Quick Start

How to write a one-line "hello, world" program

1. Create the file hello.chpl:

```
   writeln(“hello, world”);
```

2. Compile and run it:

```
   > chpl hello.chpl
   > ./a.out
   hello, world
   >
```

Comments

```
// single-line comment
/* multi-line
   comment */
```

Primitive Types

Type	Default size	Other sizes	Default init
`bool`	impl. dep.	8, 16, 32, 64	`false`
`int`	32	8, 16, 64	`0`
`uint`	32	8, 16, 64	`0`
`real`	64	32, 128	`0.0`
`imag`	64	32, 128	`0.0i`
`complex`	128	64, 256	`0.0+0.0i`
`string`	variable		`“”`

Variables, Constants and Configuration

```
var x: real = 3.14;          variable of type real set to 3.14
var isSet: bool;             variable of type bool set to false
var z = -2.0i;               variable of type imag set to -2.0i
const epsilon: real = 0.01;  runtime constant
param debug: bool = false;   compile-time constant
config const n: int = 100;   > ./a.out --n=4
config param d: int = 4;     > chpl -sd=3 x.chpl
```

Modules

```
module M1 { code; }           module definition
module M2 {
  use M1;                     module use
  def main() { body(); }      main definition
}
```

Expression Precedence and Associativity*

Operators	Uses
`. () []`	member access, call and index
`:`	cast
`**` *(right)*	exponentiation
`reduce scan`	reduction, scan
`! ~` *(right)*	logical and bitwise negation
`* / %`	multiplication, division, modulus
unary `+ -` *(right)*	positive identity, negation
`+ -`	addition, subtraction
`<< >>`	shift left, shift right
`<= >= < >`	ordered comparison
`== !=`	equality comparison
`&`	bitwise/logical and
`^`	bitwise/logical xor
`|`	bitwise/logical or
`&&`	short-circuiting logical and
`||`	short-circuiting logical or
`..`	range construction
`in`	loop expression
`by #`	range/domain stride and count
`if`	conditional expression
`forall [`	parallel iterator expression
`for`	serial iterator expression
`,`	comma separated expression

*Left-associative except where indicated

Casts and coercions

```
var i: int = 2.0:int;   cast real to int
var x: real = 2;        coerce int to real
```

Conditional and Loop Expressions

```
var half = if i%2 then i/2+1 else i/2;
writeln(for i in 1..n do i**2);
```

Assignment and Swap

Simple Assignment:	`=`

Compound Assignments:	`+= -= *= /= %= **=`
	`&= |= ^= &&= ||=`
	`<<= >>=`

Swap:	`<=>`

Statements

```
if cond then stmt1(); else stmt2();
if cond { stmt1(); } else { stmt2(); }

select expr {
  when equiv1 do stmt1();
  when equiv2 { stmt2(); }
  otherwise stmt3();
}

type select actual {
  when type1 do stmt1();
  when type2 { stmt2(); }
  otherwise stmt3();
}

while condition { … }
while condition do …;
do { … } while condition;
for index in aggregate { … }
for index in aggregate do …;
label outer for …
break; or break outer;
continue; or continue outer;
```

Functions

```
def bar(r: real, i: imag): complex {
  var c: complex = r + i;
  return c;
}
def foo(i) return i**2 + i + 1;
```

Formal Argument Intents

Intent	Semantics
`in`	copied in
`out`	copied out
`inout`	copied in and out
blank	formal arguments are constant except arrays, domains, syncs are passed by reference

Named Formal Arguments

```
def foo(arg1: int, arg2: real) { … }
foo(arg2=3.14, arg1=2);
```

Default Values for Formal Arguments

```
def foo(arg1: int, arg2: real = 3.14);
foo(2);
```

Records

```
record Point {                     record definition
  var x, y: real;                  declaring fields
}
var p: Point;                      record instance
writeln(sqrt(p.x**2+p.y**2));      field accesses
p = new Point(1.0, 1.0);           assignment
```

Classes

```
class Circle {                   class definition
  var p: Point;                  declaring fields
  var r: real;
}
var c = new Circle(r=2.0);       class construction
def Circle.area()                method definition
  return 3.14159*r**2;
writeln(c.area());               method call
class Oval: Circle {             inheritance
  var r2: real;
}
def Oval.area()                  method override
  return 3.14159*r*r2;
delete c;                        free memory
c = new Oval(r=1.0,r2=2.0);      polymorphism
writeln(c.area());               dynamic dispatch
```

Unions

```
union U {                alternatives
  var i: int;
  var r: real;
}
```

(union definition)

Tuples

```
var pair: (string, real);   heterogeneous tuple
var coord: 2*int;           homogeneous tuple
pair = ("one", 2.0);        tuple assignment
(s, r) = pair;              destructuring
coord(2) = 1;               tuple indexing
```

Enumerated Types

```
enum day {sun,mon,tue,wed,thu,fri,sat};
var today: day = day.fri;
```

Ranges

```
var every: range = 1..n;         range definition
var everyOther = every by 2;     strided range
var R = 0..#count;               counted range
var FiveBy2 = 0.. by 2 # 5;      0, 2, 4, 6, 8
```

Domains and Arrays

```
var D: domain(1) = [1..n];     arithmetic domain
var A: [D] real;               arithmetic array
var Set: domain(int);          associative domain
Set += 3;                      add index to domain
var SD: sparse subdomain(1);   sparse domain
```

Data Parallelism

```
forall i in D do A(i) = 1.0;   domain iteration
[i in D] A(i) = 1.0;           "
forall a in A do a = 1.0;      array iteration
[a in A] a = 1.0;              "
A = 1.0;                       array assignment
```

Reductions and Scans

Pre-defined: `+ * & | ^ && || min max minloc maxloc`

```
var sum = + reduce A;              1 2 3 => 6
var pre = + scan A;                1 2 3 => 1 3 6
var ml = minloc reduce (A, A.domain);
```

Iterators

```
def squares(n: int) {          serial iterator
  for i in 1..n do
    yield i**2;                yield statement
}
for s in squares(n) do …;      iterate over iterator
```

Task Parallelism

```
begin task();
cobegin { task1(); task2(); }
coforall i in aggregate do task(i);
sync { begin task1(); begin task2(); }
serial condition do stmt();
```

Synchronization Examples

1) `var lock$: sync bool;`

```
lock$ = true;       lock$ = true;
critical1();        critical2();
lock$;              lock$;
```

2) `var data$: sync int;`

```
data$ = produce1();    consume(data$);
data$ = produce2();    consume(data$);
```

3) `var go$: single real;`

```
go$=set();    use1(go$);    use2(go$);
```

Locality

Built-in Constants:

```
config const numLocales: int;      set via --n
const LocaleSpace = [0..numLocales-1];
const Locales: [LocaleSpace] locale;

var c: Circle;
on Locales(i) {           migrate task to new locale
  writeln(here.id);
  c = new Circle();       allocate class on locale
}
writeln(c.locale);        query locale of class instance
on c do { … }             data-driven task migration
```

More Information

url: `http://chapel.cs.washington.edu/`
e-mail: `chapel_info@cray.com`
bugs: `chapel_bugs@cray.com`