

# Brook Spec v0.2

Ian Buck

May 20, 2003

## 0.1 What is Brook?

Brook is an extension of standard ANSI C which is designed to incorporate the ideas of data parallel computing and arithmetic intensity into a familiar, efficient language.

1. *Data Parallelism*: Allows the programmer to specify how to perform the same operations in parallel on different data.
2. *Arithmetic Intensity*: Encourages programmers to specify operations on data which minimize global communication and maximize localized computation.

## 0.2 Streams

Brook introduces two new type-qualifier keywords to C, `stream` and `memstream` which can be applied to any standard C variable declaration.

### 0.2.1 Memstreams

Brook introduces a new type-qualifier keyword `memstream` into the C language spec [A4.4 K&R]. Declaring an object `memstream` announces that it has special properties relevant to streaming.

#### Declarations

`memstream` objects are declared the same way as any other object with a type-qualifier, such as `const` or `volatile` [A8 K&R].

#### Stream Elements

The purpose of declaring `memstreams` is to express a collection of objects which can be operated on in parallel. The collection of objects is referred to as a *stream* while each object is a *stream element*. The `memstream` type determines the stream element type as well as the number of elements contained in the stream.

If a `memstream` declaration is not an array declaration [A8.6.2 K&R], the `memstream` object contains only a single stream element of the type equal to the `memstream` object.

If the `memstream` declaration is an array declaration, then the type of the stream element is equal to the base type of the array. The base type is the type of the object after which all of the `[constant-expression]` specifiers which are directly associated with the variable name have been removed from the declaration statement (as defined by operator precedence).

For example,

```
memstream float val;
```

Stream of floats, one element.

```
memstream float a[4][5][9];
```

Stream of floats, 4\*5\*9 elements.

```
typedef float mytype[3][2];
```

```
memstream mytype c[304];
```

Stream of float[3][2], 304 elements

A few more complex examples,

```
memstream char *daytab [13];
```

Stream of char \*, 13 elements

```
memstream char (*daytab) [13];
```

Stream of pointer to char[13], one element

```
memstream char (stringtable[3])[5];
```

Stream of char[5], 3 elements

For more information on parsing C declaration types, see section 5.12 of K&R.

### Usage Restrictions

- Any expression which involves the address of any elements within a memstreams is not allowed [A7.4.2 K&R].
- Likewise, any pointer arithmetic on memstreams is not allowed [A7.1 K&R].

As a result, accessing individual elements of a memstream must occur through the object name, not through a pointer referencing the elements.

For example,

```
memstream float as [13][32];
```

```
float b = as[3][5];           Legal
```

```
as[2][5] = 4.3f;             Legal
```

```
float *p = &(as[3][5]);      Illegal: address computed
```

```
b = **(&as+5);               Illegal: pointer arithmetic
```

### Arrays of Memstreams

Arrays of memstreams is currently not supported in Brook but may be supported in the future.

### Stream Shape

The stream *shape* refers to the dimensions of a stream. For example, the declaration:

```
memstream float as[30][20][5];
```

is a stream containing 3000 floats in the shape 30 by 20 by 5. The type of *as* is a three dimensional memstream of floats with a *constant* shape of 30 by 20 by 5.

### Dynamic Shape and Size

To allocate variable sized memstreams, Brook supports memstream declarations in which some of the dimensions are either unspecified or specified by non-constant variables. For example,

```
memstream float as [n][m];
```

An *n* by *m* stream of floats.

The variables *n* and *m* must be of type `unsigned int`. The values of *n* and *m* are evaluated at the declaration statement of *as*. Expressions are permitted in the array bounds but the type must resolve to an `unsigned int`. The shape of *as* cannot change for the life of the variable *as* as defined by the scope. The type of *as* is a two dimensional memstream of floats with *static* shape.

Memstreams can also have *dynamic* shape which can be redefined after declaration. For example,

```
memstream float as[] [] = alloc_memstream(sizeof(float), 8, 5);
```

A float stream with dynamic shape 8 by 5

The `alloc_memstream` function takes a variable number of arguments. The first argument is the size of the stream element. The remaining arguments are the size of each dimension. The `alloc_memstream` function call must be present for any memstream declaration where the dimensions are not specified. The type of *as* is a two dimensional memstream of floats with *dynamic* shape. The `alloc_memstream` assignment *must* be present in any dynamic memstream declaration. `alloc_memstream` cannot be used other than for dynamic memstream declarations. The memory allocated is freed automatically when the variable goes out of scope.

The benefit of dynamic memstreams is that the programmer is permitted to redefine the shape of the stream at any point in the program. To reshape *as*, use the `reshape` operator:

```
reshape(as, 4, 10);
```

Reshaping a stream with dynamic shape using the `reshape` operator

Some notes on using `reshape`: The number of arguments to `reshape` must match the number of dimensions of the variable. The dimension arguments can be variables, expressions, or constants. The `reshape` operator does not grow or shrink the amount of memory associated with the stream variable.

The memory layout of a `memstream` is the same as a multidimensional c-array. The address calculation of `memstream` array access also operates the same way as C (See K&R). By reshaping a stream, we are simply changing the array dimensions used in the address calculation.

The `dim` operator permits access to the size of each dimension:

```
dim(as, 0);
```

Returns the least significant dimension of the stream `as`

Static or constant `memstreams` cannot be reshaped after they have been declared.

(Need to make this better defined. What about mixed declarators?)

### Passing `memstreams` to functions

```
memstream float as[3][2];
memstream float bs[n][m];
memstream float cs[][] = alloc_memstream(sizeof(float), 8, 5);

void function (memstream float arg1[3][2],
               memstream float arg2[n][m],
               memstream float arg3[void][void],
               memstream float arg4[][]);
```

`memstreams` may be passed to functions. The above example illustrates a function with four `memstream` arguments: `arg1` is a constant `memstream`, `arg2` and `arg3` are static `memstreams`, and `arg4` is a dynamic `memstream`. The syntax for static `memstreams` allows the called function access to the dimensions of a stream. `arg2`'s declaration also declares the `const unsigned int` variables `n` and `m` which are initialized with the dimension values. If these variables are not needed, the `void` keyword can be substituted as in the declaration of `arg3`. `arg4` is a dynamic `memstream` argument.

`memstreams` are passed by reference. If the function modifies any of the elements of the `memstream`, these changes are reflected in the caller. Passing dynamic `memstreams` permits reshaping of the stream, however these shape changes are *not* reflected in the caller.

Streams may be promoted to other types when passed to functions according to the following relationships:

```
Constant -> Constant, Static, Dynamic
Static    -> Static, Dynamic
Dynamic   -> Static, Dynamic
```

Note that constant and static `memstreams` may be promoted to dynamic streams, however any reshaping done by the callee does not change the shape in the caller.

`memstreams` cannot be declared inside functions and returned.

\*\*\* MORE HERE \*\*\*: Memstream c\_ptr allocator.

## 0.2.2 Streams

Brook introduces a new type-qualifier keyword `stream` into the C language spec [A4.4 K&R]. Declaring an object `stream` announces that it has special properties relevant to streaming.

### Declarations

`stream` objects are declared the same way as any other object with a type-qualifier, such as `const` or `volatile` [A8 K&R]. They are declared in the same way as dynamic `memstreams`, however they have no `alloc_memstream` assignment.

```
stream float val[];
```

Stream of floats, one dimension

```
stream float a[][][];
```

Stream of floats, three dimensions.

### Differences from `memstreams`

Streams are *not* associated with a particular region of memory. Rather, they behave as connectors between `memstreams`, kernels, and stream operators. Also, C code cannot perform array index operations on streams.

\*\*\* MORE HERE \*\*\*: shaping streams. Memstream = stream assignment. stream = memstream assignment.

### 0.3 Stream Memory Consistency Model

To be written....

Notes:

```
memstream float a[1024], c[1024];
foo(a, c); // kernel
a[435] = 3.2f; // cannot affect foo, c
float x = c[385]; // sees foo
```

Memstreams: All writes and reads to memstreams are fully ordered. Behaves sequentially

```
memstream float a[1024], c[1024];
stream float b[];
foo(a, b); // kernel
a[435] = 3.2f; // MAY affect b
bar(b, c); // kernel
```

Streams: Writes to b by kernels may be affected by writes to dependent memstreams

```
memstream float a[1024], c[1024], d[1024];
stream float b[];
foo(a, b, c); // kernel out:b, c
a[435] = 3.2f; // MAY affect b
float x = c[342]; // Sees foo
bar(b, d); // kernel
```

### 0.4 UPC Data Partitioning

To be written....

Notes:

Partitioned across machine as defined in UPC spec

```
shared [3] int A[THREADS][5];
```

Dynamic allocation

```
shared void *upc_all_alloc (size_t nblocks, size_t nbytes)
```

Same as: `shared [nbytes] char [nblocks*nbytes]`

Synchronization on Writes

Determined by UPC reference-type-qualifier.

```
strict shared int y;
```

Strict ordering semantics (code inserted by compiler/runtime)

Worst case: sync on every write

```
relaxed shared int x;
```

Unordered semantics

User inserts barriers/sync commands

```
upc_barrier(); (no upc_locks, upc_wait, upc_notify)
```

```
#pragma upc strictrelaxed determines default characteristics
```

### 0.5 Stream Operators

```
streamDomain(s, t, 2, 5, 100, 1, 90);
```

s is a copy of elements of t starting at the start offset for the dimension and up to, but not including, end. 2 refers the number of dimensions of s which was specified by the shape command. Next are the range pairs for each dimension. In this example, t will contain s[5..99][1..89].

```
streamGroup(t, s, STREAM_BOUNDS_PERIODIC, 1, 4);
```

4 elements of *s* grouped into a single element of *t*. `streamGroup` takes a variable number of arguments. *t* is the output. *s* is the input stream. Next, there is a boundary mode argument which defines what to do on the boundary of any dimension. This can be `STREAM_GROUP_CLAMP` (use a specified value for the boundary values), `STREAM_GROUP_PERIODIC` (periodic boundary condition), or `STREAM_GROUP_HALO` (do not include any elements which span the boundary). Next is the number of dimensions in the group. (This must be the same as the call to `shape` on *s* if there is more than one dimension.) Next is the number of elements in the group per dimension. If `STREAM_GROUP_CLAMP` is selected, after each dimension, a pointer is passed indicating the value to use in clamping for that dimension.

```
streamStencil(t, s, STREAM_BOUNDS_HALO, 1, -1, 2);
```

Similar to `group` however each element of *t* contains a value from *s* and all of its neighbors in *s*. The arguments work the same way except each dimension has a left and right component determining how many elements either side of element of *s* should be included in each element of *t*.

```
streamGroupExt(t, s, 1, 0, STREAM_BOUNDS_PERIODIC, 3, STREAM_BOUNDS_PERIODIC);
```

This is similar to `group` except that we provide both left and right ranges for the group with boundary conditions for each edge. The 1 refers to the number of dimensions, 0 is the left edge with periodic boundary conditions and 3 is the right edge with periodic boundary conditions.

```
streamStencilExt(t, s, 1, 0, STREAM_BOUNDS_PERIODIC, 3, STREAM_BOUNDS_PERIODIC);
```

This is similar to `stencil` except that we provide boundary conditions for each edge. The 1 refers to the number of dimensions, 0 is the left edge with periodic boundary conditions and 3 is the right edge with periodic boundary conditions.

```
streamFlatten(s, t);
```

The array elements of *t* are ungrouped into a flat linear stream of *s*.

```
streamStride(s, t, 1, 3, 16);
```

*s* is a stream derived from taking 3 elements of *t* then skipping 16 elements then taking 3 elements... The 1 indicates that this is a one dimensional stride. Multidimensional strides are supported with a different include and skip values per dimension.

```
streamReplicate(t, s, 1, 3);
```

*t* contains the elements of *s* replicated 3 times. i.e. *s*=1,2,3,4 *t*=1,1,1,2,2,2,3,3,3,4,4,4. Multidimensional replicates are also supported.

```
streamRepeat(t, s, 1, 3);
```

*t* contains the elements of stream *s* repeated 3 times. i.e. *s*=1,2,3,4 *t*=1,2,3,4,1,2,3,4,1,2,3,4. Multidimensional repeats are also supported.

\*\*\* MORE HERE \*\*\*

Notes:

Can only scatter and gather to memstreams.

```
StreamScatterOp(memstream base, memstream or stream offsets,  
memstream or stream values, OP);
```

```
StreamGatherOp(memstream or stream results, memstream base,  
memstream or stream offsets, OP);
```

Add `StreamCat`, `StreamMerge`. Delete `Repeat`?

Talk about memstream vs. stream argument difference

## 0.6 Global Memory functions

Note: This may change to be just memstreams.

These functions allow the user to allocate memory which can be accessed by the kernel functions by passing a pointer. If a kernel is executed across multiple parallel nodes, the data is replicated on each node. Allocated data is read-only inside of kernel functions.

```
void * streamGlobalAlloc (unsigned int size)
```

Allocates size bytes for passing allocated memory to kernel functions.

```
void streamGlobalFree (void *)
```

Frees memory allocated by `streamKernelAlloc`.

## 0.7 Kernel functions

Kernel functions are a special type of stream function which have certain restrictions to ensure parallel execution. Kernel functions are declared as normal C functions but with the `kernel` keyword before the function name.

Restrictions

1. No global memory reads or writes. (Except for GlobalAlloc memory)
2. Arguments can be read only, write only, or reduce.
3. No nested kernel function calls.

In general, each execution of the kernel function removes one element from each input stream. Execution completes if there are no more remaining input elements.

\*\*\* MORE HERE \*\*\*

Notes: Provide an example.

## 0.8 Kernel Arguments

Arguments considered read only unless we use one of the following keywords:

1. `out`: Specifies argument write only. System will automatically shrink or grow the stream as needed.
2. `mout`: Specifies multiple writes. Outputs must be explicitly issued using the **push** command which produces another element on this element stream.
3. `reduce`: Specifies reducible argument. Only associative operations will work correctly.

## 0.9 Return Values

Currently kernel function cannot return streams.

## 0.10 Reductions

Reduction arguments can be used to implement a simple reduction. For example:

```
void kernel foo (stream float s, reduce float max) {
    max = max > s ? max | s;
}

memstream float max;
stream float s[];
foo(s, max);
```

The `max` pointer is assigned the largest value of the stream `s`.