Libraries

Reusing Code

- All but the simplest programs reuse code already written
  - system code
    - #use <**conio**>

      C0's input/output functions
  - simple code you wrote in the past
    - **pixel-int.c0**

      Encoding and manipulation of pixels as ints
  - complex code somebody else wrote
    - **pixel.o0**

      Encoding and manipulation of pixels
    - **jquery.js**

      Javascript utilities to build dynamic web pages
- Why?
  - Writing correct code is hard and time-consuming!
- These are **libraries**
  - They separate out code used across many applications from the applications themselves

Abstraction

- Libraries promote **abstraction**
  - Focus on **what** the library code does
    - e.g., print an integer to terminal using printint
  - not on **how** it does it
    - the many minute steps to turn a integer into terminal output
- Abstraction has lots of **benefits**
  - Hide inessential details
    - writing code is hard enough without also having to know how printint works
  - Make code more manageable
    - if we find a bug in printint, there is a single place where to fix it
  - Allow for transparent improvements
    - if we find a better way of printing, update the library not the applications

Computer science is all about abstraction!

What's a Library Anyway?

What

1. The **interface**
   - Lists the functionalities the library exports and how to use them

```
void printint(int i);
```

Everything we need to use this functionality:
- name of the function
- number and type of arguments
- output type
- contracts

How

2. The **implementation**
   - The code that implements them

```
void printint(int i) {
  …
}
```

Complex low-level code

What

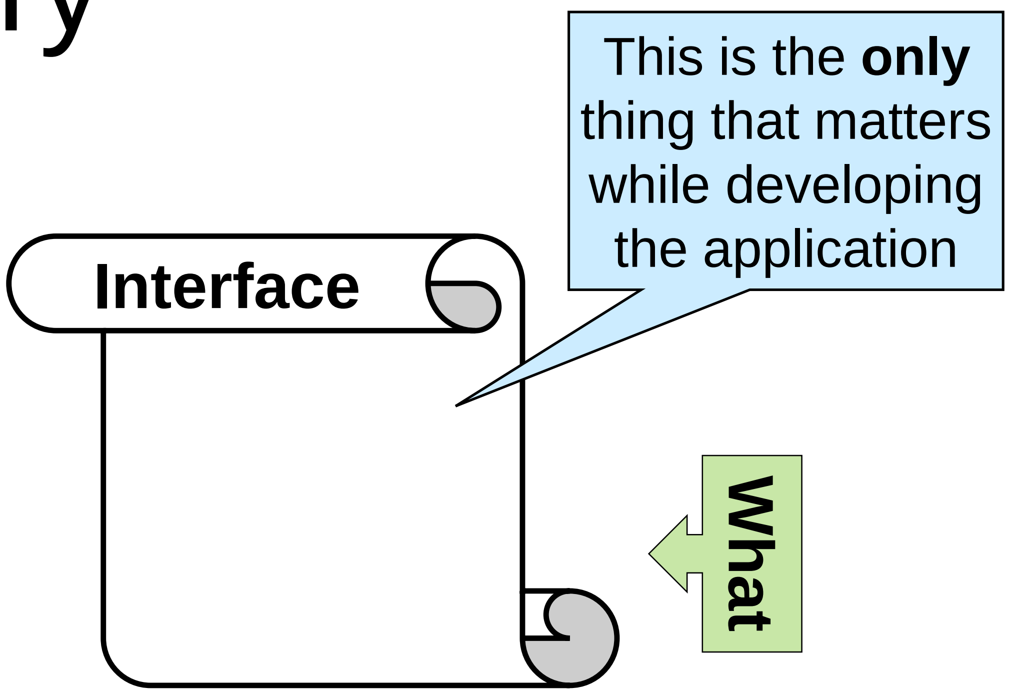
3. The **documentation**
   - The explanation of what they do

"print i to standard output"

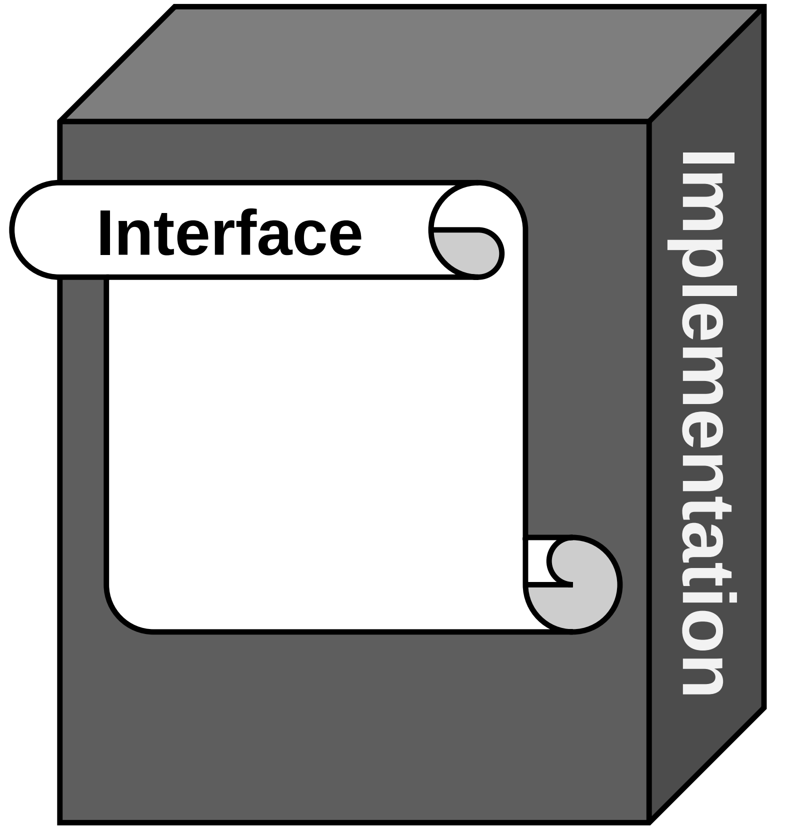
Human readable,
often in a web page or thick manual

Using a Library

- When **writing** application code, we only use the functionalities listed in the interface
  - *No reliance on implementation*



- When **compiling** the application, we involve an implementation of the library
  - Needed for the application to run



- Implementation is a **black box**

Types of Libraries

- System libraries
  - part of the programming language

    #use <**conio**>

    - No need to load any file to use them

Linux Terminal

```
# cc0 -d my-math-application.c0
```

- User-defined libraries
  - written by users or downloaded from the Internet

    pixels.c0

    or **pixels.o0**

    - must be compiled with the application

Also called an **API**

Application **P**rogramming **I**nterface

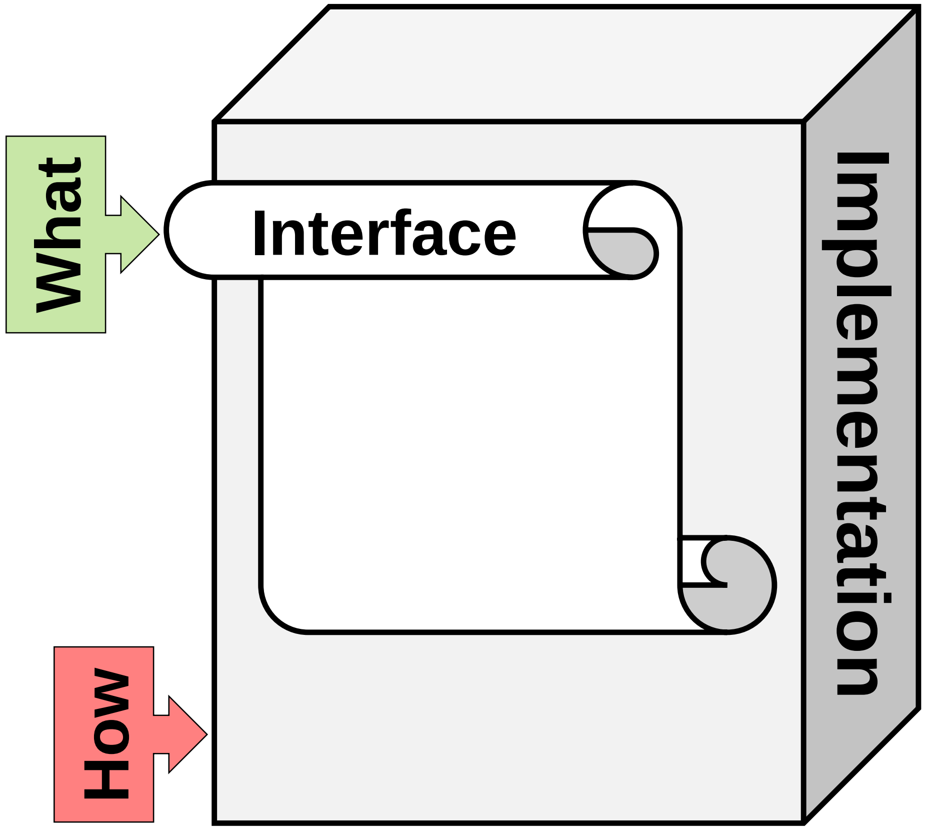
Linux Terminal

```
# cc0 -d pixels.c0 my-image-application.c0
```

Writing a Library

- When writing a library, we need to
  - decide on the interface
  - implement every functionality exported by the interface
    - Fill the black box
  - write lots of documentation

- In this class, we will be writing some of the system libraries that are native in other languages



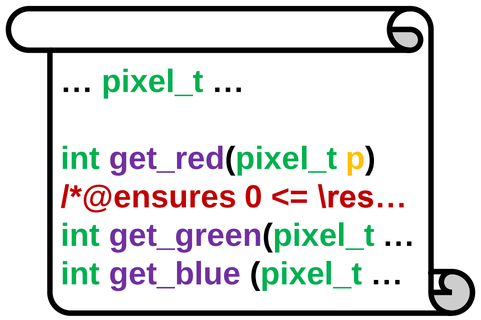
Abstract Data Types

E.g., ***pixels***

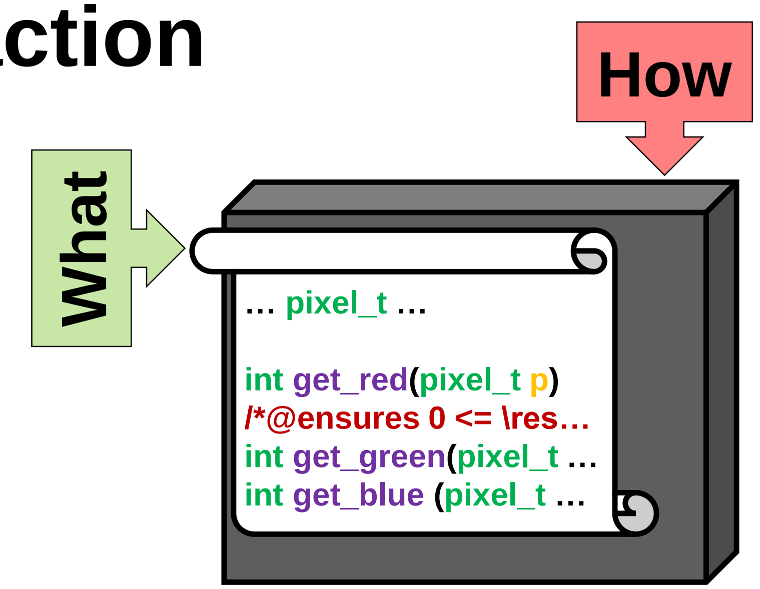
- A library that defines a **new type** and the ways to use it
  - Defines the type pixel_t of pixels
    - ➢ The **only** way we shall refer to pixel in application
  - Defines functions that manipulate pixels

```
int get_red(pixel_t p)
/*@ensures 0 <= \result && \result < 256; @*/ ;
int get_green(pixel_t p) …
int get_blue(pixel_t p) …
pixel_t make_pixel(int red, int green, int blue) …
```

This is the pixel interface



  - ➢ The **only** operations we shall use to manipulate pixels
    - ❑ Except for functions we write using them
- ADT's promote a very strong form of **abstraction**
  - If the client only uses the interface, we can use *any correct implementation* and the application will work the same!



Self-Sorting Arrays

Writing Libraries

- In this course, we will be writing many libraries
- Case study to gain familiarity
  - ➢ and define important concepts

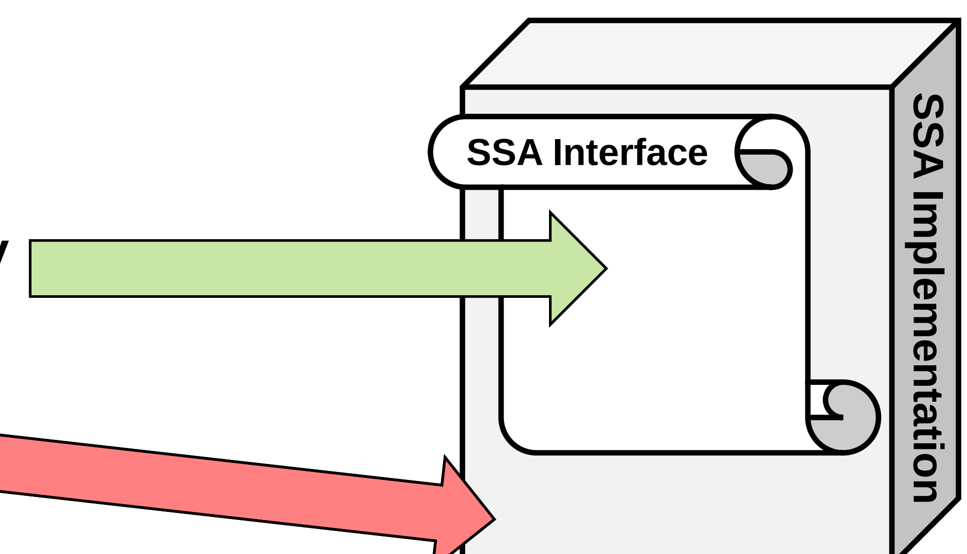
Self-sorting arrays (SSA)

- ○ a toy data structure that works just like arrays of strings but
  - ➢ has a function that reports the length of the SSA
  - ➢ guarantees that its elements are sorted

- What we need to do:

A. Define the interface of the SSA library

B. Implement it



SSA Interface

SSA Interface

// typedef ______ ssa_t;

Interface Contents

1. A **type** for self-sorting arrays

ssa_t

- SSA's are a data structure
- We need a type to refer to them in code
  - define variables that can hold an SSA
  - define functions that manipulate them, …

Convention: types exported by a library end in _t

- We do **not** want the client to learn the details of this type
  - This type is **abstract** for the client: *just a name*
  - We define it as a **pseudo-typedef**

// typedef ______ ssa_t;

  - The implementation will contain the actual definition of ssa_t
    - **Concrete** type of SSA's

A *commented-out* typedef with underscores

Another convention

What

How

Interface Contents

SSA Interface

// typedef ______ ssa_t;

// Operations

2. The **operations** provided by the library to manipulate SSA's
   - What should these be?
     - SSA's are just fancy arrays
     - We will need SSA versions of the standard operations on arrays
       - create a new array
       - read a value from an array index
       - replace the value at an array index
     - Plus *a function that returns the length*
       - not just in contracts, but in regular code

Interface Contents

2. The **operations** provided by the library to manipulate SSA's
   - Creating a new SSA

Number of elements

Newly created SSA

```
ssa_t ssa_new(int size);  // akin to alloc_array(string, size)
```

This is a **function prototype**:
a function definition without a body

Recall that our SSAs contain strings
(we'll learn later how to generalize)

   - Reading the value at an index of an SSA

```
string ssa_get(ssa_t A, int i);   // akin to … A[i] …
```

   - Replacing the value at an index of an SSA

```
void ssa_set(ssa_t A, int i, string x);    // akin to A[i] = x
```

     - unlike regular arrays, this may rearrange the contents of A to keep it sorted
   - Returning the length of an SSA

```
int ssa_len(ssa_t A);      // akin to \length(A) but better
```

     - unlike regular array, this can be used anywhere in code

Interface Contents

SSA Interface

```
// typedef ______ ssa_t;

int ssa_len(ssa_t A);

ssa_t ssa_new(int size) ;

string ssa_get(ssa_t A, int i);

void ssa_set(ssa_t A, int i, string x);
```

3. The **contracts** of each operation
   - ○ The client needs to be able to write safe code
     - ➢ Provide arguments that satisfy the preconditions of each function
     - ➢ Use the functions' postconditions to reason about follow-up code
   - ○ Reading the value at an index of an SSA
     - ➢ Same contracts as native A[i]
       - ▪ /*@requires 0 <= i && i < \length(A); @*/
     - ➢ So:

```
string ssa_get(ssa_t A, int i)   // akin to … A[i] …
/*@requires 0 <= i && i < \length(A); @*/ ;
```

Can this be right?

Interface Contents

SSA Interface

```
// typedef ______ ssa_t;

int ssa_len(ssa_t A);

ssa_t ssa_new(int size) ;

string ssa_get(ssa_t A, int i);

void ssa_set(ssa_t A, int i, string x);
```

3. The **contracts** of each operation

```
string ssa_get(ssa_t A, int i)  // akin to … A[i] …
 /*@requires 0 <= i && i < \length(A); @*/ ;
```

Can this be right?

- \length is defined only for C0 arrays
  - But SSAs are not C0 arrays
    - length can be retrieved
    - elements stay sorted
    - should be manipulated only with operations in the SSA interface
- We can however use ssa_len

```
string ssa_get(ssa_t A, int i)  // akin to … A[i] …
 /*@requires 0 <= i && i < ssa_len(A); @*/ ;
```

Interface Contents

3. The **contracts** of each operation
   - With ssa_len, we can give a meaningful precondition to ssa_get
   - and to ssa_set
   - and a postcondition to ssa_new

SSA Interface

```
// typedef ______ssa_t;

int ssa_len(ssa_t A)
/*@ensures \result >= 0; @*/ ;

ssa_t ssa_new(int size)
/*@requires 0 <= size; @*/
/*@ensures ssa_len(\result) == size; @*/ ;

string ssa_get(ssa_t A, int i)
/*@requires 0 <= i && i < ssa_len(A); @*/ ;

void ssa_set(ssa_t A, int i, string x)
/*@requires 0 <= i && i < ssa_len(A); @*/ ;
```

Interface Contents

- But what kind of type can ssa_t be?
  - An array? ✘
    - No way to get the length of an array in C0
  - An int, bool or char? ✘
    - No way to represent arbitrarily many strings
  - A string? ✘
    - Let's not go there …
  - A struct? ✘
    - Structs cannot be passed as function arguments directly

- Then, ssa_t must be a **pointer**
  - Update the pseudo-typedef to reflect this
  - Disallow NULL as a valid ssa_t
    - Every operation that takes an ssa_t has a NULL-check as a precondition
    - Every operation that returns an ssa_t has a NULL-check as a postcondition

We never use NULL for an empty data structure

SSA Interface

```
// typedef ______* ssa_t;

int ssa_len(ssa_t A)
/*@requires A != NULL;              @*/
/*@ensures \result >= 0;            @*/ ;

ssa_t ssa_new(int size)
/*@requires 0 <= size;              @*/
/*@ensures \result != NULL;         @*/
/*@ensures ssa_len(\result) == size; @*/ ;

string ssa_get(ssa_t A, int i)
/*@requires A != NULL;              @*/
/*@requires 0 <= i && i < ssa_len(A); @*/ ;

void ssa_set(ssa_t A, int i, string x)
/*@requires A != NULL;              @*/
/*@requires 0 <= i && i < ssa_len(A); @*/ ;
```

Interface Contents

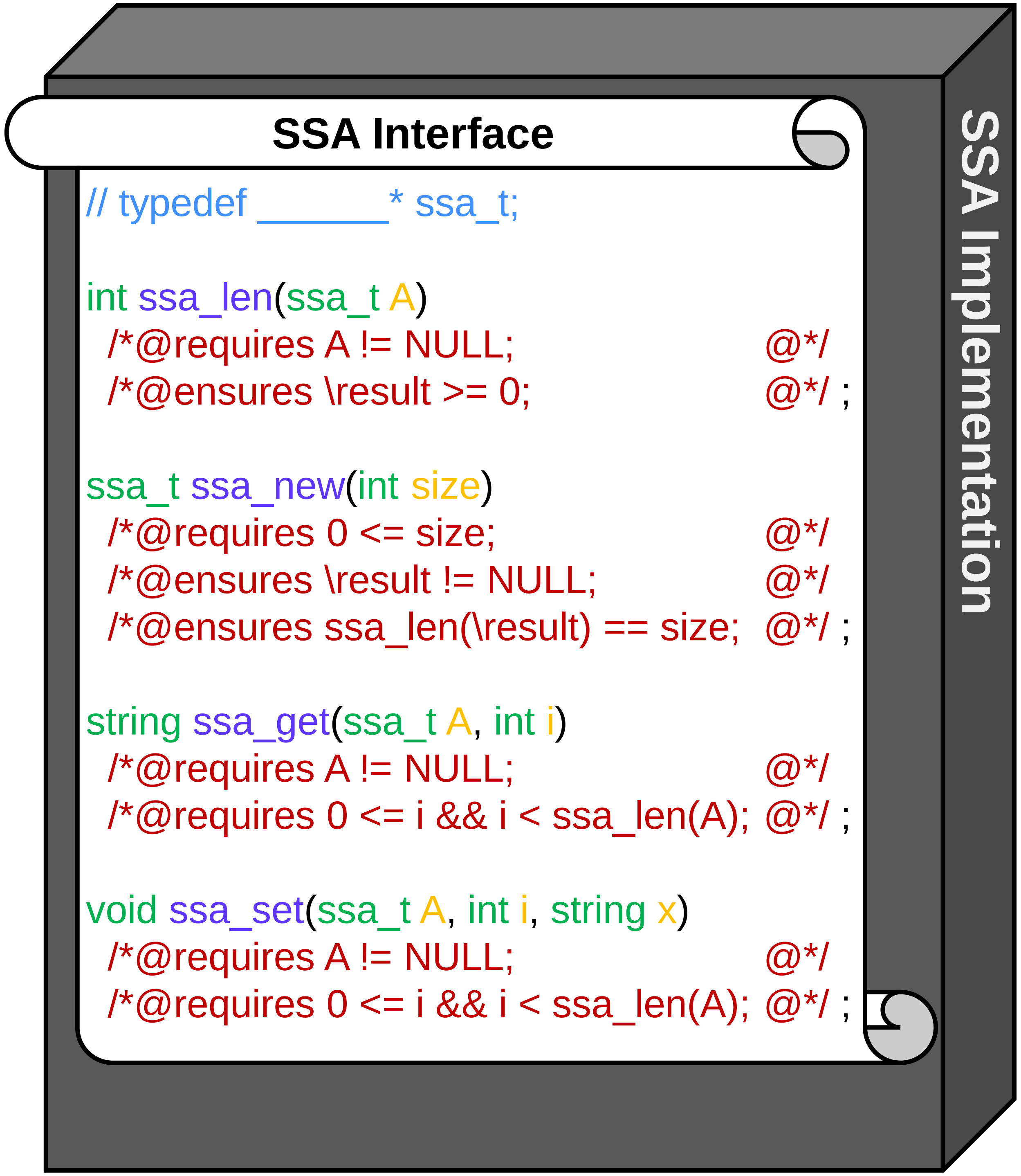
SSA Interface

```
// typedef ______* ssa_t;

int ssa_len(ssa_t A)
  /*@requires A != NULL;                        @*/
  /*@ensures \result >= 0;                      @*/ ;

ssa_t ssa_new(int size)
  /*@requires 0 <= size;                        @*/
  /*@ensures \result != NULL;                   @*/
  /*@ensures ssa_len(\result) == size;          @*/ ;

string ssa_get(ssa_t A, int i)
  /*@requires A != NULL;                        @*/
  /*@requires 0 <= i && i < ssa_len(A);         @*/ ;

void ssa_set(ssa_t A, int i, string x)
  /*@requires A != NULL;                        @*/
  /*@requires 0 <= i && i < ssa_len(A);         @*/ ;
```

What

Client Application

Using a library



- The client only knows **what** the library does
  - the library interface
  - the library documentation

- The client does <u>not</u> know **how** it does it
  - treat the implementation as a black box
    - even if its code is available
      - it may change!

Searching an SSA

- Client code that uses binary search to check if a value is in an SSA
  - This is OK because SSAs are sorted!

```
bool is_in(string x, ssa_t A, int n)
//@requires A != NULL;
//@requires n == ssa_len(A);
{
  int lo = 0;
  int hi = n;
  while (lo < hi)
  //@loop_invariant 0 <= lo && lo <= hi && hi <= n;
  {
    int mid = lo + (hi - lo) / 2;
    //@assert lo <= mid && mid < hi;
    string a = ssa_get(A, mid);
    int cmp = string_compare(a, x);
    if (cmp == 0)  return true;
    if (cmp < 0) {
     lo = mid + 1;
    } else {   //@assert cmp > 0;
      hi = mid;
    }
  }
  return false;
}
```

Precondition of ssa_len and ssa_get

returns <0 if a "less than" x, 0 if equal, >0 otherwise

SSA Interface

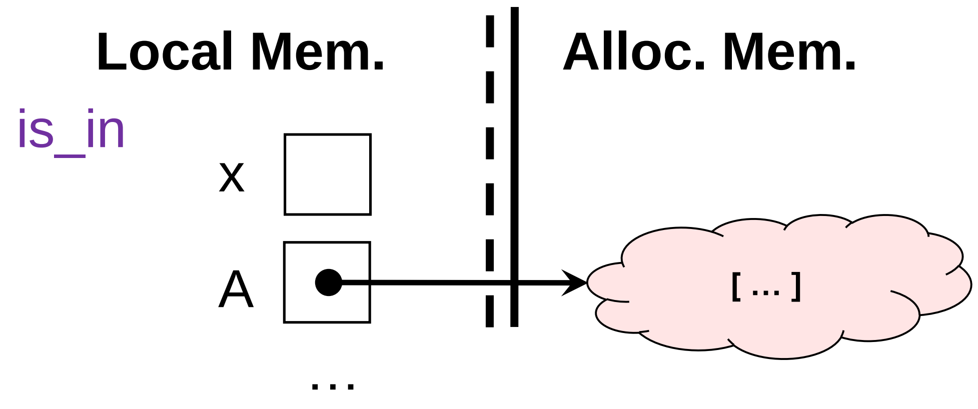
```
// typedef ______* ssa_t;

int ssa_len(ssa_t A)
/*@requires A != NULL;             @*/
/*@ensures \result >= 0;           @*/ ;

ssa_t ssa_new(int size)
/*@requires 0 <= size;             @*/
/*@ensures \result != NULL;        @*/
/*@ensures ssa_len(\result) == size;  @*/ ;

string ssa_get(ssa_t A, int i)
/*@requires A != NULL;             @*/
/*@requires 0 <= i && i < ssa_len(A); @*/ ;

void ssa_set(ssa_t A, int i, string x)
/*@requires A != NULL;             @*/
/*@requires 0 <= i && i < ssa_len(A); @*/ ;
```

- All array operations are replaced with functions from the SSA interface
- **Safety** is supported by loop invariant and assertion
- For **correctness**, we would need to implement SSA versions of gt_seg and lt_seg

Searching an SSA

```
bool is_in(string x, ssa_t A, int n)
//@requires n == ssa_len(A);
{
  int lo = 0;
  int hi = n;
  while (lo < hi)
  //@loop_invariant 0 <= lo && lo <= hi && hi <= n;
  {
    int mid = lo + (hi - lo) / 2;
    //@assert lo <= mid && mid < hi;
    string a = ssa_get(A, mid);
    int cmp = string_compare(a, x);
    if (cmp == 0)  return true;
    if (cmp < 0) {
     lo = mid + 1;
    } else {   //@assert cmp > 0;
      hi = mid;
    }
  }
  return false;
}
```

- Client view of memory



  - The client has no knowledge of how A is represented in memory

SSA Implementation

Implementing SSAs

Now we've got to fill the box

SSA Interface

```
// typedef ______*ssa_t;

int ssa_len(ssa_t A)
  /*@requires A != NULL;                @*/
  /*@ensures \result >= 0;              @*/ ;

ssa_t ssa_new(int size)
  /*@requires 0 <= size;                @*/
  /*@ensures \result != NULL;           @*/
  /*@ensures ssa_len(\result) == size;  @*/ ;

string ssa_get(ssa_t A, int i)
  /*@requires A != NULL;                @*/
  /*@requires 0 <= i && i < ssa_len(A); @*/ ;

void ssa_set(ssa_t A, int i, string x)
  /*@requires A != NULL;                @*/
  /*@requires 0 <= i && i < ssa_len(A); @*/ ;
```

SSA Implementation

- Define the type ssa_t
  - **Concrete** type
- Write code for every function
- Make sure it is safe and correct

How

Concrete Type

SSA Interface

```
// typedef ______* ssa_t;

// …
```

- Store elements in a C0 array, but keep track of the length
  - Package them together in a struct
- Define an internal nickname for it
  - So that the code is succinct and readable
  - It's convenient that it **not** be a pointer
- Define the abstract type exported to the client
  - This is what connects the concrete implementation type with the exported abstract type

SSA Implementation

```
// Implementation-side type
struct ssa_header {
  string[] data;   // sorted
  int length;      // = \length(data)
};
typedef struct ssa_header ssa;

// … rest of implementation

// Client type
typedef ssa* ssa_t;
```

This is the **concrete** implementation type

Internal nickname

Abstract client type

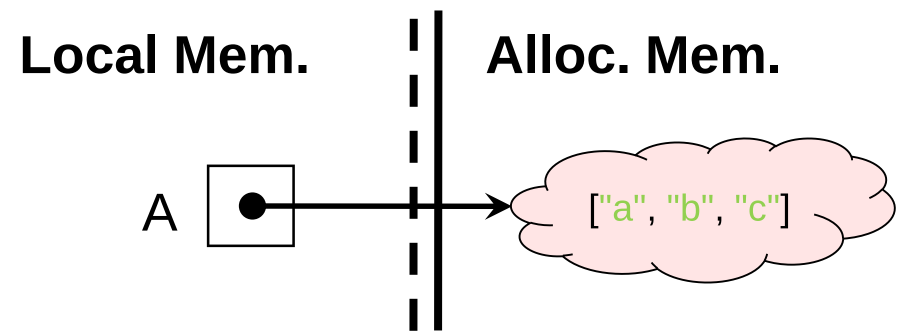
Client vs. Implementation View

An SSA containing "a", "b" and "c"

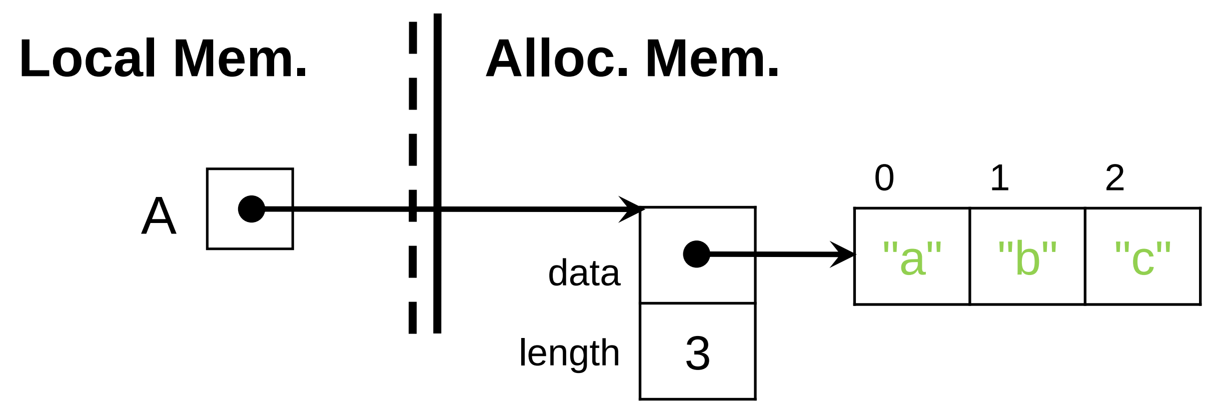
SSA Implementation

```
// Implementation-side type
struct ssa_header {
  string[] data;  // sorted
  int length;      // = \length(data)
};
typedef struct ssa_header ssa;

// … rest of implementation

// Client type
typedef ssa* ssa_t;
```

- Client view



- Implementation view



SSA Interface

```
string ssa_get(ssa_t A, int i)
/*@requires A != NULL;                @*/
/*@requires 0 <= i && i < ssa_len(A); @*/ ;
```

Implementing ssa_get

SSA Implementation

```
struct ssa_header {
  string[] data;  // sorted
  int length;     // = \length(data)
};
typedef struct ssa_header ssa;

string ssa_get(ssa* A, int i)
//@requires A != NULL;
//@requires 0 <= i && i < ssa_len(A);
{
  return A->data[i];
}

// … rest of implementation
```

- Simply return the i-th element of the underlying array

  return A->data[i]

- Is this safe? We need to check
  - A != NULL ✓
    - By 1st precondition
  - 0 <= i ✓
    - By 2nd precondition (first conjunct)
  - i < \length(A->data)
    - We know that i < ssa_len(A)
    - but we don't know how ssa_len(A) and \length(A->data) are related
    - ***Not supported!*** ✗

Let's also write ssa_len

SSA Interface

```
int ssa_len(ssa_t A)
/*@requires A != NULL;              @*/
/*@ensures \result >= 0;            @*/ ;

string ssa_get(ssa_t A, int i)
/*@requires A != NULL;              @*/
/*@requires 0 <= i && i < ssa_len(A); @*/ ;
```

SSA Implementation

```
struct ssa_header {
  string[] data;  // sorted
  int length;     // = \length(data)
};
typedef struct ssa_header ssa;

int ssa_len(ssa* A)
//@requires A != NULL;
//@ensures \result >= 0;
{
  return A->length;
}

string ssa_get(ssa* A, int i)
//@requires A != NULL;
//@requires 0 <= i && i < ssa_len(A);
{
  return A->data[i];
}

// … rest of implementation
```

- Simply return the length field
- Is this safe? We need to check
  - A != NULL ✓
    - By precondition
- Does this help us with ssa_get?
  - No useful postcondition ✗
    - Peeking at the code of ssa_len would be operational reasoning!

Let’s also write ssa_len

SSA Interface

```
int ssa_len(ssa_t A)
/*@requires A != NULL;             @*/
/*@ensures \result >= 0;           @*/ ;

string ssa_get(ssa_t A, int i)
/*@requires A != NULL;             @*/
/*@requires 0 <= i && i < ssa_len(A); @*/ ;
```

SSA Implementation

```
struct ssa_header {
  string[] data;  // sorted
  int length;     // = \length(data)
};
typedef struct ssa_header ssa;

int ssa_len(ssa* A)
//@requires A != NULL;
//@ensures \result >= 0;
//@ensures \result == \length(A->data);
{
  return A->length;
}

string ssa_get(ssa* A, int i)
//@requires A != NULL;
//@requires 0 <= i && i < ssa_len(A);
{
  return A->data[i];
}

// … rest of implementation
```

- Add a *useful* postcondition

  \result == \length(A->data)

- Is this safe? We need to check
  - A != NULL ✓
    - By precondition
- Is ssa_len correct?
  - No relation between A->length and \length(A->data) ✗

Innocent mistake: define ssa_new as

```
ssa* ssa_new(int size) {
  ssa* A = alloc(ssa);
  A->length = size;
  A->data = alloc_array(string, size+1);
  return A;
}
```

Let's also write ssa_len

SSA Interface

```
int ssa_len(ssa_t A)
/*@requires A != NULL;              @*/
/*@ensures \result >= 0;            @*/ ;

string ssa_get(ssa_t A, int i)
/*@requires A != NULL;              @*/
/*@requires 0 <= i && i < ssa_len(A); @*/ ;
```

SSA Implementation

```
struct ssa_header {
  string[] data;  // sorted
  int length;     // = \length(data)
};
typedef struct ssa_header ssa;

int ssa_len(ssa* A)
//@requires A != NULL;
//@requires A->length == \length(A->data);
//@ensures \result >= 0;
//@ensures \result == \length(A->data);
{
  return A->length;
}

string ssa_get(ssa* A, int i)
//@requires A != NULL;
//@requires 0 <= i && i < ssa_len(A);
{
  return A->data[i];
}

// … rest of implementation
```

- Add it as a precondition

  A->length == \length(A->data)

- Is this safe? We need to check
  - A != NULL ✓
    - By precondition
- Is ssa_len correct?
  - A->length == \length(A->data)
    - By new precondition
  - \result == A->length
    - By code
  - \result == \length(A->data) ✓
    - By previous two

Back to ssa_get

SSA Interface

```
int ssa_len(ssa_t A)
/*@requires A != NULL;               @*/
/*@ensures \result >= 0;             @*/ ;

string ssa_get(ssa_t A, int i)
/*@requires A != NULL;               @*/
/*@requires 0 <= i && i < ssa_len(A); @*/ ;
```

SSA Implementation

```
struct ssa_header {
  string[] data;  // sorted
  int length;     // = \length(data)
};
typedef struct ssa_header ssa;

int ssa_len(ssa* A)
//@requires A != NULL;
//@requires A->length == \length(A->data);
//@ensures \result >= 0;
//@ensures \result == \length(A->data);
{
  return A->length;
}

string ssa_get(ssa* A, int i)
//@requires A != NULL;
//@requires 0 <= i && i < ssa_len(A);
{
  return A->data[i];
}

// … rest of implementation
```

- Is the code for ssa_get safe?
  - The new postcondition of ssa_len takes care of the remaining safety check
    - i < \length(A->data) ✓
  - But ssa_len has a new precondition
    - A->length == \length(A->data)
    - we need to have a reason for why it is satisfied
      - but we don't
    - ***Not supported!*** ✗

Back to ssa_get

SSA Interface

```
int ssa_len(ssa_t A)
/*@requires A != NULL;                @*/
/*@ensures \result >= 0;              @*/ ;

string ssa_get(ssa_t A, int i)
/*@requires A != NULL;                @*/
/*@requires 0 <= i && i < ssa_len(A); @*/ ;
```

SSA Implementation

```
struct ssa_header {
  string[] data;  // sorted
  int length;     // = \length(data)
};
typedef struct ssa_header ssa;

int ssa_len(ssa* A)
//@requires A != NULL;
//@requires A->length == \length(A->data);
//@ensures \result >= 0;
//@ensures \result == \length(A->data);
{
  return A->length;
}

string ssa_get(ssa* A, int i)
//@requires A != NULL;
//@requires A->length == \length(A->data);
//@requires 0 <= i && i < ssa_len(A);
{
  return A->data[i];
}

// … rest of implementation
```

- Is the code for ssa_get safe?
  - Add

    A->length == \length(A->data)

    as a precondition to ssa_get to support the safety of ssa_len
    - A->length == \length(A->data)
      - By new precondition ✓

Representation Invariants

Where are we?

SSA Interface

```
int ssa_len(ssa_t A)
/*@requires A != NULL;              @*/
/*@ensures \result >= 0;            @*/ ;

string ssa_get(ssa_t A, int i)
/*@requires A != NULL;              @*/
/*@requires 0 <= i && i < ssa_len(A); @*/ ;
```

SSA Implementation

```
struct ssa_header {
  string[] data;  // sorted
  int length;     // = \length(data)
};
typedef struct ssa_header ssa;

int ssa_len(ssa* A)
//@requires A != NULL;
//@requires A->length == \length(A->data);
//@ensures \result >= 0;
//@ensures \result == \length(A->data);
{
  return A->length;
}

string ssa_get(ssa* A, int i)
//@requires A != NULL;
//@requires A->length == \length(A->data);
//@requires 0 <= i && i < ssa_len(A);
{
  return A->data[i];
}

// … rest of implementation
```

- All our code is safe
- Both functions have preconditions

  A != NULL
  A->length == \length(A->data)

  - ssa_set will need them too
  - and ssa_new will have them as postconditions

Where are we?

SSA Interface

```
int ssa_len(ssa_t A)
/*@requires A != NULL;              @*/
/*@ensures \result >= 0;            @*/ ;

string ssa_get(ssa_t A, int i)
/*@requires A != NULL;              @*/
/*@requires 0 <= i && i < ssa_len(A); @*/ ;
```

SSA Implementation

```
struct ssa_header {
  string[] data;  // sorted
  int length;     // = \length(data)
};
typedef struct ssa_header ssa;

int ssa_len(ssa* A)
//@requires A != NULL;
//@requires A->length == \length(A->data);
//@ensures \result >= 0;
//@ensures \result == \length(A->data);
{
  return A->length;
}

string ssa_get(ssa* A, int i)
//@requires A != NULL;
//@requires A->length == \length(A->data);
//@requires 0 <= i && i < ssa_len(A);
{
  return A->data[i];
}

// … rest of implementation
```

- They are fundamental properties an ssa* must obey to be the representation of a valid SSA
  - NULL is not a valid SSA
  - The length field must be equal to the length of the array field data
- These are **invariants** of our representation:
  - Preconditions of every library function that takes an SSA as a parameter
  - Postcondition of every library function that returns or modifies an SSA

Representation Invariants

SSA Interface

```
int ssa_len(ssa_t A)
/*@requires A != NULL;              @*/
/*@ensures \result >= 0;            @*/ ;

string ssa_get(ssa_t A, int i)
/*@requires A != NULL;              @*/
/*@requires 0 <= i && i < ssa_len(A); @*/ ;
```

SSA Implementation

```
struct ssa_header {
  string[] data;  // sorted
  int length;     // = \length(data)
};
typedef struct ssa_header ssa;

int ssa_len(ssa* A)
//@requires A != NULL;
//@requires A->length == \length(A->data);
//@requires is_sorted(A->data, 0, A->length);
//@ensures \result >= 0;
//@ensures \result == \length(A->data);
{
  return A->length;
}

string ssa_get(ssa* A, int i)
//@requires A != NULL;
//@requires A->length == \length(A->data);
//@requires is_sorted(A->data, 0, A->length);
//@requires 0 <= i && i < ssa_len(A);
{
  return A->data[i];
}
```

- **Representation invariants**
  - ➢ Preconditions of every library function that takes an SSA as a parameter
  - ➢ Postcondition of every library function that returns or modifies an SSA
  - ○ Also called **data structure invariants**

- Do ssa* have other representation invariants?
  - ○ Yes! A->data should be sorted

    is_sorted(A->data, 0, A->length)

Representation Invariants

SSA Interface

```
int ssa_len(ssa_t A)
/*@requires A != NULL;              @*/
/*@ensures \result >= 0;            @*/ ;

string ssa_get(ssa_t A, int i)
/*@requires A != NULL;              @*/
/*@requires 0 <= i && i < ssa_len(A); @*/ ;
```

SSA Implementation

```
struct ssa_header {
  string[] data;  // sorted
  int length;     // = \length(data)
};
typedef struct ssa_header ssa;

bool is_ssa (ssa* A) { … }

int ssa_len(ssa* A)
//@requires is_ssa(A);
//@ensures \result >= 0;
//@ensures \result == \length(A->data);
{
  return A->length;
}

string ssa_get(ssa* A, int i)
//@requires is_ssa(A);
//@requires 0 <= i && i < ssa_len(A);
{
  return A->data[i];
}

// … rest of implementation
```

- Factor them out into a single function that checks that they are satisfied

  is_ssa

  - **Representation invariant function**

Convention: name of representation invariant functions start with is_

Representation Invariants

SSA Interface

```
ssa_t ssa_new(int size)
/*@requires 0 <= size;                    @*/
/*@ensures \result != NULL;               @*/
/*@ensures ssa_len(\result) == size;  @*/ ;

void ssa_set(ssa_t A, int i, string x)
/*@requires A != NULL;                    @*/
/*@requires 0 <= i && i < ssa_len(A); @*/ ;
```

SSA Implementation

```
struct ssa_header {
  string[] data;  // sorted
  int length;     // = \length(data)
};
typedef struct ssa_header ssa;

bool is_ssa (ssa* A) { … }

ssa* ssa_new(int size)
//@requires size >= 0;
//@ensures is_ssa(\result);
//@ensures ssa_len(\result) == size;
{
  ssa* A = alloc(ssa);
  A->data = alloc_array(string, size);
  A->length = size;
return A;
}

void ssa_set(ssa* A, int i , string x)
//@requires is_ssa(A);
//@requires 0 <= i && i < ssa_len(A);
//@ensures is_ssa(A);
{ /* left as exercise */ }
```

- The remaining functions
  - Precondition of every ssa* parameter
  - Postcondition of
    - every returned ssa*
    - every modified ssa* parameter

Defining the internal type ssa **not** to be a pointer allows simpler allocations

The representation Invariant Function

SSA Implementation

```
struct ssa_header {
  string[] data;  // sorted
  int length;     // = \length(data)
};
typedef struct ssa_header ssa;

// Representation invariant
bool is_ssa (ssa* A) {
  return A != NULL
      && A->length == \length(A->data)
      && is_sorted(A->data, 0, A->length);
}

// … rest of implementation
```

Let's write it!

- 1^{st} attempt: simply copy the contracts it stands for
  - **Problem:** \length can only be used in contracts

✘

The representation Invariant Function

```
struct ssa_header {
  string[] data;  // sorted
  int length;     // = \length(data)
};
typedef struct ssa_header ssa;

// Representation invariant
bool is_ssa (ssa* A)
//@requires A->length == \length(A->data);
{
  return A != NULL
      && is_sorted(A->data, 0, A->length);
}

// … rest of implementation
```

SSA Implementation

Let's write it!

- 2nd attempt: move that part in the precondition of is_ssa
  - **Problem:** this is unsafe!
    - A may be NULL
    - NULL checked only *after* the precondition

✘

The representation Invariant Function

SSA Implementation

```
struct ssa_header {
  string[] data;  // sorted
  int length;     // = \length(data)
};
typedef struct ssa_header ssa;

// Representation invariant
bool is_array_expected_length(string[] A, int len)  {
   //@assert \length(A) == len;
  return true;
}

bool is_ssa (ssa* A) {
  return A != NULL
       && is_array_expected_length(A->data, A->length)
       && is_sorted(A->data, 0, A->length);
}

// … rest of implementation
```

Let's write it!

- 3rd attempt: move it into a helper function

✓

The representation Invariant Function

SSA Implementation

```
struct ssa_header {
  string[] data;  // sorted
  int length;     // = \length(data)
};
typedef struct ssa_header ssa;

// Representation invariant
bool is_ssa (ssa* A) {
  if (A == NULL) return false;
  //@assert A->length == \length(A->data);
  return is_sorted(A->data, 0, A->length);
}

// … rest of implementation
```

Let's write it!

- Alternative 3rd attempt: turn it into an //@assert in is_ssa

✓

Things to Note

- The representation invariant function is_ssa is **<u>NOT</u> part of interface**
  - Clients are allowed to manipulate SSA's **only using the interface** functions
    - If the library is correct, is_ssa(A) will always return true
    - Client must ensure the safety of library calls according to the interface
      - A != NULL only
    - Providing is_ssa to clients would encourage them to bypass the interface
      - use is_ssa to test if hacks are successful
  - The representation invariant function is an implementation device to ensure the safety and correctness of the library code
    - Used while developing the library
      - Every function that takes an SSA A must have //@requires is_ssa(A);
      - Every function that modifies an input SSA A must have //@ensures is_ssa(A);
      - Every function that returns an SSA must have //@ensures is_ssa(\result);

Things to Note

- The contracts in the interface and the implementation are **different**

Interface	Implementation
void ssa_set(ssa_t A, int i, string x) /*@requires A != NULL; @*/ /*@requires 0 <= i && i < ssa_len(A); @*/ ;	void ssa_set(ssa* A, int i , string x) //@requires is_ssa(A); //@requires 0 <= i && i < ssa_len(A); //@ensures is_ssa(A);

- The implementation contracts are more detailed
  - ➢ is_ssa(A) checks A != NULL
  - ○ The implementation contains more information, so it needs to check more things
  - ○ There is no point having //@ensures A != NULL in the interface
    - ➢ ssa_set is called with a *copy* of the address of A
    - ➢ when returning, the original has not changed, even if ssa_set modified its copy of A
      - ❑ If original A was not NULL when calling ssa_set, it will not be NULL when returning from it

Overall Implementation

SSA Implementation

```
// Implementation-side type
struct ssa_header {
  string[] data;  // sorted
  int length;     // = \length(data)
};
typedef struct ssa_header ssa;

// Representation invariant
bool is_sorted(string[] A, int lo, int hi) {
{ /* left as exercise */ }

bool is_ssa(ssa* A) {
  if (A == NULL) return false;
  //@assert A->length == \length(A->data);
  return is_sorted(A->data, 0, A->length);
}

// Implementation of interface functions
int ssa_len(ssa* A)
//@requires is_ssa(A);
//@ensures \result >= 0;
//@ensures \result == \length(A->data);
{
  return A->length;
}

ssa* ssa_new(int size)
//@requires size >= 0;
//@ensures is_ssa(\result);
//@ensures ssa_len(\result) == size;
{
  ssa* A = alloc(ssa);
  A->data = alloc_array(string, size);
  A->length = size;
return A;
}

string ssa_get(ssa* A, int i)
//@requires is_ssa(A);
//@requires 0 <= i && i < ssa_len(A);
{
  return A->data[i];
}

void ssa_set(ssa* A, int i , string x)
//@requires is_ssa(A);
//@requires 0 <= i && i < ssa_len(A);
//@ensures is_ssa(A);
{ /* left as exercise */ }

// Client type
typedef ssa* ssa_t;
```

- By **convention**, we put the interface **after** the implementation in the same file

SSA Interface

```
// typedef ______* ssa_t;

int ssa_len(ssa_t A)
/*@requires A != NULL;            @*/
/*@ensures \result >= 0;          @*/ ;

ssa_t ssa_new(int size)
/*@requires 0 <= size;            @*/
/*@ensures \result != NULL;       @*/
/*@ensures ssa_len(\result) == size;  @*/ ;

string ssa_get(ssa_t A, int i)
/*@requires A != NULL;            @*/
/*@requires 0 <= i && i < ssa_len(A); @*/ ;

void ssa_set(ssa_t A, int i, string x)
/*@requires A != NULL;            @*/
/*@requires 0 <= i && i < ssa_len(A); @*/ ;
```

How

What

Structure of a C0 Library File

```
/************ IMPLEMENTATION ************/
// Implementation-side type
struct ssa_header {
  …
};
typedef struct ssa_header ssa;

// Representation invariant
bool is_ssa(ssa* A) {
  …
}

// Implementation of interface functions
int ssa_len(ssa* A) { … }

…

// Client type
typedef ssa* ssa_t;


/*********** LIBRARY INTERFACE ***********/
// typedef ______* ssa_t;

int ssa_len(ssa_t A)
/*@requires A != NULL;              @*/
/*@ensures \result >= 0;            @*/ ;

…
```

Implementation

Interface

- Implementation
  - Concrete type definition
  - Representation invariant function
  - Implementation of interface functions
  - Client type definition
- Interface
  - Abstract type name
    - Pseudo-typedef
  - Prototype of exported functions

We will revisit this

Compiling a Library in a C0 Application

- Library file contains both implementation and interface
- When compiling,
  library files come **before** application files
  - The application uses library interface types and functions
  - They need to be defined first
    - ➢ this happens in the library