Annotated C Libraries

The C0 system libraries (which we import using the `#use` directive — for example `#use <conio>`) come with C0 contract annotations for the functions they provide. The C system libraries (e.g., `#include <stdlib.h>`) do not. This makes it harder to write safe code that uses them as we often have to consult documentation to be sure we are calling them with valid inputs, and because what constitutes a valid input is not as straightforward in C. The purpose of this guide is to provide C0-style contract annotations to functions commonly used from popular C libraries.

Quick Links *(hover over the link for a preview)*

- **<stdlib.h>**: malloc, calloc, free
- **<string.h>**: strlen, strcmp, strncmp, strcpy, strncpy, strcat, strncat
- **<stdio.h>**: printf, fopen, fclose
- **<assert.h>**: **assert**
- **"xalloc.h"**: xmalloc, xcalloc
- **"contracts.h"**: REQUIRES, ENSURES, ASSERT, IF_DEBUG

1 Conventions

1.1 Additional Contract-Only Functions

The standard C0 contract functions (e.g., `\length`) are insufficient to describe many of the constraints that values in a C program are subject to. For this reason, we will pretend we have a few other contract-only functions when annotating C library functions. **These are not valid C0 contract functions.**

- `\allocated(ptr)`:

  It takes as input a pointer `ptr` (of generic type **`void*`**) and returns a **`bool`**.

  - It evaluates to `true` if `ptr` was returned by `malloc` or similar.
  - It returns `false` otherwise.

  Examples:

```
int *p = malloc(sizeof(int) * 2);
//@assert \allocated(p)    == true;
//@assert \allocated(p+1)  == false;
//@assert \allocated(NULL) == false;
```

- `\bytes_to_end(ptr)`:

  It takes as input a pointer `ptr` (of generic type **`void*`**) and returns a **`long`**.

  - It evaluates to `-1` if `ptr` is not a pointer to or within a currently allocated block of memory (returned my `malloc` or similar).

- It evaluates to `b >= 0` if `ptr` is a pointer to or within a currently allocated block of memory and there are `b` bytes to the end of this block.

Examples (assuming ***`sizeof(int) == 4`****):*

```
int *p = malloc(sizeof(int) * 2);
//@assert \bytes_to_end(p)    == 8;
//@assert \bytes_to_end(p+1)  == 4;
//@assert \bytes_to_end(p+2)  == 0;
//@assert \bytes_to_end(p+3)  == -1;
//@assert \bytes_to_end(NULL) == -1;
int i;
int *q = &i;
//@assert \bytes_to_end(q)    == -1;
```

- `\bytes_to_first_nul(ptr)`:

  It takes as input a pointer `ptr` (of generic type `void*`) and returns a `size_t`.

  - It evaluates to `b < SIZE_MAX` if the first occurrence of a `0x00`-byte (`NUL`) is `b` bytes after where `ptr` points to.
  - It evaluates to `SIZE_MAX` if there are no `0x00`-bytes from `ptr` to the end of memory.

  Examples:

  ```
  //@assert \bytes_to_first_nul("")          == 0;
  //@assert \bytes_to_first_nul("hi")        == 2;
  //@assert \bytes_to_first_nul("hi\0there") == 2;
  ```

- `\is_string(ptr)`:

  It takes as input a pointer `ptr` (of generic type `void*`) and returns a `bool`.

  - It evaluates to `true` if `ptr != NULL` and either of the following conditions is satisfied:
    - `ptr` is a string literal or a pointer to a character within a string literal;
    - `ptr` is a pointer to or within a `NUL`-terminated heap-allocated string. This is equivalent to `\bytes_to_first_nul(ptr) < \bytes_to_end(ptr)`;
    - `ptr` is a pointer to or within a `NUL`-terminated stack-allocated string.

    All cases guarantee that the memory pointed to by `ptr` is `NUL`-terminated, i.e. that `\bytes_to_first_nul(ptr) < SIZE_MAX`.
  - It evaluates to `false` otherwise.

  Examples:

  ```
  //@assert \is_string(NULL)  == false;
  //@assert \is_string("hi")  == true;
  char *s1 = "hi";
  //@assert \is_string(s1))   == true;
  //@assert \is_string(s1+1)) == true;
  //@assert \is_string(s1+2)) == true;
  //@assert \is_string(s1+3)) == false;
  ```

```
char *s2 = calloc(2, sizeof(char));
//@assert \is_string(s2)   == true;
//@assert \is_string(s2+1) == true;
//@assert \is_string(s2+2) == false;
char[] s3 = { 'h', 'i' };
//@assert \is_string(s3)   == false;
char[] s4 = { 'h', 'i', '\0' };
//@assert \is_string(s4)   == true;
//@assert \is_string(s4+1) == true;
//@assert \is_string(s4+2) == true;
//@assert \is_string(s4+3) == false;
```

- `\no_overlap(ptr1, len1, ptr2, len2)`:

  It takes two non-`NULL` pointers `ptr1` and `ptr2` (both of type **`void`**`*`) and two numbers `len1` and `len2` of type `size_t` and returns a **`bool`**.

  - It evaluates to `true` if the `len1` bytes starting at `ptr1` are disjoint from the `len2` bytes starting at `ptr2`, i.e., if these blocks do not share any memory cells.
  - It evaluates to `false` otherwise.

  Examples:

```
char *s1 = "hi there";
//@assert \no_overlap(s1, 2, s1, 3)   == false;
//@assert \no_overlap(s1, 0, s1, 3)   == true;
//@assert \no_overlap(s1, 2, s1+1, 2) == false;
//@assert \no_overlap(s1, 2, s1+4, 2) == true;
char *s2 = "kimchi";
//@assert \no_overlap(s1, 2, s2, 2)   == true;
//@assert \no_overlap(s1, 2, s2+4, 2) == true;
char *s3 = s2+4;
//@assert \no_overlap(s1, 2, s3, 2)   == true;
char *s4 = "";
//@assert \no_overlap(s1+2, 1, s4, 1) == true;
```

1.2 Other Conventions

When giving the prototype of C library functions below, we make some simplifications that bring them closer what was discussed in class. Specifically:

- We write **`bool`** for the type of values that are understood as booleans.
- We omit type qualifiers such as **`const`**.

Follow the documentation link provided for each function to see its official prototype.

2 Standard System Libraries

2.1 `<stdlib.h>`

- `malloc`: allocates a memory block (documentation)

```
void *malloc(size_t size)
/*@ensures (\allocated(\result) && \bytes_to_end(\result) == size)
        || \result == NULL); @*/ ;
```

- `calloc`: allocates and zero-initializes a memory block (documentation)

```
void *calloc(size_t num, size_t size)
/*@requires num <= SIZE_MAX/size; @*/
/*@ensures (\allocated(\result) && \bytes_to_end(\result) == num*size)
        || \result == NULL); @*/ ;
```

- `free`: deallocates a memory block (documentation)

```
void free(void *ptr)
/*@requires \allocated(ptr) || ptr == NULL; @*/
/*@ensures !\allocated(ptr); @*/ ;
```

See documentation for a full list of the functions exported by `<stdlib.h>`.

2.2 `<string.h>`

- `strlen`: returns the length of a string (documentation)

```
size_t strlen(char *str)
/*@requires \is_string(str); @*/
/*@ensures \result == \bytes_to_first_nul(str); @*/ ;
```

- `strcmp`: compares two strings (documentation)

```
int strcmp(char *str1, char *str2)
/*@requires \is_string(str1) && \is_string(str2); @*/ ;
```

- `strncmp`: compares prefix characters of two strings (documentation)

```
int strncmp(char *str1, char *str2,  size_t num)
/*@requires \is_string(str1) && \is_string(str2); @*/ ;
```

- `strcpy`: copies a string (documentation)

```
char *strcpy(char *destination, char *source)
/*@requires \is_string(source); @*/
/*@requires destination != NULL; @*/
/*@requires \bytes_to_end(destination) >= strlen(source) + 1; @*/
/*@requires \no_overlap(source, strlen(source), destination, strlen(source)); @*/
/*@ensures \is_string(\result); @*/
/*@ensures strcmp(\result, source) == 0; @*/
/*@ensures \result == destination; @*/ ;
```

- `strncpy`: copies characters from a string (documentation)

```
char *strncpy(char *destination, char *source, size_t num)
/*@requires \is_string(source); @*/
/*@requires destination != NULL; @*/
/*@requires \bytes_to_end(destination) >= num; @*/
/*@requires \no_overlap(source, num, destination, num); @*/
/*@ensures strncmp(\result, source, num) == 0; @*/
/*@ensures \result == destination; @*/ ;
```

Note that `\result` is **not** guaranteed to be `NUL`-terminated.

- `strcat`: concatenates two strings (documentation)

```
char *strcat(char *destination, char *source)
/*@requires \is_string(source) && \is_string(destination); @*/
/*@requires \bytes_to_end(destination) >= strlen(destination) + strlen(source) + 1; @*/
/*@requires \no_overlap(source, strlen(source), destination, strlen(destination)); @*/
/*@ensures \is_string(\result); @*/
/*@ensures strcmp(\result + strlen(\result) - strlen(source), source) == 0; @*/
/*@ensures \result == destination; @*/ ;
```

- `strncat`: appends characters from a string (documentation)

```
char *strncat(char *destination, char *source, size_t num);
/*@requires \is_string(source) && \is_string(destination); @*/
/*@requires \bytes_to_end(destination) >= strlen(destination) + min(num, strlen(source)
/*@requires \no_overlap(source, strlen(source), destination, strlen(destination)); @*/
/*@ensures \is_string(\result); @*/
/*@ensures strcmp(\result + strlen(\result) - min(num, strlen(source)), source) <= 0; @
/*@ensures strlen(destination) >= strlen(source); @*/
/*@ensures \result == destination; @*/ ;
```

See documentation for a full list of the functions exported by **`<string.h>`**.

2.3 `<stdio.h>`

- `printf`: prints a string, filling placeholders with values (documentation)

```
int printf(char *format, ...)
/*@requires \is_string(format); @*/ ;
```

- `fopen`: opens a file (documentation)

```
FILE *fopen(char *filename, char *mode)
/*@requires \is_string(filename) && \is_string(mode); @*/ ;
```

- `fclose`: closes a file (documentation)

```
int fclose(FILE *stream)
/*@requires FILE != NULL; @*/ ;
```

See documentation for a full list of the functions exported by `<stdio.h>`.

2.4 <assert.h>

- **assert**: evaluates an assertion (documentation)

```
void assert(bool expression);
```

This is the only function exported by <assert.h>.

2.5 Other Standard C Libraries

15-122 routinely uses a few additional standard C libraries. These libraries do not define functions. Therefore, no C0-style annotations are provided for them. We list them for completeness.

- <limits.h>

  The standard C library <limits.h> defines macro constants such as INT_MIN and INT_MAX. It does not define any function. See the documentation for details.

- <stdbool.h>

  The standard C library <stdbool.h> defines the type **bool** and the macro constants true and false. It does not define any function. See the documentation for details.

- <stdint.h>

  The standard C library <stdint.h> defines types such as int8_t and uint32_t, as well as macro constants relevant for them. It does not define any function. See the documentation for details.

There are several other standard C libraries that are not used at all or in any direct or prominent way in 15-122. A full list can be found here.

3 15-122 Custom Libraries

3.1 "xalloc.h"

- xmalloc: allocates a memory block

```
void *xmalloc(size_t size)
/*@ensures \result != NULL; @*/
/*@ensures \allocated(\result) && \bytes_to_end(\result) == size @*/ ;
```

- xcalloc: allocates and zero-initializes a memory block

```
void *xcalloc(size_t num, size_t size)
/*@requires num <= SIZE_MAX/size; @*/
/*@ensures \result != NULL; @*/
/*@ensures \allocated(\result) && \bytes_to_end(\result) == size @*/ ;
```

3.2 "contracts.h"

`contracts.h` only defines macro functions. We include a description for each of them.

- `REQUIRES`: emulates `//@requires` when `DEBUG` is defined

  ```
  void REQUIRES(bool expression);
  ```

- `ENSURES`: emulates `//@ensures` when `DEBUG` is defined

  ```
  void ENSURES(bool expression);
  ```

- `ASSERT`: emulates `//@assert` when `DEBUG` is defined

  ```
  void ASSERT(bool expression);
  ```

- `IF_DEBUG`: execute a statement when `DEBUG` is defined

  ```
  void IF_DEBUG(statement);
  ```

  Examples:

  ```
  IF_DEBUG(printf("Entering function dfs with "));
  IF_DEBUG(printf("source=%u, target=%u and graph=", source, target));
  IF_DEBUG(print_graph(G));
  ```