

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

do { ... } while condition;
while condition { ... } single-statement forms:
for index in iterable { ... } ... do stmt();
foreach index in iterable { ... }
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) do return i**2 + i + 1;
```

Formal Argument Intents

Intent	Semantics
<code>in</code>	copy-initialized in
<code>out</code>	copied out
<code>inout</code>	copied in and out
<code>ref</code>	passed by reference
<code>const in</code>	copied in, with local modifications disabled
<code>const ref</code>	passed by reference, with local modifications disabled
<code>const</code>	passed by value or by reference; caller and callee modifications are disallowed
<i>default</i>	like <code>const</code> for most types; like <code>ref</code> for syncs and atomics

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 {
    var x, y: real;
}
var p: Point;
writeln(sqrt(p.x**2+p.y**2));
p = new Point(1.0, 1.0);
```

*record definition
declaring fields*

record variable

field accesses

*assignment
of a new instance*

Classes

```
class Circle {
    var p: Point;
    var r: real;
}
var c = new Circle(r=2.0);
proc Circle.area()
    do return 3.14159*r**2;
writeln(c.area());
class Oval: Circle {
    var r2: real;
}
override proc Oval.area()
    do return 3.14159*r*r2;
c = new Oval(r=1,r2=2);
writeln(c.area());
var nc: owned Circle? = nil;
```

*class definition
declaring fields*

initialization

method definition

method call

inheritance

method override

*polymorphism
dynamic dispatch*

*nilable type required
to store nil references*

Unions

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

*union definition
alternatives*

Tuples

```
var pair: (string, real);
var coord: 2*int;
pair = ("one", 2.0);
var (s, r) = pair;
coord(0) = 1;
```

heterogeneous tuple

homogeneous tuple

tuple assignment

destructuring

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;
var evens = every by 2;
var R = evens # 5;
var odds = evens align 1;
var open = 0..<n;
```

range definition

strided range

counted range

aligned range

open range

Domains and Arrays

```
var rectangular:domain(1);
const D = {1..n};
var A: [D] real;
var Set: domain(int);
Set += 3;
var SD: sparse subdomain(D);
```

1-d domain (index set)

domain literal

array of real numbers

associative domain

add index to domain

sparse domain

Domain Maps

```
use BlockDist;
const D = {1..n} dmapped
    blockDist(boundingBox={1..n});
var A: [D] real;
```

distrib. domain w/ block

distribution

distributed array

Data Parallelism and Task Intents

```
forall i in D do A[i] = 1.0;
[i in D] A[i] = 1.0;
forall a in A do a = 1.0;
[a in A] a = 1.0;
A = 1.0 + B;
var sumF = 0.0, factor = 3, racy = 0;
forall a in A with (const in factor,
    + reduce sumF, ref racy)
{ sumF reduce= a * factor; racy += 1; }
```

domain iteration

"

array iteration

"

promoted addition and array assignment

task intents

Reductions and Scans

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

```
var sum = + reduce A;
var pre = + scan A;
var ml = minloc reduce (A, A.domain);
```

1 2 3 => 6

1 2 3 => 1 3 6

Iterators

```
iter squares(n: int) {
    for i in 1..n do
        yield i**2;
}
for s in squares(n) do ...;
```

serial iterator

generate a value

loop over iterator

Zipper Iteration

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

Extern Declarations

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

Task Parallelism

```
begin task();
cobegin { task1(); task2(); }
coforall i in iterable 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.writeEF(produce1());
consume(data.readFE());
data.writeEF(produce2());
consume(data.readFE());
```

Locality

Built-in Constants

```
config const numLocales: int; $. /prog -nl 4
const LocaleSpace = {0..numLocales-1};
const Locales: [LocaleSpace] locale;
```

Example

```
var c: owned Circle?;
on Locales[i] {
    writeln(here);
    c = new Circle();
}
writeln(c.locale);
on c do { ... }
```

migrate task to new locale

print the current locale

allocate class on locale

query locale of class instance

data-driven task migration

User Resources

<https://chapel-lang.org/users.html>