Heap Heap Hooray
Developer Manual

Table of Contents

System Design

Introduction

The MiniJava compiler was a project for the Compiler Theory course that followed Andrew Appel's *Modern Compiler Implementation in Java*. This compiler, which we've named "MJC" ("MiniJava Compiler"), compiles MiniJava source code and produces assembly code for the SPARC architecture.

MJC includes most of the common phases of code compilation: lexing/parsing, type-checking (semantics), translation (intermediate representation), assembly code generation, register allocation, and lastly the output step. The assembly output can then be linked alongside our compiler runtime (CRT for short) to produce an executable file.

For the purposes of the Compiler Theory course, MiniJava is a sufficient language for a simple compiler. It is a subset of the Java programming language, and while it does contain powerful features such as class inheritance, it lacks other Java features such as the null keyword, and functions can only return integer values, objects cannot escape functions, etc.

For research purposes, MiniJava does not provide a great environment. As with Java, there is no way to manually free memory. However, MiniJava does not contain a garbage collector out-of-the-box as Java does, so all memory allocations are leaked and will never be freed.

As part of this project, we have implemented garbage collection in the CRT using the C programming language. We chose to implement a few unique methods of garbage collection to allow us to test configurations and determine which methods perform best for MiniJava source code, based on which data structures and algorithms are exhibited by the source code. We hope that our findings here can be applied to other programming languages.

Runtime

The CRT is written in C and compiled with GCC for the SPARC architecture. Assembled MiniJava programs must be linked with the CRT to allow memory allocation and garbage collection.

The CRT powers the runtime garbage collector and some core parts of the MiniJava language, such as its print functions.

These programs can configure the CRT to their liking by setting the garbage collection method and heap type. This is accomplished through the use of specific flags when compiling programs with MJC. These flags generate function calls in the assembly output that will configure the CRT options when the program begins execution.

Garbage Collector

The CRT implements the following methods of garbage collection:

- *None (default)*
    - No garbage collection is performed and all memory is leaked

- *Reference counting*
    - Objects maintain an atomic count of how many references to them exist in the program. When this count drops to zero, the object is provably garbage

- *Mark-sweep*
    - Objects are determined to be reachable ("marked") based on references found during a traversal of the program stack. Unreachable objects are later freed ("swept").

- *Copying*
    - Reachable objects are copied over to a second heap, and are defragmented in the process. The original heap provably contains only garbage and is destroyed.

- *Generational*
    - Built on the copying method, generational uses multiple heaps to represent "generations" of objects which have survived garbage collection cycles. Higher generations are less often attempted to be collected, saving execution time.

Heap

The CRT implements the following methods of memory management:

- *STL (default)*
    - Uses functions from the C standard library (malloc/free) to manage memory.

- *Chunk*
    - Deals out memory blocks partitioned out of a large chunk. Required for some garbage collection methods which need more control over the underlying memory of the heap.

Containerization

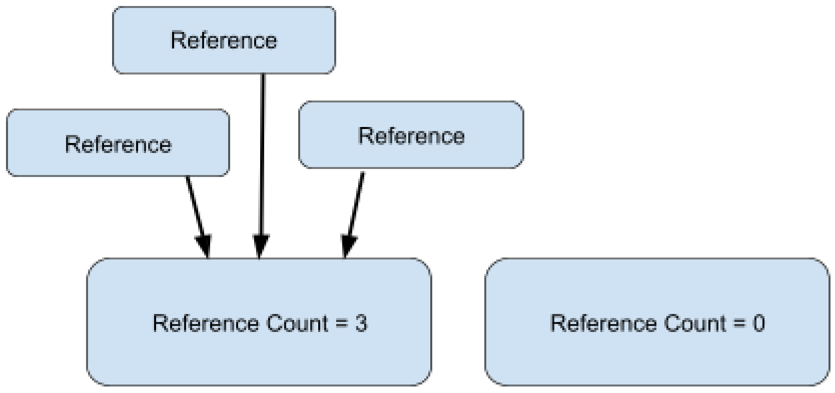
MJC can only target the SPARC architecture, which means compiled programs cannot be run natively on ARM or x86 processors. This means that an emulator is necessary for them to execute.

A previous senior design project, Jabberwocky, uses the QEMU emulator to containerize virtual environments for specific courses, such as Operating Systems Concepts, and Compiler Theory. Jabberwocky contains all necessary software to run the compiler while also allowing the user to transfer files from the native file system to the container's file system.

Algorithms

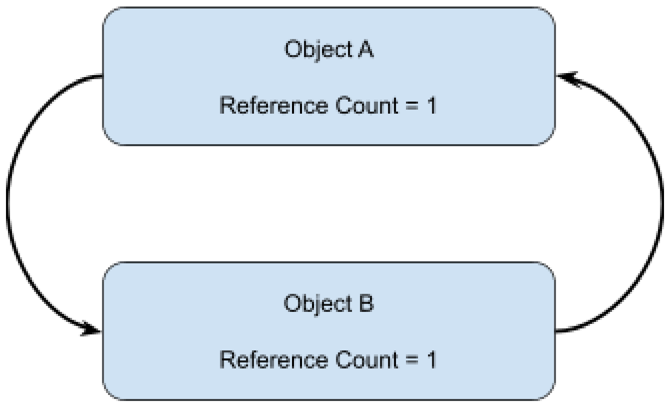
Reference Counting

Reference counting is a simple garbage collection method that relies on maintaining reference counts of objects whenever they are instanced. Each time an object is instanced, it is given a header, which stores information about the object. In this circumstance, the reference count is in the header. This counter changes depending on when the object goes out of scope, or when other objects reference the given object.



When the reference counter for an object reaches zero, it means that nothing needs to access the object anymore, so it can be freed. The problem that arises with this algorithm is that sometimes there are cyclic references.

Using an example where object A references object B, and object B references object A. In this example, both objects are always dependent on each other, which results in neither of the objects ever reaching a state where their reference counter reaches zero, which means that neither object will ever be freed, resulting in a memory leak.



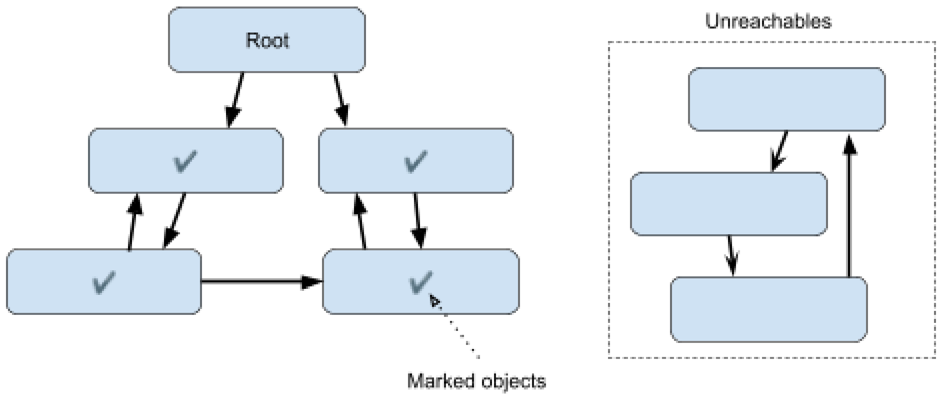
Mark and Sweep

Mark and sweep introduces a way to deal with reference counting's primary flaw. It gives the runtime a way to deal with cyclic garbage.

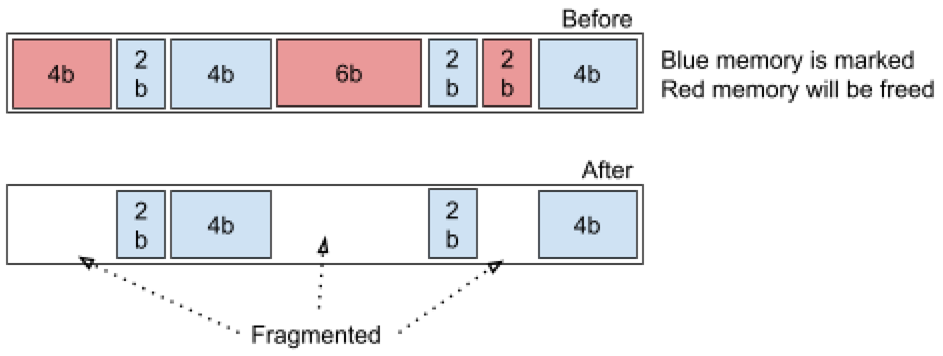
As the name suggests, mark and sweep is divided into two main phases: the mark phase, and the sweep phase.

The mark phase starts at the roots, the registers and local variables, and traverses to each object on the stack. As this phase traverses, it sets the "mark bit" in each object's header.

Lastly, the sweep phase traverses through the heap, iterating through every single object that has been allocated. If an object on the heap has not been marked, it means it is no longer needed, so it is freed.



Despite fixing reference counting's cyclical reference issue, mark-and-sweep isn't perfect. Since the unneeded memory is simply freed, this leaves holes in the heap, resulting in very fragmented memory.



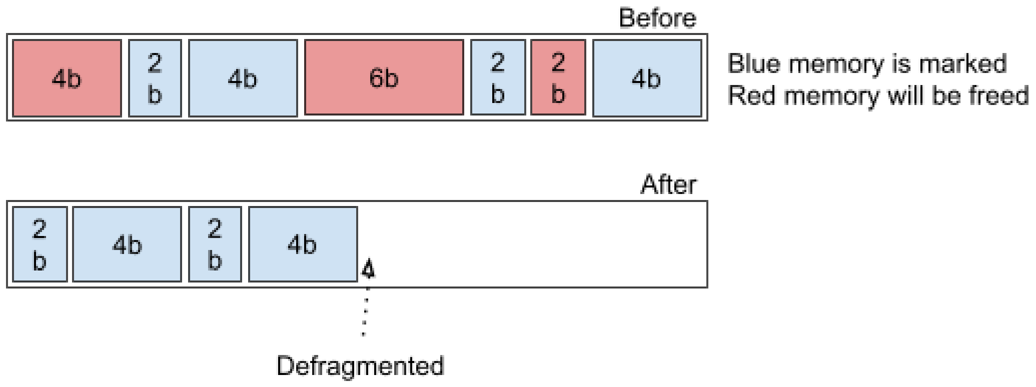
Copying

The copying algorithm acts almost identically to mark-and-sweep, and mark-and-copy might be a more fitting name. Unlike mark-and-sweep, copying splits its heap into two chunks, which are essentially sub-heaps. These are titled the “from heap” and the “to heap”. Copying is made up of two phases: the mark phase, and the copying phase.

The mark-phase is identical to mark-and-sweep, marking all objects that are still on the stack.

At the point of garbage collection, all allocations exist within the “from” heap, and the “to” heap is unused. The copying phase moves each marked object from the “from” heap to the “to” heap, placing each object directly after the last object in memory. This removes the fragmentation that occurred with the mark-and-sweep algorithm.

After all of the marked objects have been moved over, everything on the “from” is freed, and the handles of the two heaps are swapped. Lastly, existing references to the live objects are corrected as their addresses are changed as they are copied to the other heap.



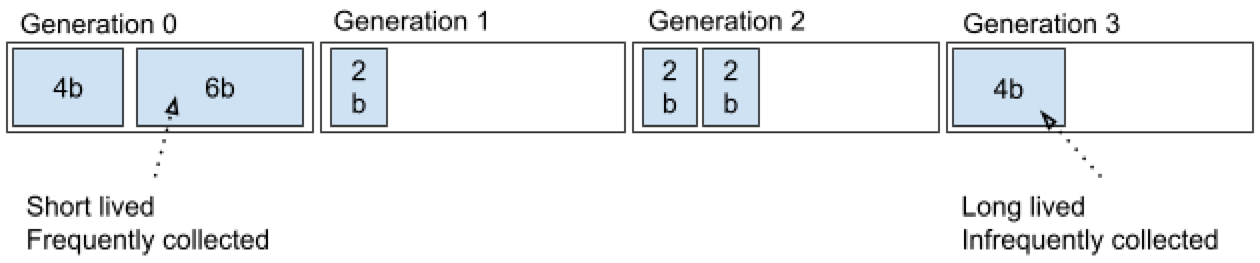
Generational

Despite copying not having any apparent flaws, generational tries to improve on the idea by shortening the duration of the garbage collection cycles. The garbage collection cycles are stop-the-world, meaning they halt the process of everything else.

Generational garbage collection shortens the duration of these cycles by splitting the heap into multiple chunks. Since newly instanced objects are typically the ones with the shortest lifetime, the generational algorithm collects the older memory less often than the younger memory.

Generation 0 is the youngest generation, where all new objects are placed in memory. Whenever a generation becomes filled, the garbage collector runs a copying garbage collection cycle, marking everything used, then copying everything still needed into the generation above the current one.

In the example below, the oldest generation (generation 3) will only fill up when all generations below it are also full, which means it has the same consequences of filling up as a normal heap.



Popular generational GCs tend to use two to three generations, and MJC uses a two-generation system.

Testing and Evaluation

Given that the MJC garbage collector was primarily designed for educational purposes, it's most aptly compared to itself. We explored the merits and drawbacks of each algorithm, highlighting how specific ones address shortcomings in others. This concept is further discussed in the Algorithms section.

To assess the algorithms, we devised four distinct test cases: RcArgumentTest, CyclicGarbageTest, FragmentedHeapTest, and NewOldTest. Although there were several more tests used for other testing purposes, we felt like these tests were the ones that showcased each algorithm’s capabilities best.

RcArgumentTest is meant to test how references work when used as function parameters. This will allow us to test the reference counter against something running without any garbage collection.

CyclicGarbageTest sets up a cyclic reference scenario, where Object A points to Object B, and vice versa.
FragmentedHeapTest deliberately fragments the heap before attempting to allocate a large memory block.
NewOldTest evaluates the performance under scenarios involving both old and new objects.

For simplicity, we set the heap size to 1024 bytes (1 kilobyte) manually. Initially, we ran MemoryGeneratorTest without a garbage collector, resulting in an expected crash after a few cycles. Subsequently, we executed the same test with the Reference Count algorithm, which ran indefinitely.

As the Reference Count algorithm fails to handle cyclic references, we assessed it with CyclicGarbageTest, observing some memory remaining unfreed. We then tested it with the Mark-and-Sweep algorithm, which effectively handles cyclic references.

Following this, we ran FragmentedHeapTest with Mark-and-Sweep, expecting a crash due to the memory fragmentation caused by Mark-and-Sweep. Subsequently, we tested the copying and generational algorithms, which both defragment the memory, allowing the program to execute without issues.

Developer’s Guide

Installation

Jabberwocky

Install Poetry from python-poetry.org.

Install Jabberwocky from https://github.com/Kippiii/jabberwocky-container-manager.

Navigate to the installed Jabberwocky folder.

```
poetry install
poetry shell

python build.py
```

Restart computer.

SPARC Container

Install the SPARC container from <DROPBOX LINK TO BE ADDED>

```
cd build\dist\
<run executable>
(e.g. installer-win32-AMD64.exe)

jab install <PATH to SPARK container .tar file> <container name>
(e.g. jab install ../ct.TAR ct)
```

Follow the instructions on the installer. QEMU will be installed if not already on the system.

Garbage Collector

gc.c/.h (Garbage collector interface)

This file contains functionality common to all garbage collector implementations.

To add an additional method of garbage collection, first extend the **GcType** enum by adding the name of your method. Lastly, create a new struct to represent your garbage collector, and in it include a **GC** instance as its first member. You will need a **create** function for your garbage collector, which must fill in the members of its **GC** object:

NOTE: Set unwanted functions to NULL and the runtime will not call them.

```
typedef struct GC {
    GcType type; // Type of this GC

    // Perform GC cycle
    void (*_collect)(GC* gc);

    // Stack frames
    void (*_stack_push)(GC* gc, void* frame, u32 size);
    void (*_stack_pop)(GC* gc);

    // Reference counting
    void (*_ref_incr)(GC* gc, Object* obj);
    void (*_ref_decr)(GC* gc, Object* obj);
    // Destroy this GC
    void (*_destroy)(GC* gc);
} GC;
```

If you would like to manually invoke some garbage collector operation as part of your implementation, please use the following public API in gc.h:

```
void gc_collect(GC* gc);
   Perform a garbage collection cycle. The specific algorithm will depend on
   which compiler flags were selected.

void gc_stack_push(GC* gc, void* frame, u32 size);
   Push a new active stack frame.

void gc_stack_pop(GC* gc);
   Pop the current active stack frame.

void gc_ref_incr(GC* gc, Object* obj);
   Increase an object’s reference count.

void gc_ref_decr(GC* gc, Object* obj);
   Decrease an object’s reference count.

void gc_destroy(GC* gc);
   Destroy this garbage collector.
```

refcount_gc.c/.h

This file contains the reference-counting garbage collector implementation. If you would like to use these functions, please use the following public API in refcount_gc.h:

```
GC* refcount_create(void);
   Create a reference counting garbage collector.

void refcount_destroy(GC* gc);
   Destroy this garbage collector.

void refcount_ref_incr(GC* gc, Object* obj);
   Increments an object’s reference counter.

```

```
void refcount_ref_decr(GC* gc, Object* obj);
   Decrements an object’s reference count. If the counter reaches zero, the
   object is freed.
```

marksweep_gc.c/.h

This file contains the mark-sweep garbage collector implementation. If you would like to use these functions, please use the following public API in marksweep.h:

```
GC* marksweep_create(void);
   Create a mark-sweep garbage collector.

void marksweep_destroy(GC* gc);
   Destroy this garbage collector.

void marksweep_collect(GC* gc);
   Perform mark-and-sweep garbage collection cycle.

void marksweep_stack_push(GC* gc, void* frame, u32 size);
   Pushes a new stack frame to traverse later using garbage collection.

void marksweep_pop_stack(GC* gc);
   Pops the current stack frame.
```

copying_gc.c/.h

This file contains the copying garbage collector implementation. If you would like to use these functions, please use the following public API in copying.h:

```
GC* copying_create(void);
   Create a copying garbage collector.

void copying_destroy(GC* gc);
   Destroy this garbage collector.

void copying_collect(GC* gc);
   Perform a copying garbage collection cycle.

void copying_stack_push(GC* gc, void* frame, u32 size);
   Push a new active stack frame to traverse later during garbage collection.
```

```
void copying_stack_pop(GC* gc);
   Pop the current active stack frame.
```

generational_gc.c/.h

This file contains the generational garbage collector implementation. If you would like to use these functions, please use the following public API in generational.h:

```
GC* generational_create(void);
   Create a generational garbage collector.

void generational_destroy(GC* gc);
   Destroy this garbage collector.

void generational_collect(GC* gc);
   Perform a generational garbage collection cycle.

void generational_stack_push(GC* gc, void* frame, u32 size);
   Push a new active stack frame to traverse later during garbage collection.

void generational_stack_pop(GC* gc);
   Pop the current active stack frame.
```

Heap

heap.c./h (Heap interface)

This file contains functionality common to all heap implementations.

To add an additional type of heap, first extend the **HeapType** enum by adding the name of your type. Lastly, create a new struct to represent your heap, and in it include a **Heap** instance as its first member. You will need a **create** function for your heap, which must fill in the following members of its **Heap** object:

NOTE: Unlike the GC interface, these function pointers are not optional. They must not be NULL or the runtime will halt.

```
typedef struct Heap {
    HeapType type;     // Type of this heap
    LinkList objects; // Live objects on this heap

    // Allocate object from this heap
    Object* (*_alloc)(Heap* heap, u32 size);
    // Free object to this heap
    void (*_free)(Heap* heap, Object* obj);
    // Dump contents of this heap
    void (*_dump)(const Heap* heap);
    // Destroy this heap
    void (*_destroy)(Heap* heap);
} Heap;
```

If you would like to manually invoke some heap operation as part of your implementation, please use the following public API in heap.h:

```
Object* heap_get_object(void* block);
   Derive a header from a memory block pointer.

void heap_dump_object(const Object* obj);
   Log object record to the console.

void* heap_alloc(Heap* heap, u32 size);
   Allocate memory from the specified heap.

void heap_free(Heap* heap, Object* obj);
   Free memory to the specified heap.

BOOL heap_is_object(const Heap* heap, const void* addr);
   Check whether an address is an object.

void heap_dump(const Heap* heap);
   Dump the contents of the heap.

void heap_destroy(Heap* heap);
   Destroy this heap.
```

chunk_heap.c./h

This file contains the “chunk” heap implementation. Chunk heaps allocate a large memory block and deal it out in blocks as you allocate/free memory. If you would like to use these functions, please use the following public API in chunk_heap.h:

```
Heap* chunkheap_create(u32 size);
   Create a chunk heap.

void chunkheap_destroy(Heap* heap);
   Destroy this chunk heap.

Object* chunkheap_alloc(Heap* heap, u32 size);
   Allocate an object from this heap.

void chunkheap_free(Heap* heap, Object* obj);
   Free an object to this heap

void chunkheap_dump(const Heap* heap);
   Dump the contents of this heap.

BOOL chunkheap_purify(Heap* src, Heap* dst);
   Copy the live allocations from one heap to another heap. References are
   updated to access the copies in the destination heap. Returns TRUE if dst has
   enough room for ALL live allocations from src, otherwise FALSE.
```

stl_heap.c/.h

This file contains the standard library heap implementation. It simply serves as a wrapper over the standard library’s malloc and free functions. If you would like to use these functions, please use the following public API in stl_heap.h:

```
Heap* stlheap_create(u32 size);
   Create a standard-library heap.

void stlheap_destroy(Heap* heap);
   Destroy this standard-library heap.

Object* stlheap_alloc(Heap* heap, u32 size);
   Allocate an object from this heap.
```

```
void stlheap_free(Heap* heap, Object* obj);
   Free an object to this heap.

void stlheap_dump(const Heap* heap);
   Dump the contents of this heap.
```

Utilities

config.c./h

This file contains the current configuration of the CRT, as specified by the compiled program. If you would like to use or modify this configuration, please use the following public API in config.h:

```
GcType config_get_gc_type(void);
  Access the current garbage collector type.

void config_set_gc_type(GcType type);
  Set the current garbage collector type.

HeapType config_get_heap_type(void);
   Access the current heap type.

void config_set_heap_type(HeapType type);
   Set the current heap type.

u32 config_get_heap_size(void);
   Access the current heap size.

void config_set_heap_size(u32 size);
   Set the current heap size.
```

debug.c./h

This file contains debugging utilities, such as macros for assertions and logging. These are very helpful in catching bugs early on in the implementation process. The following public API is available in debug.h:

NOTE: Pass NDEBUG=1 to the MJC Makefile to remove the LOG/ASSERT macros. (The ALLOC/FREE macros are unaffected.)

```
MJC_LOG(msg, ...)
   Log a message to the console.

MJC_ASSERT(expr)
   Assert a conditional expression, halting the program if it does not hold.

MJC_ASSERT_MSG(expr, ...)
   Assert a conditional expression, providing a custom message to display if and
   when the program halts.

MJC_ALLOC(size)
   Allocate memory using malloc. Useful to distinguish CRT allocations from heap
   allocations.

MJC_FREE(block)
   Free memory using free. Useful to distinguish CRT allocations from heap
   allocations.

MJC_ALLOC_OBJ(T)
   Allocate memory for a specific type. Useful to distinguish CRT allocations
   from heap allocations.
```

linklist.c./h

This file contains a doubly-linked list implementation. The following public API is provided in linklist.h:

```
void linklist_destroy(LinkList* list);
   Destroy this linked list.

void linklist_append(LinkList* list, void* object);
   Append an object to this linked list.

BOOL linklist_remove(LinkList* list, void* object);
   Remove an object from this linked list. Returns TRUE if the object was
   removed, otherwise FALSE.
```

```
void linklist_remove_iter(LinkList* list, LinkNode* at);
   Remove an object from this linked list via an iterator (node) to the object.
   This cannot fail so no return value is provided.

LinkNode* linklist_pop(LinkList* list);
   Pop the tail node from the linked list. Returns NULL if the list is empty.

void linklist_insert(LinkList* list, LinkNode* at, void* object);
   Insert an object into this linked list at a specified position (iterator).
   Object is placed AFTER the iterator at.

BOOL linklist_contains(const LinkList* list, void* object);
   Test whether this linked list contains the specified object.

void linklist_dump(const LinkList* list);
   Log the contents of this linked list to the console.

/**
 * @param list Pointer to list
 * @param T Type of list elements
 * @param stmt Code to run during each loop iteration
 */
LINKLIST_FOREACH(list, T, stmt)
   Perform a for-each loop over this linked list.
   The current NODE can be accessed through the variable 'NODE'.
   The current ELEMENT can be accessed through the variable 'ELEM'.

/**
 * @param list Pointer to list
 * @param T Type of list elements
 * @param stmt Code to run during each loop iteration
 */
LINKLIST_FOREACH_REV(list, T, stmt)
   Perform a for-each loop (in reverse) over this linked list.
   The current NODE can be accessed through the variable 'NODE'.
   The current ELEMENT can be accessed through the variable 'ELEM'.
```

runtime.c./h

This file contains core functions of the CRT, for use by the compiler in its generated code. You should not have to call these functions in your code; however, the following global variables are available as part of the public API in runtime.h:

```
extern Heap* curr_heap;
   The currently active heap.

extern GC* curr_gc;
   The currently active garbage collector.
```

stackframe.c./h

This file contains functions for traversal of the program stack. The following public API is provided in stackframe.h:

```
/**
 * @param arg User argument (optional)
 * @param obj Heap object that was found
 * @param pp_obj Address of the pointer to the object
 */
typedef void (*StackFrameTraverseFunc)(void* arg, struct Object* obj,
                                       void** pp_obj);
   Function to apply to all reachable objects.

void stackframe_push(void* frame, u32 size);
   Push a new stack frame.

void stackframe_pop(void);
   Pop the newest stack frame.

void stackframe_traverse(StackFrameTraverseFunc callback, void* callback_arg);
   Perform a traversal of the program stack, starting from the newest stack frame
   and recursing upwards. Applies the function callback to all objects found
   along the way.
```

MJC Usage

```
Usage: {compiler-jar} [option(s)] input-file
Options:
--help                  Display this information again.
--verbose               Log verbose compiler information to "/verbose.txt".
--gc=<type>             Set <type> as the program's garbage collection method.
                        (None|RefCount|MarkSweep|Copying|Generational)
--heap=<type>           Set <type> as the program's heap type.
                        (Stl|Chunk|Buddy)
```