```
//
//  main.cpp
//  SimpleCacheSim
//



#include <stdio.h>
#include <unistd.h>
#include <getopt.h>
#include <errno.h>
#include <stdlib.h>
#include <stdbool.h>
#include <inttypes.h>

#include <vector>
#include <algorithm>
using namespace std;

#include "range.h"


// In future we might want to use longs (which would be 64-bit on a 64-bit machine) or even long longs (for 64-bit even on 32-bit machines).  For our current requirements, however, 32-bits are quite suitable.
typedef unsigned int Address;
#define PRIAddress "%u"

// Similarly, we may in future want to expand the time range.  But 32-bits more than covers what we presently need.
typedef unsigned long TimeStamp;
#define PRITimeStamp "%lu"

// For debugging purposes we often want to print out the program name, outside of main().
const char *programName = "(error)";


enum { // Note that these two symbolic constants are not the be-all and end-all of possible values; indeed, any value for associativity is possible.  They are just two very common types that have explicit names beyond "N-way".
    DirectMapped = 1, // Symbolic constant for direct associativity (a.k.a. 1-way).
    FullyAssociative = 0 // Symbolic constant for full associativity.
};

enum { // Replacement policies.
    RandomReplacement, // Pick any entry within the set to replace, at random.  Results will *not* be reproducible.
    LRUReplacement, // Least Recently Used.  Pick the entry with the oldest usage (whether a read or a write).
    LRRReplacement, // Least Recently Read.  Pick the entry with the oldest read time.
    LRWReplacement, // Least Recently Written.  Pick the entry with the oldest write time.
    FIFOReplacement // First In First Out.  Pick the entry with the oldest load time.
};

enum { // Misalignment policies.
    MPNone, // Don't allow any form of misalignment.
    MPWord, // Allow misaligned words, provided they are within the same block.
    MPBlock, // Allow misaligned words that stradle multiple blocks.
    MPForced // Adjust all addresses to be appropriately aligned.
};


// At present it appears dineroIV supports only these finite number of operations, but in future we may expand the OpType to encompass additional flags or values.
typedef unsigned char OpType;
#define PRIOpType "%hhu"

enum { // Actual operation types (a.k.a. basic op types).
    OpRead = 0, // Read operation.
    OpWrite = 1, // Write operation.  Currently only the write-allocate policy is supported, meaning writes will load referenced memory into cache, marked as dirty.
    OpInstruction = 2, // Instruction fetch.  Functionally similar to a read, but for statistical purposes explicitly identified.
    OpMisc = 3, // Misc.  No idea; dineroIV defines it.  Currently treated as a bastardised read.
    OpCopyBackDirtyLines = 4, // Special cache instruction forcing a writeback of dirty cache blocks (a.k.a. lines).  Not really supported at present - currently treated as a bastardised read.
    OpInvalidateLines = 5 // Invalidate data in cache.  Not supported at present - currently treated as a bastardised read.
};

#define NUMBER_OF_OP_TYPES 8 // Only the first six are presently defined (above), and only the first three are presently used.

#define OpPrefetch 0x08 // Special bit indicating the operation is a prefetch.  In a full simulator this might kick off a longer-term prefetch operation; in ours it is noted only for statistical purposes, and the operation conducted as normal.
#define OpMultiblock 0x10 // No idea; dineroIV defines it.  Currently ignored.

/*! @function OpBasicType
    @abstract Returns the "basic type" of a given OpType.
    @discussion The basic type is the fundamental operation type - e.g. a read, a write, etc.  This information is embedded within the OpType type, and should only be accessed via this function.
    @param op The operation to retrieve the basic type of.
    @result Returns the basic type of the given operation (which is of type OpType, although don't get confused). */

#define OpBasicType(op) ((op) & 0x07)


/*! @struct CacheAccess
    @abstract Records a single cache access instance.
    @discussion Each cache access is stored in this structure, which can be easily manipulated and passed around for batch processing.

                Note that the accesses are not guaranteed to be atomic, in the sense that - because they could be misaligned for some simulations - they may generate more than one cache operation.  The necessary splitting is performed within the simulator core, at a level below which this structure is used.
    @field operation The operation the access represents (e.g. a read, write, etc).
    @field address The address the operation relates to.  Note that misalignment is not handled at the upper levels that may deal with this structure, but rather within the
```

```
core of each simulation.  As such, there are no explicit restrictions on alignment for this field (although of course trying to use a misaligned value in a simulation which doesn't tolerate misalignments will cause the simulation to abort).
    @field inputLine If debugging is enabled, this field contains the line in the original input source from which this access was read.  Useful when trying to diagnose problems (e.g. misaligned addresses), but not so good for memory usage (and consequently performance). */

typedef struct _CacheAccess {
    OpType operation;
    Address address;
#ifndef NDEBUG
    unsigned long inputLine;
#endif
} CacheAccess;


/*! @enum EvictionReason
    @abstract Indicates the reason for an eviction.
    @discussion It's no good, when a miss occurs, looking at the <i>current</i> state of the cache as to why the block wasn't already there - the current state may well have changed and not be indicitive of why the original eviction occured.  Consequently, the CacheEntry stores the reason, which can be directly retrieved when a miss does occur.
    @field EvictionNotYet The block has not actually been evicted yet - this is the first time it has been in cache.
    @field EvictionCapacity The block was evicted because the entire cache was full.
    @field EvictionConflict The block was evicted <i>while the cache was not full</i>, because it's particular set was full.
    @field EvictionInvalidated The block was evicted because the cache was instructed explicitly to invalidate the block. */

typedef enum _EvictionReason {
    EvictionNotYet = 0,
    EvictionCapacity,
    EvictionConflict,
    EvictionInvalidated
} EvictionReason;

/*! @struct CacheEntry
    @abstract Represents a single block in cache.
    @discussion This structure is used for managing blocks - that is, each instance represents a single block, which can be stored in various locations to symbolise being in or out of a cache, having been in a cache in past, etc.  The actual position of the block is not stored within the structure itself - the data flow is top down only.
    @field isDirty If true (1) the entry has been modified in cache and not yet written back to memory.
    @field wasPrefetched If true (1) this cache entry was prefetched, meaning the cache loaded it up for some reason before an actual request was made for it.
    @field wasPrefetchedExplicitly If true (1) the prefetch was requested explicitly, and not the result of, e.g. automatic read-ahead.  Meaningless if 'wasPrefetched' is false.
    @field baseAddress The base address of the entry, which will be block-aligned (i.e. an integer multiple of block size).
    @field lastLoad When the entry was last loaded into cache.  Used for FIFO algorithms, among others.
    @field lastRead When the cache entry was last read.  Used by LRR & LRU algorithms, among others.
    @field lastWritten When the cache entry was last written.  Used by LRW & LRU algorithms, among others.
    @field evictionReason The reason the block was last evicted from cache. */

typedef struct _CacheEntry {
    unsigned int isDirty:1,
                 wasPrefetched:1,
                 wasPrefetchedExplicitly:1;
    Address baseAddress;
    TimeStamp lastLoad;
    TimeStamp lastRead;
    TimeStamp lastWritten;
    EvictionReason evictionReason;

    // Might also want to, in future, record policies on the cache - e.g. instruction only, read-only, etc.  And stuff like write-through or write-back, etc.
} CacheEntry;

/*! @struct CacheSet
    @abstract Represents a single set in cache.
    @discussion This structure is pretty trivial, and could almost be snuffed out in preference to a few appropriate arrays - or even just a pair containing it's two vector elements.  But it exists for design elegance and for performance - the 'isFull' field provides an excellent shortcut in processing, at the trade off in memory footprint.  It hasn't, however, been tested - it is only my opinion that this method is faster.  In any case, in future other additions could certainly improve performance significantly.
    @field currentEntries The entries currently in cache.  Sized upon creation appropriately (i.e. using size() won't tell you anything useful about it... perhaps not the best design).
    @field pastEntries Every entry that has ever been (or currently is) in cache.  Used for differentiating between miss types (i.e. if it's not already in this vector, it's a compulsory miss).
    @field isFull If true the set is full. */

typedef struct _CacheSet {
    vector<CacheEntry*> currentEntries;
    vector<CacheEntry*> pastEntries;
    bool isFull;
} CacheSet;


enum { // Stat types.  These are used to index into the Stats structure.
    CounterHits = 0,
    CounterCapacityMisses,
    CounterCompulsoryMisses,
    CounterConflictMisses,
    CounterMisalignmentsBlock,
    CounterMisalignmentsWord
};

#define NUMBER_OF_COUNTERS 6 // As defined in the anonymous enum above.

/*! @struct Stats
    @abstract Used for recording statistics through a simulation run.
    @discussion For simplicity the current implementation uses a 2D array, rather than explicit fields.  This makes it much easier to collate results at the end, by allowing simple indexed enumeration over all the stats.
    @field stat The stats themselves. */

typedef struct _Stats {
    unsigned long stat[NUMBER_OF_OP_TYPES][NUMBER_OF_COUNTERS];
} Stats;


/*! @struct SimulationParameters
    @abstract Used for formally specifying the parameters for a single simulation run.
    @discussion Rather than pass a whole bunch of parameters to each simulation function, this parameter block is used.
    @field associativity The associativity of the cache.  Any positive integer (including 0) is valid.
    @field replacementPolicy The replacement policy to use.  Any of the defined values are valid.
    @field misalignmentPolicy The misalignment policy to use.  If a misalignment occurs that is not tolerated by the given policy, the simulation will abort.  Any of the defined values are valid.
```

```
    @field blockSize The size (in bytes) of a single block.  Should be a multiple of wordSize, or 0 (in which case it is automagically interpreted as being wordSize).
    @field cacheSize The size (in bytes) of the cache.  Should be a multiple of blockSize (or wordSize, if blockSize is 0).
    @field memorySize The size (in bytes) of memory.  Used only for validating addresses.  This validation involves checking that the memory size is a multiple of block siz
e (possibly among other things), so it must be performed on a per-simulation basis - thus the reason this isn't simply performed when reading in the address the first time.
Should be a multiple of blockSize, or 0, which indicates no range checking on memory addresses.
    @field wordSize The size (in bytes) of a single word.  Any positive integer (i.e. 1, 2, etc - <b>not</b> 0) is valid.
    @field verbosity The verbosity of the output.  Note that unlike the program parameter, which can be of either sign, this value should only be positive.  Any positive
integer (including 0) is valid - negative values have the same implicit value as 0, but are not explicitly supported and should not be used. */

typedef struct _SimulationParameters {
    unsigned int associativity,
                 replacementPolicy,
                 misalignmentPolicy;
    Address blockSize,
            cacheSize,
            memorySize,
            wordSize;
    int verbosity;
} SimulationParameters;


/*! @function fprintCounterName
    @abstract Prints to a given FILE the human-readable name of a given counter type.
    @param output The FILE to print to.  May not be NULL.
    @param counter The index of the counter in question.
    @param terse If true the output is 'terse' - that is, it does not contain any spaces or other punctuation, only alphanumerics.  Defaults to false.
    @result As for fprintf. */

int fprintCounterName(FILE *output, unsigned int counter, bool terse = false) {
    switch (counter) {
        case CounterHits:
            return fprintf(output, "Hits"); break;
        case CounterCapacityMisses:
            return fprintf(output, (terse ? "CapacityMisses" : "Capacity Misses")); break;
        case CounterCompulsoryMisses:
            return fprintf(output, (terse ? "CompulsoryMisses" : "Compulsory Misses")); break;
        case CounterConflictMisses:
            return fprintf(output, (terse ? "ConflictMisses" : "Conflict Misses")); break;
        case CounterMisalignmentsBlock:
            return fprintf(output, (terse ? "MisalignmentsStradlingBlocks" : "Misalignments straddling blocks")); break;
        case CounterMisalignmentsWord:
            return fprintf(output, (terse ? "MisalignmentsWithinBlock" : "Misalignments within block")); break;
        default:
            return fprintf(output, (terse ? "Unknown%u" : "Unknown (%u)"), counter);
    }
}

/*! @function fprintAssociativity
    @abstract Prints to a given FILE the human-readable name of a given associativity.
    @discussion It is not defined how the associativity is represented - e.g. by name (if an explicit type, e.g. direct), by value only (e.g. "4") or more verbosely (e.g.
"4-way").
    @param output The FILE to print to.  May not be NULL.
    @param associativity The associativity.
    @result As for fprintf. */

int fprintAssociativity(FILE *output, unsigned int associativity) {
    switch (associativity) {
        case DirectMapped:
            return fprintf(output, "Direct"); break;
        case FullyAssociative:
            return fprintf(output, "Full"); break;
        default:
            return fprintf(output, "%u-way", associativity);
    }
}

/*! @function fprintReplacementPolicy
    @abstract Prints to a given FILE the human-readable name of a given replacement policy.
    @param output The FILE to print to.  May not be NULL.
    @param replacementPolicy The replacement policy.
    @result As for fprintf. */

int fprintReplacementPolicy(FILE *output, unsigned int replacementPolicy) {
    switch (replacementPolicy) {
        case RandomReplacement:
            return fprintf(output, "Random"); break;
        case LRUReplacement:
            return fprintf(output, "LRU"); break;
        case LRRReplacement:
            return fprintf(output, "LRR"); break;
        case LRWReplacement:
            return fprintf(output, "LRW"); break;
        case FIFOReplacement:
            return fprintf(output, "FIFO"); break;
        default:
            return fprintf(output, "Unknown (%u)", replacementPolicy);
    }
}

/*! @function fprintOpType
    @abstract Prints to a given FILE the human-readable name of a given OpType.
    @discussion The exact form of the output is not defined.  Keep in mind that there is additional information embedded in an OpType beyond the basic type, such as whether
it is a prefetch.  The full output can be quite long.

                Note that you can safely use this function for printing out the basic type, as returned by the OpBasicType function.  It is guaranteed, in this case, not to
add anything to the output beyond the basic op name.
    @param output The FILE to print to.  May not be NULL.
    @param op The op in question.
    @param terse If true the output is 'terse' - that is, it does not contain any spaces or other punctuation, only alphanumerics.  Defaults to false.
    @result As for fprintf. */

int fprintOpType(FILE *output, OpType op, bool terse = false) {
    int err;

    switch (OpBasicType(op)) {
        case OpRead:
```

```
            err = fprintf(output, "Read"); break;
        case OpWrite:
            err = fprintf(output, "Write"); break;
        case OpInstruction:
            err = fprintf(output, "Instruction"); break;
        case OpMisc:
            err = fprintf(output, "Misc"); break;
        case OpCopyBackDirtyLines:
            err = fprintf(output, (terse ? "CopyBackDirtyLines" : "Copy back dirty lines")); break;
        case OpInvalidateLines:
            err = fprintf(output, (terse ? "InvalidateLines" : "Invalidate lines")); break;
        default:
            err = fprintf(output, (terse ? "Unknown"PRIOpType : "Unknown ("PRIOpType")"), OpBasicType(op));
    }

    if (0 > err) {
        return err;
    }

    if (op & OpPrefetch) {
        err = fprintf(output, (terse ? "Prefetch" : " [Prefetch]"));

        if (0 > err) {
            return err;
        }
    }

    if (op & OpMultiblock) {
        err = fprintf(output, (terse ? "Multiblock" : " [Multiblock]"));
    }

    return err;
}

/*! @function fprintMisalignmentPolicy
    @abstract Prints to a given FILE the human-readable name of a given misalignment policy.
    @param output The FILE to print to.  May not be NULL.
    @param misalignmentPolicy The misalignment policy.
    @result As for fprintf. */

int fprintMisalignmentPolicy(FILE *output, unsigned int misalignmentPolicy) {
    switch (misalignmentPolicy) {
        case MPNone:
            return fprintf(output, "None"); break;
        case MPWord:
            return fprintf(output, "Word"); break;
        case MPBlock:
            return fprintf(output, "Block"); break;
        case MPForced:
            return fprintf(output, "Forced"); break;
        default:
            return fprintf(output, "Unknown (%u)", misalignmentPolicy);
    }
}


/*! @function fgetline
    @abstract Obtains a single line of input from a given FILE.
    @discussion This works similarly to the C++ getline function, but works on basic C strings.  If you do provide your own buffer to it initially, you must make sure that buffer is allocated such that it can be used with the realloc function, which may be called internally to resize the buffer as necessary.  Conversely, you are responsible for freeing the returned buffer.
    @param buffer A pointer to a C string.  On input provides a C string as a starting point.  On output points to the buffer (which, if the original buffer was NULL, will be newly allocated by this function).  Should not be NULL.
    @param bufferSize On input indicates the size of the existing buffer (if *buffer is not NULL).  On output indicates the final size of the buffer (which may be resized by this function).  Should not be NULL.
    @param input The input source to read data from.  Should not be NULL.
    @result Returns the buffer if any data was read into it, otherwise returns NULL, indicating end of file.  If an error occurs this function calls exit() directly.  (yes, I know, this isn't very nice; it's on my TODO list). */

char* fgetline(char **buffer, unsigned int *bufferSize, FILE *input) {
    unsigned int i = 0;
    const unsigned int gradient = 100;
    char *endOfLine;

    /* The algorithm works pretty simple in theory - it loops indefinitely waiting for the first newline (using fgets), increasing the buffer size as necessary each time.
It has to do a few other housekeeping things - like replacing the newline fgets leaves in the buffer with a NULL terminator - but it's pretty simple overall.  Lots of room for off-by-one errors though. :) */

    if ((NULL == bufferSize) || (NULL == input) || (NULL == buffer)) {
        fprintf(stderr, "%s %s:%d - Invalid parameters (buffer = %p, bufferSize = %p, input = %p).\n", __FILE__, __func__, __LINE__, buffer, bufferSize, input);
        exit(EINVAL);
    }

    if (NULL == *buffer) {
        if (0 >= *bufferSize) {
            *bufferSize = gradient;
        }

        *buffer = (char*)malloc(*bufferSize);
    }

    do {
        if (NULL == *buffer) {
            fprintf(stderr, "Unable to allocate memory for string input buffer - tried grabbing %u bytes.\n", *bufferSize);
            exit(ENOMEM);
        }

        if (NULL == fgets(*buffer + i, *bufferSize - i, input)) {
            if (0 != ferror(input)) {
                fprintf(stderr, "Error #%u (%s) occured while reading the input.\n", errno, strerror(errno));
                exit(errno);
            } else if (0 == feof(input)) {
                fprintf(stderr, "Warning: Expected to be at the end of file at this point, but am apparently not.\n");
                // There's no need for this to be fatal... I don't think it should be possible to get here anyway.
            }
```

```
            return NULL;
        } else {
            endOfLine = (char*)memchr(*buffer + i, '\n', *bufferSize - i);

            if (NULL == endOfLine) {
                *bufferSize += gradient;
                i = *bufferSize - 1;

                *buffer = (char*)realloc(*buffer, *bufferSize);
            } else {
                *endOfLine = 0;
                return *buffer;
            }
        }
    } while (1);
}


/*! @function verifyParameters
    @abstract Verifies a given parameter block to ensure it is internally consistant.
    @discussion The various parameters are closely related - for example, block size must be an integer multiple of word size.  This function analyses the given parameter block and returns an error if it is invalid in any way.
    @param parameters The parameters to verify.  Should not be NULL.
    @param silent If true no output will be made.  If false a description of any verification failure will be printed to stderr.
    @result Returns 0 if the parameters check out okay, an error code (usually EINVAL) otherwise. */

int verifyParameters(const SimulationParameters *parameters, bool silent) {
    if (NULL != parameters) {
        Address setSize, blockSize; // These can always be derived from the other parameters, and we need them for validation and simulation, but they're not worth including in the parameter block.

        if (0 == parameters->blockSize) { // Resolve the symbolic block size of 0 (meaning, equal to word size) if necessary.
            blockSize = parameters->wordSize;
        } else {
            blockSize = parameters->blockSize;
        }

        if (FullyAssociative == parameters->associativity) { // Determine the size of each set.
            setSize = parameters->cacheSize;
        } else {
            setSize = parameters->associativity * parameters->blockSize;
        }

        if (0 != (parameters->blockSize % parameters->wordSize)) { // Ensure block size is a whole integer multiple of word size.  Note that we don't have to use 'blockSize' here because even if it is the magic value 0 the modulus will still be 0.
            if (!silent) {
                fprintf(stderr, "Block size ("PRIAddress") is not an integer multiple of word size ("PRIAddress").  See \"%s -h\" for usage information.\n", parameters->blockSize, parameters->wordSize, programName);
            }

            return EINVAL;
        }

        if (0 != (parameters->memorySize % blockSize)) { // Ensure the memory size is a whole integer multiple of block size.
            if (!silent) {
                fprintf(stderr, "Memory size ("PRIAddress") is not an integer multiple of block size ("PRIAddress").  See \"%s -h\" for usage information.\n", parameters->memorySize, blockSize, programName);
            }

            return EINVAL;
        }

        if (0 != (parameters->cacheSize % blockSize)) { // Ensure the cache size is a whole integer multiple of block size.
            if (!silent) {
                fprintf(stderr, "Cache size ("PRIAddress") is not an integer multiple of block size ("PRIAddress").  See \"%s -h\" for usage information.\n", parameters->cacheSize, blockSize, programName);
            }

            return EINVAL;
        }

        if (0 != (parameters->cacheSize % setSize)) { // Ensure the cache size is a whole integer multiple of set size.
            if (!silent) {
                fprintf(stderr, "Cache size ("PRIAddress") is not an integer multiple of set size ("PRIAddress").  See \"%s -h\" for usage information.\n", parameters->cacheSize, setSize, programName);
            }

            return EINVAL;
        }

        return 0;
    } else {
        fprintf(stderr, "%s %s:%d - Invalid parameter, \"parameters\" is NULL.\n", __FILE__, __func__, __LINE__);
        return EINVAL;
    }
}

/*! @function performSimulation
    @abstract Performs a single run through an entire simulation.
    @discussion This is a pretty beefy function.  It performs the entire simulation.  Yikes!  Luckily, it's all pretty simple... just a whole lota recursion with pretty trivial rules to evaluate.

                There aren't too many things that can cause the simulation to abort, but the most common of those few is a misalignment, when the misalignment policy is not forgiving.  These immediately force this function to return an error code.  Most other "errors" (e.g. a misalignment with a liberal misalignment policy) are part of the simulation and handled appropriately.
    @param parameters The parameters defining the simulation.  Must be verified using verifyParameters() - if parameters that don't pass verification are provided to this function, the results are undefined.  Should not be NULL.
    @param stats The statistics structure to record information in.  Is automatically reset by this function as necessary.  Should not be NULL.
    @param accessStream The series of accesses to feed into the simulation.  These are iterated over from begin() to end(), using a forward iterator.  Should not be NULL.
    @result Returns 0 if the simulation completed successfully, an error code otherwise. */

int performSimulation(SimulationParameters *parameters, Stats *stats, const vector<CacheAccess> *accessStream) {
    int err = 0;

    if ((NULL != parameters) && (NULL != stats) && (NULL != accessStream)) {
        Address setSize,
```

```
                numberOfSets,
                blocksPerSet,
                wordsPerBlock,
                currentAddress;
        Address i;
        CacheSet *cacheSets;
        TimeStamp currentTime = 1;
        Address currentBaseAddress, currentBlock, currentSet;
        vector<CacheAccess>::const_iterator accessIter, accessStopper;
        vector<CacheEntry*>::iterator cacheIter, cacheStopper;
        CacheEntry *currentEntry;
        bool foundExistingCacheEntry, needToIssueAdditionalAccess = false, cacheIsFull = false;

        // Calculate derived parameters.

        if (0 == parameters->blockSize) {
            parameters->blockSize = parameters->wordSize;
        }

        if (FullyAssociative == parameters->associativity) {
            setSize = parameters->cacheSize;
        } else {
            setSize = parameters->associativity * parameters->blockSize;
        }

        numberOfSets = parameters->cacheSize / setSize;
        blocksPerSet = setSize / parameters->blockSize;
        wordsPerBlock = parameters->blockSize / parameters->wordSize;

        // Reset stats counters.

        memset(stats, 0, sizeof(Stats));

        // Allocate memory for sets.

        cacheSets = new CacheSet[numberOfSets];

        if (NULL == cacheSets) {
            fprintf(stderr, "Unable to allocate "PRIAddress" cache sets (totalling "PRIAddress" bytes in size).\n", numberOfSets, (Address)(sizeof(CacheSet) *
numberOfSets));
            err = ENOMEM;
        } else {
            // Initialise each set appropriately.

            for (i = 0; i < numberOfSets; ++i) {
                cacheSets[i].currentEntries.resize(blocksPerSet);
                cacheSets[i].isFull = false;
            }

            // Loop through the list of cache accesses.

            for (accessIter = accessStream->begin(), accessStopper = accessStream->end(); (0 == err) && (accessIter != accessStopper); ++accessIter) {
                currentAddress = accessIter->address;

                if ((0 < parameters->memorySize) && (currentAddress >= parameters->memorySize)) { // Verify the given address is within the valid memory range.
#ifndef NDEBUG
                    fprintf(stderr, "Address "PRIAddress" on line %lu is beyond the end of memory.\n", currentAddress, accessIter->inputLine);
#else
                    fprintf(stderr, "Address "PRIAddress" is beyond the end of memory.\n", currentAddress);
#endif

                    err = EDOM;
                } else {
                    if (MPForced == parameters->misalignmentPolicy) { // If forced alignment is being used, adjust the memory address appropriately.
                        needToIssueAdditionalAccess = false;

                        if (0 != (currentAddress % parameters->wordSize)) {
                            if (2 < parameters->verbosity) {
#ifndef NDEBUG
                                printf("Address "PRIAddress" on line %lu is word-misaligned - correcting.\n", currentAddress, accessIter->inputLine);
#else
                                printf("Address "PRIAddress" is word-misaligned - correcting.\n", currentAddress);
#endif
                            }

                            currentAddress -= (currentAddress % parameters->wordSize);
                        }
                    } else { // Otherwise some other misalignment policy is being used, so check our address properly...
                        if ((currentAddress % parameters->blockSize) + parameters->wordSize > parameters->blockSize) { // Does the requested word stradle a block boundary?
                            // TODO: What about the condition where you *cannot* have both blocks in cache simultaneously?  Need to fail in that case... if there is such a
case?

                            ++(stats->stat[OpBasicType(accessIter->operation)][CounterMisalignmentsBlock]);

                            if (MPBlock == parameters->misalignmentPolicy) {
                                if (2 < parameters->verbosity) {
#ifndef NDEBUG
                                    printf("Address "PRIAddress" on line %lu stradles a block boundary (blocks "PRIAddress" and "PRIAddress") - issuing two cache accesses.
\n", currentAddress, accessIter->inputLine, currentAddress / parameters->blockSize, ((currentAddress + parameters->wordSize - 1) / parameters->blockSize));
#else
                                    printf("Address "PRIAddress" stradles a block boundary (blocks "PRIAddress" and "PRIAddress") - issuing two cache accesses.\n",
currentAddress, currentAddress / parameters->blockSize, ((currentAddress + parameters->wordSize - 1) / parameters->blockSize));
#endif
                                }

                                needToIssueAdditionalAccess = true;
                            } else {
                                if (0 <= parameters->verbosity) {
#ifndef NDEBUG
                                    fprintf(stderr, "Address "PRIAddress" on line %lu stradles a block boundary (blocks "PRIAddress" and "PRIAddress") - not permitted;
simulation terminated.\n", currentAddress, accessIter->inputLine, currentAddress / parameters->blockSize, ((currentAddress + parameters->wordSize - 1) /
parameters->blockSize));
#else
                                    fprintf(stderr, "Address "PRIAddress" stradles a block boundary (blocks "PRIAddress" and "PRIAddress") - not permitted; simulation
terminated.\n", currentAddress, currentAddress / parameters->blockSize, ((currentAddress + parameters->wordSize - 1) / parameters->blockSize));
#endif
```

```
                                }

                                err = ENODEV;
                            }
                        } else { // Won't need to issue an additional access, but the word may still be misaligned within the block...
                            needToIssueAdditionalAccess = false;

                            if (0 != (currentAddress % parameters->wordSize)) { // Is it misaligned within the block?
                                ++(stats->stat[OpBasicType(accessIter->operation)][CounterMisalignmentsWord]);

                                if (MPNone == parameters->misalignmentPolicy) {
#ifndef NDEBUG
                                    printf("Address "PRIAddress" on line %lu is word-misaligned within block - not permitted; simulation terminated.\n", currentAddress,
accessIter->inputLine);
#else
                                    printf("Address "PRIAddress" is word-misaligned within block - not permitted; simulation terminated.\n", currentAddress);
#endif

                                    err = ENODEV;
                                } else {
                                    if (2 < parameters->verbosity) {
#ifndef NDEBUG
                                        printf("Address "PRIAddress" on line %lu is word-misaligned within block.\n", currentAddress, accessIter->inputLine);
#else
                                        printf("Address "PRIAddress" is word-misaligned within block.\n", currentAddress);
#endif
                                    }
                                }
                            }
                        }
                    }

                    if (0 == err) { // If we passed all the alignment checking without error...
                        do { // Repeat indefinitely (once for each access)
                            // Calculate current block & set positions.
                            currentBlock = currentAddress / parameters->blockSize;
                            currentBaseAddress = currentBlock * parameters->blockSize;
                            currentSet = currentBlock % numberOfSets;

                            if (1 < parameters->verbosity) {
                                printf("Got ");
                                fprintOpType(stdout, accessIter->operation);
                                printf(" for address "PRIAddress", corresponding to block "PRIAddress", which falls under set "PRIAddress".\n", currentAddress, currentBlock
, currentSet);
                            }

                            // Look for the desired block already in cache...

                            foundExistingCacheEntry = false;

                            for (cacheIter = cacheSets[currentSet].currentEntries.begin(), cacheStopper = cacheSets[currentSet].currentEntries.end();
!foundExistingCacheEntry && (cacheIter != cacheStopper); ++cacheIter) {
                                currentEntry = *cacheIter;

                                if (NULL != currentEntry) {
                                    if (currentEntry->baseAddress == currentBaseAddress) {
                                        if (2 < parameters->verbosity) {
                                            printf("Cache hit.\n");
                                        }

                                        ++(stats->stat[OpBasicType(accessIter->operation)][CounterHits]);

                                        foundExistingCacheEntry = true;

                                        switch (OpBasicType(accessIter->operation)) {
                                            case OpWrite:
                                                currentEntry->isDirty = 1;
                                                currentEntry->lastWritten = currentTime;
                                                break;
                                            case OpRead:
                                            case OpInstruction:
                                                currentEntry->lastRead = currentTime;
                                                break;
                                        }
                                    }
                                }
                            }

                            if (!foundExistingCacheEntry) { // If not already in cache, it's a miss, but may have been loaded previously...
                                // Figure out why it wasn't in cache - i.e. was it a compulsory miss or otherwise?

                                foundExistingCacheEntry = false;

                                for (cacheIter = cacheSets[currentSet].pastEntries.begin(), cacheStopper = cacheSets[currentSet].pastEntries.end(); !foundExistingCacheEntry
&& (cacheIter != cacheStopper); ++cacheIter) {
                                    currentEntry = *cacheIter;

                                    if (currentEntry->baseAddress == currentBaseAddress) {
                                        foundExistingCacheEntry = true;
                                    }
                                }

                                if (foundExistingCacheEntry) { // Was in cache at least once before; must have been thrown out due to capacity or conflict...
                                    switch (currentEntry->evictionReason) {
                                        case EvictionCapacity:
                                            if (2 < parameters->verbosity) {
                                                printf("Capacity miss.\n");
                                            }

                                            ++(stats->stat[OpBasicType(accessIter->operation)][CounterCapacityMisses]);

                                            break;
                                        case EvictionConflict:
                                            if (2 < parameters->verbosity) {
                                                printf("Conflict miss.\n");
                                            }
```

```
                                        ++(stats->stat[OpBasicType(accessIter->operation)][CounterConflictMisses]);

                                        break;
                                    default:
                                        printf("Unable to interpret the reason for cache miss for address "PRIAddress" - given as %d.\n", currentEntry->baseAddress,
currentEntry->evictionReason);
                                }
                            } else { // Never been in cache before; compulsory miss by definition.  Load it up!
                                if (2 < parameters->verbosity) {
                                    printf("Compulsory miss.\n");
                                }

                                ++(stats->stat[OpBasicType(accessIter->operation)][CounterCompulsoryMisses]);

                                // Observe that this is the only place we create CacheEntry's.

                                currentEntry = new CacheEntry;

                                currentEntry->baseAddress = currentBaseAddress;
                                currentEntry->wasPrefetched = (accessIter->operation & OpPrefetch);
                                currentEntry->wasPrefetchedExplicitly = 1;
                                currentEntry->isDirty = 0;
                                currentEntry->lastLoad = currentTime;
                                currentEntry->lastWritten = currentEntry->lastRead = 0;
                                currentEntry->evictionReason = EvictionNotYet;

                                cacheSets[currentSet].pastEntries.push_back(currentEntry);
                            }

                            // Update cache entry

                            switch (OpBasicType(accessIter->operation)) {
                                case OpWrite:
                                    currentEntry->isDirty = 1;
                                    currentEntry->lastWritten = currentTime;
                                    break;
                                case OpRead:
                                case OpInstruction:
                                    currentEntry->lastRead = currentTime;
                                    break;
                            }

                            currentEntry->lastLoad = currentTime;

                            // Load into cache

                            if (cacheSets[currentSet].isFull) { // If it's full, we'll need to evict a page.
                                if (RandomReplacement == parameters->replacementPolicy) { // Random is a special replacement case, as it doesn't need to iterate over th
e blocks in cache like all the other algorithms.
                                    Address randomVictim = random() % blocksPerSet; // On any decent system the low bits of random() are actually random.. unfortunately
some older Linux's (and possibly other *nix's) had issues in this regard.  Their problem, not mine.

                                    if (2 < parameters->verbosity) {
                                        printf("Displacing [randomly] page with baseAddress "PRIAddress" (block "PRIAddress").\n", cacheSets[currentSet].
currentEntries[randomVictim]->baseAddress, cacheSets[currentSet].currentEntries[randomVictim]->baseAddress / parameters->blockSize);
                                    }

                                    if (cacheIsFull) {
                                        cacheSets[currentSet].currentEntries[randomVictim]->evictionReason = EvictionCapacity;
                                    } else {
                                        cacheSets[currentSet].currentEntries[randomVictim]->evictionReason = EvictionConflict;
                                    }

                                    cacheSets[currentSet].currentEntries[randomVictim] = currentEntry;
                                } else {
                                    vector<CacheEntry*>::iterator victim;

                                    // Sort through the list and find out which ones are to be evicted.

                                    for (victim = cacheSets[currentSet].currentEntries.begin(), cacheIter = victim + 1, cacheStopper = cacheSets[currentSet].
currentEntries.end(); cacheIter != cacheStopper; ++cacheIter) {
                                        switch (parameters->replacementPolicy) {
                                            case LRUReplacement:
                                                if (max((*victim)->lastRead, (*victim)->lastWritten) > max((*cacheIter)->lastRead, (*cacheIter)->lastWritten)) {
                                                    victim = cacheIter;
                                                }

                                                break;
                                            case LRRReplacement:
                                                if ((*victim)->lastRead > (*cacheIter)->lastRead) {
                                                    victim = cacheIter;
                                                }

                                                break;
                                            case LRWReplacement:
                                                if ((*victim)->lastWritten > (*cacheIter)->lastWritten) {
                                                    victim = cacheIter;
                                                }

                                                break;
                                            case FIFOReplacement:
                                                if ((*victim)->lastLoad > (*cacheIter)->lastLoad) {
                                                    victim = cacheIter;
                                                }

                                                break;
                                        }
                                    }

                                    if (2 < parameters->verbosity) {
                                        printf("Displacing page with baseAddress "PRIAddress" (block "PRIAddress").\n", (*victim)->baseAddress, (*victim)->baseAddress /
parameters->blockSize);
                                    }

                                    if (cacheIsFull) {
```

```
                                        (*victim)->evictionReason = EvictionCapacity;
                                    } else {
                                        (*victim)->evictionReason = EvictionConflict;
                                    }

                                    // Replace our victim with our new entry... note that the victim isn't lost - it's still in the pastEntries vector.
                                    *victim = currentEntry;
                                }
                            } else { // Find an empty spot, given the cache set is not full.
                                CacheEntry *symbolicNULL = NULL;

                                // Find the first empty spot.

                                cacheIter = find(cacheSets[currentSet].currentEntries.begin(), cacheSets[currentSet].currentEntries.end(), symbolicNULL);

                                if (cacheIter == cacheSets[currentSet].currentEntries.end()) {
                                    fprintf(stderr, "Internal error - isFull is not set for set %u, but find found no empty spots.\n", currentSet);
                                    exit(EFAULT);
                                }

                                // Insert our new entry.

                                *cacheIter = currentEntry;

                                // See if there's any more empty spots...

                                cacheIter = find(cacheSets[currentSet].currentEntries.begin(), cacheSets[currentSet].currentEntries.end(), symbolicNULL);

                                if (cacheIter == cacheSets[currentSet].currentEntries.end()) { // If there isn't, mark our set as full, and then see if the whole cache
is full.
                                    bool cacheIsNowFull = true; // Our hypothesis

                                    if ((1 < blocksPerSet) && (2 < parameters->verbosity)) {
                                        printf("Set "PRIAddress" is now full.\n", currentSet);
                                    }

                                    cacheSets[currentSet].isFull = true;

                                    // Our proof (one way or another)
                                    for (i = 0; cacheIsNowFull && (i < numberOfSets); ++i) {
                                        if (0 == cacheSets[i].isFull) {
                                            cacheIsNowFull = false;
                                        }
                                    }

                                    if (cacheIsNowFull && !cacheIsFull) {
                                        if (0 < parameters->verbosity) {
                                            printf("Cache is now full (after %lu accesses).\n", currentTime);
                                        }

                                        cacheIsFull = true; // Used at present to distinguish between conflict and capacity misses.
                                    }
                                }
                            }
                        }

                        // Increment the current simulation time.  Note that this happens once for every actual access, not for every CacheAccess processed - remember
that CacheAccess's are not necessarily atomic, from the cache's internal point of view.

                        ++currentTime;

                        // If we're being at all verbose, print out a progress marker every 10000 units of time.
                        if (0 < parameters->verbosity) {
                            if (0 == (currentTime % 10000)) {
                                printf(".");
                                fflush(stdout);
                            }
                        }

                        if (needToIssueAdditionalAccess) { // Adjust our current address to the start of the next block, so we can load that in as well on the next time
through.
                            // TEMPORARY
                            //printf("Issuing additional access...\n");

                            currentAddress = currentAddress + parameters->wordSize - 1;
                            currentAddress -= currentAddress % parameters->blockSize;

                            needToIssueAdditionalAccess = false; // And make sure we don't loop indefinitely.
                        } else {
                            break; // Escape our do loop.  This is the only way out (aside from err being set to non-zero).
                        }
                    } while (0 == err);
                }
            }
        }

        // Clean up memory.

        for (currentSet = 0; currentSet < numberOfSets; ++currentSet) {
            for (cacheIter = cacheSets[currentSet].pastEntries.begin(), cacheStopper = cacheSets[currentSet].pastEntries.end();  cacheIter != cacheStopper; ++cacheIter)
{
                if (NULL != *cacheIter) {
                    delete *cacheIter;
                }
            }
        }

        delete[] cacheSets;
    }
} else {
    fprintf(stderr, "%s %s:%d - Invalid parameters (parameters = %p, stats = %p, accessStream = %p).\n", __FILE__, __func__, __LINE__, parameters, stats, accessStream);
    err = EINVAL;
}

return err;
}
```

```
/*! @function readAccessPattern
    @abstract Reads a series of CacheAccess's from a given FILE.
    @discussion At present only one file format is supported, which is a trivial ASCII format - each line contains two numbers, the first being the OpType and the second being the address.  Any line not containing or starting with a number is ignored.
    @param input The FILE to read data from.  Should not be NULL.
    @param addressBase The explicit base of the addresses.  Useful in cases where base 16 is used but no "0x" prefix is used (as is commonly done - dineroIV always assumes base 16 addresses, so it's input sources usually suffer from this problem).  If 0 the base is auto-detected as best as possible, with base 10 (decimal) being the default if no other indication is available.  Any value which is valid for strtol is valid, which at time of writing is 0 or 2 through 36.
    @param accessStream The vector on which to append new accesses read from the file.  This vector is not cleared by this function - consequently you can use this function multiple times to concatenate data from multiple sources.  Should not be NULL.
    @result Returns 0 if successful, an error code otherwise (in which case the state of accessStream is undefined). */

int readAccessPattern(FILE *input, int addressBase, vector<CacheAccess> *accessStream) {
    int err = 0;

    if ((NULL != input) && (NULL != accessStream)) {
        unsigned int bufferSize = 0;
        char *buffer = NULL, *readBuffer, *end;
        long long currentRawInput;
        unsigned long currentLine = 1;
        CacheAccess currentAccess;

        while ((0 == err) && (NULL != (readBuffer = fgetline(&buffer, &bufferSize, input)))) { // While we've got more input to parse...
            // See if we can get an initial integer... if we can't we ignore this line (assume it's empty, or a comment, or whatever else).

            currentRawInput = strtoll(buffer, &end, 0);

            if (buffer != end) {
#ifndef NDEBUG
                currentAccess.inputLine = currentLine;
#endif

                // The first integer read is the OpType.

                currentAccess.operation = (OpType)currentRawInput;

                if (currentAccess.operation != currentRawInput) { // Check that the OpType read is valid, using the cast as an implicit mask.
                    fprintf(stderr, "Op type of %lld on line %lu is too large - not recognised as a valid operation.\n", currentRawInput, currentLine);
                    err = EIO;
                } else {
                    // Read the next integer (which will be the memory address the access refers to).

                    readBuffer = end;

                    currentRawInput = strtoll(readBuffer, &end, addressBase);

                    if (readBuffer != end) { // Ensure we did actually read something...
                        currentAccess.address = (Address)currentRawInput;

                        if (currentAccess.address != currentRawInput) {
                            fprintf(stderr, "Address of %lld on line %lu is too large for this particular version of SimpleCacheSim (maximum size is %lu bits).\n", currentRawInput, currentLine, sizeof(Address) * 8);
                            err = EIO;
                        }
                    } else { // Otherwise we have to barf... sorry, can't have an OpType with an address (well.. at least not that I'm aware).
                        fprintf(stderr, "Unable to read address corresponding to operation ");
                        fprintOpType(stderr, currentAccess.operation);
                        fprintf(stderr, " on line %lu - found \"%s\" instead.\n", currentLine, readBuffer);
                        err = EIO;
                    }

                    if (0 == err) { // If all went well, add our new access to the vector.
                        accessStream->push_back(currentAccess);
                    }
                }
            }

            // Count the lines, for debugging purposes.
            ++currentLine;
        }

        // Free our buffer, which could be very big by now (i.e. the length of the longest line in our input).

        if (NULL != buffer) {
            free(buffer);
        }
    } else {
        fprintf(stderr, "%s %s:%d - Invalid parameters (input = %p, addressBase = %d, accessStream = %p).\n", __FILE__, __func__, __LINE__, input, addressBase, accessStream);
        err = EINVAL;
    }

    return err;
}


int main(int argc, char* const argv[]) {
    static struct option longopts[] = { { "addressbase", required_argument, NULL, 'A' },
                                        { "misalignment", required_argument, NULL, 'M' },
                                        { "SQL", required_argument, NULL, 'S' },
                                        { "associativity", required_argument, NULL, 'a' },
                                        { "blocksize", required_argument, NULL, 'b' },
                                        { "cachesize", required_argument, NULL, 'c' },
                                        { "help", no_argument, NULL, 'h' },
                                        { "memorysize", required_argument, NULL, 'm' },
                                        { "replacement", required_argument, NULL, 'r' },
                                        { "verbosity", required_argument, NULL, 'v' },
                                        { "wordsize", required_argument, NULL, 'w' },
                                        { NULL, 0, NULL, 0 } };
    int chosenOption;
    const unsigned int defaultAssociativity = FullyAssociative,
                       defaultReplacementPolicy = LRUReplacement,
                       defaultMisalignmentPolicy = MPNone;
    const Address defaultBlockSize = 0, // Defer to word size
                  defaultCacheSize = 16 * 1024, // 16kiB
```

```
                defaultMemorySize = 0, // No defined limit
                defaultWordSize = 4;
    int defaultVerbosity = 0;
    SimulationParameters parameters = { 0, 0, defaultMisalignmentPolicy, 0, 0, defaultMemorySize, 0, defaultVerbosity };
    FILE *input;
    Stats stats;
    unsigned long total[NUMBER_OF_COUNTERS], totalAccesses, totalMisses, totalMisalignments, j, k;
    const int defaultAddressBase = 0;
    int addressBase = defaultAddressBase;
    vector<CacheAccess> accessStream;
    range<unsigned int> associativities, replacementPolicies;
    range<unsigned int>::iterator associativitiesIter, associativitiesStopper, replacementPoliciesIter, replacementPoliciesStopper;
    range<Address> blockSizes, cacheSizes, wordSizes;
    range<Address>::iterator blockSizesIter, blockSizesStopper, cacheSizesIter, cacheSizesStopper, wordSizesIter, wordSizesStopper;
    char *currentWord, *end;
    int err;
    bool didRunAtLeastOneSimulation = false, formatAsSQLInsert = false;

    // We use the global variable programName throughout the program for debugging purposes.
    programName = argv[0];

    // Scan through all our CLI options and handle them appropriately.

    while (-1 != (chosenOption = getopt_long(argc, argv, "A:M:S:a:b:c:hm:r:v:w:", longopts, NULL))) {
        switch (chosenOption) {
            case 'A': // Address base
                addressBase = strtol(optarg, &end, 0);

                if ((0 != *end) || (optarg == end) || (0 > addressBase) || (1 == addressBase) || (36 < addressBase)) {
                    fprintf(stderr, "Invalid address base, \"%s\".  See \"%s -h\" for usage information.\n", optarg, programName);
                    exit(EINVAL);
                }

                break;
            case 'M': // Misalignment
                if (0 == strcasecmp(optarg, "none")) {
                    parameters.misalignmentPolicy = MPNone;
                } else if (0 == strcasecmp(optarg, "word")) {
                    parameters.misalignmentPolicy = MPWord;
                } else if (0 == strcasecmp(optarg, "block")) {
                    parameters.misalignmentPolicy = MPBlock;
                } else if (0 == strcasecmp(optarg, "forced")) {
                    parameters.misalignmentPolicy = MPForced;
                } else {
                    fprintf(stderr, "Unknown misalignment policy, \"%s\".  See \"%s -h\" for usage information.\n", optarg, programName);
                    exit(EINVAL);
                }

                break;
            case 'S': // SQL
                // We include these in the program, even though they're technically static, because they can be generated directly from the internal names here, ensuring
that any changes we make (e.g. by adding an extra op type, or extra counters) can be automatically adopted.

                if (0 == strcasecmp(optarg, "create")) { // SQL command to create results table (and summary views).
                    // CREATE TABLE

                    printf("CREATE TABLE SimpleCacheSim (\n\tAccesses BIGINT,\n\tAssociativity VARCHAR(100),\n\tReplacementPolicy VARCHAR(100),\n\tBlockSize
INT,\n\tCacheSize INT,\n\tWordSize INT,\n");

                    for (k = 0; k < NUMBER_OF_COUNTERS; ++k) {
                        for (j = 0; j < NUMBER_OF_OP_TYPES; ++j) {
                            printf("\t");
                            fprintCounterName(stdout, k, true);
                            fprintOpType(stdout, j, true);
                            printf(" BIGINT,\n");
                        }
                    }

                    printf("\tPRIMARY KEY (Accesses, Associativity, ReplacementPolicy, BlockSize, CacheSize, WordSize)\n);\n");

                    // CREATE COLLATION

                    printf("CREATE VIEW SimpleCacheSimCollated (\n\tAccesses,\n\tAssociativity,\n\tReplacementPolicy,\n\tBlockSize,\n\tCacheSize,\n\tWordSize,\n");

                    for (k = 0; k < NUMBER_OF_COUNTERS; ++k) {
                        printf("\t");
                        fprintCounterName(stdout, k, true);

                        if (k != (NUMBER_OF_COUNTERS - 1)) {
                            printf(",\n");
                        }
                    }

                    printf("\n) AS SELECT Accesses, Associativity, ReplacementPolicy, BlockSize, CacheSize, WordSize, ");

                    for (k = 0; k < NUMBER_OF_COUNTERS; ++k) {
                        for (j = 0; j < NUMBER_OF_OP_TYPES; ++j) {
                            fprintCounterName(stdout, k, true);
                            fprintOpType(stdout, j, true);

                            if (j != (NUMBER_OF_OP_TYPES - 1)) {
                                printf(" + ");
                            } else {
                                if (k != (NUMBER_OF_COUNTERS - 1)) {
                                    printf(", ");
                                }
                            }
                        }
                    }

                    printf(" FROM SimpleCacheSim;\n");

                    // CREATE SUMMARY

                    printf("CREATE VIEW SimpleCacheSimSummary
(\n\tAccesses,\n\tAssociativity,\n\tReplacementPolicy,\n\tBlockSize,\n\tCacheSize,\n\tWordSize,\n\tHits,\n\tMisses\n) AS SELECT Accesses, Associativity, ReplacementPolicy,
```

```
BlockSize, CacheSize, WordSize, ");

                    fprintCounterName(stdout, CounterHits, true);
                    printf(", ");
                    fprintCounterName(stdout, CounterCapacityMisses, true);
                    printf(" + ");
                    fprintCounterName(stdout, CounterCompulsoryMisses, true);
                    printf(" + ");
                    fprintCounterName(stdout, CounterConflictMisses, true);

                    printf(" FROM SimpleCacheSimCollated;\n");

                    exit(0); // Doesn't really make too much sense to continue operation after doing this; it's unlikely the user will want to generate these statements as part of a normal simulation run.
                //} else if (0 == strcasecmp(optarg, "alter")) { // SQL command to modify results table to new format, as may be necessary when upgrading to a new versiono of SimpleCacheSim.  Not yet needed as this is the first version.
                } else if (0 == strcasecmp(optarg, "insert")) { // Make each output in the form of an SQL insert statement
                    formatAsSQLInsert = true;
                } else { // No idea
                    fprintf(stderr, "Unknown SQL directive, \"%s\".  See \"%s -h\" for usage information.\n", optarg, programName);
                    exit(EINVAL);
                }

                break;
            case 'a': // Associativity
                while (NULL != optarg) {
                    currentWord = strsep(&optarg, ", \t\n\r");

                    if (0 != currentWord) {
                        if (0 == strcasecmp(currentWord, "direct")) {
                            associativities.insert(DirectMapped);
                        } else if (0 == strcasecmp(currentWord, "full")) {
                            associativities.insert(FullyAssociative);
                        } else {
                            err = readRange(associativities, currentWord);

                            if (0 != err) {
                                fprintf(stderr, "Unable to read range of associativities from \"%s\".  See \"%s -h\" for usage information.\n", currentWord, programName);
                                exit(EINVAL);
                            }
                        }
                    }
                }

                break;
            case 'b': // Block size
                err = readRange(blockSizes, optarg);

                if (0 != err) {
                    fprintf(stderr, "Unable to read block size(s) from \"%s\".  See \"%s -h\" for usage information.\n", optarg, programName);
                    exit(EINVAL);
                }

                break;
            case 'c': // Cache size
                err = readRange(cacheSizes, optarg);

                if (0 != err) {
                    fprintf(stderr, "Unable to read cache size(s) from \"%s\".  See \"%s -h\" for usage information.\n", optarg, programName);
                    exit(EINVAL);
                }

                break;
            case 'h': // Help
                printf("Usage: %s [OPTIONS] [INPUT]\n\nOPTIONS:\n", programName);

                printf("\t--addressbase/-A\tThe numerical base of addresses provided as input.  Useful if the addresses are hexadecimal but without a preceding \"0x\", which would otherwise confuse the input parser.  Valid values are 2 through 36 inclusive, or the special value 0, which allows any base by trying to detect the base in use (e.g. any input starting with \"0x\" is hexadecimal, any starting with \"0\" is assumed to be octal, and if all else fails the input is assumed to be decimal.  Defaults to %d.\n", defaultAddressBase);
                printf("\t--misalignment/-M\tThe types of misalignment to permit, if any.  Valid values are \"none\", \"word\", \"block\" (which implies allowing word misalignment as well) or \"forced\" (where all addresses are forced to align).  Defaults to ");
                fprintMisalignmentPolicy(stdout, defaultMisalignmentPolicy);
                printf(".\n\t--SQL/-S\t\tSQL functionality.  Takes a string as an argument, which may be either \"create\" or \"insert\".  The \"create\" directive instructs the program to output the SQL CREATE statements that generate a table and summary views suitable for holding the tabular output from this program.  These statements can be executed in a database client - e.g. psql for PostgreSQL.  The \"insert\" directive instructs the program to output all tabular results as SQL INSERT statements.  The output can then be executed directly in a database client to import data into the table(s) generated by the \"create\" directive.  Note: if your database supports a COPY command, it is usually much faster to use this on the normal tabular output than to use INSERT statements.  The \"insert\" directive is only meaningful if a negative verbosity is specified.\n");
                printf("\t--associativity/-a\tThe associativity of the cache, which may be either \"direct\", \"full\" or a numeric value indicating the number of ways mapped.  Multiple associativities may be separated by commas, and/or provided as ranges distinguished by hyphens.  Defaults to ");
                fprintAssociativity(stdout, defaultAssociativity);
                printf(".\n\t--blocksize/-b\t\tThe size (in bytes) of each cache block.  This is the smallest unit that the cache will be capable of holding (not to be confused with the smallest unit which can be transferred, which is the word size).  Must be an integer multiple of the word size.  Multiple block sizes may be separated by commas, and/or provided as ranges distinguished by hyphens.  Defaults to %u.\n", defaultBlockSize);
                printf("\t--cachesize/-c\t\tThe size (in bytes) of the cache.  Must be an integer multiple of block size and associativity.  Multiple cache sizes may be separated by commas, and/or provided as ranges distinguished by hyphens.  Defaults to %u.\n", defaultCacheSize);
                printf("\t--help/-h\t\tPrints this usage information and exits.\n");
                printf("\t--memorysize/-m\t\tThe size (in bytes) of the memory behind the cache.  Optional, used only to validate input values.  A value of 0 indicates no defined limit.  Note that only one value is permitted, unlike other parameters, as this is purely for input validation.  Defaults to %u.\n", defaultMemorySize);
                printf("\t--replacement/-r\tThe replacement policy in N-way associative caches (N > 0).  May be any of \"random\", \"LRU\" (Least Recently Used), \"FIFO\" (First In First Out), \"LRR\" (Least Recently Read) or \"LRW\" (Least Recently Written).  Multiple policies may be separated by commas.  Defaults to ");
                fprintReplacementPolicy(stdout, defaultReplacementPolicy);
                printf(".\n\t--verbosity/-v\t\tSets the verbosity of the output, where 0 corresponds to silent operation, 1 prints out the simulation parameters and other key information, 2 also prints out information about every cache operation as it is simulated, and 3 also prints out even more details about what's happening with each and every cache access.  A negative verbosity indicates tabular output; -1 outputs only the raw data, while -2 or lower also outputs a header on the first row, indicating what value each column represents.  Use high verbosities with caution - the volume of output can be tremendous.  Defaults to %d.\n", defaultVerbosity);
                printf("\t--wordsize/-w\t\tThe size (in bytes) of one word, which is the smallest unit of data that can be transferred to or from cache.  Multiple word sizes may be separated by commas, and/or provided as ranges distinguished by hyphens.  Defaults to %u.\n", defaultWordSize);

                printf("\nINPUT:\n\tThe name of the file to read requests from.  If not provided, stdin is used.\n");

                exit(0); // Makes no sense to print out usage and then continue execution; typical invocations of this help don't provide any other parameters.

                break;
            case 'm': // Memory size
```

```
                parameters.memorySize = strtol(optarg, &end, 0);

                if ((0 != *end) || (optarg == end)) {
                    fprintf(stderr, "Invalid memory size, \"%s\".  See \"%s -h\" for usage information.\n", optarg, programName);
                    exit(EINVAL);
                }

                break;
            case 'r': // Replacement policy
                while (NULL != optarg) {
                    currentWord = strsep(&optarg, ", \t\n\r");

                    if (0 != currentWord) {
                        if (0 == strcasecmp(currentWord, "random")) {
                            replacementPolicies.insert(RandomReplacement);
                        } else if (0 == strcasecmp(currentWord, "LRU")) {
                            replacementPolicies.insert(LRUReplacement);
                        } else if (0 == strcasecmp(currentWord, "LRR")) {
                            replacementPolicies.insert(LRRReplacement);
                        } else if (0 == strcasecmp(currentWord, "LRW")) {
                            replacementPolicies.insert(LRWReplacement);
                        } else if (0 == strcasecmp(currentWord, "FIFO")) {
                            replacementPolicies.insert(FIFOReplacement);
                        } else {
                            fprintf(stderr, "Unknown replacement policy, \"%s\".  See \"%s -h\" for usage information.\n", currentWord, programName);
                            exit(EINVAL);
                        }
                    }
                }

                break;
            case 'v': // Verbose
                parameters.verbosity = strtol(optarg, &end, 0);

                if ((0 != *end) || (optarg == end)) {
                    fprintf(stderr, "Invalid verbosity level, \"%s\".  See \"%s -h\" for usage information.\n", optarg, programName);
                    exit(EINVAL);
                }

                break;
            case 'w': // Word size
                err = readRange(wordSizes, optarg);

                if (0 != err) {
                    fprintf(stderr, "Unable to read word size(s) from \"%s\".  See \"%s -h\" for usage information.\n", optarg, programName);
                    exit(EINVAL);
                }

                break;
            default:
                fprintf(stderr, "See \"%s -h\" for usage information.\n", programName);
                exit(EINVAL);
        }
    }

    // Invoke default values for any parameters which weren't provided explicitly.

    if (associativities.empty()) {
        associativities.insert(defaultAssociativity);
    }

    if (replacementPolicies.empty()) {
        replacementPolicies.insert(defaultReplacementPolicy);
    }

    if (wordSizes.empty()) {
        wordSizes.insert(defaultWordSize);
    }

    if (cacheSizes.empty()) {
        cacheSizes.insert(defaultCacheSize);
    }

    if (blockSizes.empty()) {
        blockSizes.insert(defaultBlockSize);
    }

    // See if we have a file name argument after all those options...

    if ((optind + 1) < argc) {
        fprintf(stderr, "Too many arguments from \"%s\" onwards.  See \"%s -h\" for usage information.\n", argv[optind + 1], programName);
        exit(EINVAL);
    } else if (optind < argc) {
        input = fopen(argv[optind], "r");

        if (NULL == input) {
            fprintf(stderr, "Unable to open the file \"%s\" for input, error #%d (%s).\n", argv[optind], errno, strerror(errno));
            exit(errno);
        }
    } else {
        input = stdin;
    }

    // Read in our list of cache accesses from file (whether that be an actual file or stdin).

    err = readAccessPattern(input, addressBase, &accessStream);

    if (0 != err) {
        fprintf(stderr, "Reading access stream failed, error #%d (%s).\n", err, strerror(err));
        exit(err);
    }

    if (!formatAsSQLInsert && (-1 > parameters.verbosity)) { // In tabular (and non-SQL-formatted) operation, high verbosities include a table header.
        printf("Accesses\tAssociativity\tReplacement\tBlock Size\tCache Size\tWord Size");

        for (k = 0; k < NUMBER_OF_COUNTERS; ++k) {
```

```
            for (j = 0; j < NUMBER_OF_OP_TYPES; ++j) {
                printf("\t");
                fprintCounterName(stdout, k);
                printf(" ");
                fprintOpType(stdout, j);
            }
        }

        printf("\n");
        fflush(stdout);
    }

    // Now iterate over each parameter recursively, creating every possible permutation for the parameters given.  Simulate each one where possible (ignoring any errors, provided we can perform at least one simulation for the given parameters).

    for (associativitiesIter = associativities.begin(), associativitiesStopper = associativities.end(); associativitiesIter != associativitiesStopper; ++associativitiesIter) {
        parameters.associativity = *associativitiesIter;

        for (replacementPoliciesIter = replacementPolicies.begin(), replacementPoliciesStopper = replacementPolicies.end(); replacementPoliciesIter != replacementPoliciesStopper; ++replacementPoliciesIter) {
            parameters.replacementPolicy = *replacementPoliciesIter;

            for (cacheSizesIter = cacheSizes.begin(), cacheSizesStopper = cacheSizes.end(); cacheSizesIter != cacheSizesStopper; ++cacheSizesIter) {
                parameters.cacheSize = *cacheSizesIter;

                for (wordSizesIter = wordSizes.begin(), wordSizesStopper = wordSizes.end(); wordSizesIter != wordSizesStopper; ++wordSizesIter) {
                    parameters.wordSize = *wordSizesIter;

                    for (blockSizesIter = blockSizes.begin(), blockSizesStopper = blockSizes.end(); blockSizesIter != blockSizesStopper; ++blockSizesIter) {
                        parameters.blockSize = *blockSizesIter;

                        // See if our current permutation of the parameters is valid.

                        err = verifyParameters(&parameters, true);

                        if (0 == err) { // If it is, try performing the simulation.
                            err = performSimulation(&parameters, &stats, &accessStream);

                            if (0 == err) { // Simulation successful - output the results.
                                didRunAtLeastOneSimulation = true;

                                if (0 <= parameters.verbosity) { // If we're outputting in non-tabular form...
                                    if (0 < parameters.verbosity) { // If we're being verbose we output the simulation parameters as well as the results...
                                        Address wordsPerBlock = parameters.blockSize / parameters.wordSize, numberOfSets = ((0 == parameters.associativity) ? parameters.cacheSize : (parameters.cacheSize / parameters.associativity)), setSize = parameters.cacheSize / numberOfSets, blocksPerSet = setSize / parameters.blockSize;

                                        printf("Simulation parameters:\n\tExplicit cache accesses: %lu\n\tAssociativity: ", accessStream.size());
                                        fprintAssociativity(stdout, parameters.associativity);
                                        printf("\n\tBlock size: "PRIAddress" bytes / "PRIAddress" words\n\tCache size: "PRIAddress" bytes / "PRIAddress" words / "PRIAddress" blocks / "PRIAddress" sets\n\tMemory size: "PRIAddress" bytes / "PRIAddress" words / "PRIAddress" blocks / %f caches\n\tReplacement policy: ", parameters.blockSize, wordsPerBlock, parameters.cacheSize, parameters.cacheSize / parameters.wordSize, parameters.cacheSize / parameters.blockSize, numberOfSets, parameters.memorySize, parameters.memorySize / parameters.wordSize, parameters.memorySize / parameters.blockSize, (double)parameters.memorySize / (double)parameters.cacheSize);
                                        fprintReplacementPolicy(stdout, parameters.replacementPolicy);
                                        printf("\n\tMisalignment policy: ");
                                        fprintMisalignmentPolicy(stdout, parameters.misalignmentPolicy);
                                        printf("\n\tSet size: "PRIAddress" bytes / "PRIAddress" words / "PRIAddress" blocks\n\tWord size: "PRIAddress" bytes\n\n", setSize, setSize / parameters.wordSize, blocksPerSet, parameters.wordSize);
                                    }

                                    // Output the actual results.

                                    for (k = 0; k < NUMBER_OF_COUNTERS; ++k) {
                                        total[k] = 0;

                                        for (j = 0; j < NUMBER_OF_OP_TYPES; ++j) {
                                            total[k] += stats.stat[j][k];
                                        }

                                        fprintCounterName(stdout, k);
                                        printf(": %lu\n", total[k]);

                                        for (j = 0; j < NUMBER_OF_OP_TYPES; ++j) {
                                            if (0 < stats.stat[j][k]) {
                                                printf("\t");
                                                fprintOpType(stdout, j);
                                                printf(": %lu\n", stats.stat[j][k]);
                                            }
                                        }
                                    }

                                    totalMisses = total[CounterConflictMisses] + total[CounterCompulsoryMisses] + total[CounterCapacityMisses];
                                    totalAccesses = totalMisses + total[CounterHits];

                                    if (MPNone != parameters.misalignmentPolicy) {
                                        totalMisalignments = total[CounterMisalignmentsWord] + total[CounterMisalignmentsBlock];
                                        printf("\nTotal misalignments: %lu", totalMisalignments);
                                    }

                                    printf("\nTotal misses: %lu\n", totalMisses);

                                    if (accessStream.size() != totalAccesses) {
                                        printf("Total cache accesses: %lu (of %lu direct accesses)\n\n", totalAccesses, accessStream.size());
                                    } else {
                                        printf("Total cache accesses: %lu\n\n", totalAccesses);
                                    }

                                    printf("Miss rate: %f\n\tCompulsory: %f\n\tCapacity: %f\n\tConflict: %f\n", ((double)totalMisses) / (double)totalAccesses, (double)total[CounterCompulsoryMisses] / (double)totalAccesses, (double)total[CounterCapacityMisses] / (double)totalAccesses, (double)total[CounterConflictMisses] / (double)totalAccesses);
                                } else { // Tabular form.
                                    // Note that tabular form does not have a heading for each simulation run - if the verbosity was high enough a single heading would have been outputted at the start.

                                    if (formatAsSQLInsert) { // The SQL insert just has a few extra words squished in, really.
                                        printf("INSERT INTO SimpleCacheSim ");
```

```
                                    if (-1 > parameters.verbosity) {
                                        printf("(Accesses, Associativity, ReplacementPolicy, BlockSize, CacheSize, WordSize");

                                        for (k = 0; k < NUMBER_OF_COUNTERS; ++k) {
                                            for (j = 0; j < NUMBER_OF_OP_TYPES; ++j) {
                                                printf(", ");
                                                fprintCounterName(stdout, k, true);
                                                fprintOpType(stdout, j, true);
                                            }
                                        }

                                        printf(") ");
                                    }

                                    printf("VALUES (");

                                    printf("%lu, '", accessStream.size());
                                    fprintAssociativity(stdout, parameters.associativity);
                                    printf("', '");
                                    fprintReplacementPolicy(stdout, parameters.replacementPolicy);
                                    printf("', "PRIAddress", "PRIAddress", "PRIAddress"", parameters.blockSize, parameters.cacheSize, parameters.wordSize);

                                    for (k = 0; k < NUMBER_OF_COUNTERS; ++k) {
                                        for (j = 0; j < NUMBER_OF_OP_TYPES; ++j) {
                                            printf(", %lu", stats.stat[j][k]);
                                        }
                                    }

                                    printf(");\n");
                                } else {
                                    printf("%lu\t", accessStream.size());
                                    fprintAssociativity(stdout, parameters.associativity);
                                    printf("\t");
                                    fprintReplacementPolicy(stdout, parameters.replacementPolicy);
                                    printf("\t"PRIAddress"\t"PRIAddress"\t"PRIAddress"", parameters.blockSize, parameters.cacheSize, parameters.wordSize);

                                    for (k = 0; k < NUMBER_OF_COUNTERS; ++k) {
                                        for (j = 0; j < NUMBER_OF_OP_TYPES; ++j) {
                                            printf("\t%lu", stats.stat[j][k]);
                                        }
                                    }

                                    printf("\n");
                                }
                            }

                            fflush(stdout);
                        } else if (ENODEV == err) { // ENODEV indicates a memory alignment (or similar) failure, which cannot be detected earlier but should not be considered fatal for batch purposes
                            printf("\n");
                        } else { // Some other error occured (possibly a memory error), so just fail fatally.
                            fprintf(stderr, "Simulation failed, error #%d (%s).\n", err, strerror(err));
                            exit(err);
                        }
                    }
                }
            }
        }
    }

    // Check that we did run at least one simulation - otherwise output an appropriate error message.

    if (!didRunAtLeastOneSimulation) {
        fprintf(stderr, "No possible permutations of the input parameters are simulatable.\n");
        exit(EINVAL);
    }

    return 0;
}
```