Data Structure Invariants

LAST

Faster sort
- Merge sort
- Quicksort

TODAY

Pointers
Structs
Abstraction barrier
- client/interface/library
- data structure invariants

NEXT

Using libraries

END OF PART 1

Dataypes so far

- Basic (a.k.a. small types) : `int, bool, char`
- `string`
- Arrays *t*`[]`
- `struct` (we will learn about them today)

Thought exercise: multiple return values

- Suppose we want to add up all elements in an array and return the sum along with whether the sum is 42 or not

```
// Return both the sum of all numbers in A and whether 42 occurs
??? sum_and_42(int[] A, int n)
//@requires n == \length(A);
{
  int sum = 0;
  bool has_42 = false;
  for (int i = 0; i < n; i++) {
    sum += A[i];
    if (A[i] == 42) has_42 = true;
  }
}

int main() {
  int[] A = alloc_array(int, 10);
  for (int i = 0; i < 10; i++){
    A[i] = i - 5;
  }
  ??? = sum_and_42(A, 10);
  return 0;
}
```

Pointer syntax

```
int* x = alloc(int);
bool* y = alloc(bool);
```

Create a pointer

```
*x = 7;
*y = true;
```

Assign value to its contents

```
if(*y) {
   printint(*x);
   int z = 3 + *x;
}
```

Use value of its contents

Aliasing

```
int* x = alloc(int);
*x = 3;
int* y = x;
*x = 4;
printint(*y);
```

What is printed?

Using allocated memory to “return” the result

```
void sum_and_42(int[] A, int n, int* sum, bool* has_42)
 //@requires n == \length(A);
 {
   *sum = 0;
   *has_42 = false;
   for (int i = 0; i < n; i++) {
     *sum += A[i];
     if (A[i] == 42) *has_42 = true;
   }
}

int main() {
   int[] A = alloc_array(int, 10);
   for (int i = 0; i < 10; i++){
     A[i] = i - 5;
   }
   int*  sum  = alloc(int);
   bool* fourtytwo = alloc(bool);
   sum_and_42(A, 10, sum, fourtytwo);
   return 0;
}
```

```
??? sum_and_42(int[] A, int n)
//@requires n == \length(A);
{
  int sum = 0;
  bool has_42 = false;
  for (int i = 0; i < n; i++) {
    sum += A[i];
    if (A[i] == 42) has_42 = true;
  }
}

int main() {
  int[] A = alloc_array(int, 10);
  for (int i = 0; i < 10; i++){
    A[i] = i - 5;
  }
  ??? = sum_and_42(A, 10);
  return 0;
}
```

NULL

```
$ coin
--> int* x = alloc(int);
x is 0x1DBA640 (int*)
--> *x;
0 (int)
--> bool* y = alloc(bool);
y is 0x1DBA660 (bool*)
--> *y;
false (bool)
--> int** z = alloc(int*);
z is 0x1DBA680 (int**)
--> *z;
NULL (int*)
```

Pointer safety

```
int* x = alloc(int);
bool* y = alloc(bool);

*x = 7;
*y = true;
```

SAFETY: check x != NULL

SAFETY: check x != NULL

```
if(*y) {
  printint(*x);
  int z = 3 + *x;
}
```

SAFETY: check x != NULL

SAFETY: check x != NULL

*p has implicit precondition that p != NULL

"All those problems, in the end, I didn't want to deal with, and that let me to suggest that the NULL pointer was a possible value of every reference variable; a possible mistake on every use of that reference variable. And perhaps it was a billion dollar mistake.“

–Tony Hoare

http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare
(begins at 27:30)

Using allocated memory to “return” the result

```
void sum_and_42(int[] A, int n, int* sum, bool* has_42)
//@requires n == \length(A);
//@requires sum != NULL && has_42 != NULL;
{
  *sum = 0;
  *has_42 = false;
  for (int i = 0; i < n; i++) {
    *sum += A[i];
    if (A[i] == 42) *has_42 = true;
  }
}

int main() {
  int[] A = alloc_array(int, 10);
  for (int i = 0; i < 10; i++){
    A[i] = i - 5;
  }
  int*  sum  = alloc(int);
  bool* fourtytwo = alloc(bool);
  sum_and_42(A, 10, sum, fourtytwo);
  return 0;
}
```

```
??? sum_and_42(int[] A, int n)
//@requires n == \length(A);
{
  int sum = 0;
  bool has_42 = false;
  for (int i = 0; i < n; i++) {
    sum += A[i];
    if (A[i] == 42) has_42 = true;
  }
}

int main() {
  int[] A = alloc_array(int, 10);
  for (int i = 0; i < 10; i++){
    A[i] = i - 5;
  }
  ??? = sum_and_42(A, 10);
  return 0;
}
```

A new data type: `struct`

```
struct image_header {
   int width;
   int height;
   pixel[] data;
};
```

- Local memory can only hold values that can fit in a word
- Structs must live in allocated memory, accessed though pointers

```c
struct image_header {
  int width;
  int height;
  pixel[] data;
};
```

```c
struct image_header* IMG = alloc(struct image_header);
IMG->width = 320;
IMG->height = 240;
IMG->data = alloc_array(pixel, 320*240);

typedef struct image_header image;
```

The same thing with a shorter type name

```
struct image_header {
  int width;
  int height;
  pixel[] data;
};
```

```
typedef struct image_header image;
```

```
image* IMG = alloc(image);
IMG->width = 320;
IMG->height = 240;
IMG->data = alloc_array(pixel, 320*240);
```

```
struct image_header {
  int width;
  int height;
  pixel[] data;
};
```

```
typedef  struct image_header image;
```

```
image* IMG = alloc(image);
IMG->width = 320;
IMG->height = 240;
IMG->data = alloc_array(pixel,
320*240);
```

Alternatively,

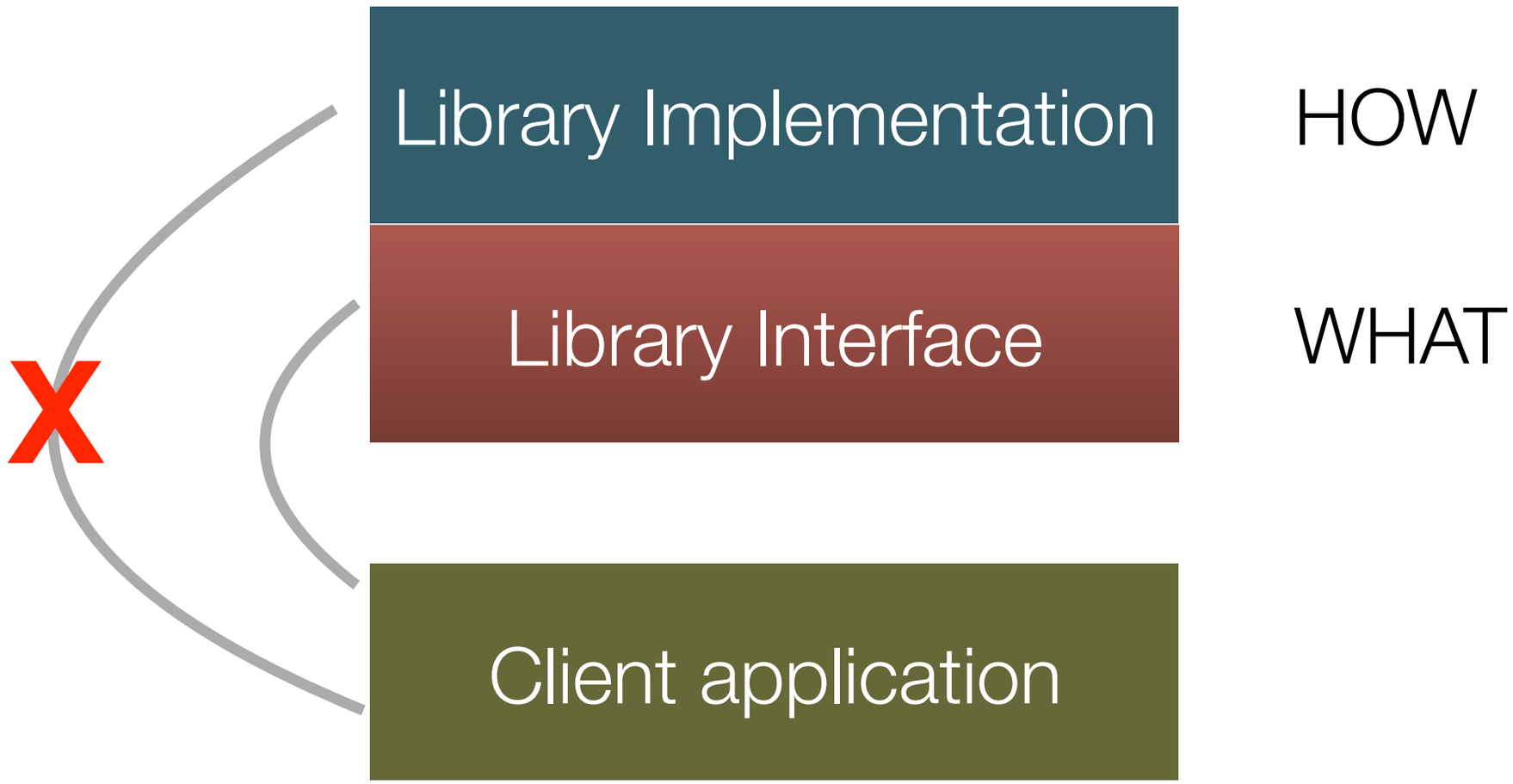
```
image* IMG = alloc(image);
(*IMG).width = 320;

….
```

Not a recommended style

Self-sorting arrays

— works mostly like string arrays with O(log n) search time

Libraries


Library Implementation
HOW
Library Interface
WHAT
X
Client application

Self-sorting arrays: Interface

```
/******************************* 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);
```

Self-sorting arrays: Interface

```
/******************************* 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);
```

What would be a reasonable precondition for `ssa_set`?

Self-sorting arrays: Interface

```
/********************************* 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);
/*@requires 0 <= i && i < ssa_len(A); @*/;
```

Why not use `\length(A)`?

Self-sorting arrays: Interface

```
/******************************* Interface *******************************/
// typedef ______ ssa_t;

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

ssa_t ssa_new(int size)



string ssa_get(ssa_t A, int i)



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

Self-sorting arrays: Interface

```
/******************************* Interface *******************************/
// typedef ______* ssa_t;

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

ssa_t ssa_new(int size)



string ssa_get(ssa_t A, int i)



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

Self-sorting arrays: Interface

```
/******************************* Interface *******************************/
// typedef ______* ssa_t;

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

ssa_t ssa_new(int size)
```

```
string ssa_get(ssa_t A, int i)


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

Self-sorting arrays: Interface

```
/*************************** 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); @*/;
```

Self-sorting arrays: Implementation

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

typedef struct ssa_header ssa;


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

Self-sorting arrays: Implementation

```
struct ssa_header {
  string[] data;
  int length;
};
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];
}
```

SAFE?

Self-sorting arrays: Implementation

```
struct ssa_header {
  string[] data;
  int length;
};
typedef struct ssa_header ssa;
```

```
int ssa_len(ssa* A)
//@requires A!= NULL;
//@requires A->length = \length(A->data);
//@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];
}
```

Self-sorting arrays: Implementation

```
struct ssa_header {
  string[] data;
  int length;
};
typedef struct ssa_header ssa;
```

```
int ssa_len(ssa* A)
//@requires A!= NULL;
//@requires A->length = \length(A->data);
//@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];
}
```

SAFE?

Self-sorting arrays: Implementation

```
struct ssa_header {
  string[] data;
  int length;
};
typedef struct ssa_header ssa;
```

```
int ssa_len(ssa* A)
//@requires A!= NULL;
//@requires A->length = \length(A->data);
//@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);
//@requires A->length == \length(A->data);
{
  return A->data[i];
}
```

Self-sorting arrays: Implementation

```
struct ssa_header {
  string[] data;
  int length;
};
typedef struct ssa_header ssa;
```

```
int ssa_len(ssa* A)
//@requires A! == NULL;
//@requires A -> length = \length(A->data);
//@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);
//@requires A->length == \length(A->data);
{
  return A->data[i];
}
```

Every function that takes `ssa` as a parameter needs the same preconditions!

`ssa* A` is well-formed if

- A is a non-NULL pointer
- length of `A == \length(A->data)`
- A is sorted

Data structure invariants (representation invariants)

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

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

assume this is given

Data structure invariants (representation invariants)

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

- Every function that takes a self-sorting array `A` as input must have `//@requires is_ssa(A);`

- Every function that modifies a self-sorting array `A` must have `//@ensures is_ssa(A);`

- Every function that returns a self-sorting array `A` must have `//@ensures is_ssa(A);`

Self-sorting arrays: Implementation

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


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