x2c Cheat Sheet

x2c-lang.dev

Commands

`x2c run hello.x -- args`	Build and run; args follow `--`.
`x2c build -o app a.x b.x`	An executable from its units.
`x2c translate --dump-ast a.x`	Print the expanded, bound AST; omit flag to emit C.
`#!/usr/bin/env -S x2c script`	File-scope statements run.
`x2c <command>`	Also `new repl lsp install remove list env`.
`x2c help <command>`	`x2c help` lists every command.

Values

`Var v = 42;`	Eight-byte tagged value; not arbitrary C data.
`v is int`	Test the tag; `Var` on the left.
`int n = v;  (long) v`	A typed destination converts.
`printf("%d", v)`	Formats convert a `Var` too.
`v is void`	No value. Test with `is`, not `if`.
`a == b  a === b`	Value equality; exact type/value or object identity.
`x in c`	Membership in a collection.
`<name>`	A `Symbol`; quoted bare names are `Atoms`.

Text

`"text"  s + t  s.len()`	Immutable `String`; `+` joins.
`%"a $x $(f(y))"`	Interpolate `$x` or `$(expr)`.
`s.split(",")  s.lower()`	Also `strip upper words lines`.
`a[start:stop:step]`	Negative bounds count from the end; negative step reverses.

Collections

`%(1 2 (3) "s" $x @xs)`	An immutable `List`; `$x` inserts, `@xs` splices.
`cons(0, xs)  xs.cdr()`	Also `car map filter`.
`[1, 2, 3]  a.push(4)`	A mutable `Array`; `pop sort`.
`{key: 1, "s": 2}`	A `Map`; bare keys are `Atoms`.
`m[<key>]  m.del(<key>)`	A missing key reads `void`.
`$(a, b)  ${k: v}`	Evaluated items and keys.

Functions

`int f(int x) => x * 2;`	Expression body; forward calls need no prototype.
`%!(a, b) => a + b`	Lambda; `using &k` by reference.
`Func g = f;  g(21)`	First-class; calls return `Var`.
`T f(String s, Map @opts)`	Call `f("x", port: 80)`.
`T f(List @rest)`	Rest arguments as a `List`.
`void bump(int &n)`	By reference; `&?p` allows `NULL`.

Lisp

`Lisp l = Lisp.new();`	Embedded Lisp interpreter.
`defer l.destroy();`	Free the interpreter at scope exit.
`l.bind("f", fn);`	Expose a native function.
`int r = l.eval(%(f $n 2));`	`$n` inserts a C value; the result converts back.

Memory and cleanup

`auto { ... }`	Group allocations; free them on leaving the block.
`for (...) auto { ... }`	A scope per iteration.
`T t = auto T.new();`	Clean up this value at exit.
`Scope.open(); defer Scope.close();`	A scope by hand.
`defer f.close();`	At exit, `return`, or a raise.
`xmalloc(n);  xfree(p);`	Tracked allocation; `xfree` releases it early.

Flow

`foreach (T x in c)`	Collections; a `File` by line.
`range(0, n, 2)`	Lazy `int`s; stop is excluded.
`foreach ((String k, int v) in m)`	Each key and value.
`(int a, int b) = list;`	Unpack; `(a, b) = list;` assigns.
`match (x) { ... }`	One statement per case.
`case %(tag ?a *r):`	`?a` one, `*r` the rest, `?` skip.
`case %(p ?a) if (a > 0):`	A guard; no fall-through.
`if (x ~= %(p ?(int a) ?b))`	Binds `a` and `b` on a match.
`switch (s) { case "a": }`	Compares `String` content.
`with (n = 5) { ... }`	Set for the block, then restored.
`alias total to a + b;`	Substitute an expression; re-evaluate at each use.

Errors

`raise %(bad-arg (reason "r"));`	An `Error` is a `List`: cause, details.
`try stmt catch %(not-found *): stmt`	Catch by pattern.
`catch %(?c *d): stmt`	Bind the cause and details.
`catch: stmt  finally stmt`	Anything else; then cleanup.
`$assert(n > 0)`	Raises `<invariant>`; in `system-macros.x`.

Types

`int T.f(T self, int n)`	A method: `value.f(3)`.
`class Point { int x; int y; };`	A value class; generated constructor, `==`, printing.
`class Node { int v; } *;`	A pointer class in the scope.
`void Node.init(Node n, int v)`	Runs inside `Node.new(v)`.
`String Point.repr(Point p)`	Replaces a default method.
`Point Point.add(Point a, Point b)`	Overloads `+`; also `== < []`.
`delegate T field;`	Forwards unknown methods to it.
`protocol Base(T) { long T.m(T); }`	A contract named for its base.
`protocol Base(Impl);`	Adopt; provide required methods, replace defaults.
`$derive(T, <json>, <repr>)`	Methods from fields; `<derive.x>`.
`ArrayInt a = [1, 2, 3];`	Packed `int`s; `typed-array.x`.
`MapStringInt m = {"a": 1};`	Unboxed; also `ArrayDbl ListInt`, more.

Concurrency

`Thread.run([%!(void) => { ... }, ...])`	Start all, then join all.
`Thread.start(fn, v)`	One thread; `t.join()`.
`Channel.new(<i32>, 2)`	Typed messages; `foreach` drains until closed.
`$lock(m) { ... }`	Hold a `Mutex` for a block.
`threaded int counter;`	One copy per thread.

Compile time

`macro Expression $m(Expr $e) => $e * 60;`	`$m(2)` becomes `2 * 60`.
`macro Stmt $s(Expr $a, Expr $b) { ... }`	Names resolve where it lands; or `using name`.
`macro Decorator $d(Function $f) { ... }`	Rewrites the item after `$d()`.
`keyword repeat $ns.repeat;`	Now `repeat (3) stmt` parses.
`meta int f(int n) => ...;`	`$f(3)` runs in the compiler.
`meta List g(TypeInfo t)`	Reads a type's fields, methods.
`$!( { @items } )  $!Stmt{ ... }`	Quoted syntax; `$` inserts, `@` splices.

Library

`%(git status).job().output()`	`process.x`: run a command.
`Path p = ".";  p.join("*.x").glob()`	`path.x`: files, globs, dirs.
`Socket.connect(host, port, ms)`	`net.x`: blocking TCP.
`Router.new().get(path, %!(req, res) => ...)`	`http.x`: routes and a server.
`Json.parse(s)  v.json()`	`json.x`: parse and print.
`Regex re = Regex.compile("a+");  re.match(s)`	`regex.x`: match, split, replace.
`Url u = Url.parse(s);  u.host  u.port`	`url.x`: parsed fields.
`Time t = Time.now();  t.since(t0)`	`time.x`: clocks, `Duration`, sleep.

Tests

`$test() static void t(void) { ... }`	`test.x`; end with `$test.main();`.
`$expect(c)  $expect_eq(a, b)`	Continue after a failure.
`$expect_raises(<cause>, { ... })`	Pass if that cause is raised.
`x2c test --filter name`	Run the project's tests.

C interop

`#include "unit.x"`	Its public names; no header.
`#include "lib.h"`	A C header, unchanged.
`import "pcre2" as rx;`	A wrapped library: `rx.Regexp`.
`p.field  p->field`	`.` works through a pointer.
`String.new(cstr)`	To `String`; out as `char *`.
`x2c build -I dir -O2 a.x`	Flags pass to cc.
`x2c bind --package p --header p.h`	Experimental, checkout only: a starting wrapper.