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
  
  ```
  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><code>bool</code></td>
<td>boolean value</td>
<td>false</td>
<td>impl.-dependent</td>
<td>8, 16, 32, 64</td>
</tr>
<tr>
<td><code>int</code></td>
<td>signed integer</td>
<td>0</td>
<td>32 bits</td>
<td>8, 16, 32, 64</td>
</tr>
<tr>
<td><code>uint</code></td>
<td>unsigned integer</td>
<td>0</td>
<td>32 bits</td>
<td>8, 16, 32, 64</td>
</tr>
<tr>
<td><code>real</code></td>
<td>real floating point</td>
<td>0.0</td>
<td>64 bits</td>
<td>32, 64</td>
</tr>
<tr>
<td><code>imag</code></td>
<td>imaginary floating point</td>
<td>0.0i</td>
<td>64 bits</td>
<td>32, 64</td>
</tr>
<tr>
<td><code>complex</code></td>
<td>complex value</td>
<td>0.0 + 0.0i</td>
<td>128 bits</td>
<td>64, 128</td>
</tr>
<tr>
<td><code>string</code></td>
<td>character string</td>
<td>&quot;&quot;</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

- **Syntax**
  ```
  scalar-type: scalar-type-name [(width)]
  ```

- **Examples**
  ```
  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
  'PRACE' // 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
                     // supported
    flag = false;    // supported
  ```
Console Output

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

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

Hello world: simplest version

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

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

- Syntax (readln versions also supported):
  ```plaintext
  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:
  ```plaintext
  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
  ```plaintext
  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
  ```plaintext
  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, PRACE!!"
  Hello, PRACE!!
  ```
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 guide in establishing these rules

Explicit Conversions / Casts

- **Syntax**
  
  \[
  \text{cast-expr:} \\
  \text{expr : type}
  \]

- **Semantics**
  - convert \text{expr} to the type specified by \text{type}

- **Examples**

  ```
  \text{const three = pi: int,} \\
  \text{age = ”3”: int;}
  ```
Basic Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ - * / %</td>
<td>arithmetic ops: plus, minus, multiply, divide, C-style modulus</td>
</tr>
<tr>
<td>**</td>
<td>exponentiation</td>
</tr>
<tr>
<td>&amp;</td>
<td>^</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>assignment</td>
</tr>
<tr>
<td>+= -= *= /= %= *= &amp;=</td>
<td>= ^= &lt;&lt;= &gt;&gt;= &amp;&amp;=</td>
</tr>
<tr>
<td>&lt;&lt;=&gt;= &amp;=&amp;</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**
  ```plaintext
  range-expr: [lo].[hi] [by stride]
  ```

- **Semantics**
  - represents a regular sequence of integers
    - if `stride > 0`: `lo, lo+stride, lo+2*stride, ... ≤ hi`
    - if `stride < 0`: `hi, hi–stride, hi–2*stride, ... ≥ lo`
  - `lo` or `hi` can be omitted if `stride` has the appropriate sign

- **Examples**
  ```plaintext
  1..6        // 1, 2, 3, 4, 5, 6
  1..6 by -1  // 6, 5, 4, 3, 2, 1
  6..1        // an empty sequence
  1..6 by 2   // 1, 3, 5
  1..6 by -4  // 6, 2
  1..          // 1, 2, 3, 4, 5, 6, 7, 8, ...
  ```
The # operator

- **Syntax**
  
  ```
  count-expr: 
  range-expr # count-expr
  ```

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

- **Examples**
  
  ```
  0..#6 // 0, 1, 2, 3, 4, 5
  0..#6 by 2 // 0, 2, 4
  0..by 2 #6 // 0, 2, 4, 6, 8, 10
  1..6 #3 // 1, 2, 3
  1..6 by -1 # 3 // 6, 5, 4
  ```

Array Types

- **Syntax**
  
  ```
  array-type: 
  [index-set] elemtype
  ```

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

- **Examples**
  
  ```
  var A: [1..3] int; // a 3-element array of ints
  B: [1..3, 1..5] string; // a 2D 3x5 array of strings
  C: [1..3] [1..5] real; // a 3-element array of
  // 5-element arrays of reals
  D: [1..3] int = (1, 2, 3); // initialized array
  ```
### Array Indexing

- **Syntax**
  
  \[
  \text{index-expr:} \\
  \quad \text{array-expr[index-expr]} \\
  | \quad \text{array-expr(index-expr)}
  \]

- **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] = "PRACE";
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-scoped variable/const named `identifier`

- **Examples**

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

- **Syntax**

  ```chapel
tensor-for-loop:
  for index-decl in [iter-expr1, iter-expr2, ...] loop-body

zippered-for-loop:
  for index-decl 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**
  
  ```plaintext
  function-decl-stmt:
  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**
  
  ```plaintext
  def square(x: real): real {
    return x**2;
  }
  const pi2 = square(pi);
  ```

---

Formal Arguments

- **Syntax**
  
  ```plaintext
  formal-argument:
  [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**
  
  ```plaintext
  def label(x, name: string, end = "\n") {
    write(name, " is ", x, end);
  label(n, "n", " and ");
  label(msg, "msg");
  }
  ```

- **Output**
  
  ```plaintext
  n is 1000 and msg is Hello, PRACE!
  ```
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>pass by reference and disallow modification</td>
<td>copy in and disallow modification</td>
<td>copy reference in and disallow modification to reference</td>
<td></td>
</tr>
<tr>
<td>(blank)</td>
<td>see const</td>
<td>pass-by-reference and permit modification</td>
<td>see const</td>
<td>see const</td>
<td></td>
</tr>
</tbody>
</table>

### Motivation for Iterators

Given a program with a bunch of similar loops...

```plaintext
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...

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

Or to tile the loops...

```plaintext
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)...
```
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 make them into fragmented loops for an MPI program...

```
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 distribution of the work/arrays in that MPI program...

```
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 label them as parallel for OpenMP or a vectorizing compiler...

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

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

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:

  ```
  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:

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

  ```
  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
- Use-ing 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");
}
use M;
foo();
```

### 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`, choose one using compiler flags

### Example

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

```chapel
module M2 {
  def main() {
    writeln("Running M2");
  }
}
```

### Output

```
chpl --main-module=M1
a.out
Running M1
```

```
chpl --main-module=M2
a.out
Running M2
```
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
  e2 = e1;        // e2 is a distinct copy of e1
  e2.name = "Joe";
  writeln(e1.name);  // prints "Brad"
  ```
Class Types

- **Syntax**
  
  ```chapel
  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
  e2 = e1; // 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/record

- 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);
}
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**

```plaintext
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);
```

**Use**

```
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
5.6
7.8
(-5.6, -7.8)
```

---

**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);
```

**Note:** recall that eliding a formal argument’s type also results in a function generic in that argument)
Other Base Language Features

- config params -- can be set on the compiler command-line
- function/operator overloading
  - where clauses to select between overloads using type/param exprs
- argument query syntax

```plaintext
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

Base Language Status

- **Stable features:**
  - just about everything

- **Incomplete features:**
  - OOP features are limited in certain ways (e.g., multiple inheritance)
  - performance of base language is decent, but could be improved
  - compiler introduces memory leaks in some cases
  - semantic checks are not always complete
    - e.g., constness checking for arrays, domains, class members
  - in many cases error messages could use improvement

- **Future directions:**
  - improved memory management strategy (GC and/or region-based)
Questions?