CiviC (Civilised C) — One-Page Cheat Sheet

Compact reference + runnable-style snippets for the course language.

Program Structure

- **Units:** function declarations/definitions and global var decls/defs.
- **Visibility:** `export` makes a def visible to other modules; `extern` imports a symbol.
- **Entry point:** exactly one exported `main` with no params returning `int`.
- **Preprocessor:** C preprocessor runs; include headers like `#include "civic.h"`.

```
// Declaration (prototype) in this unit for a symbol defined elsewhere
extern int add(int a, int b);

// Definition visible only in this compilation unit
int twice(int x) { return x + x; }

// Exported definition (visible program-wide)
export int main() { return 0; }
```

Types, Literals, Casts

- **Basic types:** `bool`, `int` (32-bit), `float` (64-bit).
- **Literals:** `true`, `false`, decimal/binary/octal/hex ints, floats with optional exponent.
- **No implicit conversions.** Use explicit casts: `(Type) expr`.

```
bool b = true;
int  k = 0b1010;    // 10
float f = 3.14e0;   // 3.14
int  t = (int) b;   // true -> 1, false -> 0
bool nz = (bool) 42; // -> true
```

Expressions & Operators

- **Precedence/associativity:** like C.
- **Arithmetic:** `+ - * / %` on `int`; `+ - * /` on `float`; `%` only on `int`.
- **Relational:** `== != < <= > >=` → `bool`.
- **Boolean:** `&& ||` (short-circuit), unary `!`.

- **Unary minus:** `-` on `int` / `float`.

```
bool ok = (x != 0 && y / x > 42);
float g = -(f * 2.0) + 1.0;
```

Statements & Blocks

- **Kinds:** assignment, procedure call, `if/else`, `while`, `do/while`, counted `for`, `return`.
- **Blocks:** `{ ... }` or single statement.
- **Dangling `else`:** binds to the *nearest* unmatched `if` (like C).

```
x = 5;
printInt(x);            // procedure call (return value ignored)
if (x > 0) { printInt(1); } else printInt(0);
```

Counted `for` Loop

Syntax: `for (int i = start, stop [, step]) Block` - `start` inclusive; `stop` **exclusive**. - `step` optional (default `1`); **must not be 0**; may be negative. - All three expressions are **evaluated once** before the loop. - `i` is scoped to the loop; **assigning to `i` inside the loop is illegal**.

```
int sum = 0;
for (int i = 0, 10) {      // i = 0..9
    sum = sum + i;
}
for (int j = 10, 0, -2) {  // j = 10,8,6,4,2
    printInt(j);
}
```

Functions

- **Definition order:** all functions on the same level are mutually visible (no forward decls needed).
- **Return rules:**
- `void` functions: `return;` only (optional).
- Non-void: every control path must end in `return <expr>` of matching type.
- **Procedure call:** call a function in statement position (discard result).

```
int inc(int x) { return x + 1; }
void shoutInt(int v) { printInt(v); }

export int main() {
  int v = inc(41);
  shoutInt(v); // procedure call; ignores any result
  return 0;
}
```

Scoping Rules (Important Gotcha)

- **Locals:** traditionally declared at function start; compilers may also allow mixed order.
- **Initializer scope:** the initializer **is not** in scope of the variable being declared.

```
int a = 1;
int b = a + 1;      // ok: reads outer a
int c = c + 1;      // ERROR in CiviC: initializer cannot see the new c
```

- **Functions vs variables:** separate name spaces; mutual recursion allowed for functions.

Comments

- Single-line and multi-line like C/C++: `// ...` and `/* ... */`.

Standard Library I/O (via `civic.h`)

Available functions: `printInt(int)`, `printFloat(float)`, `scanInt()`, `scanFloat()`, `printSpaces(int)`, `printNewlines(int)`.

```
#include "civic.h"
export int main() {
  printInt(42);
  printSpaces(1);
  printFloat(3.5);
  printNewlines(1);
  int x = scanInt();
  printInt(x);
  return 0;
}
```

Arrays (Extension 2)

- Vectors with element type in {`bool`, `int`, `float`}.
- Indexing `0..n-1`; index expr is `int`.
- **Carry their size**; locals may have dynamic size; memory managed automatically.
- **Pass-by-reference** to functions.
- **Initialization:** array literal `[e0, e1, ...]` or **scalar fill** with a single `Expr`.
- **Restrictions:** array values appear only in variable initializers; arrays cannot be returned.

```
void fillOnes(int[n] a) {     // n is available as a param
  for (int i = 0, n) a[i] = 1;
}

export int main() {
  int[5] v = [1,2,3,4,5];     // literal init
  int[3] w = 7;               // scalar fill -> [7,7,7]
  v[0] = 42;
  fillOnes(v);                // by reference
  return 0;
}
```

Declaration vs Definition Syntax: - Decls/params use **identifiers** for extents: `extern int[n] buf;` / `void f(int[m] x)`. - Defs/locals may use **full expressions** for extents: `int[2*size] tmp;`.

Multi-Dimensional Arrays (Extension 3)

- Use comma-separated extents and indices.
- Initialization uses nested brackets; or scalar fill replicates to all elements.

```
int[2,3] M = [[1,2,3],
              [4,5,6]];
M[1,2] = 99;            // row 1, col 2

void zero2d(int[r,c] A) {
  for (int i = 0, r)
    for (int j = 0, c)
      A[i,j] = 0;
}
```

Nested Functions (Extension 1)

- Functions may be defined inside a function body (after local var decls, before statements).
- Local functions are visible only within the outer function and can access its params/locals.
- Local functions at the same nesting level can call each other (order doesn't matter).

```
void outer(int a) {
  int base = a;
  void helper(int b) { printInt(base + b); }
  void twice() { helper(2); }
  helper(10);
  twice();
}
```

Complete Mini-Program

```
#include "civic.h"

int sum(int[n] xs) {
  int s = 0;
  for (int i = 0, n) s = s + xs[i];
  return s;
}

export int main() {
  int[5] a = [1,2,3,4,5];
  void showSum() { printInt(sum(a)); printNewlines(1); } // nested
  showSum();
  int[2,3] m = [[1,2,3],[4,5,6]];
  m[1,2] = 42;
  printInt(m[1,2]); printNewlines(1);
  return 0;
}
```

Quick Rules Recap

- **No implicit casts.**
- `for`: start inclusive, stop exclusive, step≠0, pre-evaluated once, `i` is read-only inside.
- **Initializers** can't see the variable being declared.
- **Functions**: mutual visibility; separate name space from variables.
- **Arrays**: pass-by-ref; literals only in inits; locals can be dynamic; cannot return arrays.