

## Quick Start: one-line “hello, world”

```
1. Create the file hello.chpl:
   writeln("hello, world");
2. Compile and run it:
   $ chpl hello.chpl
   $ ./hello
   hello, world
   $
```

## Comments

```
// single-line comment
/* multi-line
   comment /*can be nested*/ */
```

## Primitive Types

Type	Default size	Other sizes	Default init
bool	impl. dep.	8, 16, 32, 64	false
int	64	8, 16, 32	0
uint	64	8, 16, 32	0
real	64	32	0.0
imag	64	32	0.0i
complex	128	64	0.0+0.0i
string	n/a		""

## 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; $. / prog -n=4
config param d: int = 4; $ chpl -sd=3 x.chpl
```

## Modules

```
module M1 { var x = 10; } module definition
module M2 {
  use M1; module use
  proc main() { writeln(x); } main function
}
```

## Expression Precedence and Associativity\*

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

\*Left-associative except where indicated

## Casts and coercions

```
var i = 2.0: int; explicit conversion real to int
var x: real = 2; implicit conversion 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);
```

## Assignments

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

## Statements

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

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

while condition do ...;
while condition { ... }
do { ... } while condition;
for index in aggregate do ...;
for index in aggregate { ... }
try { ... } catch error { ... }
label outer for ...
break; or break outer;
continue; or continue outer;
```

## Procedures

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

## Formal Argument Intents

Intent	Semantics
in	copy-initialized in
out	copied out
inout	copied in and out
ref	passed by reference
const	passed by value or reference, with local modifications disabled
const in	copied in, with local modifications disabled
const ref	passed by reference, with local modifications disabled
<i>blank</i>	like ref for arrays, syncs, singles, atomics; otherwise like const

## Named Formal Arguments

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

## Default Values for Formal Arguments

```
proc 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); initialization
proc Circle.area() method definition
    return 3.14159*r**2;
writeln(c.area()); method call
class Oval: Circle { inheritance
    var r2: real;
}
override proc Oval.area() method override
    return 3.14159*r*r2;
c = new Oval(r=1,r2=2); polymorphism
writeln(c.area()); dynamic dispatch
var nc: owned Circle? = nil; nilable type required
to store nil references
```

## Unions

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

## Tuples

```
var pair: (string, real); heterogeneous tuple
var coord: 2*int; homogeneous tuple
pair = ("one", 2.0); tuple assignment
var (s, r) = pair; destructuring
coord(0) = 1; tuple indexing, 0-based
```

## Enumerated Types

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

## Ranges

```
var every: range = 0..n; range definition
var evens = every by 2; strided range
var R = evens # 5; counted range
var odds = evens align 1; aligned range
var open = 0.. $\infty$ ; open range
```

## Domains and Arrays

```
var rectangular: domain(1); 1-d domain (index set)
const D = {1..n}; domain literal
var A: [D] real; array of real numbers
var Set: domain(int); associative domain
Set += 3; add index to domain
var SD: sparse subdomain(D); sparse domain
```

## Domain Maps

```
use BlockDist;
const D = {1..n} dmapped distrib. domain w/
    Block(boundingBox={1..n}); block distribution
var A: [D] real; distributed array
```

## 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  
                  minmax 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

```
iter squares(n: int) { serial iterator
    for i in 1..n do
        yield i**2; generate a value
    }
for s in squares(n) do ...; loop over iterator
```

## Zipper Iteration

```
for (i,s) in zip(1..n, squares(n)) do ...
```

## Extern Declarations

```
extern proc C_function(x: int);
extern var C_variable: real;
extern { /* c code here */ }
```

## Task Parallelism

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

## Atomic Example

```
var count: atomic int;
if count.fetchAdd(1)==n-1 then
    done = true; nth task to arrive
```

## Synchronization Examples

```
var data$: sync int;
data$ = produce1(); consume(data$);
data$ = produce2(); consume(data$);
```

```
var go$: single real;
go$=set(); use1(go$); use2(go$);
```

## Locality

### Built-in Constants

```
config const numLocales: int; $. /prog -nl4
const LocaleSpace = {0..numLocales-1};
const Locales: [LocaleSpace] locale;
```

### Example

```
var c: owned Circle?;
on Locales[i] { migrate task to new locale
    writeln(here); print the current locale
    c = new Circle(); allocate class on locale
}
writeln(c.locale); query locale of class instance
on c do { ... } data-driven task migration
```

## More Information

**www:** <https://chapel-lang.org>  
**user resources:** <https://chapel-lang.org/users.html>