Sneaker Documentation

Release 0.3.1

Yanzheng Li

September 17, 2016

Contents

CHAPTER 1

Overview

Sneaker is a collection of modern and versatile C++ components that are useful in many application development scenarios.

Components are designed as independent entities that each has its own responsibility in answering one or more of the six aspects of all software systems, namely the "*what*", "*when*", "*where*", "*who*", "*why*" and "*how*".

The elegance and succinctness of the code intrinsically demonstrates the "*how*", and the separation of functionalities among components and the interactions between them focus on the "*who*", "*when*" and "*where*". Finally, good documentation and comments reflects the "*what*" and "*why*" to users.

In addition, the design of many components are heavily inspired from features and models from other high level programming languages that are either absent or not first-class citizens in the C++ world, such as the use of scoped context management and function decorators, which are all prevalent features in Python.

CHAPTER 2

Get Started

1. Checkout the project documentations
    - Web Version
    - PDF Version
2. Build from source and install library

    make && make install
3. Read the wiki if you'd like to contribute

- Wiki home page
- Styles and Guidelines
- Development and Release Workflow
- Build Environments and Dependencies

CHAPTER 3

Content

3.1 Build and Installation

This reference manual contains useful information for developers who wish to build the project from source.

The project strives the make the build and installation process as straightforward as possible. Currently, the build system requires GNU Make and CMake.

Steps for building Sneaker:

1. Update submodules for all dependencies

   git submodule update –init –recursive

   Sneaker currently has the following dependencies referenced as submodules:

   - Google Test version 1.7.0

2. Build and install Google Test from source

   make gtest

3. Build libraries, unit tests and documentations

   make

   This builds the following main binaries:

Binary	Description
libsneaker.a	Sneaker static library (release version)
libsneaker_dbg.a	Sneaker static library (debug version)
libsneaker_dbg_cov.a	Sneaker static library with coverage support (debug version)

   If you are unsure whether your system meets the requirements for building Sneaker, please refer to this page.

4. Install the library binaries and header files

   make install

   This places the libraries *libsneaker.a* and *libsneaker_dbg.a* to under */usr/local/lib*, and copies the header files to */usr/local/include/sneaker/*.

5. Uninstall the library

   make uninstall

   This will undo the actions done in step 4) above.

3.2 C Utilities

Types, functions and data structures in C.

3.2.1 Assertions

Various utilities for dynamic and static assertions.

Header file: *sneaker/libc/assert.h*

ASSERT(expr)
: Asserts that *expr* evaluates to a truthy value during runtime. Aborts the program if the assertion fails, along with print out the line number and file name of where the assertion failed.

ASSERT_STREQ(str1, str2)
: Asserts that both specified strings *str1* and *str2* are equal. Aborts the program if the assertion fails, along with print out the line number and file name of where the assertion failed.

STATIC_ASSERT(cond)
: Asserts that the specified expression *cond* evaluates to a truthy value during compilation. Compilation fails if the assertion fails.

3.2.2 Array

Array abstraction that can dynamically increase and shrink capacity.

Header file: *sneaker/libc/vector.h*

vector_t

---

vector_t **vector_create**()
: Creates an instance of *vector_t* using dynamically allocated memory.

void **vector_free**(*vector_t**)
: Frees memory from the pointer of an instance of *vector_t* specified.

int **vector_append**(*vector_t*, void*)
: Appends an element to the end of a *vector_t* instance specified as the first argument that holds a pointer which points to the value specified in the second argument. Returns *1* if successful, *0* otherwise.

void* **vector_get**(*vector_t*, int)
: Retrieves the element pointer from a specified *vector_t* instance as the first argument by the zero-based index from the second argument. If the index given is out of bound, then the code exits, so it's better to use *vector_size()* to make sure the index is within the bound of the vector.

void* **vector_remove**(*vector_t*, int)
: Removes an element from the specified *vector_t* instance at the index in the second argument. If the index is within the bound of the vector, the element is removed and the pointer held in that element is returned. If the index is out of bound, then the code exits, so it's better to use *vector_size()* to make sure the index is within the bound of the vector.

void* **vector_set**(*vector_t*, int, void*)
: Sets the pointer held of a particular element from the specified *vector_t* instance. The new pointer specified in the third argument will replace the existing one in the encapsulating element, and the

old pointer will be returned. Likewise, if the index in the second argument is out of bound, then the code will exit.

int **vector_size** (*vector_t*)
Gets the number of elements in the instance of *vector_t* provided.

const void** **vector_content** (*vector_t*)
Retrieves a list of the element pointers contained in the instance of *vector_t* specified.

3.2.3 Bitmap

Two-dimensional bitmap abstraction.

Header file: *sneaker/libc/bitmap.h*

bitmap_t

---

bitmap_t **bitmap_create** (size_t, size_t)
Creates an instance of *bitmap_t* using dynamically allocated memory by specifying the width and height of the bitmap, which are the number of bits in each respective dimension.

void **bitmap_free** (*bitmap_t* *)
Frees memory from the pointer of an instance of *bitmap_t* specified.

size_t **bitmap_width** (*bitmap_t*)
Gets the width of the *bitmap_t* instance specified.

size_t **bitmap_height** (*bitmap_t*)
Gets the height of the *bitmap_t* instance specified.

int **bitmap_clear_bit** (*bitmap_t*, size_t, size_t)
Clears a particular bit to *0* in the *bitmap_t* instance specified. The second and third arguments respectively specify the row and column of the bit to clear, both are zero-based indices. Returns *1* if successful, *0* otherwise.

If either the row or column specified are out of bound, the function returns *0*.

int **bitmap_is_set** (*bitmap_t*, size_t, size_t)
Returns a value indicating whether a particular bit in the *bitmap_t* specified is set to *1*. The second and third arguments respectively specify the row and column of the bit, both are zero-based indices. Returns *1* if the bit is set, *0* otherwise.

If either the row or column specified are out of bound, the function returns *0*.

void **bitmap_clear** (*bitmap_t*)
Sets every bit in the *bitmap_t* specified to *0*.

3.2.4 C-String Types

Short-hand notations for C-String types.

Header file: *sneaker/libc/cstr.h*

c_str

---

typedef of *char**

cc_str

typedef of *const char**

3.2.5 Standard Library Functions

Standard functions in C that might not be available in some compilers.

Header file: *sneaker/libc/cutils.h*

char* **itoa** (int, char*, int)
Integer to ASCII string conversion.

Converts an integer value specified as the first argument to a null-terminated string specified as the second argument. The base of the integer can be optionally specified as the third argument, which defaults to *10* if not specified. Returns the converted ASCII string.

If base is *10* and value is negative, the resulting string is preceded with a minus sign (-). With any other bases, value is always considered unsigned.

char* **atoi** (const *char**)
ASCII string to integer conversion.

Converts an ASCII string specified as the first argument to an integer number.

The function first discards as many whitespace characters as necessary until the first non-whitespace character is found. Then, starting from this character, takes an optional initial plus or minus sign followed by as many numerical digits as possible, and interprets them as a numerical value.

3.2.6 Dictionary

A dictionary implementation with C-string typed keys, based on the `hashmap_t` implementation.

Header file: *sneaker/libc/dict.h*

dict_t

dict_t **dict_create** ()
Creates an instance of *dict_t* using dynamically allocated memory.

size_t **dict_size** (*dict_t*)
Gets the number of key-value pairs in the *dict_t* instance specified.

void **dict_free** (*dict_t**)
Frees memory from the pointer of an instance of *dict_t* specified.

void* **dict_put** (*dict_t*, const *char**, void*)
Insert a key-value pair into the *dict_t* instance specified as the first argument. The second argument is the key and the third argument is a pointer of the value to be stored. Returns the pointer in the pair inserted. If the key already exists, then the existing value is replaced.

void* **dict_get** (*dict_t*, const *char**)
Retrieves the value associated with a key specified as the second argument from the *dict_t* instance specified as the first argument. Returns the pointer of the value associated with the key if the key exists, *NULL* otherwise.

3.2.7 Hashing

Hashing algorithms.

Header file: *sneaker/libc/hash.h*

unsigned long int **linear_horners_rule_str_hash** (const *char**)
Calculates the hash value of the specified string using a linear version of Horner's Rule.

unsigned long int **hash32shift** (unsigned *int*)
Calculates the hash of a 32-bit unsigned integer.

unsigned long int **hash64shift** (unsigned *long*)
Calculates the hash of a 64-bit unsigned integer.

unsigned long int **hash_str_jenkins_one_at_a_time** (const *char**)
Calculates the hash value of the specified string using the "Jenkins's one-at-a-time" algorithm.

unsigned long int **hash_robert_jenkin** (unsigned *int*)
Calculates the hash value of the specified string using the "Robert Jenkin" algorithm.

3.2.8 Hash Map

A hash map implementation that offers near constant-time lookups and inserts.

Header file: *sneaker/libc/hashmap.h*

hashmap_t

HashFunc
Hash function used to hash keys in *hashmap_t*, defined as *unsigned long int(*HashFunc)(void*)*.

KeyCmpFunc
Key comparison function used to compare keys in the hash map, defined as *int(*KeyCmpFunc)(void*, void*)*.

hashmap_t **hashmap_create** (size_t, *HashFunc*, *KeyCmpFunc*)
Creates an instance of *hashmap_t* using dynamically allocated memory, and by specifying the initial capacity as the first argument, as well as the hashing and key comparison functions as the second and third arguments.

size_t **hashmap_size** (*hashmap_t*)
Gets the number of elements in the specified *hashmap_t* instance.

void **hashmap_lock** (*hashmap_t*)
Locks the specified *hashmap_t* instance so that updates from other threads are blocks until *hashmap_unlock* is called on the instance.

void **hashmap_unlock** (*hashmap_t*)
Unlocks the specified *hashmap_t* instance following a previous call of *hashmap_lock* so that updates from other threads are unblocked.

void **hashmap_free** (*hashmap_t**)
Frees memory from the pointer of an instance of *hashmap_t* specified.

void* **hashmap_put** (*hashmap_t*, void*, void*)
Inserts a key-value pair into the *hashmap_t* instance as the first argument. If the key already exists, its associated value will be updated. Returns the value inserted if successful, *NULL* otherwise.

void* **hashmap_get** (*hashmap_t*, void*)
Retrieves the value associated with the key specified as the second argument from the *hashmap_t* instance specified as the first argument. Returns the value if its associated key is in the instance, *NULL* otherwise.

int **hashmap_contains_key** (*hashmap_t*, void*)
Returns a value indicating if the key specified as the second argument exists in the *hashmap_t* instance specified as the first argument. Returns *1* if the key is in the instance, *0* otherwise.

void* **hashmap_remove** (*hashmap_t*, void*)
Removes the key-value pair from the *hashmap_t* instance specified as the first argument using the key as the second argument. Returns the value associated with the key if found, *NULL* otherwise.

void* **hashmap_lookup** (*hashmap_t*, void*, void*)
Looks up a particular key-value pair from the *hashmap_t* instance as the first argument. The lookup function is the second argument that has the signature *int(*lookup)(void *key, void *value, void* arg)*, which takes a key, its associated value and an optional argument which is specified as the third argument. The lookup function returns *1* for a key-value pair considered as found, at which the function stops the searching and returns the value found. If no key-value is found, returns *NULL*.

void **hashmap_iterate** (*hashmap_t*, void*, int)
Iterates through all key-value pairs in the *hashmap_t* instance as the first argument. The second argument specifies a function of signature *int(*callback)(void *key, void *value)* that receives every key-value pair iterated. The third argument specifies a value that indicates whether the iteration should continue when the callback function returns a non-truthy value.

size_t **hashmap_bucketcount** (*hashmap_t*)
Returns the number of buckets in the *hashmap_t* instance specified as the first argument.

size_t **hashmap_capacity** (*hashmap_t*)
Returns the capacity of the *hashmap_t* instance specified as the first argument.

int **hashmap_equal** (*hashmap_t*, *hashmap_t*)
Determines equality between the two instances of *hashmap_t* specified. The two instances would only be considered equal if they have exactly the same keys and all associated values of the same keys must be identical as well. Returns *1* if they are considered equal, *0* otherwise.

3.2.9 Math

General functions for mathematical computations.

Header file: *sneaker/libc/math.h*

unsigned int **nearest_exp2_ceil** (unsigned *int*)
Computes the least largest exponent of base 2 that is closest to the input value.

unsigned int **nearest_exp2_floor** (unsigned *int*)
Computes the most largest exponent of base 2 that is closest to the input value.

3.2.10 Queue

A FIFO storage container.

Header file: *sneaker/libc/queue.h*

queue_t

queue_t **queue_create**()
Creates an instance of *queue_t* using dynamically allocated memory.

size_t **queue_size**(*queue_t*)
Gets the number of elements in the *queue_t* instance specified.

void* **queue_front**(*queue_t*)
Gets the element pointer at the front of the *queue_t* instance specified. If the queue is empty, *NULL* is returned.

void* **queue_back**(*queue_t*)
Gets the element pointer at the back of the *queue_t* instance specified. If the queue is empty, *NULL* is returned.

int **queue_push**(*queue_t*, void*, size_t)
Pushes an element pointer to the back of the *queue_t* instance specified as the first argument. The second argument is a pointer that points to the value to be pushed onto the queue, and the third argument is the size of the value to be pushed, in number of bytes. Returns *-1* if the push failed, *1* if successful.

void* **queue_pop**(*queue_t*)
Pops the element pointer at the front of the *queue_t* instance specified and returns it. If the queue is empty, *NULL* is returned.

void **queue_free**(*queue_t* *)
Frees memory from the pointer of an instance of *queue_t* specified.

3.2.11 Stack

A FILO storage container.

Header file: *sneaker/libc/stack.h*

sstack_t

---

sstack_t **stack_create**()
Creates an instance of *sstack_t* using dynamically allocated memory.

size_t **stack_size**(*sstack_t*)
Returns the number of elements in the *sstack_t* instance specified.

void* **sstack_top**(*sstack_t*)
Gets the element pointer at the top of the *sstack_t* instance provided. Returns *NULL* if the stack is empty.

int **stack_push**(*sstack_t*, void*, size_t)
Pushes an element pointer to the top of the *sstack_t* instance specified as the first argument. The second argument is a pointer that points to the value to be pushed onto the stack, and the third argument is the size of the value to be pushed, in number of bytes. Returns *-1* if the push failed, *1* if successful.

void* **stack_pop**(*sstack_t*)
Pops the element pointer at the top of the *sstack_t* instance specified and returns it. If the stack is empty, *NULL* is returned.

void **stack_free**(stack*)
Frees memory from the pointer of an instance of *sstack_t* specified.

3.2.12 String Buffer

A C-string buffer abstraction that can dynamically change capacity.

Header file: *sneaker/libc/strbuf.h*

strbuf_t

---

strbuf_t **strbuf_t_create** ()
Creates an instance of *strbuf_t* using dynamically allocated memory.

void **strbuf_free** (*strbuf_t**)
Frees memory from the pointer of an instance of *strbuf_t* specified.

void **strbuf_empty** (*strbuf_t*)
Empties the *strbuf_t* instance specified as the argument and reduces its capacity to *0*.

size_t **strbuf_len** (*strbuf_t*)
Gets the length of the string currently held in the *strbuf_t* instance specified as the argument.

const char* **strbuf_cstr** (*strbuf_t*)
Gets the C-string held in the *strbuf_t* instance specified as the argument.

size_t **strbuf_capacity** (*strbuf_t*)
Gets the capacity of the *strbuf_t* instance specified as the argument.

int **strbuf_append** (*strbuf_t*, const *char**)
Appends a C-string into the *strbuf_t* instance specified as the first argument. The second argument is the C-string to be appended. Returns *1* if the append is successful, *0* otherwise.

3.2.13 String Manipulation

String manipulation functions.

Header file: *sneaker/libc/strutils.h*

char* **strtoupper** (char *)
Converts the C-string specified as the argument to its uppercase form. Returns the converted string.

char* **strtolower** (char *)
Converts the C-string specified as the argument to its lowercase form. Returns the converted string.

char* **strtrim** (char *)
Trims off empty spaces at the front and end of the string specified. Returns the trimmed string.

char* **strcpy_hard** (char*, const *char**)
Copies the source string specified as the second argument to the destination string specified as the first argument. If the length of the source string is greater than the length of the destination string, then the destination string is freed and re-allocated to have the same length as the source string. Returns the destination string.

char* **strncpy_safe** (char*, const *char**, size_t)
Copies the source string specified as the second argument to the destination string specified as the first argument. The number of characters copied is the minimum between the length of the destination string and the size specified as the third argument. Returns the destination string.

size_t **strlcpy2** (char*, const *char**, size_t)
Copies the source string specified as the second argument to the destination string specified as the first argument. The number of characters copied is at most the number specified as the third argument minus *1*. Returns the number of characters tried to be copied, which is the size of the source string.

3.2.14 General Utilities

General utility functions.

Header file: *sneaker/libc/utils.h*

void **set_nth_bit** (int*, char)
: Set a specific bit of a given number to 1. The first argument specifies a pointer to the number to be set, and the second argument is the ‘n‘th bit to set from the LSB.

void **clear_nth_bit** (int*, char)
: Clears a specific bit of a given number to 0. The first argument specifies a pointer to the number to be cleared, and the second argument is the ‘n‘th bit to clear from the LSB.

int **is_bit_set** (int, char)
: Checks if a specific bit of a given number is set to 1. The first argument specifies a pointer to the number to be checked, and the second argument is the *n‘th bit to check from the LSB. Returns ‘1* if the bit is set to *1*, *0* otherwise.

void **set_nth_bit_uint32** (uint32_t*, char)
: 32-bit version of *set_nth_bit*.

void **clear_nth_bit_uint32** (uint32_t*, char)
: 32-bit version of *clear_nth_bit*.

int **is_bit_set_uint32** (uint32_t, char)
: 32-bit version of *is_bit_set*.

void **set_nth_bit_uint8** (uint8_t **val*, char *bit*)
: 8-bit version of *set_nth_bit*.

void **clear_nth_bit_uint8** (uint8_t **val*, char *bit*)
: 8-bit version of *clear_nth_bit*.

int **is_bit_set_uint8** (uint8_t *val*, char *bit*)
: 8-bit version of *is_bit_set*.

int **rand_top** (int)
: Returns an pseudo-random integer that’s in the range between 1 and the number specified as the argument, inclusively.

int **rand_range** (int, int)
: Returns an pseudo-random integer that’s in the range between two integers specified as the two arguments, inclusively.

double **randf_top** (double)
: Returns an pseudo-random double-precision floating number that’s in the range between 1 and the number specified as the argument, inclusively.

double **randf_range** (double, double)
: Returns an pseudo-random double-precision floating number that’s in the range between two double-precision floating numbers specified as the two arguments, inclusively.

char* **generate_text** (size_t=0, size_t=0)
: Generates a random string of an optional length specified as the first argument, up to the optional maximum length specified as the second argument.

  If both the length and max length are non-zero, the generated text will have a length between the two numbers.

  If only the length is non-zero, the generated text will have exactly the specified length.

  If only the max length is non-zero, the generated text will have a length between *1* and the max length, inclusively.

If both the length and max length are zero, *NULL* is returned.

char* **generate_loremipsum**()
Generates a string of Lorem Ipsum of arbitrary length.

3.2.15 Universally Unique Identifier

A 128-bits implementation of UUID.

Header file: *sneaker/libc/uuid.h*

uuid128_t

---

uuid128_t **uuid_create**()
Creates an instance of *uuid128_t* using dynamically allocated memory.

int **uuid_compare**(const *uuid128_t*, const *uuid128_t*)
Evaluates equality between two instances of *uuid128_t*. Returns *1* if the first argument is considered greater than the second, *0* if they are equivalent, and *-1* if the first is less than the second.

__uint128_t **uuid_to_hash**(const *uuid128_t*)
Hashes the *uuid128_t* instance provided as the arugment to an 128-bit unsigned integer.

__uint128_t **uuid_create_and_hash**()
Creates an instance of *uuid128_t* and returns its equivalent hash.

3.3 STL Allocator

Components responsible for memory allocation and deallocation for STL containers.

3.3.1 Allocation Policy

Abstraction that encapsulates the logic to allocate and deallocate memory.

Header file: *sneaker/allocator/alloc_policy.h*

class sneaker::allocator::**standard_alloc_policy**<T>

A default implementation of allocation policy using dynamic memory allocation and deallocation from the operating system.

type value_type
The underling type that memory will be allocated for.

type pointer
Pointer type of the memory allocated by this policy.

type const_pointer
Constant pointer type of the memory allocated by this policy.

type reference
Reference type of memory allocated by this policy.

type const_reference
Constant reference type of memory allocated by this policy.

type **size_type**
Type used for denoting the quantity of elements allocated by this policy.

type **difference_type**
Type used for denoting the size difference between two pointers allocated by this policy.

type **propagate_on_container_move_assignment**
Indicates that the policy shall propagate when the STL container is move-assigned.

type **rebind**<U>
Equivalent allocator type to allocate elements of type *U*.

explicit **standard_alloc_policy**()
Explicit constructor.

~standard_alloc_policy()
Destructor.

explicit **standard_alloc_policy**(standard_alloc_policy **const**&)
Explicit copy constructor. The argument is an instance of *standard_alloc_policy* with the same encapsulating type. Nothing is copied over.

template<typename U>
explicit **standard_alloc_policy**(standard_alloc_policy<U> **const**&)
Explicit copy constructor. The argument is an instance of *standard_alloc_policy* with a different encapsulating type. Nothing is copied over.

pointer **allocate**(size_type, **typename** std::allocator<void>::const_pointer = 0)
Allocates a specified number of memory in bytes. The first argument specifies the number of bytes requested for allocation. The second argument may be a value previously obtained by another call to *allocate* and not yet freed with *deallocate*. This value may be used as a hint to improve performance by allocating the new block near the one specified.

If allocation is successful, a pointer that points to the memory allocated is returned, otherwise *std::bad_alloc* is raised.

void **deallocate**(pointer, size_type)
Deallocates the pre-allocated memory. The first argument is a pointer that points to the memory that needs to be freed, and the second argument specifies the size of the memory in bytes.

size_type **max_size**() **const**
Get the maximum amount of memory that can be allocated, in number of bytes.

template<typename T, typename T2>
bool **operator==**(standard_alloc_policy<T> **const**&, standard_alloc_policy<T2> **const**&)
Equality comparison between two instances of *standard_alloc_policy* with potentially different encapsulating type. Returns *true* by default.

template<typename T, typename T2>
bool **operator!=**(standard_alloc_policy<T> **const**&, standard_alloc_policy<T2> **const**&)
Inequality comparison between two instances of *standard_alloc_policy* with potentially different encapsulating type. Returns *false* by default.

template<typename T, typename other_allocator>
bool **operator==**(standard_alloc_policy<T> **const**&, other_allocator **const**&)
Equality comparison between two an instances of *standard_alloc_policy* and a different type. Returns *false* by default.

template<typename T, typename other_allocator>

bool **operator!=**(standard_alloc_policy<T> **const**&, other_allocator **const**&)
Inequality comparison between two an instances of *standard_alloc_policy* and a different type. Returns *true* by default.

3.3.2 Object Traits

Abstraction that control the construction and destruction of the encapsulating type.

Header file: *sneaker/allocator/object_traits.h*

class sneaker::allocator::**object_traits**<T>

A default implementation of object traits, where the construction and destruction of the underlying type is carried over by calling the type's constructor and destructor respectively.

type rebind<U>
Equivalent object traits type of encapsulating type *U*.

explicit object_traits()
Explicit constructor.

~object_traits()
Destructor.

T ***address**(T&)
Get the address on a reference of an instance of the encapsulating type.

T **const** ***address**(T **const**&)
Get the address on a constant reference of an instance of the encapsulating type.

void **construct**(T *, **const** T&)
Instantiates an instance of the encapsulating type through copy semantics by using the specified allocated memory and an instance of the encapsulating type to be passed to the copy constructor of the instance to be created.

template<class U, class... Args>
void **construct**(U *, Args&&...)
Instantiates an instance of type *U* by using the specified allocated memory and the arguments used for instantiation. The first argument is a pointer that points to the pre-allocated memory and the remain arguments are arguments passed to the constructor of type *U*.

void **destroy**(T *)
Destroy an instantiated instance of the encapsulating type by calling the type's destructor. The specified argument is a pointer that points to an instance of the encapsulating type.

3.3.3 Allocator

Abstraction that's responsible for allocating and deallocating memory used by STL containers.

Header file: *sneaker/allocator/allocator.h*

class sneaker::allocator::**allocator**<T, Policy, Traits>

A default implementation of STL allocator based on the abstraction of allocation policy and object traits.

type value_type
The underlying type that memory will be allocated for.

type `pointer`
Pointer type of the memory allocated by this allocator.

type `const_pointer`
Constant pointer type of the memory allocated by this allocator.

type `reference`
Reference type of memory allocated by this allocator.

type `const_reference`
Constant reference type of memory allocated by this allocator.

type `size_type`
Type used for denoting the quantity of elements allocated by this allocator.

type `difference_type`
Type used for denoting the size difference between two pointers allocated by this allocator.

explicit `allocator`()
Explicit constructor.

`~allocator`()
Destructor.

`allocator`(allocator **const** &*rhs*)
Copy constructor that takes another allocator of the same encapsulating type. Nothing is copied over.

```
template<typename U>
```
`allocator`(allocator<U> **const**&)
Copy constructor that takes another allocator of a different encapsulating type. Nothing is copied over.

```
template <typename U, typename P, typename T2>
```
`allocator`(allocator<U, P, T2> **const** &*rhs*)
Copy constructor that takes another allocator of a different encapsulating type, as well as different allocation policy and object traits. Nothing is copied over.

```
template<typename T, typename P, typename Tr>
```
bool `operator==`(allocator<T, P, Tr> **const**&, allocator<T, P, Tr> **const**&)
Equality operator that evaluates equality between two instances of *allocator* that have the same encapsulating type, and are based on the same allocation policy and object traits types. Equality is evaluated based on the equality between the allocation policy types.

```
template<typename T, typename P, typename Tr, typename T2, typename P2, typename Tr2>
```
bool `operator==`(allocator<T, P, Tr> **const**&, allocator<T2, P2, Tr2> **const**&)
Equality operator that evaluates equality between two instances of *allocator* that have different encapsulating types, and are based on different allocation policy and object traits. Equality is evaluated based on the equality between the allocation policy types.

```
template<typename T, typename P, typename Tr, typename other_allocator>
```
bool `operator==`(allocator<T, P, Tr> **const**&, other_allocator **const**&)
Equality operator that evaluates equality between two instances of *allocator* that have potentially the same encapsulating types, and are based on the same allocation policy and object traits. Equality is evaluated based on the equality between the allocation type of the first argument and the second argument.

```
template<typename T, typename P, typename Tr>
```
bool `operator!=`(allocator<T, P, Tr> **const**&, allocator<T, P, Tr> **const**&)
Inequality operator that evaluates inequality between two instances of *allocator* that have the same

encapsulating type, and are based on the same allocation policy and object traits. Inequality is evaluated based on the inequality between the allocation policy types.

template<typename T, typename P, typename Tr, typename T2, typename P2, typename Tr2>
bool **operator!=**(allocator<T, P, Tr> **const**&, allocator<T2, P2, Tr2> **const**&)
Inequality operator that evaluates inequality between two instances of *allocator* that have different encapsulating types, and are based on different allocation policy and object traits. Inequality is evaluated based on the inequality between the allocation policy types.

template<typename T, typename P, typename Tr, typename other_allocator>
bool **operator!=**(allocator<T, P, Tr> **const**&, other_allocator **const**&)
Inequality operator that evaluates inequality between two instances of *allocator* that have potentially different encapsulating types, and are based on different allocation policy and object traits. Inequality is evaluated based on the equality between the allocation type of the first argument and the second argument.

3.4 In-memory Cache Management

Components that manage caching of in-memory objects.

3.4.1 Cache Interface

Sneaker provides a generic and extensible cache interface that allows the use of different caching schemes by the choices of users. This is the public interface that users whom wish to use the caching mechanism will interact with.

Header file: *sneaker/cache/cache_interface.h*

class sneaker::cache::**cache_interface**<**class** CacheScheme, **class** OnInsert, **class** OnErase>

This class allows users to define a cache scheme, as well as handler types that are to be invoked during cache value insertions and erasures.

type key_type
Type of the keys in the cache.

type value_type
Type of the values in the cache.

cache_interface(**const** OnInsert &*on_insert*, **const** OnErase &*on_erase*)
Constructor that takes a reference of *OnInsert* and *OnErase* instances each.

bool **empty**() **const**
Determines whether the cache has any elements. Returns *true* if the cache has any element, *false* otherwise.

bool **full**() **const**
Determines whether the cache is full.

size_t **size**() **const**
Gets the number of elements in the cache.

bool **find**(key_type) **const**
Determines if the value associated with the specified key is in the cache. Returns *true* if there is a value associated with the key in the cache, *false* otherwise.

bool **get**(key_type, value_type&)
: Retrieves the element associated with the specified key in the cache. The first argument is the key, and the second argument is a reference of the result value associated with the key. Returns *true* if the value is found, *false* otherwise.

void **insert**(key_type, **const** value_type&)
: Inserts a key-value pair into the cache.

3.4.2 Cache Schemes

Sneaker provides abstractions of some of the most well-known caching schemes to users. These abstractions are meant to be used with the cache interface described above.

LRU Cache

This class encapsulates the logic of the *Least-Recently Used* caching scheme.

Header file: *sneaker/cache/lru_cache.h*

sneaker::cache::lru_cache<typename K, typename V, size_t N>

---

type key_type
: Type of the keys in the cache.

type value_type
: Type of the values in the cache.

size_t **N**
: The size of the cache.

bool **empty**() **const**
: Determines whether the cache has any elements. Returns *true* if the cache has any element, *false* otherwise.

bool **full**() **const**
: Determines whether the cache is full.

size_t **size**() **const**
: Gets the number of elements in the cache.

bool **find**(key_type) **const**
: Determines if the value associated with the specified key is in the cache. Returns *true* if there is a value associated with the key in the cache, *false* otherwise.

bool **get**(key_type, value_type&)
: Retrieves the element associated with the specified key in the cache. The first argument is the key, and the second argument is a reference of the result value associated with the key. Returns *true* if the value is found, *false* otherwise.

void **next_erasure_pair**(key_type **key_ptr*, value_type ***value_ptr*)
: Gets the next key-value pair to be erased when inserting a new key-value pair while the cache is full.

void **insert**(key_type, **const** value_type&)
: Inserts a key-value pair into the cache.

bool **erase**(key_type)
Erase the element associated with the specified key in the cache. The first argument is the key associated with the value that needs to be erased. Returns *true* is the key-value pair is erased, *false* otherwise.

void **clear**()
Clears the cache by erasing all elements within.

Example showing using *lru_cache* with *cache_interface*:

```
#include <sneaker/cache/cache_interface.h>
#include <sneaker/cache/lru_cache.h>

typedef int KeyType;
typedef const char* ValueType;

struct InsertHandler
{
    bool operator()(KeyType key, const ValueType& value)
    {
        // Do something here.
    }
};

struct EraseHandler
{
    bool operator()(KeyType key, const ValueType& value)
    {
        // Do something here.
    }
};

typedef sneaker::cache::cache_interface<
  sneaker::cache::lru_cache<KeyType, ValueType, 10>, InsertHandler, EraseHandler> CacheType;

// Make use of CacheType..
CacheType cache;
cache.insert(1, "Hello world");
```

3.5 Containers

Storage containers of objects that serve a broad range of purposes.

3.5.1 Reservation-based Containers

Container types that store objects on a reservation-based system. Users must reserve spots before objects are requested to be stored in these containers.

class sneaker::container::**reservation_map**<T>

Header file: *sneaker/container/reservation_map.h*

Note: The internal implementation is based on *std::map<token_t, T>*, hence elements of type *T* must have their comparators defined for comparison.

type `token_t`
The token type used by the reservation container.

type `generator_type`
The type that is used to generate reservation tokens internally.

`reservation_map()`
Constructor.

`~reservation_map()`
Destructor.

size_t `size()` **const**
Gets the number of elements that are currently reserved.

token_t `reserve()`
Reserve a spot in the container. Returns a token that can be used later for storage. The user must call this method before attempts to store an object in the container.

bool `member`(token_t) **const**
Determines if an object is a member of the container by using a token. Returns *true* if a spot has been reserved for the token specified, *false* otherwise.

bool `put`(token_t, T)
Attempts to store an object into the container by using a token. The storage fails if the token specified is invalid. Returns *true* if the storage is successful, *false* otherwise.

bool `get`(token_t, T *)
Attempts to retrieve an object from the container by using a token. The retrieval fails if the token specified is invalid. Returns *true* if the retrieval is successful, *false* otherwise.

bool `unreserve`(token_t)
Attempts to unreserve a previously reserved spot by using a token. The un-reservation fails if no previously reservation has been made by the token specified. Returns *true* if the un-reservation is successful, *false* otherwise.

void `clear()`
Removes all the reserved elements. After invoked, all tokens previously obtained are no longer valid.

3.5.2 Assorted-values Map Containers

Key-value(s) based map containers where each key can be mapped to an assortment of multiple values of different statically defined types.

class `sneaker::container::assorted_value_map`<K, ... ValueTypes>

An implementation of assorted-values map container based on *std::map*.

Header file: *sneaker/container/assorted_value_map.h*

type `core_type`
The core mapping type used internally. This type is *std::map<K, boost::tuple<ValueTypes ...>>*.

type `key_type`
The type of the keys in the mapping.

type `mapped_type`
The type of the assortment of values in the mapping.

type `value_type`
The type of the key-value(s) pairs in the mapping.

type `key_compare`
Key comparison type.

type `value_compare`
Value comparison type.

type `reference`
Reference type of the values in the mapping.

type `const_reference`
Constant reference type of the values in the mapping.

type `pointer`
Pointer type of the values in the mapping.

type `const_pointer`
Constant pointer type of the values in the mapping.

type `iterator`
Forward iterator type.

type `const_iterator`
Constant forward iterator type.

type `reverse_iterator`
Reverse iterator type.

type `const_reverse_type`
Constant reverse iterator type.

type `difference_type`
The type that indicates the difference of number of elements between two iterators of the mapping.

type `size_type`
The type that indicates the number of elements in the mapping.

explicit `assorted_value_map`()
Constructor.

explicit `assorted_value_map`(**const** assorted_value_map&)
Copy constructor. The mapping is copied over.

explicit `assorted_value_map`(**const** core_type&)
Constructor that takes a reference of core mapping type. The mapping is copied over.

`~assorted_value_map`()
Destructor. All elements in the mapping are freed.

`static template<class Compare, class Alloc>`
sneaker::container::*assorted_value_map*<K, ... ValueTypes> `create`()
Static factory method that creates an instance with the specified *Compare* key comparator type, and *Alloc* value allocation type.

`static template<class Compare, class Alloc>`
sneaker::container::*assorted_value_map*<K, ... ValueTypes> `create`(**const** Compare&, **const** Alloc&)
Static factory method that creates an instance with the specified *Compare* key comparator type and *Alloc* value allocation type, and a reference of each type respectively.

bool **empty**() **const**
Determines whether the mapping is empty. Returns *true* if there are no key-value(s) pairs in the mapping, *false* otherwise.

size_type **size**() **const**
Determines the number of key-value(s) pairs in the mapping.

size_type **max_size**() **const**
Determines the maximum number of key-value(s) pairs that can be in the mapping.

void **insert**(K, ValueTypes)
Inserts a key-value(s) pair into mapping. If the specified key already exists in the mapping, its value(s) will be overwritten.

void **erase**(iterator)
Erases a particular key-value(s) pair in the mapping by an iterator. The iterator must point to a valid pair in the mapping to be effective.

size_type **erase**(**const** K&)
Erases a particular key-value(s) pair in the mapping by a key. Returns the number of elements erased. Note if the specified key does not exist in the mapping, then the number of elements returned is *0*.

void **erase**(iterator, iterator)
Erases a range of key-value(s) in the mapping in between in the two specified iterators, inclusively.

void **swap**(assorted_value_map&)
Swaps the mapping with another instance of *assorted_value_map* with the same types for the key and values.

void **clear**() **noexcept**
Clears the content in the mapping.

mapped_type &**at**(K)
Retrieves the value(s) associated with the specified key by reference. Note if the key specified doees not exist in the mapping, *std::out_of_range* is raised.

const mapped_type &**at**(K) **const**
Retrieves the value(s) associated with the specified key by constant reference. Note if the key specified doees not exist in the mapping, *std::out_of_range* is raised.

template<class A, size_t Index>
A &**get**(K)
Gets the *Index* th element associated with the specified key in the container by reference. Note if the key specified doees not exist in the mapping, *std::out_of_range* is raised.

template<class A, size_t Index>
const A &**get**(K) **const**
Gets the *Index* th element associated with the specified key in the container by reference. Note if the key specified doees not exist in the mapping, *std::out_of_range* is raised.

mapped_type &**operator[]**(**const** K&)
Retrieves the value(s) associated with the specified key by reference. Note if the key does not match the key of any element in the container, the function inserts a new element with that key and returns a reference to its mapped value

iterator **begin**()
Gets a forward iterator that marks the beginning of the mapping.

const_iterator **begin**() **const**
Gets a constant forward iterator that marks the beginning of the mapping.

iterator **end**()
: Gets a forward iterator that marks the end of the mapping.

const_iterator **end**() **const**
: Gets a constant forward iterator that marks the end of the mapping.

reverse_iterator **rbegin**()
: Gets a reverse iterator that marks the beginning of the mapping in reverse order.

const_reverse_iterator **rbegin**()
: Gets a constant reverse iterator that marks the beginning of the mapping in reverse order.

reverse_iterator **rend**()
: Gets a reverse iterator that marks the end of the mapping in reverse order.

const_reverse_order **rend**() **const**
: Gets a constant reverse iterator that marks the end of the mapping in reverse order.

iterator **find**(K)
: Attempts to find the value(s) associated in the specified key. Returns an instance of forward iterator that points to the key-value(s) pair. If the key does not exist in the mapping, then the iterator returned points to *end()*.

const_iterator **find**(K) **const**
: Attempts to find the value(s) associated in the specified key. Returns an instance of constant forward iterator that points to the key-value(s) pair. If the key does not exist in the mapping, then the iterator returned points to *cend()*.

class sneaker::container::**unordered_assorted_value_map**<K, ... ValueTypes>

An implementation of assorted-values map container based on *std::unordered_map*.

Header file: *sneaker/container/unordered_assorted_value_map.h*

type core_type
: The core mapping type used internally. This type is *std::unordered_map<K, boost::tuple<ValueTypes ...>>*.

type key_type
: The type of the keys in the mapping.

type mapped_type
: The type of the assortment of values in the mapping.

type value_type
: The type of the key-value(s) pairs in the mapping.

type hasher
: The type used to hash the keys.

type key_equal
: The type used to evaluate equality between two keys.

type allocator_type
: The type of allocator used to allocate memory.

type reference
: The reference type for a key-value(s) pair.

type const_reference
: The constant reference type for a key-value(s) pair.

type `iterator`
A bidirectional iterator to *value_type*.

type `const_iterator`
A bidirectional iterator to *const value_type*.

type `reverse_iterator`
A reverse order iterator to *value_type*.

type `const_reverse_iterator`
A reverse order iterator to *const value_type*.

type `size_type`
The type that indicates the number of elements in the mapping.

type `difference_type`
A type that represents the difference between two iterators.

`unordered_assorted_value_map`()
Constructor.

`unordered_assorted_value_map`(**const** core_type&)
Constructor that takes a reference of core mapping type. The mapping is copied over.

`unordered_assorted_value_map`(**const** unordered_assorted_value_map&)
Copy constructor. The mapping from the argument is copied over.

`~unordered_assorted_value_map`()
Destructor. All elements in the mapping are freed.

`static template<size_type N, class Hash, class Pred, class Alloc>`
sneaker::container::*unordered_assorted_value_map*<K, ... ValueTypes> **`create`**()
Static factory method that creates an instance with the specified initial capacity *N*, key hash object of type *Hash*, value comparison object of type *Pred* and value allocation object of type *Alloc*.

`static template<size_type N, class Hash, class Pred, class Alloc>`
sneaker::container::*unordered_assorted_value_map*<K, ... ValueTypes> **`create`**(**const** Hash&, **const** Pred&, **const** Alloc&)
Static factory method that creates an instance with the specified initial capacity *N*, key hash object of type *Hash*, value comparison object of type *Pred* and value allocation object of type *Alloc*, and a reference of each type respectively except for *N*.

bool **`empty`**() **const**
Determines whether the mapping is empty. Returns *true* if there are no key-value(s) pairs in the mapping, *false* otherwise.

size_type **`size`**() **const**
Determines the number of key-value(s) pairs in the mapping.

size_type **`max_size`**() **const**
Determines the maximum number of key-value(s) pairs that can be in the mapping.

void **`insert`**(K, ValueTypes)
Inserts a key-value(s) pair into mapping. If the specified key already exists in the mapping, its value(s) will be overwritten.

void **`erase`**(iterator)
Erases a particular key-value(s) pair in the mapping by an iterator. The iterator must point to a valid pair in the mapping to be effective.

size_type **erase**(**const** K&)
: Erases a particular key-value(s) pair in the mapping by a key. Returns the number of elements erased. Note if the specified key does not exist in the mapping, then the number of elements returned is *0*.

void **erase**(iterator, iterator)
: Erases a range of key-value(s) in the mapping in between in the two specified iterators, inclusively.

void **swap**(assorted_value_map&)
: Swaps the mapping with another instance of *assorted_value_map* with the same types for the key and values.

void **clear**() **noexcept**
: Clears the content in the mapping.

const mapped_type &**at**(K) **const**
: Retrieves the value(s) associated with the specified key by constant reference. Note if the key specified does not exist in the mapping, *std::out_of_range* is raised.

template<class A, size_t Index>
A **get**(K)
: Retrieves a particular value among the assortment of values associated with the specified key. Type *A* is the type of the value, and *Index* is a zero-based index that specifies the position of the value to be retrieved, among the list of values. Note if the key specified does not exist in the mapping, *std::out_of_range* is raised.

template<class A, size_t Index>
const A &**get**(K) **const**
: Retrieves a particular value by constant reference among the assortment of values associated with the specified key. Type *A* is the type of the value, and *Index* is a zero-based index that specifies the position of the value to be retrieved, among the list of values. Note if the key specified does not exist in the mapping, *std::out_of_range* is raised.

mapped_type &**operator[]**(**const** K&)
: Retrieves the value(s) associated with the specified key by reference. Note if the key does not match the key of any element in the container, the function inserts a new element with that key and returns a reference to its mapped value

iterator **begin**()
: Gets a forward iterator that marks the beginning of the mapping.

const_iterator **begin**() **const**
: Gets a constant forward iterator that marks the beginning of the mapping.

iterator **end**()
: Gets a forward iterator that marks the end of the mapping.

const_iterator **end**() **const**
: Gets a constant forward iterator that marks the end of the mapping.

iterator **find**(K)
: Attempts to find the value(s) associated in the specified key. Returns an instance of forward iterator that points to the key-value(s) pair. If the key does not exist in the mapping, then the iterator returned points to *end()*.

const_iterator **find**(K) **const**
: Attempts to find the value(s) associated in the specified key. Returns an instance of constant forward iterator that points to the key-value(s) pair. If the key does not exist in the mapping, then the iterator returned points to *cend()*.

float **load_factor**() **const noexcept**
Gets the current load factor of the mapping, which is the ratio between the number of key-value(s) pair in the mapping and the number of buckets.

float **max_load_factor**() **const noexcept**
Get the maximum load factor the mapping can have.

void **rehash**(size_type)
Sets the number of buckets in the mapping to *n* or more by enforcing a rehash on all the keys in the mapping.

If *n* is greater than the current number of buckets in the mapping (bucket_count), a rehash is forced. The new bucket count can either be equal or greater than *n*.

If *n* is lower than the current number of buckets in the mapping (bucket_count), the function may have no effect on the bucket count and may not force a rehash.

Rehashes are automatically performed by the container whenever its load factor is going to surpass its max_load_factor in an operation.

void **reserve**(size_type)
Request a capacity change on the mapping by setting the number of buckets in the mapping to the most appropriate to contain at least the number of key-value(s) pairs specified by the first argument.

If *n* is greater than the current bucket_count multiplied by the *max_load_factor*, the container's bucket_count is increased and a rehash is forced.

If *n* is lower than that, the function may have no effect.

hasher **hash_function**() **const**
Returns the hash function object used by the mapping.

3.6 Context Management

Components that faciliate automatic context management.

Context management unfetters users from the cumbersomness and efforts associated with trivial cleanup actions, such as deallocations or destructions of resources after exiting the scope where they are no longer needed.

3.6.1 Context Manager Interface

An interface of context manager implementations.

Header file: *sneaker/context/context_manager.h*

class sneaker::context::**context_manager**

---

virtual void **__enter__**()
Method invoked when entering the scope of the context. This method needs to be overriden with code that prepares the context, such as resource allocations, aquiring and opening file handles, etc.

virtual void **__exit__**()
Method invoked when exiting the scope of the context. This method needs to be overriden with code that closes the context, such as resource deallocations, closing and releasing file handles, etc.

3.6.2 Scoped Context

```
template<class F, class Args ...>
void scoped_context(context_manager*, F, ... Args)
```

Function that allows a block of code to run under the context of a context manager, in which the scope of the context is bounded by the scope of this function. In other words, the context starts immediately after the invocation of this function, and ends right before this function goes out of scope.

The first argument specifies a pointer of a context manager, the second argument specifies the block of code or a function that executes under the context of the context manager, and the remaining arguments are passed to the function when it's invoked under the context.

Header file: *sneaker/context/scoped_context.h*

3.6.3 Nested Contexts

Function that allows a block of code to run under the nested contexts of a list of context managers, in which the scope of the contexts are bounded by the scope of this function. In other words, the contexts start immediately after the invocation of this function, and end right before this function goes out of scope.

The contexts from the list of context managers are nested, meaning that the context of the second context manager in the list executes under the context of the first context manager, and so on.

The first argument specifies the list of context managers, the second argument specifies the block of code or a function that executes under the contexts of the context managers, and the remaining arguments are passed to the function when it's invoked under the contexts.

Header file: *sneaker/context/scoped_context.h*

```
template<class F, class Args ...>
void nested_context(std::vector<context_manager*>, F, ... Args)
```

3.7 Functional Abstractions

These functional abstractions improve the incompatibilites and interoperabilities between function pointers, functors and lambdas.

3.7.1 Function abstractions

Function abstractions that offer interoperabilities between function pointers, functors and lambdas, as well as asynchronous invocations.

Header file: *sneaker/functional/function.h*

class `sneaker::functional::`**function**<R, ... Args>

A function wrapper that takes statically defined return type and argument types, also supports asynchronous invocation.

type `implicit_type`
The underlying type of the function pointers, functors and lambas that it's compatible with by signature.

type `return_value`
The return type of this function.

`function` (implicit_type)
Constructor that takes a compatible function pointer, functor or lambda, and makes itself a owner of a copy of the argument. Note that this constructor is not declared as *explicit* as that implicit conversion from compatible types are possible.

`template<class Functor>`
`function` (Functor)
Constructor that takes a compatible function pointer, functor or lambda, and makes itself a owner of a copy of the argument. Note that this constructor is not declared as *explicit* as that implicit conversion from compatible types are possible.

const function<R, ... Args> &`operator=` (implicit_type)
Assignment operator that marks assignment from compatible function types.

`R operator() (... Args) const`
Invocation operator that takes arguments that are compatible with this function type and returns the result of the function.

`operator implicit_type` ()
Conversion operator that converts this instance to the underlying compatible function type.

`void invoke_async(... Args)`
Invokes the function asynchronously.

class `sneaker::functional::call`

A variant of *sneaker::functional::function* that is compatible with functions, functors and lambdas whose signatures take no arguments and has no return type.

class `sneaker::functional::action`<...Args>

A variant of *sneaker::functional::function* that is compatible with functions, functors and lambdas whose signatures take a list of statically typed arguments but has no return type.

class `sneaker::functional::predicate`<...Args>

A variant of *sneaker::functional::function* that is compatible with functions, functors and lambdas whose signatures take a list of statically typed arguments, and has a return type of *bool*.

3.7.2 Decorators

Function abstractions that facilitate the use of the decorator pattern. These decorators provide a higher level of operations on top of the encapsulating functions without having to modifiy their functionalities. Examples such as retries, error handling and logging are good examples of using decorators. Multiple decorators can be chained together together so that different operations can be stacked on top of each other.

class `sneaker::functional::retry`<R, ...Args>

A decorator that retries on the encapsulating function upon invocations that has an exception thrown. User can specify the type of exception to catch and the number of retries allowed for the encapsulating function.

Here is an example:

```
#include <vector>
#include <mysql> // ficticious
#include <sneaker/functional/retry.h>

// Suppose we have a function that takes an instance of a
// MySQL connection object, tries to connect to it, and
// queries some results. This can potentially have a connection
// issue sporadically, so we want to issue a maximum of 5 retries.
retry<void> wrapper = [](mysql::db_connection& connection) -> void {
  mysql::connection_result conn_result = connection.connect();
  mysql::query_result = conn_result.query(MyModel.list());
  std::vector<MyModel> models = query_result.normalize();
  printModels(models);
};

const int MAX_RETRY = 5;

// Invokes the function above to connect to the MySQL instance
// and queries the results, can retry 5 timees on connection error.
wrapper.operator()<mysql::connection_error, MAX_RETRY>();
```

template<typename ExceptionType, uint32_t MaxRetries>
R operator() (... Args) const

Invocation operator that takes arguments that are compatible with this encapsulating function type and returns the result of the function. Also specifies the exception type and max count on retry.

3.8 I/O

Components that faciliates I/O operations.

3.8.1 File Reader

Class that facilitates reads from files.

class `sneaker::io::`**file_reader**

Header file: *sneaker/io/file_reader.h*

file_reader()

Default constructor.

explicit file_reader(**const** char *)

Constructor. Takes a path of a file to be read.

explicit file_reader(**const** std::string &*path*)

Constructor. Takes a path of a file to be read.

const char ***file_path**() **const**

Gets the path of the file to be read.

void **set_path**(**const** char *)
Sets the path of the file to be read.

bool **read_file**(**const** char **p, size_t *size_read = NULL)
Reads the file and copies the content read to the pointer specified as the first argument. Also can specify an optional output argument that gets the number of bytes read. Returns *true* if the read is successful, *false* otherwise.

bool **read**(std::vector<char> *buf, size_t *size_read = NULL) **const**
Reads the file and copies content read to the byte array specified as the first argument. Also can specify an optional output argument that gets the number of bytes read. Returns *true* if the read is successful, *false* otherwise.

3.8.2 Data Stream Abstractions

Abstractions and interfaces for manipulating and interacting with data streams.

Input Stream

Header file: *sneaker/io/input_stream.h*

class sneaker::io::**input_stream**

Abstract class of an input stream.

input_stream()
Default constructor.

virtual ~input_stream()
Destructor.

virtual bool **next**(**const** uint8_t **data, size_t *len) = 0
Reads data from the stream.

Returns true if some data is successfully read, false if no more data is available or an error has occurred.

virtual void **skip**(size_t *len*) = 0
Skips a specified number of bytes.

virtual size_t **bytes_read**() **const** = 0
Returns the number of bytes read from this stream so far.

class sneaker::io::**stream_reader**

Convinience class that facilitates reading from an instance of *input_stream*.

explicit stream_reader(input_stream *)
Constructor. Takes an instance of *input_stream*.

bool **read**(uint8_t *)
Read one byte from the underlying stream. Returns *true* if the read is successful, *false* otherwise.

bool **read_bytes**(uint8_t *blob, size_t *n*)
Reads the given number of bytes from the underlying stream. Returns *true* if there are enough bytes to read, *false* otherwise.

void **skip_bytes**(size_t *n*)
Skips the given number of bytes.

bool **has_more**()
Returns *true* if and only if the end of stream is not reached.

Helper functions:

std::unique_ptr<*input_stream*> sneaker::io::**file_input_stream**(**const** char **filename*, size_t *buffer_size*)
Returns a new instance of *input_stream* whose contents come from the specified file. Data is read in chunks of given buffer size.

std::unique_ptr<*input_stream*> sneaker::io::**istream_input_stream**(std::istream &*stream*, size_t *buffer_size*)
Returns a new instance of *input_stream* whose contents come from the given *std::istream*. The *std::istream* object should outlive the returned result.

std::unique_ptr<*input_stream*> sneaker::io::**memory_input_stream**(**const** uint8_t **data*, size_t *len*)
Returns a new instance of *input_stream* whose data comes from the specified byte array.

Output Stream

Header file: *sneaker/io/output_stream.h*

class sneaker::io::**output_stream**

Abstract class of an output stream.

output_stream()
Default constructor.

~output_stream()
Destructor.

virtual bool **next**(uint8_t ***data*, size_t **len*) = 0
Returns a buffer that can be written into. On successful return, *data* has the pointer to the buffer and *len* has the number of bytes available at data.

virtual size_t **bytes_written**() **const** = 0
Returns the number of bytes written so far into this stream. The whole buffer returned by *next()* is assumed to be written.

virtual void **backup**(size_t *len*) = 0
"Returns" back to the stream some of the buffer obtained from in the last call to *next()*.

virtual void **flush**() = 0
Flushes any data remaining in the buffer to the stream's underlying store, if any.

class sneaker::io::**stream_writer**

Convinience class that facilitates writing to an instance of *output_stream*.

explicit stream_writer(output_stream *)
Constructor. Takes an instance of *output_stream*.

bool **write**(uint8_t *c*)
Writes a single byte to the stream.

bool **write_bytes**(**const** uint8_t *_blob_, size_t _n_)
Writes the specified number of bytes to the stream.

void **flush**()
Backs up upto the currently written data and flushes the underlying stream. Users should call this member method before finalizing the writing operation.

Helper functions:

std::unique_ptr<*output_stream*> sneaker::io::**file_output_stream**(**const** char *_filename_, size_t _buffer_size_)
Returns a new instance of *output_stream* whose contents are to be written to a file, in chunks of given buffer size.

If there is a file with the given name, it is truncated and overwritten. If there is no file with the given name, it is created.

std::unique_ptr<*output_stream*> sneaker::io::**ostream_output_stream**(std::ostream &_stream_, size_t _buffer_size_)
Returns a new instance *output_stream* whose contents are to be written to the specified *std::ostream*.

3.8.3 Temporary Files Management

Interfaces that facilitate handling and management of temporary files.

Header file: *sneaker/io/tmp_file.h*

const char *sneaker::io::**get_tmp_file_path**()
Generates a unique file path for storing a temporary file. The generated path is not used by any existing files and is a platform-specific location. Note that files stored at this file path are not persisted between program invocations and system reboots. If such persistency are desired, please use *sneaker::io::get_persistent_tmp_file_path()*.

const char *sneaker::io::**get_persistent_tmp_file_path**()
Generates a unique file path for storing a temporary file. The generated path is not used by any existing files and is a platform-specific location. Note that files stored at this file path are persistent across program invocations and system reboots.

3.9 Logging

Extensible general-purpose logging facilities.

3.9.1 Exception-safety Tags

Tags that signify exception-safeties of logging facilities.

Header file: *sneaker/logging/exception_safety_tag.h*

type sneaker::logging::**exception_safe_tag**
Tag that signifies exception-safe logging facilities.

type sneaker::logging::**exception_unsafe_tag**
Tag that signifies non exception-safe logging facilities.

3.9.2 Thread-safety Tags

Tags that signify thread-safeties of logging facilities.

Header file: *sneaker/logging/thread_safety_tag.h*

type `sneaker::logging::`**`thread_safe_tag`**
Tag that signifies thread-safe logging facilities.

type `sneaker::logging::`**`thread_unsafe_tag`**
Tag that signifies non thread-safe logging facilities.

3.9.3 Log Levels

Various levels of log severity.

Header file: *sneaker/logging/log_level.h*

enum class `sneaker::logging::`**`LogLevel`**
Enumerations of various levels of log severity. Values are:

- *LOG_LEVEL_DEBUG*
- *LOG_LEVEL_INFO*
- *LOG_LEVEL_WARN*
- *LOG_LEVEL_ERROR*
- *LOG_LEVEL_FATAL*

3.9.4 Log Schemes

Mechanisms of logging.

Header file: *sneaker/logging/log_scheme.h*

class `sneaker::logging::`**`log_scheme`**

Abstraction of a particular mechanism of storing log records.

virtual void **`write`**(**const** char **msg*) = 0
Base member of writing a log record. To be defined in subclasses.

virtual **`~log_scheme`**()
Base destructor.

class `sneaker::logging::`**`stream_log_scheme`**

Logging scheme that sends log records to *std::ostream*.

virtual void **`write`**(**const** char **msg*)
Writes the log message to *std::ostream*.

class `sneaker::logging::`**`stdout_log_scheme`** : **public** **`stream_log_scheme`**

Logging scheme that sends log records to stdout.

stdout_log_scheme()
Constructor.

virtual ~stdout_log_scheme()
Destructor.

class sneaker::logging::**stderr_log_scheme** : **public stream_log_scheme**

Logging scheme that sends log records to stderr.

stderr_log_scheme()
Constructor.

virtual ~stderr_log_scheme()
Destructor.

class sneaker::logging::**file_log_scheme** : **public stream_log_scheme**

Logging scheme that sends log records to a file.

file_log_scheme(const char* filename);
Constructor that takes a file path.

virtual ~file_log_scheme()
Destructor.

3.9.5 Logger

Class that encapsulates the core logging mechanism. Can be customized with particular exception-safety and thread-safety tags.

Header file: *sneaker/logging/logger.h*

template<typename thread_safety_tag, typename exception_safety_tag> sneaker::logging::logge

---

explicit logger(log_scheme **log_scheme*)
Constructor.

template<size_t LINE_SIZE=1024> void write(LogLevel log_lvl, const cha
Writes a free-format log record message.

3.9.6 Helper Utilities

Utility functions that faciliate logging.

Header file: *sneaker/logging/logging.h*

LOG(lvl, file, line, format, ...)
Convenience macro that logs a log record message by taking the log level, file and line number of where the log occurs, and the format and arguments that goes into the log message.

LOG_DEBUG(format, ...)
Convenience macro that logs a log record in *LOG_LEVEL_DEBUG* level.

LOG_INFO(format, ...)
Convenience macro that logs a log record in *LOG_LEVEL_INFO* level.

LOG_WARN (format, ...)
Convenience macro that logs a log record in *LOG_LEVEL_WARN* level.

LOG_ERROR (format, ...)
Convenience macro that logs a log record in *LOG_LEVEL_ERROR* level.

LOG_FATAL (format, ...)
Convenience macro that logs a log record in *LOG_LEVEL_FATAL* level.

LOG_DETAILED (lvl, file, line, format, ...)
Convenience macro that logs a log record message by taking the log level, file and line number of where the log occurs, and the format and arguments that goes into the log message. Also logs the timestamp of the occurance of log.

LOG_DEBUG_DETAILED (format, ...)
Convenience macro that logs a log record in *LOG_LEVEL_DEBUG* level.

LOG_INFO_DETAILED (format, ...)
Convenience macro that logs a log record in *LOG_LEVEL_INFO* level.

LOG_WARN_DETAILED (format, ...)
Convenience macro that logs a log record in *LOG_LEVEL_WARN* level.

LOG_ERROR_DETAILED (format, ...)
Convenience macro that logs a log record in *LOG_LEVEL_ERROR* level.

LOG_FATAL_DETAILED (format, ...)
Convenience macro that logs a log record in *LOG_LEVEL_FATAL* level.

3.10 JSON

Interfaces for manipulating, validating and conversating data in JSON format.

3.10.1 JSON Serialization and Deserialization

JSON data object abstraction, and interfaces for JSON serialization and deserialization.

This module provides a set of interfaces for interacting with raw JSON data, and converting that data into data objects, and vice versa.

The top level function *sneaker::json::parse()* takes a JSON blob in string format and returns the parsed JSON object, which is an instance of *sneaker::json::JSON*.

Internally, instances of *sneaker::json::JSON* depends on several internal types to encapsulate JSON data of various formats. For example, *std::vector* is used to capture JSON arrays and *std::map* is used to encapsulate JSON objects, and so on.

This module is largely based on some of the concepts and implementations borrowed from the open source project "json11" from Dropbox, with minor changes and fixes. Please refer to https://github.com/dropbox/json11 for more information.

Here is an example of parsing a JSON blob into its corresponding data object.

```
#include <sneaker/json/json.h>
#include <cassert>
#include <iostream>
#include <string>
```

```
using namespace sneaker::json;

const std::string str = "{"
  "\"k1\": \"v1\","
  "\"k2\": -42,"
  "\"k3\": [\"a\", 123, true, false, null]"
"}";

auto json = sneaker::json::parse(str);

assert(std::string("\"v1\"") == json["k1"].dump());
assert(std::string("-42") == json["k2"].dump());
assert(std::string("[\"a\", 123, true, false, null]") == json["k3"].dump());

assert(std::string("v1") == json["k1"].string_value());
assert(-42 == json["k2"].number_value());
assert(std::string("a") == json["k3"][0].string_value());
assert(123 == json["k3"][1].number_value());
assert(true == json["k3"][2].bool_value());
assert(false == json["k3"][3].bool_value());

// Conversely, here is an example of serializing a JSON data object:

JSON json = JSON::object {
  { "key1", "value1" },
  { "key2", 123.456 },
  { "key3", false },
  { "key4", JSON::array { 1, "a", true, nullptr } },
};

std::cout << json.dump() << std::endl;
```

Header file: *sneaker/json/json.h*

sneaker::json::**parse**(**const** std::string &*in*)

Top level function for parsing a JSON blob and returns the deserialized JSON data object.

class sneaker::json::**invalid_json_error**

Error thrown when parsing an invalid JSON blob.

class sneaker::json::**JSON**

---

type JSON::**Type**
The type of the JSON object. Values are *NUL*, *NUMBER*, *BOOL*, *STRING*, *ARRAY* and *OBJECT*.

type JSON::**string**
The underflying JSON string type.

type JSON::**array**
The underlying JSON array type.

type JSON::**object**
The underlying JSON object type.

JSON()
Default constructor.

JSON(null)
Constructor.

JSON(double)
Constructor.

JSON(int)
Constructor.

JSON(**const** string&)
Constructor.

JSON(string&&)
Constructor.

JSON(**const** char *)
Constructor.

JSON(**const** array&)
Constructor.

JSON(array&&)
Constructor.

JSON(**const** object&)
Constructor.

JSON(object&&)
Constructor.

```
template<class T, class = decltype(&T::to_json)>
```
JSON(**const** T &*t*)
Implicit constructor: anything with a to_json() function.

```
template<class M, typename std::enable_if<
std::is_constructible<std::string, typename M::key_type>::value &&
std::is_constructible<JSON, typename M::mapped_type>::value, int>::type = 0
>
```
JSON(**const** M &*m*)
Implicit constructor: map-like objects (std::map, std::unordered_map, etc).

```
template<class V, typename std::enable_if<
std::is_constructible<JSON, typename V::value_type>::value, int>::type = 0
>
```
JSON(**const** V &*v*)
Implicit constructor: vector-like objects (std::list, std::vector, std::set, etc).

Type **type**() **const**
Gets the type of the JSON object.

bool **is_null**()
Determines if this instance represents a JSON null value.

bool **is_number**()
Determines if this instance represents a JSON numeric value.

bool **is_bool**()
Determines if this instance represents a JSON boolean value.

bool **is_string**()
Determines if this instance represents a JSON string value.

bool **is_array**()
Determines if this instance represents a JSON array value.

bool **is_object**()
Determines if this instance represents a JSON object value.

double **number_value**() **const**
Gets the encapsulating floating numeric value of this JSON object.

int64_t **int_value**() **const**
Gets the encapsulating integer numeric value of this JSON object.

bool **bool_value**() **const**
Gets the encapsulating boolean value of this JSON object.

const string &**string_value**() **const**
Gets the encapsulating string value of this JSON object.

const array &**array_items**() **const**
Gets the encapsulating array value of this JSON object.

const object &**object_items**() **const**
Gets the encapsulating object value of this JSON object.

const JSON &**operator[]**(size_t *i*) **const**
JSON array type element accessor.

const JSON &**operator[]**(**const** std::string &*key*) **const**
JSON object type element accessor.

bool **operator==**(**const** JSON &*other*) **const**
Equality operator.

bool **operator<**(**const** JSON &*other*) **const**
Less Than equality operator.

bool **operator!=**(**const** JSON &*other*) **const**
Inequality operator.

bool **operator<=**(**const** JSON &*other*) **const**
Less Than or Equal equality operator.

bool **operator>**(**const** JSON &*other*) **const**
Greater Than equality operator.

bool **operator>=**(**const** JSON &*other*) **const**
Greater Than or Equal equality operator.

void **dump**(std::string &*out*) **const**
Serializes the JSON data object and dumps the result into the provided string.

std::string **dump**() **const**
Serializes the JSON data object and returns the result string.

3.10.2 JSON Schema Validation

Interface for validating JSON blobs using JSON schemas.

The validation mechanisms are implemented based on the JSON Schema Validation specification. More information can be found at:

http://json-schema.org/documentation.html

The implementation is strictly based on the latest specification found at http://json-schema.org/latest/json-schema-validation.html. All features specified in the specification are supported.

Example:

```
#include <sneaker/json/json.h>
#include <sneaker/json/json_schema.h>
#include <string>

using namespace sneaker::json;

const std::string json_str = "{"
  "\"name\": \"Tomiko Van\","
  "\"age\": 28,"
  "\"interests\": ["
    "music",
    "swimming,"
    "reading"
  "],"
  "\"married\": false,"
  "\"languages\": {"
    "\"Japanese\": \"fluent\","
    "\"Chinese\": \"beginner\","
    "\"English\": \"fluent\""
  "}"
"}";

const std::string schema_str = "{"
  "\"type\": \"object\","
  "\"properties\": {"
    "\"name\": {"
      "\"type\": \"string\","
      "\"maxLength\": 50"
    "},"
    "\"age\": {"
      "\"type\": \"number\","
      "\"minimum\": 0,"
      "\"maximum\": 120"
    "},"
    "\"married\": {"
      "\"type\": \"boolean\""
    "},"
    "\"interests\": {"
      "\"type\": \"array\","
      "\"uniqueItems\": ["
        "\"music\","
        "\"dancing\","
        "\"swimming\","
        "\"reading\""
      "]"
    "},"
    "\"languages\": {"
      "\"type\": \"object\""
    "}"
  "}"
"}";
```

```
JSON json = sneaker::json::parse(json_str);
JSON schema = sneaker::json::parse(schema_str);

sneaker::json::json_schema::validate(json, schema);
```

class sneaker::json::**json_validation_error**

Error thrown when parsing an JSON schema validation fails.

sneaker::json::json_schema::**validate**(**const** *JSON*&, **const** *JSON*&)

Interface for validating a JSON blob with a specified JSON schema. The first argument is the JSON blob to be validated, and the second argument is the JSON schema. The JSON schema passed in must be valid, as no validation is performed on the schema object itself, and an invalid schema will cause undefined behaviors during validation.

If validation is successful, nothing happens. Otherwise an instance of *json_validation_error* is thrown.

3.11 Thread Management and Daemons

Components that faciliate thread management and asynchronous executions.

3.11.1 Daemon Service

An abstraction that provides the necessary functionalities to execute code as a daemon service running in a background thread.

Header file: *sneaker/threading/daemon_service.h*

class sneaker::threading::**daemon_service**

---

explicit daemon_service(bool *wait_for_termination* = false)
Contructor. Takes a boolean argument specifying whether the foreground thread is be blocked for the during which the code in *handle* runs in the background thread. Defaults to *false*.

virtual ~daemon_service()
Destructor. The background thread created is destroyed.

protected virtual void **handle**() = 0
Encapsulates the code that is executed when *start* is called to run the daemon service in the background thread. This method should be overriden in deriving classes.

virtual bool **start**()
Starts the daemon service in a background thread and executes the code in defined in *handle* immediately after the background thread is created.

Returns a boolean indicating whether the background thread was created successfully, *true* is successful, *false* otherwise.

3.11.2 Fixed-time Interval Daemon Service

A deriving type of *sneaker::threading::daemon_service* where the code is executed over a regular time interval.

Header file: *sneaker/threading/fixed_time_interval_daemon_service.h*

class `sneaker::threading::`**`fixed_time_interval_daemon_service`**

`typedef void(*ExternalHandler)(void*)`
Type of the external handler to be invoked.

`fixed_time_interval_daemon_service`(size_t, ExternalHandler, bool, size_t)
Constructor. The first argument specifies the interval duration in milliseconds. The second argument takes an external handler which gets called once during each interval, of type *void(*ExternalHandler)(void)*. The third argument specifies if the foreground thread should wait for the background thread which it is running, and the last argument specifies the maximum number of iterations to run.

virtual `~fixed_time_interval_daemon_service`()
Destructor. The background thread created is destroyed.

size_t **`interval`**() **const**
Returns the time interval of the daemon.

3.11.3 Atomic Incrementor

An abstraction that encapsulates the logic to handle atomic incremental operations.

Header file: *sneaker/threading/atomic.h*

`sneaker::threading::atomic<T, T UPPER_LIMIT>`

`atomic`()
Constructor that takes no arguments and initializes the initial value of the encapsulating type to *0*.

`atomic`(T *value*)
Constructor that takes a value of the encapsulating type and makes that the initial value.

`atomic`(**const** atomic<T, UPPER_LIMIT>&)
Copy constructor.

atomic<T, UPPER_LIMIT> &**`operator=`**(**const** T&)
Assignment operator.

atomic<T, UPPER_LIMIT> &**`operator++`**()
Pre-increment operator. If the value exceeds *UPPER_LIMIT* after the increment, the value wraps backs to *0*.

atomic<T, UPPER_LIMIT> &**`operator++`**(int)
Post-increment operator. If the value exceeds *UPPER_LIMIT* after the increment, the value wraps backs to *0*.

`operator T`() **const**
Conversion operator.

bool **`operator==`**(**const** T&) **const**
Equality operator.

bool **operator!=**(**const** T&) **const**
Inequality operator.

3.12 Algorithms

Implementations of advanced algorithms that are handy for solving frequently encountered problems.

3.12.1 Tarjan's Strongly Connected Graph Algorithm

Abstraction that implements Tarjan's Strongly Connected Graph Algorithm.

Header file: *sneaker/algorithm/tarjan.h*

class sneaker::algorithm::**tarjan**<T>

This class is a templated class that takes a type of the encapsulating value in the vertices. The method *get_components()* takes a list of vertices, assuming that the vertices form the graph in mind. It returns an instance of *strongly_connected_component_list*, which has information on the independent components as well as cycles in the entire graph.

NOTE: An instance of the class can only perform cycle detection once. Running the method on the same instance more than once will result in inaccurate results.

```
#include <sneaker/algorithm/tarjan.h>

/* Tests the following object graph:
 * (1) -> (2) -> (3)
 * ^              |
 * |______________|
 */

using namespace sneaker::algorithm;

std::vector<tarjan<int>::vertex*> vertices;

tarjan<int>::vertex v1(1);
tarjan<int>::vertex v2(2);
tarjan<int>::vertex v3(3);

v1.dependencies().push_back(&v2);
v2.dependencies().push_back(&v3);
v3.dependencies().push_back(&v1);

vertices.push_back(&v1);
vertices.push_back(&v2);
vertices.push_back(&v3);

tarjan<int> algo;
auto components = algo.detect_cycle(vertices);

// This graph has one component, which is an independent component,
// thus no cycles detected.
assert(1 == components.size());
assert(0 == components.independent_components().size());
assert(1 == components.cycles().size());
```

typedef std::vector<T> Enumerable
Type represents each strongly connected component in the graph.

typedef std::vector<vertex*> VerticesSet
Type of a set of vertices in the graph.

class sneaker::algorithm::tarjan<T>::**vertex**

Type represents a vertex in a graph.

vertex()
Constructor.

explicit vertext(T)
Constructor that takes an instance of the encapsulating type.

type iterator
The iterator for the list of neighbor vertices.

bool **operator==**(**const** vertex&)
Equality operator.

bool **operator!=**(**const** vertex&)
Inequality operator.

iterator **begin**()
Returns an iterator that points to the beginning of the neighbor vertices.

iterator **end**()
Returns an iterator that points to the end of the neighbor vertices.

int **index**() **const**
Returns the index value associated with this vertex.

int **lowlink**() **const**
Returns the low link value associated with this vertex.

T **value**() **const**
Returns a copy of the encapsulating value.

void **set_index**(int)
Sets the index value of this vertex.

void **set_lowlink**(int)
Sets the low link value of this vertex.

VerticesSet &**dependencies**()
Returns the list of neighbor vertices.

class sneaker::algorithm::tarjan<T>::**strongly_connected_component_list**

Represents a set of strongly connected components in a directed graph.

strongly_connected_component_list()
Constructor.

void **add**(**const** Enumerable&)
Adds a strongly connected component list to the collection.

size_t **size**() **const**
Returns the number of strongly connected components in the graph.

std::vector<Enumerable> **independent_components**() **const**
Gets the set of independent components in the graph.

std::vector<Enumerable> **cycles**() **const**
Gets the set of cycles in the graph.

tarjan()
Constructor.

strongly_connected_component_list **get_components**(**const** VerticesSet&)
Given a set of vertices in a graph, returns a set of connected components in the graph.

3.13 Utilities

Utility components that are useful in day-to-day developments.

3.13.1 Command-line Program Interface

A base class that encapsulates logic for instantiating and running command-line programs and parsing command-line arguments.

Header file: *sneaker/utility/cmdline_program.h*

class sneaker::utility::**cmdline_program**

---

int **run**(int *argc*, char ***argv*)
Member function that takes the command-line argument parameters from *int main()* and pass them through to the program to run.

void **add_string_parameter**(**const** char **name*, **const** char **description*, std::string **res*)
Adds a string-value named parameter to the program, along with a description.

void **add_uint64_parameter**(**const** char **name*, **const** char **description*, std::string **res*)
Adds a 64-bit integer-value named parameter to the program, along with a description.

void **add_uint32_parameter**(**const** char **name*, **const** char **description*, std::string **res*)
Adds a 32-bit integer-value named parameter to the program, along with a description.

void **add_float_parameter**(**const** char **name*, **const** char **description*, std::string **res*)
Adds a floating point named parameter to the program, along with a description.

void **add_boolean_parameter**(**const** char **name*, **const** char **description*, std::string **res*)
Adds a boolean-value named parameter to the program, along with a description.

void **add_positional_parameter**(**const** char **name*, int *n*)
Adds a named positional parameter to the program, at index *n*.

template<typename T>
void **add_array_parameter**(**const** char **name*, **const** char **desc*, std::vector<T> **res*)
Adds a named list of parameters to the program, along with a description.

bool **option_provided**(**const** char **name*) **const**
Determines if a named parameter has been specified.

protected explicit cmdline_program(**const** char *)
Protected constructor that takes the name of the program.

private virtual int **do_run**()
Member function that invokes the program to run. Returns a status value once the program finishes. A value of *0* is returned upon successful run, non-zero values otherwise.

Default implementation does nothing, and is intended to be overriden in subclasses.

private virtual bool **check_parameters**() **const**
Member function that checks the parameters passed to the program, and returns a boolean indicating if the parameters are valid.

Default implementation does nothing, and is intended to be overriden in subclasses.

3.13.2 OS Utilities

Utilities that provide OS-level information and services.

Header file: *sneaker/utility/os.h*

void sneaker::utility::**get_process_mem_usage**(uint64_t **vm_peak*, uint64_t **vm_size*, uint64_t **vm_hwm*, uint64_t **vm_rss*)
Gets the memory usages of the current process, in number of kB.

- *vm_peak*: Peak virtual memory size.
- *vm_size*: Virtual memory size.
- *vm_hwm*: Peak resident set size ("High Water Mark").
- *vm_rss*: Resident set size.

uint64_t sneaker::utility::**get_process_vm_peak**()
Gets the peak virtual memory size of the current process, in kB.

uint64_t sneaker::utility::**get_process_vm_size**()
Gets the current virtual memory size of the current process, in kB.

uint64_t sneaker::utility::**get_process_vm_hwm**()
Gets the peak resident set size of the current process, in kB.

uint64_t sneaker::utility::**get_process_vm_rss**()
Gets the current resident set size of the current process, in kB.

3.13.3 Stacktrace Utilities

Utilities that provide runtime stack trace information.

Header file: *sneaker/utility/stack_trace.h*

class sneaker::utility::**stack_trace**
A class that encompasses the utility functions.

static void **print_stack_trace**(std::ostream *&ost*, unsigned int *max_frames*)
Retrieves stack trace information and forwards to an output stream, with a maximum value on the number of frames to inspect.

3.13.4 Numerics Utilities

Utilities that deal with numerics.

Header file: *sneaker/utility/util.numeric.h*

template<typename T> bool sneaker::utility::floats_equal(T lhs, T rhs, T tolerance=EPSILON
Safe floating-point equality comparisons.

3.13.5 Uniform Table

A utility class for managing formatting and printing data into a uniform table. This class is templatized by the sizes of the columns that form the uniform table.

Header file: *sneaker/utility/uniform_table.h*

Example:

```
#include <sneaker/utility/uniform_table.h>
#include <iostream>

sneaker::utility::uniform_table<2, 10, 10> uniform_table;

uniform_table.write_separator();
uniform_table.write(2, "Ocean", "Blue");
uniform_table.write(3, "Forest", "Green");
uniform_table.write(1, "Volcano", "Red");
std::cout << uniform_table.str();


```
will give the following output:

```
| -- | ---------- | ---------- |
|  2 |      Ocean |       Blue |
|  3 |     Forest |      Green |
|  1 |    Volcano |        Red |
```

template<size_t... ColumnSizes> sneaker::utility::uniform_table

---

std::ostream **stream**() **const**
Returns the stream that content has been written to.

std::string **str**() **const**
Returns a copy of the string written so far.

template<typename... Args> void write(Args... args)
Writes a row in the table, with the supplied variadic arguments.

void **write_separator**()
Writes a horizontal row separator.

3.13.6 I/O Utilities

A set of utility functions for I/O operations.

void sneaker::utility hex_to_bytes(const unsigned char* src, size_t src_len, unsigned char
Converts a string of hexidecimal digits to its byte representation.

void `sneaker::utility::`**hex_to_bytes**(**const** std::string &*src*, std::string **dst*)
Converts a string of hexidecimal digits to its byte representation.

void sneaker::utility::bytes_to_hex(const unsigned char* src, size_t src_len, unsigned char
Converts a byte array to its hexidecimal representation in string.

void `sneaker::utility::`**bytes_to_hex**(**const** std::string &*src*, std::string **dst*)
Converts a byte array to its hexidecimal representation in string.

3.14 Testing

Utilities for tests based on Google Test.

Header file: *sneaker/testing/testing.h*

3.14.1 Fixture Based Test

Base class of test suite that allows one or more fixtures of the same type, and automatically manages their teardowns.

Header file: *sneaker/testing/fixture_based_test.h*

class `sneaker::testing::`**fixture_based_test**<T>

---

type T
Type of the fixture managed by this class.

typedef void(*FixtureTeardownHandler)(T)
Type of fixture teardown handler.

fixture_based_test(FixtureTeardownHandler)
Constructor that takes a fixture teardown handler.

void **add_fixture**(T)
Adds a fixture to be managed by this class.

virtual void **TearDown**()
Handles teardown of fixtures managed by the class.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

U

V

W