays in data parallelism part}
Array Examples

- examples/primers/arrays.chpl
For Loops

• Syntax

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

• Semantics

• Executes loop body once per loop iteration
• Indices in `index-expr` are new variables

• Examples

```chapel
var A: [1..3] string = (" DO", " RE", " MI");

for i in 1..3 do write(A(i));  // DO RE MI
for a in A { a += "LA"; write(a); }  // DOLA RELA MILA
```
Zipper "()" and Tensor "[]" Iteration

- **Syntax**

  ```chapel
  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**

  ```chapel
  for i in (1..2, 1..2) do // (1,1), (2,2)
  for i in [1..2, 1..2] do // (1,1), (1,2), (2,1), (2,2)
  ```
Traditional Control

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

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

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

- High-Level Comments
- Elementary Concepts
- Data Structures and Control
- Miscellaneous
  - Functions and iterators
  - Records and classes
  - Generics
  - Other basic language features
Function Examples

• Example to compute the area of a circle

```chapel
def area(radius: real)
    return 3.14 * radius**2;

writeln(area(2.0)); // 12.56
```

• Example of function arguments

```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)
```
What is an Iterator?

- An abstraction for loop control
- Yields (generates) values for consumption
- Otherwise, like a function

**Example**

```chapel
def string_chars(s: string) {
    for i in 1..length(s) do
        yield s.substring(i);
}

for c in string_chars(s) do ...
```
Iterator Examples

- examples/primers/iterators.chpl
Records

- Value-based objects
  - Value-semantics (assignment copies fields)
  - Contain variable definitions (fields)
  - Contain function definitions (methods)
  - Similar to C++ classes

- Example

```chapel
record circle { var x, y, radius: real; }
var c1, c2: circle;
c1.x = 1.0; c1.y = 1.0; c1.radius = 2.0;
c2 = c1; // copy of value
```
Classes

- Reference-based objects
- Reference-semantics (assignment aliases)
- Dynamic allocation
- Dynamic dispatch
- Similar to Java classes

Example

```chapel
class circle { var x, y, radius: real; }
var c1, c2: circle;
c1 = new circle(x=1.0, y=1.0, radius=2.0);
c2 = c1; // c2 is an alias of c1
delete c1;
```
Classes Examples

- examples/primers/classes.chpl
Methods are functions associated with types.

```chapel
def circle.area()
  return 3.14 * radius**2;

writeln(c1.area());
```

Methods can be defined for any type.

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

writeln(5.square());
```
Generic functions can be defined by explicit type and param arguments:

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

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

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

Generic functions are replicated for each unique instantiation:

```chapel
    foo(int, x);  // copy of foo() with t==int 
    foo(string, x);  // copy of foo() with t==string 
    goo(4, 2.2);  // copy of goo() with int and real args 
```
Generic Types

Generic types can be defined by explicit type and param fields:

```chapel
class Table { param numFields: int; ... }
class Matrix { type eltType; ... }
```

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

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

Generic types are replicated for each unique instantiation:

```chapel
// copy of Table with 10 fields
var myT: Table(10);
// copy of Triple with x:int, y:int, z:real
var my3: Triple(int,int,real) = new Triple(1,2,3.0);
```
Generics Examples

- examples/primers/genericClasses.chpl
Other Basic Language Features

- Unions
- Enumerated types
- Range and domain by and # operators
- Expression forms of conditionals and loops
- Type select statements
- Function instantiation constraints (where clauses)
- Formal argument intents (in, out, inout, const)
- User-defined compiler warnings and errors
Future Directions

- Fixed length strings
- Binary I/O
- Parallel I/O
- Interoperability with other languages
- More advanced OO features
• High-Level Comments
• Elementary Concepts
  • Lexical structure
  • Types, variables, and constants
  • Input and output
• Data Structures and Control
  • Tuples
  • Ranges
  • Arrays
  • For loops
  • Traditional constructs

• Miscellaneous
  • Functions and iterators
  • Records and classes
  • Generics
  • Other basic language features