Neat is basically a smaller version of D1 with some experimental syntax and a focus on terseness without losing the basic C-like syntax.

Read more here.

```
// single line comments start with //
/*
  multiline comments look like this
*/
/+
  or this
  /+ these can be nested too, same as D +/
+/

// Module name. This has to match the filename/directory.
module LearnNeat;

// Make names from another module visible in this one.
import std.file;
// You can import multiple things at once.
import std.math, std.util;
// You can even group up imports!
import std.(process, socket);

// Global functions!
void foo() { }

// Main function, same as in C.
// string[] == "array of strings".
// "string" is just an alias for char[],
void main(string[] args) {
  // Call functions with "function expression".
  writeln "Hello World";
  // You can do it like in C too... if you really want.
  writeln ("Hello World");
  // Declare a variable with "type identifier"
  string arg = ("Hello World");
  writeln arg;
  // (expression, expression) forms a tuple.
  // There are no one-value tuples though.
  // So you can always use () in the mathematical sense.
  // (string) arg; <- is an error

  /*
    byte: 8 bit signed integer
      char: 8 bit UTF-8 byte component.
    short: 16 bit signed integer
    int: 32 bit signed integer
    long: 64 bit signed integer

    float: 32 bit floating point
    double: 64 bit floating point
    real: biggest native size floating point (80 bit on x86).

    bool: true or false
```

```
*/
int a = 5;
bool b = true;
// as in C, && and || are short-circuit evaluating.
b = b && false;
assert(b == false);
// "" are "format strings". So $variable will be substituted at runtime
// with a formatted version of the variable.
writeln "$a";
// This will just print $a.
writeln `$a`;
// you can format expressions with $()
writeln "$(2+2)";
// Note: there is no special syntax for characters.
char c = "a";
// Cast values by using type: expression.
// There are three kinds of casts:
// casts that just specify conversions that would be happening automatically
// (implicit casts)
float f = float:5;
float f2 = 5; // would also work
// casts that require throwing away information or complicated computation -
// those must always be done explicitly
// (conversion casts)
int i = int:f;
// int i = f; // would not work!
// and, as a last attempt, casts that just reinterpret the raw data.
// Those only work if the types have the same size.
string s = "Hello World";
// Arrays are (length, pointer) pairs.
// This is a tuple type. Tuple types are (type, type, type).
// The type of a tuple expression is a tuple type. (duh)
(int, char*) array = (int, char*): s;
// You can index arrays and tuples using the expression[index] syntax.
writeln "pointer is $(array[1]) and length is $(array[0])";
// You can slice them using the expression[from .. to] syntax.
// Slicing an array makes another array.
writeln "$(s[0..5]) World";
// Alias name = expression gives the expression a name.
// As opposed to a variable, aliases do not have an address
// and can not be assigned to. (Unless the expression is assignable)
alias range = 0 .. 5;
writeln "$(s[range]) World";
// You can iterate over ranges.
for int i <- range {
  write "$(s[i])";
}
writeln " World";
// Note that if "range" had been a variable, it would be 'empty' now!
// Range variables can only be iterated once.
// The syntax for iteration is "expression <- iterable".
// Lots of things are iterable.
for char c <- "Hello" { write "$c"; }
writeln " World";
```

```
// For loops are "for test statement";
alias test = char d <- "Hello";
for test write "$d";
writeln " World\t\x05"; // note: escapes work
// Pointers: function the same as in C, btw. The usual.
// Do note: the pointer star sticks with the TYPE, not the VARIABLE!
string* p;
assert(p == null); // default initializer
p = &s;
writeln "$(*p)";
// Math operators are (almost) standard.
int x = 2 + 3 * 4 << 5;
// Note: XOR is "xor". ^ is reserved for exponentiation (once I implement that).
int y = 3 xor 5;
int z = 5;
assert(z++ == 5);
assert(++z == 7);
writeln "x $x y $y z $z";
// As in D, ~ concatenates.
string hewo = "Hello " ~ "World";
// == tests for equality, "is" tests for identity.
assert  (hewo == s);
assert !(hewo is s);
// same as
assert  (hewo !is s);

// Allocate arrays using "new array length"
int[] integers = new int[] 10;
assert(integers.length == 10);
assert(integers[0] == 0); // zero is default initializer
integers = integers ~ 5; // This allocates a new array!
assert(integers.length == 11);

// This is an appender array.
// Instead of (length, pointer), it tracks (capacity, length, pointer).
// When you append to it, it will use the free capacity if it can.
// If it runs out of space, it reallocates - but it will free the old array automatically.
// This makes it convenient for building arrays.
int[auto~] appender;
appender ~= 2;
appender ~= 3;
appender.free(); // same as {mem.free(appender.ptr); appender = null;}

// Scope variables are automatically freed at the end of the current scope.
scope int[auto~] someOtherAppender;
// This is the same as:
int[auto~] someOtherAppender2;
onExit { someOtherAppender2.free; }

// You can do a C for loop too
// - but why would you want to?
for (int i = 0; i < 5; ++i) { }
// Otherwise, for and while are the same.
while int i <- 0..4 {
```

```
  assert(i == 0);
  break; // continue works too
} then assert(false); // if we hadn't break'd, this would run at the end
// This is the height of loopdom - the produce-test-consume loop.
do {
  int i = 5;
} while (i == 5) {
  assert(i == 5);
  break; // otherwise we'd go back up to do {
}

// This is a nested function.
// Nested functions can access the surrounding function.
string returnS() { return s; }
writeln returnS();

// Take the address of a function using &
// The type of a global function is ReturnType function(ParameterTypeTuple).
void function() foop = &foo;

// Similarly, the type of a nested function is ReturnType delegate(ParameterTypeTuple).
string delegate() returnSp = &returnS;
writeln returnSp();
// Class member functions and struct member functions also fit into delegate variables.
// In general, delegates are functions that carry an additional context pointer.
// ("fat pointers" in C)

// Allocate a "snapshot" with "new delegate".
// Snapshots are not closures! I used to call them closures too,
// but then my Haskell-using friends yelled at me so I had to stop.
// The difference is that snapshots "capture" their surrounding context
// when "new" is used.
// This allows things like this
int delegate(int) add(int a) {
  int add_a(int b) { return a + b; }
  // This does not work - the context of add_a becomes invalid
  // when add returns.
  // return &add_a;
  // Instead:
  return new &add_a;
}
int delegate(int) dg = add 2;
assert (dg(3) == 5);
// or
assert (((add 2) 3) == 5);
// or
assert (add 2 3 == 5);
// add can also be written as
int delegate(int) add2(int a) {
  // this is an implicit, nameless nested function.
  return new  (int b) { return a + b; }
}
// or even
auto add3(int a) { return new  (int b) -> a + b; }
```

```
// hahahaaa
auto add4 = (int a) -> new (int b) -> a + b;
assert(add4 2 3 == 5);
// If your keyboard doesn't have a   (you poor sod)
// you can use \ too.
auto add5 = \(int a) -> new \(int b) -> a + b;
// Note!
auto nestfun = () { } // There is NO semicolon needed here!
// "}" can always substitute for "};".
// This provides syntactic consistency with built-in statements.


// This is a class.
// Note: almost all elements of Neat can be used on the module level
//       or just as well inside a function.
class C {
  int a;
  void writeA() { writeln "$a"; }
  // It's a nested class - it exists in the context of main().
  // so if you leave main(), any instances of C become invalid.
  void writeS() { writeln "$s"; }
}
C cc = new C;
// cc is a *reference* to C. Classes are always references.
cc.a = 5; // Always used for property access.
auto ccp = &cc;
(*ccp).a = 6;
// or just
ccp.a = 7;
cc.writeA();
cc.writeS(); // to prove I'm not making things up
// Interfaces work same as in D, basically. Or Java.
interface E { void doE(); }
// Inheritance works same as in D, basically. Or Java.
class D : C, E {
  override void writeA() { writeln "hahahahaha no"; }
  override void doE() { writeln "eeeee"; }
  // all classes inherit from Object. (toString is defined in Object)
  override string toString() { return "I am a D"; }
}
C cd = new D;
// all methods are always virtual.
cd.writeA();
E e = E:cd; // dynamic class cast!
e.doE();
writeln "$e"; // all interfaces convert to Object implicitly.

// Templates!
// Templates are parameterized namespaces, taking a type as a parameter.
template Templ(T) {
  alias hi = 5, hii = 8;
  // Templates always have to include something with the same name as the template
  // - this will become the template's _value_.
  // Static ifs are evaluated statically, at compile-time.
```

```
    // Because of this, the test has to be a constant expression,
    // or something that can be optimized to a constant.
    static if (types-equal (T, int)) {
      alias Templ = hi;
    } else {
      alias Templ = hii;
    }
  }
  assert(Templ!int == 5);
  assert(Templ!float == 8);
}
```

Topics Not Covered

- Extended iterator types and expressions
- Standard library
- Conditions (error handling)
- Macros