Programmieren 1

Modules

Human-Computer Interaction Group

Prof. Dr. Michael Rohs
michael.rohs@hci.uni-hannover.de

Lectures

#	Date	Topic	HÜ→	HÜ←
1	14.10.	Organization, computers, programming, algorithms, PostFix introduction (execution model, IDE, basic operators, booleans, naming)	1	20.10. 23:59
2	21.10.	PostFix (primitive types, functions, parameters, local variables, tests), recipe for atomic data	2	27.10. 23:59
3	28.10.	PostFix (operators, array operations, string operations), recipes for enumerations, intervals, and itemizations	3	3.11. 23:59
4	4.11.	Recipes for compound and variant data, iteration and recursion, PostFix (loops, association arrays, data definitions)	4	10.11. 23:59
5	11.11.	C introduction (if, variables, functions, loops), Programming I C library	5	17.11. 23:59
6	18.11.	Data types, infix expressions, C language (enum, switch)	6	24.11. 23:59
7	25.11.	Compound and variant data, C language (formatted output, struct, union)	7	1.12. 23:59
8	2.12.	C language (arrays, pointers) arrays: fixed-size collections, linear and binary search	8	8.12. 23:59
9	9.12.	Dynamic memory (malloc, free), recursion (recursive data, recursive algorithms)	9	15.12. 23:59
10	16.12.	Linked lists, binary trees, search trees	10	22.12. 23:59
11	13.1.	C language (program structure, scope, lifetime, linkage), function pointers, pointer lists	11	12.1. 23:59
12	20.1.	List operations (filter, map, reduce), modules, documentation	12	19.1. 23:59
13	27.1.	Dynamic data structures (stacks, queues, maps, sets), iterators	(13)	

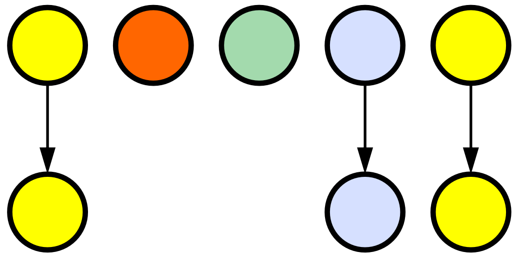
Preview

- List processing operations (filter, map, reduce)
- Modules
- Comments and Documentation

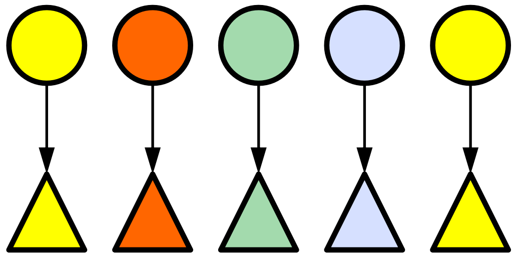
LIST PROCESSING OPERATIONS

List-Processing Operations: filter, map, fold/reduce

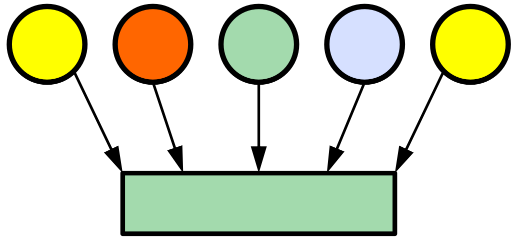
filter:



map:



fold / reduce:



- Operations on each element of a list
- Treat the list as a whole
- Powerful form of abstraction

Map: Create a New List with Transformed (Mapped) Elements

- Function that transforms elements

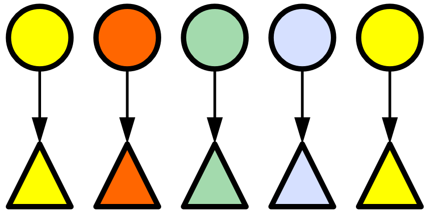
```
typedef void* (*MapFunc)(void* element, int i, void* x);
```

- Map each list element to f(element, index, x)

```
Node* map_list(Node* list, MapFunc f, void* x) {
    if (list == NULL) return NULL; // handle case of empty list first
    Node* first = new_node(f(list->value, 0, x), NULL);
    Node* last = first;
    int i = 1;
    for (Node* node = list->next; node != NULL; node = node->next, i++) {
        last->next = new_node(f(node->value, i, x), NULL);
        last = last->next;
    }
    return first;
}
```

call transformation function f for each element

map:



Map a List of Books to a List of Book Titles

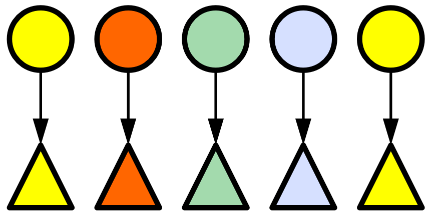
- Transformation function

```
// Book -> String
// Maps a book to its title.
void* book_title(void* element, int i, void* x) {
    Book* b = (Book*) element;
    return b->title;
}
```

- Performing the mapping

```
Node* titles = map_list(list, book_title, NULL); // titles is a list of strings
```

map:



Filter a List according to some Predicate

- Predicate function specifies what to keep

```
typedef bool (*FilterFunc)(void* element, int i, void* x);
```

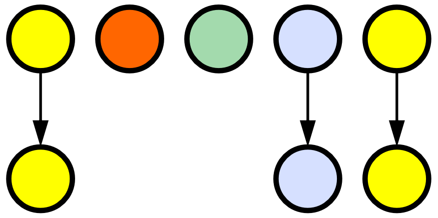
- Produce new list of filtered elements

```
Node* filter_list(Node* list, FilterFunc pred, void* x) {
    Node* result = NULL;
    int i = 0;
    for (Node* node = list; node != NULL; node = node->next, i++) {
        if (pred(node->value, i, x)) {
            result = new_node(node->value, result);
        }
    }
    return reverse_list(result);
}
```

include only those that satisfy the predicate

inefficient, but less code ☺

filter:



Keep only Books Published at or after Year

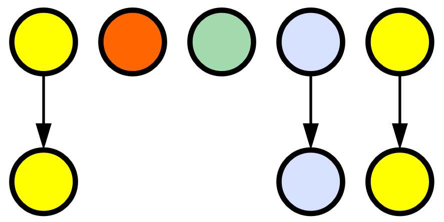
- Predicate function

```
bool published_after(void* element, int i, void* x) {
    Book* b = (Book*) element;
    int* year = (int*) x;
    return b->year > *year;
}
```

- Perform the filtering

```
int year = 1990;
Node* selected_books = filter_list(list, published_after, &year);
println_list(selected_books, book_to_string);
free_list(selected_books, NULL);
```

filter:

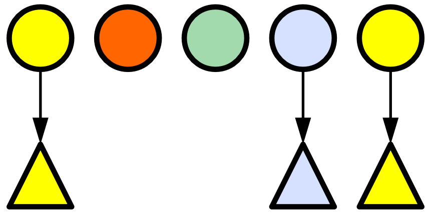


Combining Filtering and Mapping

- Filtering and mapping often done in sequence
- More efficient to combine them
- Needs a predicate and a transformation function

```
Node* filter_map_list(Node* list, FilterFunc pred, MapFunc map, void* x) {
    Node* result = NULL;
    int i = 0;
    for (Node* node = list; node != NULL; node = node->next, i++) {
        if (pred(node->value, i, x)) {
            result = new_node(map(node->value, i, x), result);
        }
    }
    return reverse_list(result);
}
```

filter + map:



Reduce/Fold: Aggregating a List

- Aggregation function modifies state variable

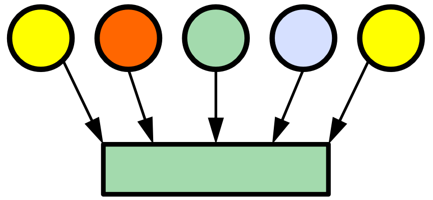
```
typedef void (*ReduceFunc)(void* state, void* element, int index);
```

- Aggregate the list using f, modifies state variable

```
void reduce_list(Node* list, ReduceFunc f, void* state) {
    int i = 0;
    for (Node* node = list; node != NULL; node = node->next, i++) {
        f(state, node->value, i);
    }
}
```

side effect:
f modifies state
on each call

fold / reduce:



Computing the Average Publication Year

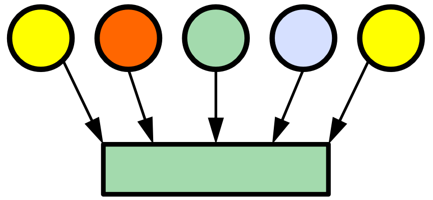
- Aggregation function

```
void sum_years(void* state, void* element, int index) {
    int* sum = (int*)state;          read current state
    Book* b = (Book*)element;
    *sum = *sum + b->year;           update state (add year
}                                    of current element)
```

- Performing the aggregation

```
int years = 0;          initial state
reduce_list(list, sum_years, &years);
printf("average year: %d\n", years / length_list(list)); // assumes length > 0
```

fold / reduce:



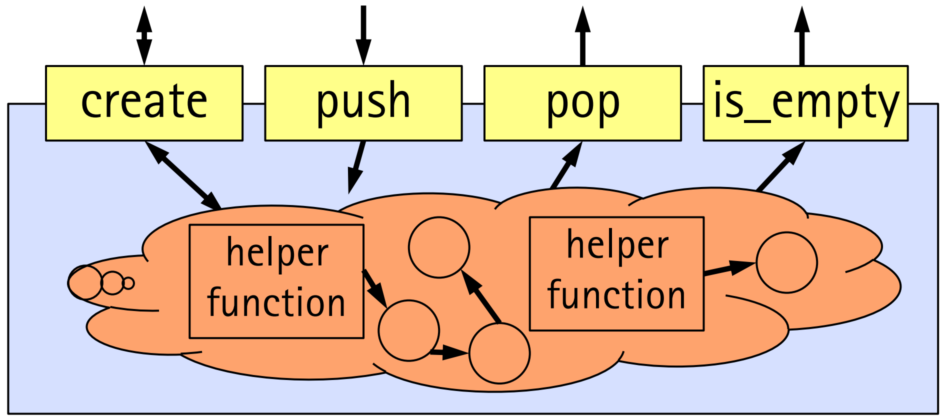
MODULES

Modules

- The above example involves many functions related to different objects.
- How to organize these into different .c and .h files?
  - Implementation: .c files, separate compilation
  - Interface: .h files, included by .c files and .h files
- How to document the public interface of a module?
  - So that others can use it
  - So that it internals are hidden
- How to use implementation comments

Information Hiding Principle

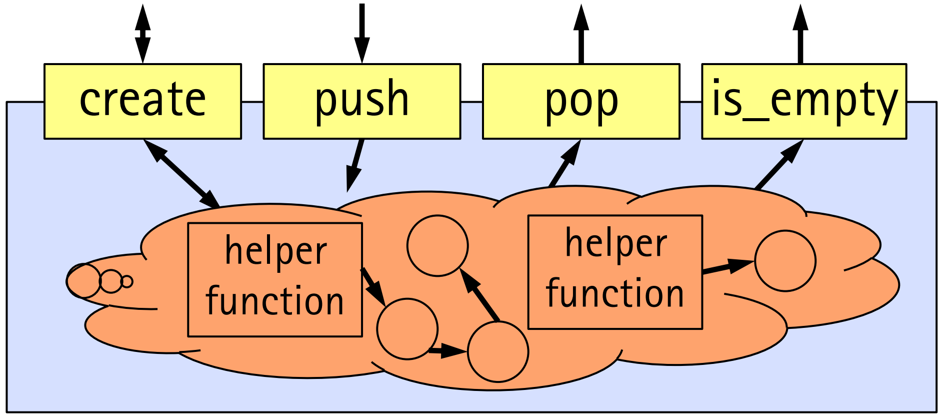
- Hide implementation details of a module and provide access to the module only via its public functions
  - Capture knowledge about design decisions within a module
  - Hide details about how to implement some mechanism
- Public interface
  - Simple relative to the functionality the module provides
  - Should be stable over time
- Private implementation
  - May be complex
  - May change over time



David Parnas: On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM, December 1972.

Information Hiding Principle

- Advantages of information hiding
  - The public interface is stable, simple, and clean
  - No need to learn implementation details
  - Implementation details can change without breaking client code (if public interface does not change)
  - Invariants can be ensured (no direct access to internal data)
- When designing a module, think about
  - What design decisions can be hidden inside the module?
  - How to simplify the interface?
  - What do users need to know to use the module?
  - How to minimize coupling between modules?



David Parnas: On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM, December 1972.

TOKEN-ARRAY MODULE EXAMPLE

Module Implementation – token_array.c

```c
// A token can either be a word or a number.
typedef enum { Word, Number } TokenType;

// Represents an instance of a token of a certain type.
struct Token {
    char* value;            // '\0'-terminated string, dynamically allocated, owned by Token
    TokenType type;
};

// Represents an array of tokens.
struct TokenArray {
    int count;              // the number of tokens in the array
    Token tokens[];         // array of count tokens (flexible array member must be last)
};
```

Token-Array Client

```
char* text = s_read_file(argv[1]);
TokenArray* ta = parse_text(text);
free(text);
printf("%d tokens\n", ta->count);
if (ta->count > 0) {
    Token* t = ta->tokens;
    token_print(t);
}
if (ta->count > 1) {
    Token* t = ta->tokens + 1;
    token_print(t);
    if (t->type == Number) {
        printf("type = %d\n", atoi(t->value));
    }
}
...
```

direct access to internal representation (struct members),

changing the representation breaks clients

```
typedef enum {
    Word,
    Number
} TokenType;

struct Token {
    char* value;
    TokenType type;
};

struct TokenArray {
    int count;
    Token tokens[];
};
```

Token-Array Client

```
char* text = s_read_file(argv[1]);
TokenArray* ta = parse_text(text);
free(text);
printf("%d tokens\n", token_array_count(ta));
if (token_array_count(ta) > 0) {
    Token* t = token_array_get(ta, 0);
    token_print(t);
}
if (token_array_count(ta) > 1) {
    Token* t = token_array_get(ta, 1);
    token_print(t);
    if (token_type(t) == Number) {
        printf("type = %d\n", atoi(token_value(t)));
    }
}
...
```

use accessor functions

```
typedef enum {
    Word,
    Number
} TokenType;

struct Token {
    char* value;
    TokenType type;
};

struct TokenArray {
    int count;
    Token tokens[];
};
```

Module Interface – token_array.h

```
#ifndef TOKEN_ARRAY_H
#define TOKEN_ARRAY_H

typedef struct Token Token;
typedef enum {Word, Number} TokenType;
typedef struct TokenArray TokenArray;

// The value of a token.
char* token_value(Token* t);

// The type of a token.
TokenType token_type(Token* t);

// Prints a token and its type.
void token_print(Token* t);
```

hide representation (struct definitions) in module implementation

enumeration constants are part of the interface

functions provide access to the token array

Module Interface – token_array.h

```
// Creates a token array of the given capacity.
// All elements are initialized to zero.
TokenArray* token_array_create(int count);

// Frees the token array, including the string values.
void token_array_free(TokenArray* a);

// Sets the value of the token at index. The value must have been dynamically
// allocated with xmalloc/xcalloc. Does not copy the value.
// Token array owns the value (i.e., is responsible for deletion).
void token_array_set(TokenArray* a, int index, char* value, int type);

// Returns a pointer to the Token at index i.
// Exits the program if i is not a valid index.
Token* token_array_get(TokenArray* a, int i);
```

documentation needs to be sufficiently detailed to be able to use the module

Module Interface – token_array.h

```
// Returns the number of tokens in the array.
int token_array_count(TokenArray* a);

// The index of the token in the array.
// Returns -1, if the token is not in the array.
int token_array_index(TokenArray* a, Token* t);

// Prints the array of tokens, one token per line.
void token_array_print(TokenArray* a);

#endif
```

Token-Array Client

static: internal linkage, because not to be used outside of this module

```
// Counts the tokens of the given type in the array. Internal helper function.
static int count_tokens_of_type(TokenArray* a, int type) {
    int result = 0;
    int n = token_array_count(a); // use public function of module token_array
    for (int i = 0; i < n; i++) {
        Token* t = token_array_get(a, i); // public function of token_array
        if (token_type(t) == type) result++; // public function of token_array
    }
    return result;
}
```

DOCUMENTATION GENERATOR DOXYGEN

pointer_queue.h: Interface of a First-In-First-Out Queue

```
#ifndef POINTER_QUEUE_H
#define POINTER_QUEUE_H
#include "base.h"

typedef struct Queue Queue;

Queue* new_queue(void);
void free_queue(Queue* q);
void put_queue(Queue* q, void* x);
void* get_queue(Queue* q);
bool is_empty_queue(Queue* q);

#endif
```

Image source: Getty

pointer_queue.h: Interface, with Doxygen Annotations

```
/** @file
A queue that points to elements of arbitrary type.
@author Michael Rohs
@date 20.1.2018
*/
#ifndef POINTER_QUEUE_H
#define POINTER_QUEUE_H
#include "base.h"

// Opaque type. Actual components specified in .c-file.
typedef struct Queue Queue;
```

pointer_queue.h: Interface, with Doxygen Annotations

```
/**
Creates an empty stack.
@return the dynamically allocated queue
*/
Queue* new_queue(void);

/**
Frees the queue itself, but not the elements.
@param q the queue
*/
void free_queue(Queue* q);
```

pointer_queue.h: Interface, with Doxygen Annotations

```
/**
Adds element x to the back of the queue.
@param[in,out] q the queue
@param[in] x element to enqueue
*/
void put_queue(Queue* q, void* x);

/**
Gets the next element from the front of the queue.
@param[in,out] q the queue
@return the element at the front of the queue
*/
void* get_queue(Queue* q);
```

pointer_queue.h: Interface, with Doxygen Annotations

```
/**
Checks if the queue is empty.
@param[in] q the queue
@return true if the queue is empty, false otherwise
*/
bool is_empty_queue(Queue* q);

#endif
```

pointer_queue.c: Representation, Data Definitions

```c
#include "pointer_queue.h"

// Helper structure. Not mentioned in header file.
typedef struct PointerNode {
    void* value;
    struct PointerNode* next;
} PointerNode;

// Helper function. Not mentioned in header file.
static PointerNode* new_pointer_node(void* value, PointerNode* next) {...}

struct Queue { // Representation of the abstract data type
    PointerNode* head; // front of queue (first to be served)
    PointerNode* tail; // back of queue (last to be served)
};
```

static: internal linkage, because not to be used outside of this module

pointer_queue.c: Function Implementations

```
// Implementations of "public" functions.

Queue* new_queue(void) {...}

void free_queue(Queue* q) {...}

void put_queue(Queue* q, void* x) {...}

void* get_queue(Queue* q) {...}

bool is_empty_queue(Queue* q) {...}
```

Documentation of a Module

- Doxygen Tool for generating documentation
  - http://doxygen.org
  - Command line tool: doxygen, generates HTML files
  - Configuration file: Doxyfile
- Special comments and annotations in the source code

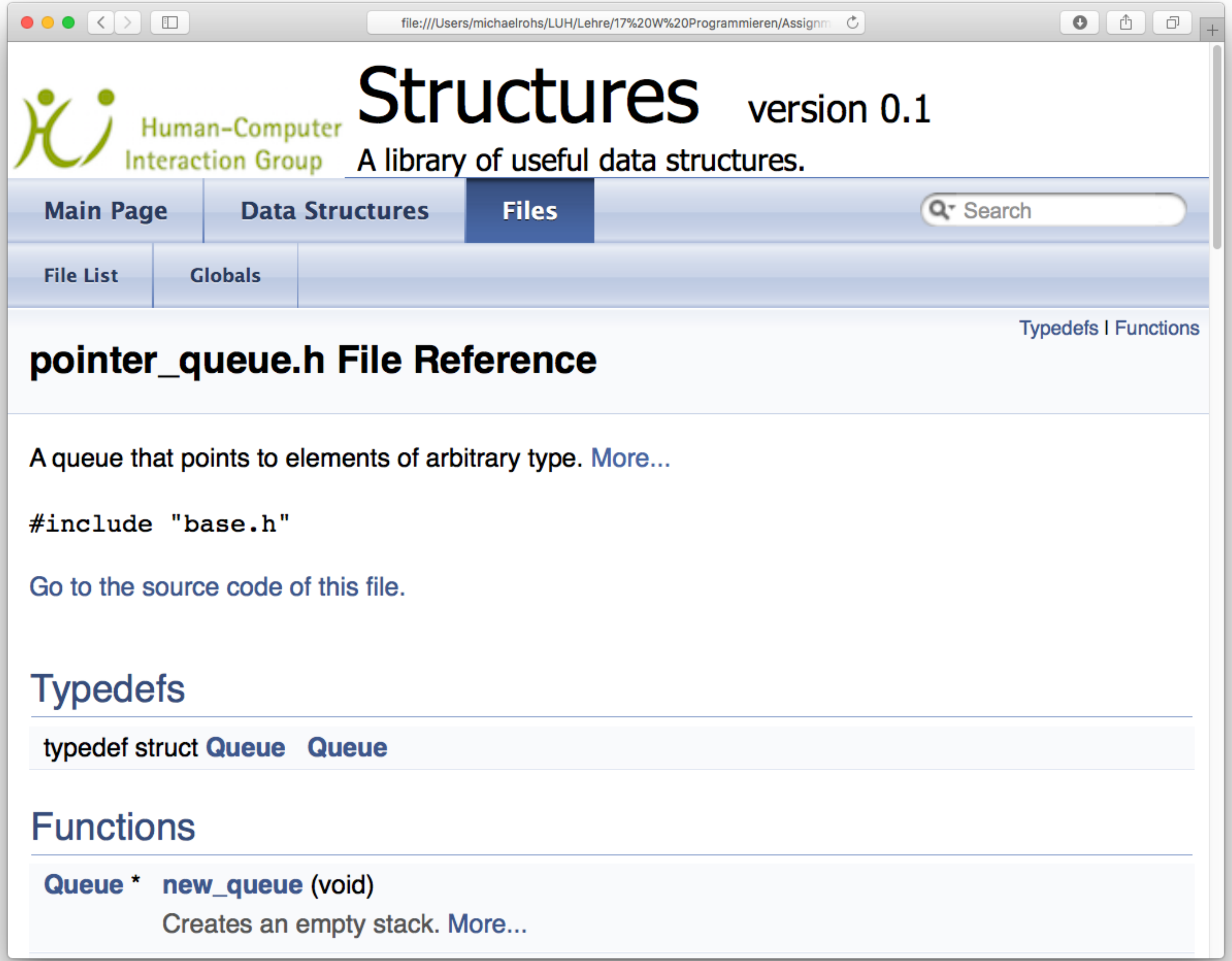
/** ... */	encloses a Doxygen comment, /// single line comment, ///< preceding entity
@param[in,out]	parameter, optional: in, out, inout
@return	return value
@pre	precondition
@post	postcondition
@see	cross reference
@code ... @endcode	code example

  - Markdown (lists, tables, hyperlinks, diagrams, etc.)

https://www.doxygen.nl/manual/commands.html

Doxyfile: Doxygen Configuration File

```
...

# The PROJECT_NAME tag is a single word (or a sequence of words surrounded by
# double-quotes, unless you are using Doxywizard) that should identify the
# project for which the documentation is generated.

PROJECT_NAME           = "Structures"

# The PROJECT_NUMBER tag can be used to enter a project or revision number. This
# could be handy for archiving the generated documentation or if some version
# control system is used.

PROJECT_NUMBER         = "version 0.1"

# Using the PROJECT_BRIEF tag one can provide an optional one line description
# for a project that appears at the top of each page and should give viewer a
# quick idea about the purpose of the project. Keep the description short.

PROJECT_BRIEF          = "A library of useful data structures."

...
```


file:///Users/michaelrohs/LUH/Lehre/17%20W%20Programmieren/Assignm
Human-Computer Interaction Group
Main Page
Data Structures
Files
Search
File List
Globals
Typedefs | Functions

Structures version 0.1

A library of useful data structures.

pointer_queue.h File Reference

A queue that points to elements of arbitrary type. More...

```
#include "base.h"
```

Go to the source code of this file.

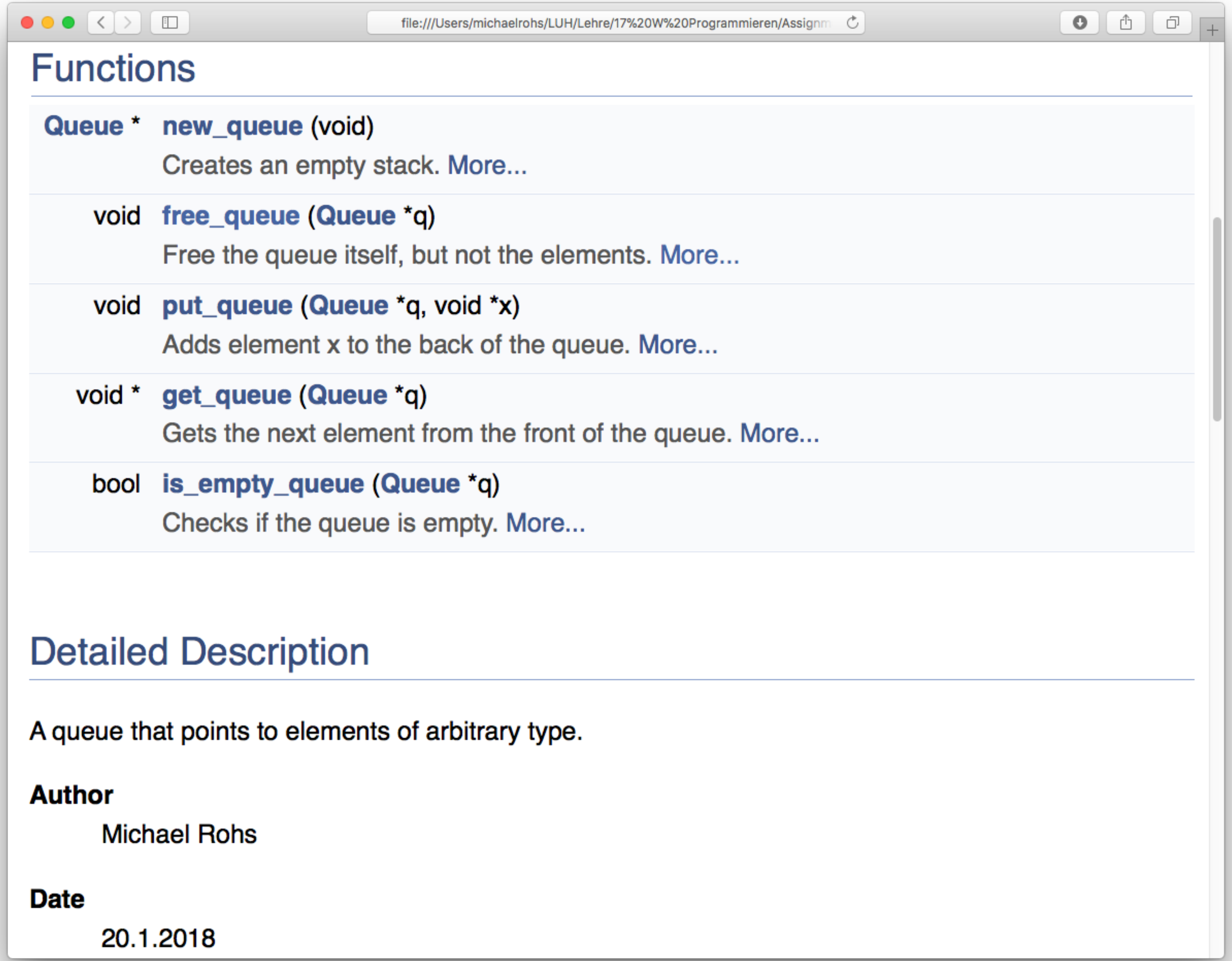
Typedefs

typedef struct Queue Queue

Functions

Queue * new_queue (void)

Creates an empty stack. More...


file:///Users/michaelrohs/LUH/Lehre/17%20W%20Programmieren/Assignm

Functions

Queue *	**new_queue** (void) Creates an empty stack. More...
void	**free_queue** (Queue *q) Free the queue itself, but not the elements. More...
void	**put_queue** (Queue *q, void *x) Adds element x to the back of the queue. More...
void *	**get_queue** (Queue *q) Gets the next element from the front of the queue. More...
bool	**is_empty_queue** (Queue *q) Checks if the queue is empty. More...

Detailed Description

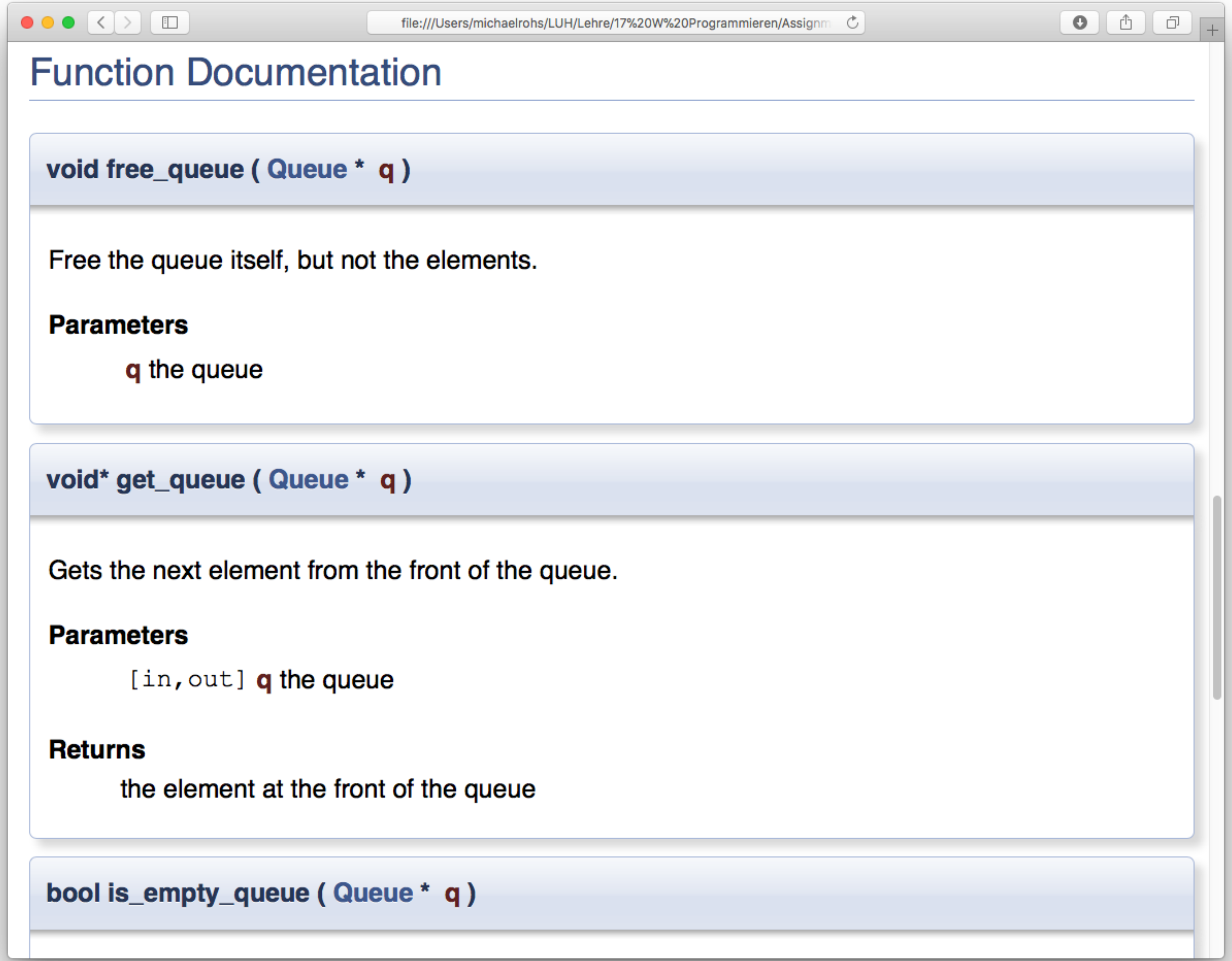
A queue that points to elements of arbitrary type.

Author

Michael Rohs

Date

20.1.2018


file:///Users/michaelrohs/LUH/Lehre/17%20W%20Programmieren/Assignm

Function Documentation

void free_queue (Queue * q)

Free the queue itself, but not the elements.

Parameters

q the queue

void* get_queue (Queue * q)

Gets the next element from the front of the queue.

Parameters

`[in,out]` **q** the queue

Returns

the element at the front of the queue

bool is_empty_queue (Queue * q)

DOCUMENTATION OF A MODULE

Documentation of a Module

- A module should be usable just by looking at the documentation of its public functions and types
- Module's interface provides an abstraction of some functionality
- Abstraction: Simplified view of an entity, which preserves essential information, but omits details that safely can be ignored
  - Comments support abstraction by providing a simpler, higher-level view
- Documentation (as comments) is necessary, because a significant amount of design information cannot be expressed in code
  - Without comments the only abstraction of a function is its declaration
  - Human language is less precise, but more expressive in describing the meaning of an entity

John Ousterhout: A Philosophy of Software Design. 2018.

Writing Comments

- Writing comments during the design process improves the design
  - What was in the mind of the designer, but couldn't be represented in code
  - When comment is difficult to write → design might be too complicated
  - The simpler the comments can be, the better
- Good comments describe things at a different level of detail than the code
  - Don't repeat the code, provide additional information
  - Lower level: precision, clarifying the exact meaning of the code
  - Higher level: intuition, the reasoning behind the code, overall intent and structure, conceptual framework

John Ousterhout: A Philosophy of Software Design. 2018.

Interface Comments and Implementation Comments

- Interface comments
  - Don't describe implementation details in interface comments
  - Describes the module's interface: parameters and return types of functions, constraints
  - What does the function do? What do the parameters mean? What does the function return? Are interval bounds inclusive or exclusive? Is NULL permitted? Does the value have to be freed? Who is responsible for freeing? May the parameter get modified? Are there preconditions on the parameters? What does the function guarantee (postconditions)? Does the function have side effects (e.g., changing a global variable)? What is the computational complexity of the function? etc.
- Implementation comments
  - What and why: describe what the code is doing, why it was done in this way
  - Implementation comment inside a function, describes how the code works internally

John Ousterhout: A Philosophy of Software Design. 2018.

Comments

```
String class_name(Object* object);
```

- What happens if object is NULL?
- Will the function modify object?
- Do I have to free the returned string?
- May I modify the returned string?
- May the function return NULL?

object.h with Free Form Comments

```
// Returns the class name of the object.
// The same pointer is returned for each instance of the same class.
// The returned String is immutable.
String class_name(Object* object);

// Creates a copy of the object.
Object* copy(Object* object);

// Deletes the object.
void delete(Object* object);

// Checks if two objects are equal.
bool equal(Object* obj1, Object* obj2);

// Returns a String representation of the object.
// The returned String is dynamically allocated and has to be freed by the caller.
String to_string(Object* object);
```

object.h with Doxygen Comments

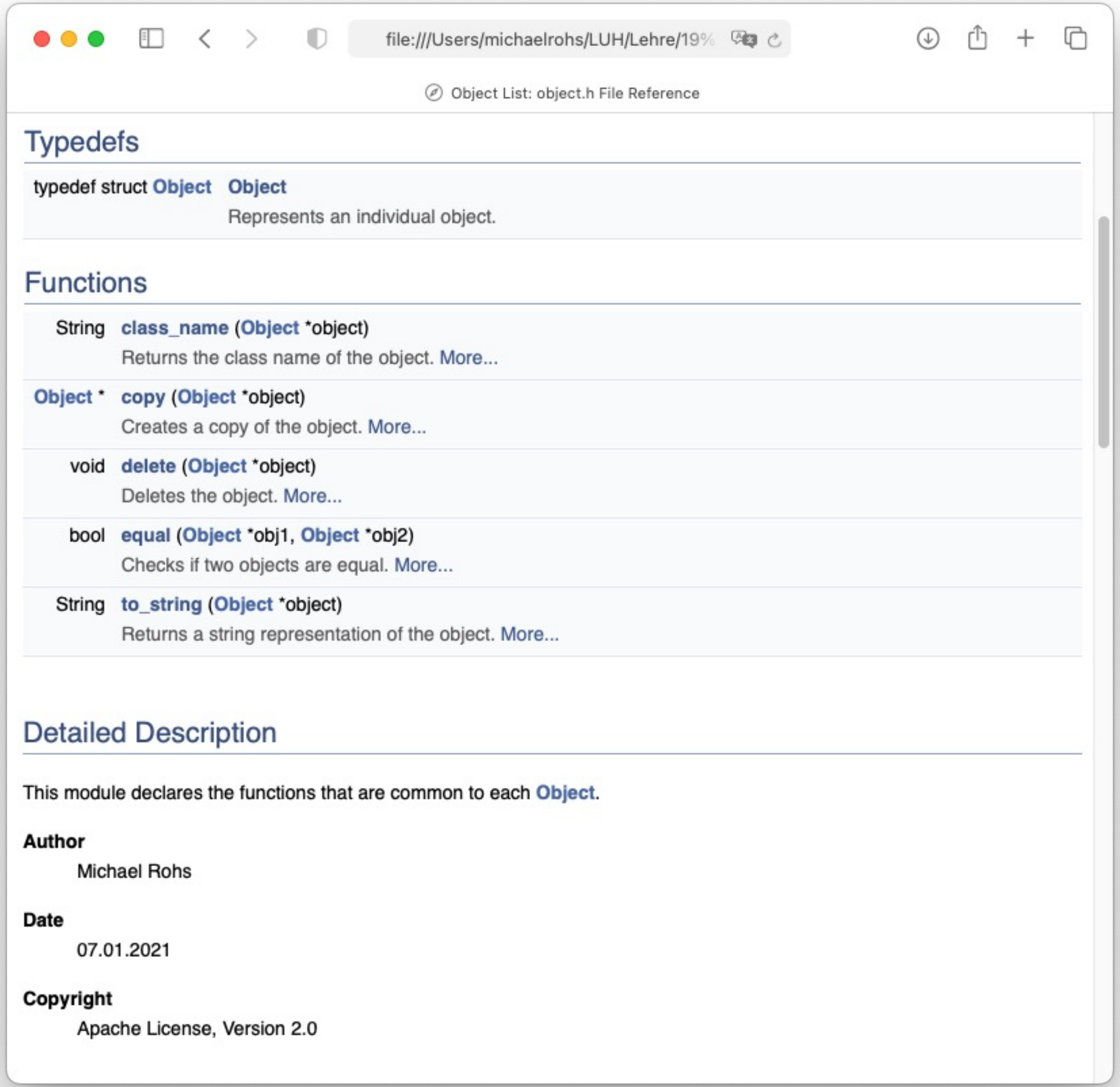
```
/** @file
This module declares the functions that are common to each object.
@author Michael Rohs
@date 07.01.2021
@copyright Apache License, Version 2.0
*/
#ifndef OBJECT_H
#define OBJECT_H
#include "base.h"
typedef struct Object Object;
/**
Returns the class name of the object. The same pointer is returned for each instance of the same class.
@param[in] object
@return the class name as an immutable string
@pre "not null", object
*/
String class_name(Object* object);
...
#endif
```

Module comments: Documentation of the purpose of the module. Metadata about the module.

Interface comments: Documentation of the interface of each public function.

object.h with Doxygen Comments

```
/** @file
This module declares functions that are common...
@author Michael Rohs
@date 07.01.2021
@copyright Apache License, Version 2.0
*/
```


file:///Users/michaelrohs/LUH/Lehre/19%

Object List: object.h File Reference

Typedefs

typedef struct Object Object
Represents an individual object.

Functions

String	class_name (Object *object) Returns the class name of the object. More...
Object *	copy (Object *object) Creates a copy of the object. More...
void	delete (Object *object) Deletes the object. More...
bool	equal (Object *obj1, Object *obj2) Checks if two objects are equal. More...
String	to_string (Object *object) Returns a string representation of the object. More...

Detailed Description

This module declares the functions that are common to each Object.

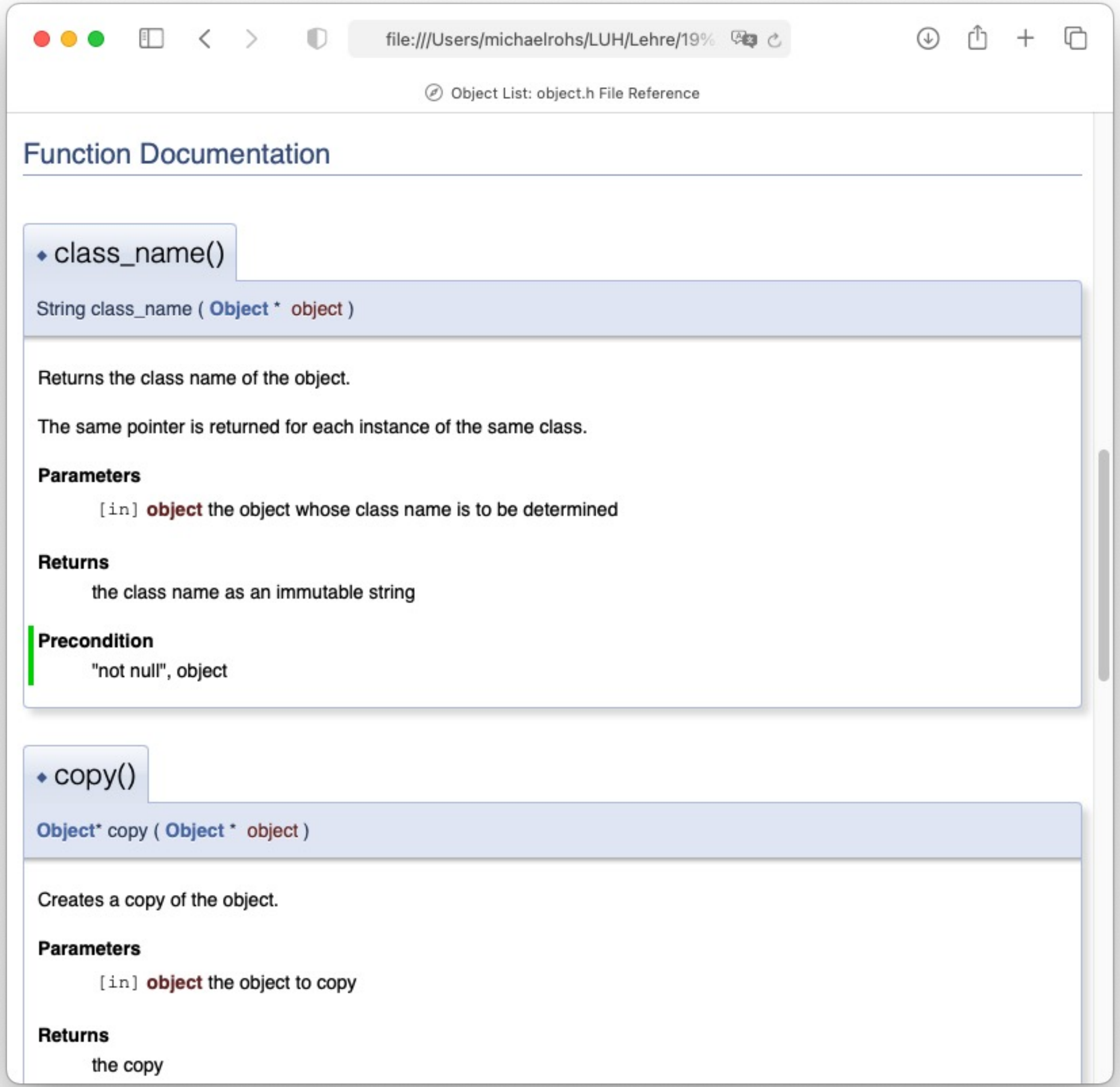
Author
Michael Rohs

Date
07.01.2021

Copyright
Apache License, Version 2.0

object.h with Doxygen Comments

```
/**
Returns the class name of the object. The same
pointer is returned for each instance of the same
class.
@param[in] object the object whose class name
is to be determined
@return the class name as an immutable string
@pre "not null", object
*/
String class_name(Object* object);
```


file:///Users/michaelrohs/LUH/Lehre/19%
Object List: object.h File Reference

Function Documentation

◆ class_name()

String class_name (Object * object)

Returns the class name of the object.

The same pointer is returned for each instance of the same class.

Parameters

[in] **object** the object whose class name is to be determined

Returns

the class name as an immutable string

Precondition

"not null", object

◆ copy()

Object* copy (Object * object)

Creates a copy of the object.

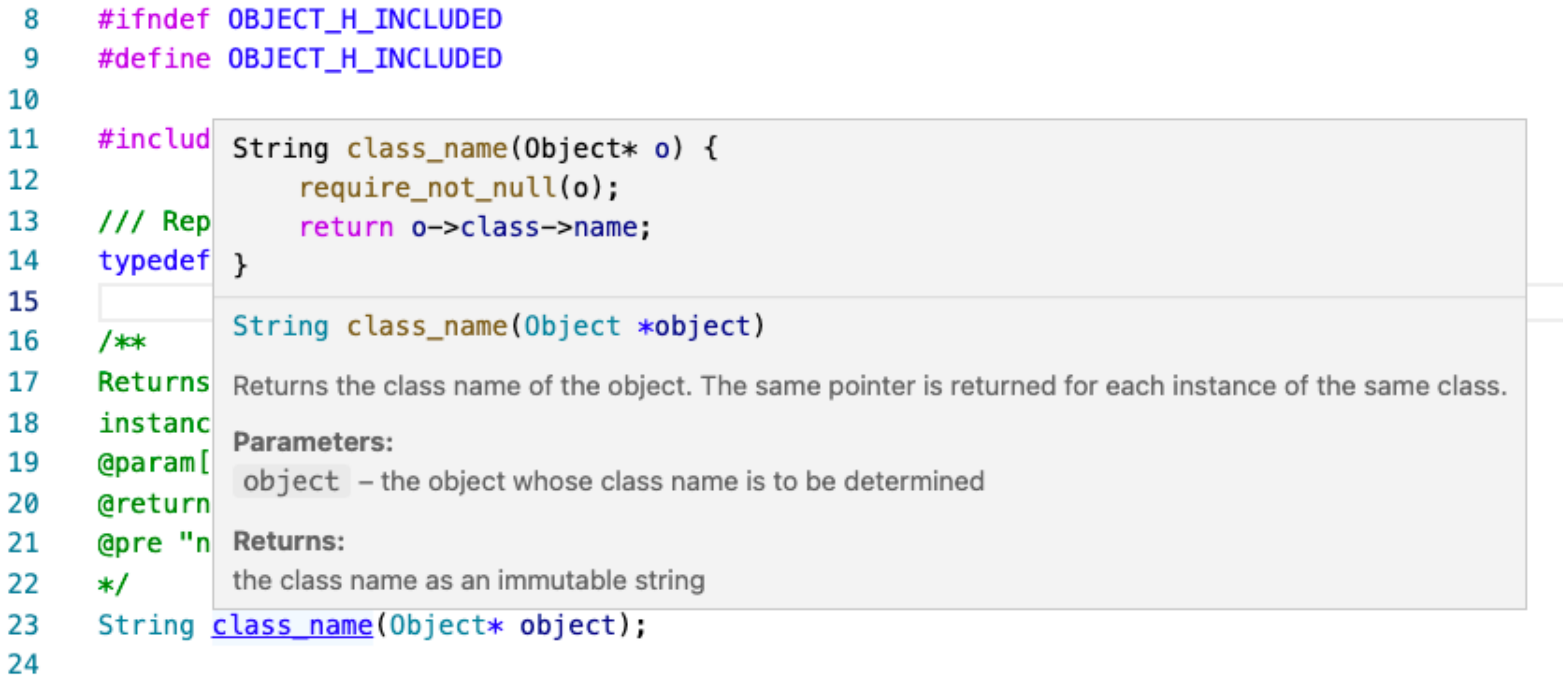
Parameters

[in] **object** the object to copy

Returns

the copy

Some editors understand documentation comments.
Example: Visual Studio Code



```
#ifndef OBJECT_H_INCLUDED
#define OBJECT_H_INCLUDED

#includ

/// Rep
typedef

/**
Returns
instanc
@param[
@return
@pre "n
*/
String class_name(Object* object);

```

```
String class_name(Object* o) {
    require_not_null(o);
    return o->class->name;
}
```

```
String class_name(Object *object)
```

Returns the class name of the object. The same pointer is returned for each instance of the same class.

Parameters:

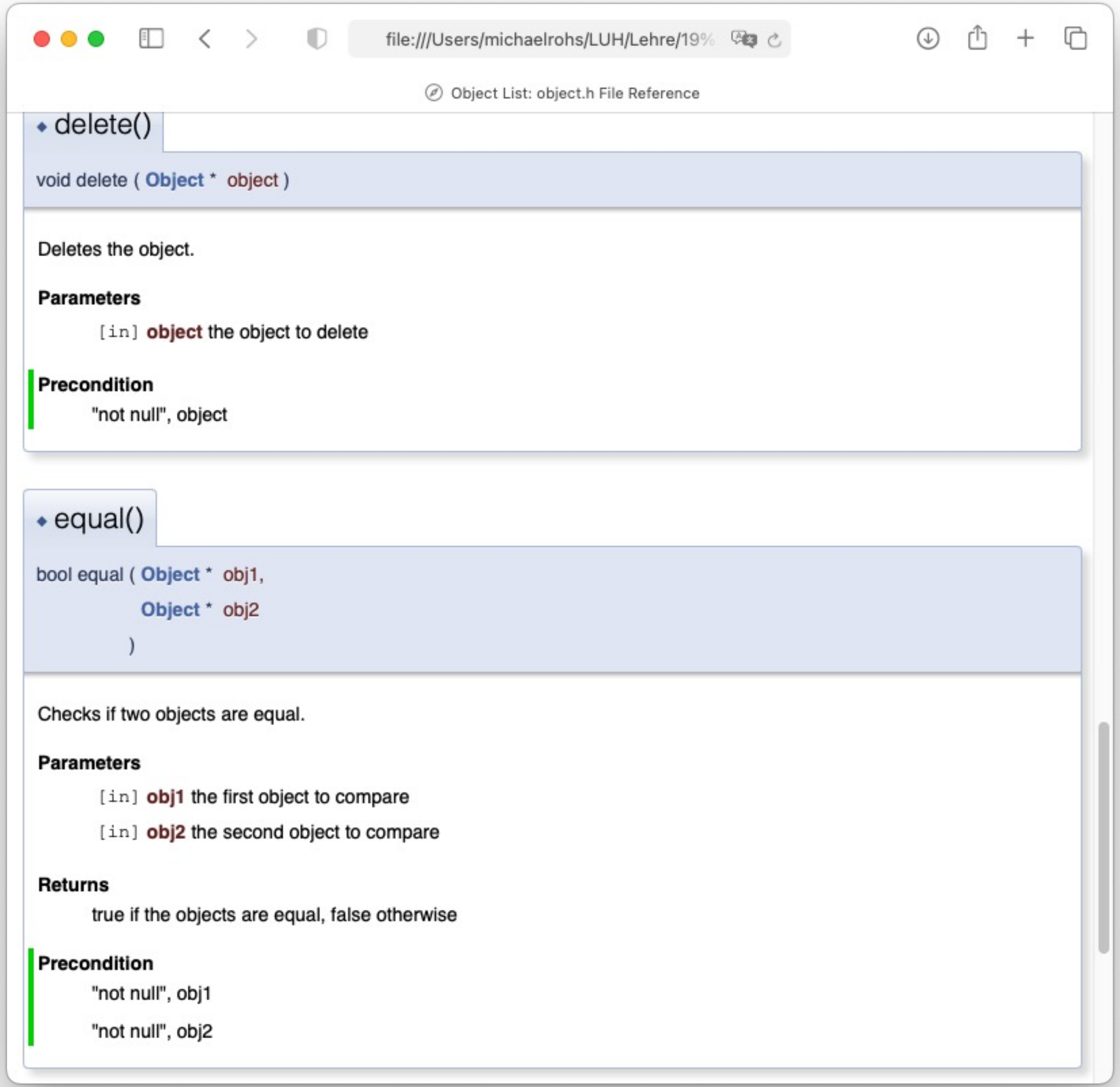
`object` – the object whose class name is to be determined

Returns:

the class name as an immutable string

object.h with Doxygen Comments

```
/**
Checks if two objects are equal.
@param[in] obj1 the first object to compare
@param[in] obj2 the second object to compare
@return true if the objects are equal,
           false otherwise
@pre "not null", obj1
@pre "not null", obj2
*/
bool equal(Object* obj1, Object* obj2);
```


file:///Users/michaelrohs/LUH/Lehre/19%

Object List: object.h File Reference

◆ delete()

```
void delete ( Object * object )
```

Deletes the object.

Parameters

[in] **object** the object to delete

Precondition

"not null", object

◆ equal()

```
bool equal ( Object * obj1,
             Object * obj2
           )
```

Checks if two objects are equal.

Parameters

[in] **obj1** the first object to compare

[in] **obj2** the second object to compare

Returns

true if the objects are equal, false otherwise

Precondition

"not null", obj1

"not null", obj2

Summary

- List processing operations
  - Filter, map, reduce
- Modularization
  - Projects consist of multiple files
  - The .h file contains the public interface
    - There may be a need for "private" header files as well
  - The .c file contains the implementation
    - Helper functions should be `static` (private to the module)
    - Struct definitions should be private to the module (if possible)
- Comments and Documentation
  - Documentation helps to create an abstraction