Cmajor Language Reference

This guide aims to be a technical reference guide for all aspects of the Cmaj language.

It's written for developers who have a decent level of familiarity with common languages and the concepts involved. Beginners who are looking for a more friendly "getting started" tutorial should probably start with the example code walkthroughs.

Cmaj was designed with these main goals:

- To allow simple procedural DSP code to be easily composed into graph structures
- To offer performance that *at least* matches C/C++
- To make it impossible to write code that can crash or break real-time safety rules
- To use a simple, familiar syntax which will be very easily learned by anyone who's dabbled with C/C++, javascript or other common languages

Lexical Rules

Whitespace

Whitespace in Cmaj code is ignored by the parser, except where it's needed to separate tokens that would otherwise be ambiguous. Tabs and spaces are treated equally.. but we encourage you to make the smart choice and use spaces for your indentation rather than tabs :)

Comments

Cmaj uses the good old familiar C-family-style comment that we all know and love:

```
/* Multi-line comments

    use the slash-star syntax.. */
```

```
// Single-line comments use double-slashes
```

Identifiers

The names of variables, types, functions, etc. are limited to the following characters: `A-Z a-z 0-9 _` Names must begin with a letter - that cannot start with a number or an underscore. The compiler is case-sensitive when matching names.

No restrictions are placed on how you use upper/lower case letters, but the in-house style we use for naming Cmaj objects is:

- Variables (local or global): `lowerCamelCase`
- Types, processors, structures: `UpperCamelCase`
- Namespaces: `snake_case`

Reserved keywords

The following words are reserved for language constructs (including future ones), so cannot be used as user-defined names:

```
bool, break, case, catch, class, complex, complex32, complex64, connection, const, continue, default, do,
double, else, enum, event, external, false, fixed, float, float32, float64, for, graph, if, import, input,
int, int32, int64, let, loop, namespace, node, operator, output, private, processor, public, return,
string, struct, switch, throw, true, try, using, var, void, while
```

Literals

The rules for literals follow common conventions from the C/C++/Javascript family, and shouldn't cause many surprises.

Integer Literals

32-bit signed integers are written without a suffix:

```
-12345      // 32-bit signed decimal
0x12345     // 32-bit hex
0b101101    // 32-bit binary
```

64-bit integers have the suffix `L` or `_L` or `i64` or `_i64` (NB: lower-case `l` is not allowed as it's too similar to a 1)

```
12345L      // 64-bit decimal    (NB: lower-case 'l' is not allowed, since it looks too much like '1')
12345_i64   // 64-bit decimal
0x12345_L   // 64-bit hex
0b101101L   // 64-bit binary
```

Currently only signed integers are supported, and will use a twos-complement representation.

If a signed integer overflow occurs, the precise result will depend on the CPU, the runtime, and any compiler optimisations that were applied to the code while being built. It will however never cause an error or exception.

Boolean Literals

`true` and `false`. These must be lower-case.

Floating-point Literals

Floating point literals must contain a point. 64-bit floats are written either without a suffix or using `f64` or `_f64`, e.g. `1234.0` or `1234.0_f64`. 32-bit floats can use the suffix `f` or `f32` or `_f32`, e.g `1234.0f` or `1234.0_f32`.

The imaginary part of a complex number can be written with the suffix `i` or `_i` or `f32i`, e.g. `123.0i` or `123.0fi`. You can add imaginary literals to normal integers to form complete complex numbers, e.g. `(2.5 + 0.5i)`

String Literals

String literals are written with double-quotes, and escape characters follow the JSON rules, e.g. `string myString = " Hello\n World\n \uD83D\uDE00";`

Aggregate Literals

Data for initialising a structure or array is written in parentheses, e.g.

```
int[5] x = (2, 3, 4, 5, 6);
MyStruct y = (3, 6.5f, "hello", (3, 4, false));
```

```
var z = bool<4> (true, false, false, true);
```

Null Literals

Any type of null or zero value can be represented with an empty pair of parentheses `()` e.g.

```
var x = int[5](); // creates an array of 5 zeros
x = (); // sets all 5 elements of x to 0

var y = MyStruct(1, "hello", 3); // creates an object of type MyStruct.
y = ();   // resets all elements of the object to zero or null values

int[] slice = (1, 2, 3); // creates a slice with 3 elements
slice = ();  // resets the variable to be an empty slice with size 0

float64 i = 123.0;
i = (); // this syntax works for numeric types too, which can be useful in generic code
```

Types

Primitive Types

The primitive types are:

- `int32` (or just `int`) - a signed 32-bit integer
- `int64` - a signed 64-bit integer
- `float32` (or just `float`)
- `float64`
- `complex32` (or just `complex`)
- `complex64`
- `bool`

Limited-range Integer Types

`wrap<N>` and `clamp<N>` represent 32-bit integers which can only hold values between 0 and N - 1.

The value of `N` must be a compile-time constant.

When values are assigned to a wrap type that would exceed its size, they are automatically limited: a `wrap` type will apply a modulo operation to keep it in range, and a `clamp` type will stop at the end.

```
wrap<5> w;
clamp<5> c;

loop (7)
{
    ++w;
    ++c;
}

// after being incremented 7 times, w == 2 and c == 4

w = 4 - 5;  // w == 4
c = 4 - 5;  // c == 0
```

Having limited-range integers is useful for allowing safe but efficient array indexing. There is also a version of the `for` loop syntax that uses a `wrap` type as a counter variable – see the section on loops for more details.

Complex Numbers

There are built-in types for 32 or 64-bit complex numbers. The types are `complex32`, `complex64` and `complex` (which is the same as `complex32`). Imaginary literals are declared with an `i` suffix.

```
// These 3 statements are different ways to create the same complex value:
complex32 c1 = 2.0fi + 3.0f;
let c2 = 3.0f + 2.0fi;
let c3 = complex (3.0f, 2.0f);

let c4 = complex (4.0);  // creates a value (4 + 0i)

let ci = c1.imag;  // extracts the imaginary part a complex number
let cr = c1.real;  // extracts the real part a complex number
```

```
complex64<4> v = (2.0i + 5.0);  // declares a vector of 4 complex 64-bits numbers.

let r = v.real;     // extracts the real elements from the vector so has type float64<4>
```

Arrays

Arrays use square-bracket syntax, e.g.

```
int[3] x;       // an array of 3 integers
float64[6] y;   // an array of 6 64-bit floats
MyStruct[4] z;  // an array of 4 'MyStruct' objects
```

NB: If you are a C/C++ programmer, notice that the brackets are associated with the type not following the variable name.

They can be declared and initialised in various ways:

```
let x = int[4] (1, 2, 3, 4);    // these all declare an array containing 1, 2, 3, 4
let x = int[] (1, 2, 3, 4);
int[] x = (1, 2, 3, 4);

int[4] y;           // these are all zero-initialised
int[4] y = ();
var y = int[4]();
```

The size of an array must be a compile-time constant, and arrays cannot be re-sized. You can read the size of an array using the `.size` pseudo-property.

Use the familiar `[]` operator to get an element from an array.

If the compiler can't prove that the element index is guaranteed to be in-range, it will generate code to wrap any out-of-bounds index values to within the array size at runtime. When the compiler does this, it'll emit a warning: `Performance warning: the type of this array index could not be proven to be safe, so a runtime check was added.`

To avoid this warning, either make sure that you pass a `wrap<N>` type as your index (where `N` is less than the size of the array), or a constant. Because a `wrap` type can never be out-of-range, this avoids any runtime overhead. Or, if you don't mind the fact that the index is being

checked, and you want to hide the warning, use the `.at(index)` operator instead of `[]` to read the element.

```
int[8] x;

let sizeOfX = x.size;      // This is 8

let a1 = x[3];             // OK
let a2 = x[-1];            // OK: this returns the last element of the array
let a3 = x[10];            // Error: the compiler sees that this constant integer is out of bounds.

wrap<8> wrappedInt = ...   // 'wrappedInt' is a wrap<8> so has the range 0 to 7
int normalInt = ...        // 'normalInt' is a normal integer so could have any value

let a4 = x[wrappedInt];    // This produces efficient code because the compiler knows that the wrap<8>
                           // value can never exceed the bounds of an array with size 8.
let a5 = x[normalInt];     // This compiles, but will emit a performance warning because the compiler
                           // needs to insert a wrap operation to make sure the integer index is in-range.
let a6 = x.at(normalInt);  // Using .at() instead of [] tells the compiler not to emit a performance warni
let a7 = x[wrap<8> (normalInt)];  // Casting the integer to a wrap<8> also removes the performance warning
```

As well as reading elements of arrays, you can extract or modify sub-regions:

```
int[8] x;

x = 7;       // Sets all elements of x to 7
x[:] = 7;    // Sets all elements of x to 7
x[3:5] = 7;  // Sets elements 3 and 4 to 7

let a1 = x[1:5];   // Returns a copy of elements 1 to 4
let a1 = x[3:];    // Returns a copy of elements 3 to 7
let a3 = x[:3];    // Returns a copy of elements 0 to 2
let a4 = x[:-1];   // Returns a copy of all elements apart from the last one
let a5 = x[3:-2];  // Returns a copy of elements 3 to 5

z[3:6] = y[5:8];  // If the sizes and types match, you can copy sub-sections between arrays
```

Multi-Dimensional Arrays

Multi-dimensional arrays can be declared with a list of comma-separated sizes inside the square brackets:

```
int[3, 4, 5] x;

x[1, 2, 3] = 99;
```

An alternative syntax is to use multiple nested brackets:

```
int[5][4][3] x;  // same as int[3, 4, 5]

x[1][2][3] = 99;   // same as x[1, 2, 3]
x[1, 2][3] = 99;   // same as x[1, 2, 3]
x[1][2, 3] = 99;   // same as x[1, 2, 3]
```

You can access one of the inner dimensions as a sub-array by indexing into it with fewer arguments than there are dimensions:

```
int[3, 4, 5] x;

let y = x[1];     // the type of y is int[4, 5]
let z = x[1, 2];  // the type of z is int[5];
```

Arrays vs Slices

Arrays are copied around by value, but often you want to work with references to sections of arrays. This is done with *slices* (also sometimes called “fat pointers” in other languages).

A slice type is written as an array without a size, e.g.

```
int[] x;   // a slice
int[3] y;  // an array

// Functions can take slices as parameters
void myFunction (int[] x) { ...
```

Unlike an array, a slice can be empty, and can be assigned new values that point to different targets and sizes, e.g.

```
int[4] originalArray = (1, 2, 3, 4);

int[4] arrayCopy  = originalArray; // creates a copy of the original
int[]  arraySlice = originalArray; // creates a slice of the original

console <- arrayCopy[1]  <- " "    // prints 2
        <- arraySlice[1] <- " ";   // prints 2

originalArray[1] = 456; // modifying the original array

console <- arrayCopy[1]  <- " "    // prings 2
        <- arraySlice[1] <- " ";   // prints 456

arraySlice = (); // sets the slice to be empty

console <- arraySlice.size <- " "   // prints 0
        <- arraySlice[1]   <- " ";  // prints 0

arraySlice = originalArray[2:4]; // now make our slice point at the last two elements

console <- arraySlice.size <- " " // prints 2
        <- arraySlice[1];         // prints 4
```

Slices of multi-dimensional arrays are not yet supported.

Scope of slice data

Because slices are *references* to another array, there are some rules in place to stop you creating dangling pointers to arrays that have gone out of scope.

- If you create a slice of an array which is a global constant, or a processor state variable, then it can be used anywhere.
- If you create a slice of a local array, the slice cannot be assigned to a global variable or returned from a function.

- When a function takes slices in any of its parameters, all calls to the function are checked to see whether it can ever receive local slice data. If so, the slice parameters are treated as having local scope, so they cannot be assigned to global variables or returned. If all calls to the function are proven to only provide it with global slice data, then those parameters can be used for any purpose.

The compiler is fairly conservative when it comes to deciding what is safe, so sometimes a legitimate use of a slice may be flagged as illegal!

Vectors

Vectors are similar to arrays, but with a few differences:

- The maximum number of elements is small (this is platform-specific but is unlikely to be more than about 128)
- Vectors can only contain primitive numeric types such as integers, floats, bools or complex numbers.
- They support parallel mathematical operators which are applied to all their elements
- On suitable hardware, vector operations may be implemented with SIMD instructions for better performance

Vectors use an angle-bracket syntax, e.g.

```
int<4> x = (1, 2, 3, 4);     // a vector of 4 int32s
var x = int64<2> (7L, 8L);   // a vector of 2 int64s
let f = float<8> (2.0f);     // creates a vector of 8 floats, all with the value 2.0f

let y = x[2];                // retrieve an index from the vector, y = 3
let g = f[2:4];              // you can copy vector ranges, g is of type float<2> with value (2.0f, 2.0f)

int<> h;                     // syntax error - you cannot create vector slices

let s = sum (x);             // s = 10
                             // The standard library includes sum() and product() functions to produce
                             // sum and products of vector and array types
```

Structures

Structs are declared in the familiar old C-style of `<type> <name> [, <name>];`

Members cannot have initialiser values (maybe this will be added in a future version of the language), but are all initialised to zero when a struct is created.

```
struct ExampleStruct
{
    int<5>       member1, member2;
    float[]      thisMemberIsASlice;
    float64[4]   thisMemberIsAnArray;
    int64        thisMemberIsABigInteger;
    OtherStruct  anotherEmbeddedStructMember;
}
```

NB: if you're a C/C++ programmer, note the lack of a semi-colon after the brace!

To create an object with a struct type, just use its name as a type:

```
struct Position { float x, y; }

Position getMovedPosition (Position p)
{
    Position newPos; // creates a zero-initialised object of type Position
    newPos.x = p.x + 10.0f;
    newPos.y = p.y - 5.0f;
    return newPos;
}
```

You can also initialise a struct using a parenthesised list of its member values, e.g.

```
struct Position { float x, y; }

Position getMovedPosition (Position p)
{
    let newPos = Position (p.x + 10.0f,
                           p.y - 5.0f);
    return newPos;
}
```

And where the target type is known, you can also implicitly create a struct object from a list of values, e.g.

```
struct Position { float x, y; }

Position getMovedPosition (Position p)
{
    return (p.x + 10.0f,
            p.y - 5.0f);
}
```

You can also declare member functions inside structs: see the later section about functions for more details.

String Type

The language supports very limited use of strings, so the `string` type is essentially a token representing a read-only string literal. They can be passed around, but no concatenation or other run-time mutations are supported, as these would require dynamic memory allocation.

Enums

An enum can be declared with the syntax:

```
enum Animal
{
    cat, dog
}

let c = Animal::cat;
let d = Animal::dog;

void isThisACat (Animal a)    { return a == Animal::cat; }
```

Enums are strong, abstract types, and cannot be cast to/from integers. (This might be a feature that is added in future versions of the language, but for now if you want to declare a constant which must have a specific integer value, just declare it as a `const int` rather than an `enum`).

Constant Types

A type can be marked as constant by prefixing it with the `const` keyword.

Reference Types

A reference type is declared by adding the `&` suffix to a type.

```
struct MyObject
{
    float y;
}

void mangle (int& x, const MyObject& o)
{
    x = 2;
    o.y = 3.0;
}

int x = 1;
MyObject o;
mangle (x, o);
console <- x <- o.y;   // prints "2, 3.0"
```

Note that the compiler is restrictive on where references may be used, because it must be able to check at compile time that no dangling references are possible. Currently references are only permitted in function parameters, but this rule may be relaxed in the future as more permissive lifetime checking is added.

Type Aliases

The `using` keyword lets you declare an alias for a type.

```
using MyInt = int64;
using VectorOfInts = int<4>;
using MyThingArray = some_namespace::Thing[10];
```

Type Metafunctions

A set of basic operations are provided for interrogating and getting variations on types at compile time.

These can all be written as free functions, or using dot notation, e.g.

```
using MyArrayType = int64[10];
using T1 = MyArrayType.elementType;    // T1 is now int64.
using T2 = elementType (MyArrayType);  // (alternative syntax)
int[MyArrayType.size] x; // x is an array of 10 ints
```

Metafunctions can be called with either a type or a value as their argument, e.g.

```
using T = int32[4];
T x;                // declares x as an int32[4]
x.elementType[5] y; // declares y as an int32[5]
static_assert (MyType.isFixedSizeArray, "MyType must be a fixed-size array!")
```

Available meta-functions are:

Function	Description
`size`	For an array or vector, this returns the number of elements
`type`	Returns the type of its argument. Can be helpful in metaprogramming situations.
`makeConst`	Returns a const version of a type
`removeConst`	Returns a non-const version of a type
`makeReference`	Returns a reference version of a type
`removeReference`	Returns a non-reference version of a type
`elementType`	If the type is an array or vector, this extracts the element type
`primitiveType`	If the type is a primitive type or a vector, this returns the primitive type
`isStruct`	Returns true if its argument is a struct
`isArray`	Returns true if its argument is an array or slice
`isSlice`	Returns true if its argument is a slice
`isFixedSizeArray`	Returns true if its argument is a fixed-size (non-slice) array
`isVector`	Returns true if its argument is a vector
`isPrimitive`	Returns true if its argument is a primitive type

Function	Description
`isFloat`	Returns true if its argument is a float32 or float64
`isFloat32`	Returns true if its argument is a float32
`isFloat64`	Returns true if its argument is a float64
`isInt`	Returns true if its argument is an int32 or int64
`isInt32`	Returns true if its argument is an int32
`isInt64`	Returns true if its argument is an int64
`isScalar`	Returns true if its argument is a scalar type: i.e. an int, float, or vector of int or float
`isString`	Returns true if its argument is a string
`isBool`	Returns true if its argument is a bool
`isComplex`	Returns true if its argument is a complex
`isReference`	Returns true if its argument is a reference
`isConst`	Returns true if its argument is a const

Functions

Function syntax uses the trusty old C/C++/Java/C# format:

```
void doSomething (int parameter1, bool parameter2)
{
    // ...
}

float calculateAverage (float f1, float f2)
{
    return (f1 + f2) / 2.0f;
}
```

Cmajor supports function overloading, that is, multiple functions can be declared with the same name, so long as their arguments are different.

```
void handleMessage(std::notes::NoteOn n){}
```

```
void handleMessage(std::notes::NoteOff n){}
```

Overloaded functions enable you to supply different semantics for a function, depending on the types and number of arguments.

Cmaj doesn't currently support default values for function parameters.

Member Functions

Member functions for structures can be written in two ways:

```
struct Thing
{
    float a, b;

    // You can declare a function inside the struct, and use the
    // special variable 'this' to  refer to  the object.
    // Note that the 'const' keyword optionally can be appended in the
    // same way as C++ to allow the member to be called on a const object.
    float getBiggest() const   { return max (this.a, this.b); }
}

struct Thing
{
    float a, b;
}

// Any function which takes a struct as its first argument is treated as
// a member function.
float getBiggest (const Thing& t)    { return max (t.a, t.b); }
```

The two styles of declaration are equivalent. Either one can be invoked either using the dot operator, or as a free function with the object as its first argument:

```
Thing t;
let biggest1 = t.getBiggest();   // these two calls do
let biggest2 = getBiggest (t);   // exactly the same thing
```

Special Functions

`advance()`

The `advance()` function is a special built-in function which moves time forward by one frame for the processor that calls it. `advance()` may only be called from `main()` (or from functions which are exclusively called from `main()`).

`void main()`

A processor must declare a function called `main` which returns `void` and takes no arguments.

The job of the `main` function is to read from the processor's input streams (if it has any), do whatever processing is needed to that data, and then to write some kind of data to the processor's outputs.

It is also responsible for repeatedly calling `advance()` to step the processor forward to the next frame.

Most `main` functions will be written as an infinite loop, but the function is allowed to return if it doesn't need to do any more processing (however, once the function exits, it can't be re-started within this run of the program).

```
// A processor that adds 1 to each value that passes through it
processor AddOne
{
    input stream int in;    // an input stream of integers
    output stream int out;  // an output stream of integers

    void main()
    {
        loop
        {
            out <- in + 1;  // reads the next value from 'in', adds 1, and writes the result to 'out'
            advance();  // moves forward to the next frame
        }
    }
}
```

A processor which only has `event` endpoints, and which performs all its work in response to event callback functions is allowed to skip having a `main` function.

```
void init()
```

If you have a large amount of setup work to be done before the processor starts, you can add an `init()` function to your processor, and this is called at initialisation time, outside of the normal real time callback.

For very simple setup tasks, this can be done at the start of your `main()` function.

Note that an `init()` method can't do any work which involves endpoints, so it can't call `advance()`. But processor properties (such as the `frequency` and `id` are available).

Recursion

Recursion isn't allowed! (Well, not at the moment, at least...)

This is a deliberate policy: without recursion, the compiler can determine the maximum stack size needed by a program, and avoid the overhead of runtime checking and reallocations/errors if there's an overflow.

> Note that it *is* legal to write a generic function which calls a version of itself with different specialisation parameters. This loophole does allow finite recursion to be done if written cunningly, and we use it in some of our library code.

Generic functions

Generic functions are written by appending one or more template names in angle-brackets after the function name:

```
Type add<Type> (Type a, Type b)   { return a + b; }

let x = add (1, 2); // x has type int32
let y = add (1.0f, 2.0f);  // y has type float
let z = add (1, "nope") // error! can't resolve the 'Type' template!

// You can use the templates within more complex type declarations such as
// arrays, vectors or references, e.g.
void myFunction<T1, T2> (const T1& a, T2[3] b, T1<4> c)   {  ...  }
```

The compiler will attempt to pattern-match all of the template parameters against the function parameters and the function's return type, and if successful, will generate a specialised version of the function for the resulting types.

It is also often handy to apply metafunctions and `static_assert`s to generic types, e.g.

```
Type.elementType getFirstElement<Type> (Type array)
{
    static_assert (Type.isArray, "The argument supplied to this function must be an array!");
    return array[0];
}

console <- getFirstElement (int[] (2, 3, 4));  // prints "2"
console <- getFirstElement (123.0f);  // An error is thrown by the static_assert failing
```

Variables and Constants

Local Variables and Parameters

Local variables and parameters are declared in a way that will cause no surprises for anyone who's written any C/C++/C#/Java/Javascript/etc.

The `let` and `var` keywords are used to declare auto-deduced constant and mutable variables.

```
void myFunction (int param1, const float param2)
{
    int64 a = 1;         // a is a mutable int64
    const int c = 2;     // c is a const int32
    let b = 2;           // b is a const int32
    var d = 2;           // d is a mutable int32
    var e = bool[10]();  // e is a mutable array of 10 bools
    bool[10] f;          // f is a mutable array of 10 bools
    complex64 g;         // g is a complex number, initialised to zero
    MyStruct h;          // h is an object, with all its fields zero-initialised.
}
```

See also the sections on literals and types for details on the syntax for initialising values.

Processor State Variables

A processor can also contain variables whose state persists over the lifetime of an instance of the processor, and which can be modified by any of the processor's functions. (If you're familiar with object-oriented programming, these are a *bit* like member variables in a class).

These are declared inside a processor's definition:

```
processor NumberGenerator
{
    output stream float out;

    // These are all processor state variables.
    float value;
    int counter = 10;
    let increment = 2.5f; // this one is a constant

    void main()
    {
        value = 100.0f;

        while (--counter > 0)
        {
            emitNextNumber();
            advance();
        }
    }

    void emitNextNumber()
    {
        // Any function in a processor can read and modify its state variables
        out <- value;
        value += increment;
    }
}
```

State variables may have an initial value provided, which will be applied when the processor instance is created. Any variables without an initial value are set to zero.

Global Constants

Variables with a constant value that is known at compile time can also be declared inside namespaces.

```
namespace N
{
    let x = 1234; // ok
    const int[4] y = (1, 2, 3, 4);
    int z;  // error! This would be OK in a processor, but a namespace can only contain constants
}
```

`external` Constants

The `external` keyword can be applied to variables declared in any namespace or processor. They cannot have an initial value, because their value will be supplied by the hosting environment that loads the program. For patches, see the patch specification document for details of how external data can be provided.

```
namespace N
{
    external int[] someData; // The contents of this array will be supplied by whatever runtime is loading
}

void myFunction()
{
    let x = N::someData[3];   // externals can be referenced from anywhere in the program as global consta
}
```

External values are implicitly constant, so you don't need to add a `const` to their type.

Control-flow and Loops

`while`

It's a classic. Cmaj supports the good old trusty while loop:

```
int i = 0;
while (i < 5)
{
    console <- i;   // prints 0, 1, 2, 3, 4
    ++i;
}
```

Note that rather than writing `while (true)`, you're encouraged to use the `loop` keyword to avoid the redundant conditional expression.

`for`

The `for` loop construct follows the tried-and-tested C/C++/Java/Javascript syntax:

```
for (<initialiser>; <condition>; <iterator>) <loop body>
```

```
for (int i = 0; i < 5; ++i)
    console <- i;             // prints 0, 1, 2, 3, 4

for (;;)        // an infinite loop
   advance();

int i = 0;
for (; i < j; ++i)
    console <- i;             // prints 0, 1, 2, 3, 4
```

In addition, `for` supports a syntax for visiting the values in a range type

```
for (wrap<5> i)
    console <- i;               // prints 0, 1, 2, 3, 4

for (clamp<5> i = 2)            // you can set an initial value
    console <- i;               // prints 2, 3, 4
```

`loop`

The `loop` statement can be used for infinite loops, or a fixed number of iterations.

```
loop { advance(); }   // infinite loop


int i = 0;
loop (5)
    console <- ++i; // prints 1, 2, 3, 4, 5
```

`break` / `continue`

Unsurprisingly, `break` jumps out of the current loop, and `continue` returns to the start of the current loop.

Cmaj also supports the use of target labels for `break` and `continue`, to escape from a named parent block. For this, a block must be given a label, e.g.

```
my_outer_loop: loop (100) // prefixing a `loop` or `for` block with a label gives it a name
{
    loop (200)
    {
        break; // this would escape the inner loop

        break my_outer_loop; // this will escape from the outer loop

        continue my_outer_loop; // this will jump back to the start of the outer loop
    }
}
```

You can also use break to jump forwards out of a normal block if it has a label:

```
my_block:
{
    break my_block; // this will jump forwards to the statement after the block

    // any code here will be skipped
}

// execution resumes here
```

Conditions

In Cmaj, `if` / `else` statements shouldn't provide any surprises:

```
if (x == 1)
    doSomething();
else if (x == 2)
    doSomethingElse();
else if (x == 3)
    doSomethingElseAgain();
```

Ternaries are also supported. They use short-circuiting semantics, so only one of the branches of the ternary will have its code executed.

```
let x = b ? getValueIfTrue() : getValueIfFalse();
```

`if const`

Cmaj supports compile-time conditionals with the `if const` syntax:

```
void genericFunction<Type> (Type x)
{
    if const (x.isArray)
        doSomething (x[1]);  // this code will only parsed if x is an array
    else if const (x.isFloat)
        doSomething (x + 3.5f);   // this code will only be parsed if x is a float
}
```

`if const` is very handy in generic functions where different pieces of code can be used for a variable depending on its type.

Arithmetic Operators

Cmaj supports the following binary operators:

Operator	Description
`+`	Add

Operator	Description
`-`	Subtract
`*`	Multiply
`/`	Divide
`%`	Modulo
`++`	Pre/post increment
`--`	Pre/post decrement
`**`	Exponentiation
`&`	Bitwise AND
`|`	Bitwise OR
`^`	Bitwise XOR
`<<`	Bitwise left shift
`>>`	Bitwise right shift
`>>>`	Bitwise unsigned right shift
`&&`	Logical AND
`||`	Logical OR
`<`	Less than
`<=`	Less than or equal
`>`	Greater than
`>=`	Greater than or equal
`==`	Equal
`!=`	Not equal

And the following unary operators:

Operator	Description
`-`	Numeric negation
`!`	Logical (boolean) NOT
`~`	Bitwise NOT

Note that these operators can all be applied to both scalars and vectors. When vectors are provided, the result will be a vector.

Casts

Casts are written in a functional style:

```
let x = int (2.5);  // x has value 2 (int32)
let y = float<3> (int<3> (1, 2, 3));   // y is a float<3> vector 1.0f, 2.0f, 3.0f
let z = int<3> (3);   // z is (3, 3, 3)
```

Namespaces, Processors and Graphs

At the top level, a program consists of a set of `processor`, `graph` or `namespace` declarations.

Namespaces

Types, processors, graphs, variables and functions are declared inside namespaces, as is standard in most modern languages.

When writing a qualified name to indicate where a symbol is found, use a double-colon `::` as the separator:

```
namespace N1
{
    namespace N2
    {
        int myFunction() { ... }
        let myConstant = 1234;
    }

    namespace N3
    {
        void myOtherFunction()    { let x = N2::myFunction() + N2::myConstant; }
    }
```

```
}

void yetAnotherFunction()    { let x = N1::N2::myFunction() + N1::N2::myConstant; }
```

Namespaces can contain:

- Other namespace declarations
- `processor` or `graph` declarations
- Function definitions
- Global constant variables
- `struct` and `using` type declarations

```
namespace animals
{
    namespace dogs
    {
        processor Woof
        {
            ...
        }

        string getName()    { return "dog"; }
    }

    namespace cats
    {
        processor Miaow
        {
            ...
        }

        string getName()    { return "cat"; }

        struct Cat
        {
            string name, breed;
            float scratchiness;
```

```
        }

        using CatType = Cat;
    }
}

animals::dogs::Woof  // The double-colon separator is used when referring to a namespace path
animals::dogs::getName()  // returns "dog"
animals::cats::getName()  // returns "cat"
```

When declaring namespaces, you can combine nested declarations into a single `namespace` statement like this:

```
namespace A
{
    namespace B
    {
        namespace C
        {
            void myFunction() {}
        }
    }
}

// The above declaration can be written like this:

namespace A::B::C
{
    void myFunction() {}
}
```

Processors

A processor is an execution unit which takes streams of input and generates output.

A `processor` declaration contains:

- A list of input and output endpoint declarations

- Functions, including a `main()` function
- Types (`struct`s, `using` declarations)
- Processor state variables: variables that are used for the lifetime of a processor instance

```
processor MyProcessor
{
    // Input and output endpoints are always declared first in the processor
    output event   int   myOutput;
    input  stream  float myInput1;
    input  value   bool  myInput2;

    // then you can declare types, functions and variables in any order you fancy
    struct MyStruct
    {
        int x, y;
        float[20] buffer;
    }

    MyStruct thing1, thing2;
    int someKindOfCounter;
    let myConstants = int[] (10, 20, 30);

    void function1() { ... }
    bool function2 (int x, int y) { ... }

    // Every processor must declare a main() function - see the section about its format
    void main()  { ... }
}
```

Processor Aliases

You can declare short aliases for processors (or graphs) using the syntax:

```
processor MyAlias1 = some_namespace::MyProcessor(1234),
          MyAlias2 = some_namespace::MyGraph(float64, bool);
```

Input/Output Endpoint Declarations

Processors and graphs can only communicate with the outside world via their endpoints, so must always declare at least one output.

The syntax for an endpoint is:

```
<direction> <type> <data-type(s)> <name> [optional array size] [optional annotation];
```

- The direction can be `input` or `output`
- The type can be `stream`, `event` or `value`
- The data type is a type (or a list of types) for the data that the endpoint carries.
- The name is whatever you want to call your endpoint. The name must be unique, and not clash with any other variables or functions in the processor.
- An array of endpoints can be declared using a square-bracket syntax after the name.
- An endpoint can be given a set of annotations which a host program may use to help interpret the purpose of the endpoint (see elsewhere in this guide for more details)

Examples:

```
processor P
{
    input stream float  input1;      // a simple stream of floats
    input stream float<4> input2;    // a stream where each element is a float<4> vector
    input value int64 in3, in4;      // two input value streams that hold int64s
    output event int out1;           // an output which sends simple integers as events
    output event MyStruct out2;      // an output of more complex object events
    output event (string, int) out3; // an output event stream which can accept either strings or ints
    output stream float<2> out4[4];  // an array of 4 output streams which each hold float<2> vectors
    input event void in;             // An input event with no value

    // If you have a lot of endpoints, you can also use braces to group together definitions:
    output
    {
        stream int x;
        stream float y;
    }

    // ..or..
```

```
        input event
        {
            int<2> x;
            float64 y;
        }
    }
```

Stream Endpoints

A `stream` endpoint transmits a continuous sequence of sample-accurate values, providing one value per frame. Currently, the type of a stream must be scalar (i.e. a float or integer, or a vector of floats/ints), so that they can be summed.

Streams involve storing and updating values for every frame, which makes them expensive, This makes them a bad choice for values that rarely change, but are ideal for continuously changing signals like audio data.

Value Endpoints

A `value` endpoint can hold any type of data, and allows a fixed value to be sent or received in a way that is not sample-accurate. They have effectively zero overhead when the value is not updated. This makes them the best choice for values that don't often change, and where it doesn't matter if there's some inaccuracy in the times at which they change, so for example you might use one to control a master volume level.

Values also have a feature that allows an automatic ramp to be applied to scalar values - the Cmaj API can specify a target and a number of frames, and it will smoothly interpolate to reach the target level.

Event Endpoints

When an input event is declared, a processor or graph can define a handler function for it. These special functions are prefixed with the `event` keyword.

```
processor P
{
    input event float<2> myInput;

    // when declaring an event handler, the name and type must match that of the endpoint.
    event myInput (float<2> e)
```

```
    {
        // do something here with the incoming event value e
    }

}
```

If the event has multiple types, you should declare a handler for each type. Any endpoint type without an event handler defined will be ignored when written to.

```
processor P
{
    input event (string, float<2>) myInput;

    event myInput (string e) { ... }
    event myInput (float<2> e) { ... }
}
```

Event endpoints also support the `void` datatype, for situations where the event does not include any data. Event handlers for `void` datatypes do not take a datatype, and there is special `void` syntax for send an event to such an endpoint:

```
processor P
{
    input event (void, int) myInput;        // Input endpoint taking either a void or int value
    output event void myOutput;

    event myInput() {}                      // Handler for the void datatype
    event myInput (int i) {}                // Handler for the int datatype

    void main()
    {
        myOutput <- void;                   // Write a void event to myOutput
        advance();
    }
}
```

Graph event handlers are more limited than processor event handlers because of the restriction that graphs do not contain state. However, graph event handlers can do useful work such as filtering what is forwarded, or scaling values

```
graph G
{
    input event float paramIn;
    output event float filteredOut;
    output value float scaledOutput;

    event paramIn (float f)
    {
        // Only send some param values through
        if (f > 0.5)
            filteredOut <- f;

        // Scale the normalised parameter to the range 10 .. 100
        scaledOutput <- 10.0f + (90.0f * f);
    }
}
```

“Hoisted” Endpoints

If you have a graph containing child nodes, and you want to allow the endpoint of a child node as a top-level endpoint, there is a shortcut syntax to make this possible.

So for example, if you have a processor with some endpoints like this:

```
processor MyChildProcessor
{
    output stream int out1, out2;

    ...etc...
}
```

then to expose these outputs from its parent graph, you could write it explicitly like this:

```
graph Parent
{
    output stream int out1, out2;

    node child = MyChildProcessor;

    connection child.out1 -> out1;
    connection child.out2 -> out2;
}
```

...or using the hoisted endpoint syntax, shorten it to this:

```
graph Parent
{
    output child.out1, child.out2;

    node child = MyChildProcessor;
}
```

...and for multiple endpoints, it can be shortened even further by using a wildcard pattern to match the endpoint names:

```
graph Parent
{
    output child.out*; // expose all of this node's outputs that begin with the characters "out"

    node child = MyChildProcessor;
}
```

(Wildcard patterns can use `*` to match any number of characters, or `?` to match a single character)

You can also directly hoist the endpoints of deeply-nested children of sub-graphs, without needing to do anything for the intermediate levels, e.g.

```
graph Parent
{
```

```
    input childNode.otherChild.yetAnotherLevel.*;
}
```

Renaming hoisted endpoints

It is possible to rename a hoisted enpoint, and to provide a pattern to rename wildcarded hoisted endpoints, e.g:

```
graph Parent
{
    output child1.outL child1outL;      // Rename outL as child1outL
    output child1.outR child1outR;      // Rename outR as child1outR;

    output child2.* child2*;            // Hoist child2 outL and outR renamed as child2outL and child2outR

    node child1 = Child;
    node child2 = Child;
}

graph Child
{
    output stream float outL, outR;
}
```

When renaming a wildcarded endpoint, unique names are generated by substituting the hoisted endpoint name into the provided name, replacing the `*` character.

Writing to Outputs

To write to any kind of output endpoint, use the left-arrow operator:

```
void main()
{
    loop
    {
        if (isTimeToSendEvent())
```

```
            myEventOut <- 1.0f <- 2.0f;


        myOutputStream <- 1.0f;
        advance();
    }
}
```

You can write to outputs at any time during your run loop, or inside an event handler in response to an incoming event.

If you write to the same output multiple times within the same frame, then

- For event outputs, multiple events are sent in the order they were added
- For value outputs, the current value is overwritten each time, so the last value written is the one that is used
- For stream outputs with a scalar type, values written are summing, and then flushed when the processor calls advance(). (NB: Currently only scalar types are allowed in streams, but when non-scalar types are supported, these will overwrite in the same way as for value endpoints)

Multiple writes to the same output can be chained into a single statement, which can be helpful for writing sequences of events.

Graphs

A graph is a collection of processors, and a description of how their endpoints are connected.

A `graph` declaration contains:

- A list of input and output endpoint declarations (exactly like a `processor`)
- A set of `node` declarations to define the set of child processor nodes it uses.
- A set of `connection` statements to define how the input and output nodes are connected to each other and to the graph's inputs and outputs.
- Pure functions used by the graph connections
- Global constant variables
- Types (`struct`s, `using` declarations)
- Optional event handlers for input endpoints - event handlers or connection routing can be provided for each input event endpoint, but not both

For example:

```
// This example graph just applies a gain of 0.5 to a mono stream of floats
graph GainExample
{
    // Declare the graph's inputs and outputs first - this is done with exactly the same
    // syntax as used in a processor declaration
    output stream float out;
    input stream float in;

    // now declare the nodes that the graph contains:
    node attenuatorNode = std::levels::ConstantGain (float, 0.5f);

    // now declare how the nodes are connected:
    connection in -> attenuatorNode -> out;
}
```

Graph Nodes

A node declaration has the form:

```
node <name> = <processor type> [optional parameters] [optional array size] [optional over/under-sampling f
```

For example:

```
node myNode1 = MyProcessor;  // simple instance of a processor
node myNode2 = SomeProcessor[4]; // declares a node which has an array of 4 instances of this processor
node myNode3 = OtherProc (float, 100); // declares an instance of a processor which has some specialisatic
node n1 = MyProcessor, n2 = MyProcessor; // you can use a comma to declare more than one node

// An alternative syntax for declaring multiple nodes is to use braces like this:
node
{
    n1 = MyProcessor;
    n2 = OtherProcessor;
}
```

Each node declares an instance of a processor type - if you're familiar with object-oriented programming, think of processors as being like classes, and nodes as being instances of those classes.

If you're only planning to use a single instance of a processor, then as a shortcut you can skip declaring a node, and just use the processor name in your connection list instead of the node name. This will implicitly create an unnamed instance node for that processor. If you need to refer to the processor more than once in your connection list, you'll need to declare a node for it and give it a name.

Connections

Connections use the right-arrow operator `->` and can be chained together.

```
// Node endpoints use the syntax <node name>.<endpoint name>
connection node1.output1 -> node3.input2;

// The graph's top-level endpoints are referred to by using their name on its own
connection node3.output7 -> output3;

// For nodes that only have a single input or output, the endpoint name can
// be omitted, and you can write chains of connections as a single statement:
connection node1.output1 -> node2 -> node3 -> output3;

// You can use a comma-separated list to send an output to multiple destinations:
connection node.output1 -> node2.in1, node3.in3;

// As for nodes and endpoints, you can also use a braced block to declare the connections:
connection
{
    node3.output7 -> output3;
    node1.output1 -> node2 -> node3 -> output3;
}
```

If you connect multiple sources to the same destination, then:

- If it's a stream with a data type of float or int, their values will be summed together
- If they are events, then all the events will be multiplexed together
- In other cases, this will be an error

When connecting endpoints or nodes which are arrays, you must make sure that either:

- You specify a connection between particular indexes of the source and destination arrays
- Or, if both ends have the same number of elements, each pair will be connected together
- Or, if one end of the connection is an array and the other is a single stream, connecting them will create a set of connections that perform a fan-in/out

Connection Functions

For stream and value connections, the source of the connection (the left hand element) can be specified as an expression. Endpoints can appear within these expressions, and their type will be the corresponding type of the endpoint.

```
/// specify a constant
0.5f -> node1.in;

/// Perform an arithmetic expression on multiple input values or streams
in1 * in2 -> node2.in;

/// Take the minimum of two inputs
std::min (in1, in2) -> node2.in;

/// Apply a custom function to convert a mid side input into a left/right stereo out
graph ConvertMidSide
{
    input stream float32<2> midSideInput;
    output stream float32<2> stereoOut;

    float32<2> convert (float32<2> v)
    {
        return float32<2> (v[0] - v[1], v[0] + v[1]);
    }

    connection
        convert (midSideIn) -> stereoOut;
}

// Merge a left/right input stream into a stereo output stream
```

```
graph MergeStereo
{
    input stream float leftIn, rightIn;
    output stream float<2> stereoOut;

    connection
        float<2> (leftIn, rightIn) -> stereoOut;
}
```

If the endpoint is on an array node, the type will be an array of the endpoint type. If the endpoint type is itself an array, this will produce a syntax error since the language does not support arrays of arrays. In this situation, you can use a node index, but you cannot take an endpoint index to resolve the issue as the type will still be an array of arrays which is not supported.

```
graph ReturnsArray
{
    output stream float32 out1[3];
    output stream float32 out2;
}

graph Test
{
    output stream float32 out;
    output stream float32 instanceOut[3];
    output stream float32 arrayElement[10];

    node n = ReturnsArray[10];

    connection
    {
        n.out1       -> arrayElement;    // invalid - the node is an array, and the endpoint is also an ar
        n.out1[1]    -> arrayElement;    // invalid - as above, n.out1 is an invalid type, so you can't ta
        n[2].out1    -> instanceOut;     // valid - n[2] selects a node, so the type is float32[3]
        n[2].out1[2] -> out;             // valid - n[2] selects a node, out1[2] selects an array member,

        n.out2        -> arrayElement;   // valid - type is float[10]
        n[1].out2     -> out;            // valid - type is float
```

```
    }
}
```

There are limits to what can be written as a graph function, specifically the function must be pure. Unlike processors, graphs cannot contain state, so you cannot, for example, implement an FIR filter, as the function cannot depend on previous input values.

Conditional connections

It is possible to use an `if` statement within the connection block to create connections which are optionally included in the graph. This allows, for example, for the order of processors to be changed, or for additional processors to be included in some paths. There are currently restrictions on the condition so that it must be a compile time constant. For now, the constant value can be passed to the graph as a parameterised valuee, or can be specified as an external.

```
graph Processor (bool distortionBeforeCompressor)
{
    input stream float in;
    output stream float out;

    node compressor = Compressor;
    node distortion = Distortion;

    connection
    {
        if (distortionBeforeCompressor)
        {
            in -> distortion -> compressor -> out;
        }
        else
        {
            in -> compressor -> distortion -> out;
        }
    }
}
```

Support will be extended in the future to allow for fully dynamic graph routing.

Processor composition

We can use the `node` concept within a processor, allowing processors to utilise DSP within another processor. Here is a simple example:

```
processor FilterComposition
{
    input stream float in;
    output stream float out;

    node lowPass = std::filters::tpt::onepole::Processor (0, 1000);

    void main()
    {
        loop
        {
            lowPass.in <- in;
            lowPass.advance();
            out <- lowPass.out;

            advance();
        }
    }
}
```

The main loop writes to the `in` input endpoint of the node, and then `advances` the node. This call triggers the `node` to step forward one frame, running the contained DSP, making it's outputs contain the processed result of the filter.

All input types are supported (`stream`, `value` and `event`), but only `stream` and `value` outputs can be read in the wrapping processor.

Like graph nodes, processor nodes can be marked as oversampled, and an array of nodes can also be declared.

```
processor Test
{
    input stream float in;
    output stream float out;

    // An array of 10 one pole filters, all 2x oversampled
    node lowPassArray = std::filters::tpt::onepole::Processor (0, 1000)[10] * 2;
}
```

Processor composition caveats

Processor composition is an advanced feature, and it is possible to write very confusing DSP this way! Since the wrapping processor has control over how time is advanced within the wrapped processor, it is possible to created unexpected results. For example, if `advance` is not called on the wrapped processor, or if `advance` is not called for a number of frames, the wrapped processor is likely to have old internal state which is likely to produce surprising results.

When to use a processor vs a graph

The key to understanding this is to think about how a general audio model may look. An effect might use several audio processor nodes, with the signal flowing though these in parallel or series. These processor nodes are linked together in a graph structure to create a larger graph layout.

Several processor nodes may graph together in clusters that are then fed into the main graph as one larger processor node. It can be as complicated or simple as a program requires.

It's important to note that although graphs seem like a higher-level abstraction than processors, they are *just as fast*! During compilation, the data-flow through graph connections becomes function calls, and the compiler optimises these as aggressively as hand-written functions. So for example, a graph containing a node which applies a gain-change to an input stream is quite likely to be reduced to a single assembly-language operation, just like a hand-written loop would be.

For more examples, most of the example code patches use a mixture of graphs and processors.

Delays and Feedback Loops

If you want to insert a delay between two endpoints in a graph, you can use the syntax:

```
connection mySource -> [100] -> myDest; // this adds a 100 frame delay between these two endpoints
```

Streams, events and values can all have delays added.

If you want to create a feedback loop in a graph, this is possible as long as you have a delay of at least 1 sample somewhere in the loop, e.g.

```
graph G
{
    // declare 3 nodes
    node p1 = MyProcessor,
         p2 = MyProcessor,
         p3 = MyProcessor;

    // Declaring a loop like this would cause a compile error:
    connection p1 -> p2 -> p3 -> p1;

    // ...but if you insert a delay somewhere, it becomes legal:
    connection p1 -> p2 -> [1] -> p3 -> p1;
}
```

OVER/UNDER-SAMPLING

A node in a graph can be made to run at a higher or lower frequency than its parent graph, and any connections between the node and other nodes will have their data resampled automatically.

```
// To declare a node that runs at a multiple of the parent frequency, use the multiply/divide operators:
node myOversampledNode = MyProcessor * 4; // 4x oversampling
node myUndersampledNode = MyProcessor / 2; // 2x undersampling
```

When a node is being resampled, you can set the resampling policy used by stream connections to that node like this:

```
connection [latch]  node2 -> out;       // chooses latched interpolation (repeats the last value, very low
connection [linear] node1.out -> out;   // chooses linear interpolation (low quality but quick)
```

```
connection [sinc]   node3.out2 -> out;  // chooses sinc interpolation (highest quality but slowest)
```

If no policy is specified, default policies are applied. For an oversampled node, `sinc` interpolation is used in and out of the processor to provide high quality alias free streams. For an undersampled node, `latch` is used on input connections, and `linear` is used on output connections.

Note that for obvious reasons, only streams with scalar data types can be interpolated. If you try to use other types, you'll get a compile error.

DECLARING AND DETECTING PROCESSOR LATENCY

The special property `processor.latency` is used to get and set the latency of a processor.

Some signal-processing algorithms require the use of an internal buffer, which delays the timing of the signal that passes through it. If you create a processor which adds latency, you should tell the system about its length by setting the `processor.latency` property like this:

```
processor P
{
    output stream float<2> output;
    input stream float<2> input;

    // Set a value for processor.latency alongside your other processor state variables.
    // It must be a compile-time constant, and cannot change dynamically.
    // The units are frames.
    processor.latency = 1000;

    void main()
    {
        float[processor.latency + 100] x;  // the property can also be read as
                                           // a constant anywhere in the processor

       // ...etc
    }
}
```

AUTOMATIC DELAY COMPENSATION

When a graph contains any nodes with non-zero latency, the system will automatically insert other delays into the signal-chain to compensate for differences in latency across the graph, so that all the events and streams remain in-sync. In DAWs, this is often referred to as PDC or Plugin Delay Compensation.

WRITING TO THE CONSOLE OUTPUT

A special event stream is available in processors, called `console`, which can be used to write messages to the output. Exactly what happens to the messages depends on the runtime: they may be printed to the console, or logged to a file, or just ignored, depending on what the host wants to do with them.

You can use `console` in any processor without needing to declare it, and you can write any type of value to it, e.g.

```
processor P
{
    output stream int out;

    void writeSomeLogging()
    {
        console <- "Hello " <- "World\n" <- 123 <- 1.5f <- myObject;
    }
}
```

Processor and Namespace Specialisation Parameters

Processor, graph and namespace declarations can declare parameters so that specialised instances of them can be created with custom types and constant values. (If you're familiar with templates in languages such as C++, this is the same kind of pattern).

```
// When anything tries to create an instant of this processor, the type MySampleType
// and the integer myConstant must be provided.
processor MyProcessor (using MySampleType, int myConstant)
{
    output stream MySampleType out;

    void main()
```

```
        {
            MySampleType x[myConstant];

            out <- MySampleType (myConstant + 10);
            advance();
        }
    }

    graph G
    {
        // when instantiating a processor that takes parameters, you put the values
        // in parentheses after the name. In this example, the compiler will create
        // two different versions of MyProcessor with these two sets of parameters.
        node p1 = MyProcessor (float, 100),
             p2 = MyProcessor (float64<2>, 200);
    }
```

A parameter can be:

- A type, by using the `using` keyword
- A processor, by using the `processor` keyword
- A namespace, using the `namespace` keyword
- A constant value, by declaring it as a type + name

```
// Examples of all the types of parameter that can be used:
processor Example (using TypeName, processor MyProcessor, namespace MyNamespace, float<2> myConstant)
{
    ...
```

Default values for trailing parameters can be provided like this:

```
graph G (int v = 100, processor P = MyProcessor)
{
```

When using a parameterised namespace, you'll need to provide parameters wherever you use its name. A way to avoid repeated code when doing this is to create a namespace alias:

```
namespace ExampleNamespace (using Type, int value)
{
    Type addValue (Type x)   { return x + Type (value); }
}

void f()
{
    // When using a parameterised namespace, it can be quite long-winded...
    let x1 = ExampleNamespace (int, 10)::addValue (100);

    // ...so you can use 'namespace' to declare a local alias:
    namespace N = ExampleNamespace (int, 10);

    let x2 = N::addValue (101);
    let x3 = N::addValue (102);
}
```

When creating parameterised modules, it's often a good idea to use `static_assert` to sanity-check the types provided and give helpful errors if a caller tries to use them with types that aren't suitable:

```
processor P (using T, int i)
{
    static_assert (T.isFloat || T.isVector, "T must be a float or vector type!");
    static_assert (i >= 0 && i < 100, "i is out of range!");
```

As a shortcut for writing out a parameterised processor or namespace name, you can create aliases for them with this syntax:

```
processor MyAlias1 = my_namespace::MyProcessor(float<2>, 1234),
          MyAlias2 = my_namespace::MyProcessor(float<3>, 5432);

namespace n123 = some_namespace(float64)::other_namespace(1.0f);
```

Annotations

At various places in a Cmaj program, an *annotation* can be attached to a definition, such as a processor, endpoint, function, variable, etc.

Annotations are enclosed in double-square-brackets `[[ ]]` and can contain an arbitrary set of comma-separated key-value pairs. The compiler generally ignores the content of annotations, but allows the runtime to access them.

Some examples of attributes:

```
// A processor annotation is written after its name and before the open-brace:
processor P [[ name: "hello", animal: "cat", size: 123 ]]
{
    // An endpoint or variable annotation goes between its name and the semi-colon:
    input event float in [[ name: "input", min: 10.0, max: 100.0 ]];

    int x [[ desc: "blah", number: 1234 ]];
}
```

Annotation property names must be valid Cmaj identifiers, and the values must be valid Cmaj values. They can contain expressions as long as they can be evaluated as compile-time constants.

Using the `[[ main ]]` Annotation

A commonly-used annotation is to add `[[ main ]]` to one of the processors in a program, as a hint to the runtime that this is the one that should be chosen as the entry point.

Built-in Constants

Inside a processor, the following special constants are available:

Name	Type	Description
`processor.frequency`	`float64`	The frequency of the processor in frames-per-second
`processor.period`	`float64`	The length in seconds of one frame for this processor
`processor.id`	`int32`	A value which is unique for every instance of a particular processor.
`processor.session`	`int32`	A value which is unique for each run of the program

Some global numerical constants are also defined:

Name	Type	Description
`nan`	`float32`	NaN (Not A Number)
`inf`	`float32`	Inf (Infinity)
`pi`	`float64`	Pi
`twoPi`	`float64`	Pi * 2

(To get a float32 version of pi, just use a cast: `float32 (pi)`)

Intrinsic Functions

The Cmajor standard library provides a set of basic built-in intrinsic functions that can be called like their C/C++ equivalents.

For documentation on the many other functions and processors that the Cmajor standard library contains, please see the online documentation.

ARITHMETIC FUNCTIONS

Function	Description
`abs()`	Absolute value
`sqrt()`	Square root of a value
`pow()`	Base value raised by exponent
`fmod()`	Floating point remainder of a division
`remainder()`	Remainder of numerator/denominator, rounded
`roundToInt()`	Rounds a float to the closest integer
`floor()`	Rounds a float down to closest integer
`ceil()`	Rounds a float up to the closest integer
`rint()`	Rounds a float to the closest integer
`log10()`	Returns a base 10 logarithm of value
`log()`	Returns a natural logarithm of value
`exp()`	exponential (e) raised by given value

TRIG FUNCTIONS

Function	Description
`sin()`	Sine of an angle
`sinh()`	Hyperbolic sine of an angle
`asin()`	Arc sine (inverse sine) of an angle
`asinh()`	Arc hyperbolic sine
`cos()`	Cosine of an angle
`cosh()`	Hyperbolic cosine of an angle
`acos()`	Arc cosine (inverse cosine) of an angle
`acosh()`	Arc hyperbolic cosine
`tan()`	Tangent of an angle
`tanh()`	Hyperbolic tangent of an angle
`atan()`	Arc tangent (inverse tangent) of x
`atanh()`	Arc hyperbolic tangent
`atan2()`	Arc tangent of y/x

COMPARISON & OTHER FUNCTIONS

Function	Description
`max()`	Returns largest value
`min()`	Returns smallest value
`select()`	Compares two input vectors and choses based on boolean input
`lerp()`	Linear Interpolation

Calling native functions from Cmajor

When embedding a Cmajor engine into a custom app, you may want to provide some native functions that can be called directly from the Cmajor code.

Doing this is obviously very much an “expert” level feature, and opens up many interesting and subtle ways to make your app crash, and it’s also only available on some back-ends (e.g. LLVM).

If you really need to do this, then in your Cmajor code, you define a placeholder for the function using the `external` keyword, e.g.

```
external int32 myNativeFunction (float x, double y, float[] z);  // no function body is declared
```

When building a program containing externals, the `cmaj::Engine::link()` method takes a functor which your app uses to provide raw C function pointers for any external functions that need to be resolved. Because these function pointers are simply a `void*`, it's your responsibilty to make sure the parameter and return types exactly match those of the Cmajor function! Only primitive types are permitted as parameters, but you can pass a slice (e.g. `int[]`) to allow arrays to be passed in. These come through as a raw C pointer, so it's up to you to not access it out-of-bounds.