Pointers and Structs

Returning Multiple Values

Returning two Values from a Function

- We want to return
  - the sum of all the elements in an array (an int) *and*
  - whether 42 is in the array (a bool)

C0 functions return at most **one** value

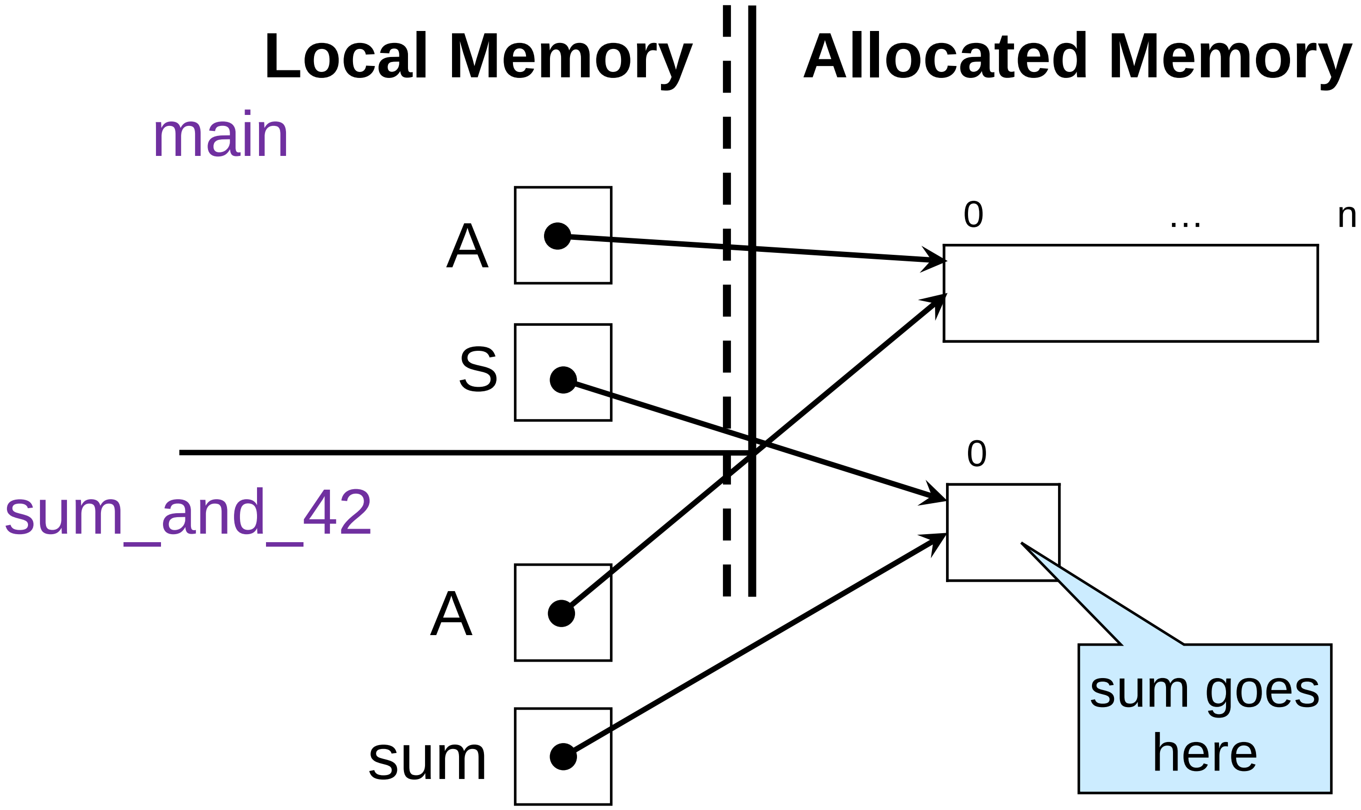
```
??? sum_and_42(int[] A, int n)
//@requires n == \length(A);
{
  int sum = 0;
  bool has_42 = false;
  for (int i = 0; i < n; i++) {
    sum += A[i];
    if (A[i] == 42) has_42 = true;
  }
}
```

```
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;
}
```

- How can we do that?

Returning two Values from a Function

- A C0 function can communicate with its caller
  - by returning a value to it *or*
  - by ***modifying a value in allocated memory*** the caller shared with it
- Idea:
  - main passes a 1-element int array S to sum_and_42
  - sum_and_42 *stores* the sum in S
  - it *returns* whether 42 is in the array as a bool



sum goes here

Returning two Values from a Function

- A C0 function can communicate with its caller
  - o by returning a value to it *or*
  - o by modifying a value in allocated memory the caller shared with it
    - ➢ Idea: caller pass a 1-element int array to store the sum and function return a bool

```
bool sum_and_42(int[] A, int n, int[] sum)
//@requires n == \length(A);
//@requires \length(sum) == 1;
{
  sum[0] = 0;
  bool has_42 = false;
  for (int i = 0; i < n; i++) {
    sum[0] += A[i];
    if (A[i] == 42) has_42 = true;
  }
  return has_42;
}
```

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

  int[] S = alloc_array(int, 1);
  bool b = sum_and_42(A, 10, S);
  return 0;
}
```

Returning two Values from a Function

- Idea
  - caller pass a 1-element int array to store the sum and
  - function return a bool
- This is clunky: invoke the whole array machinery for a single cell in allocated memory!

```
bool sum_and_42(int[] A, int n, int[] sum)
//@requires n == \length(A);
//@requires \length(sum) == 1;
{
  sum[0] = 0;
  bool has_42 = false;
  for (int i = 0; i < n; i++) {
    sum[0] += A[i];
    if (A[i] == 42) has_42 = true;
  }
 return has_42;
}
```

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

  int[] S = alloc_array(int, 1);
  bool b = sum_and_42(A, 10, S);
  return 0;
}
```

Yuck!

Pointers

Memory Cells and Pointers

- C0 provides
  - a way to create individual cells in allocated memory

Creates a new cell in allocated memory

```
alloc(int)
```

Returns the memory address of the new cell

Type of the values we can put in this cell

  - and **pointers** to manipulate them

The memory address of the new cell is stored in p

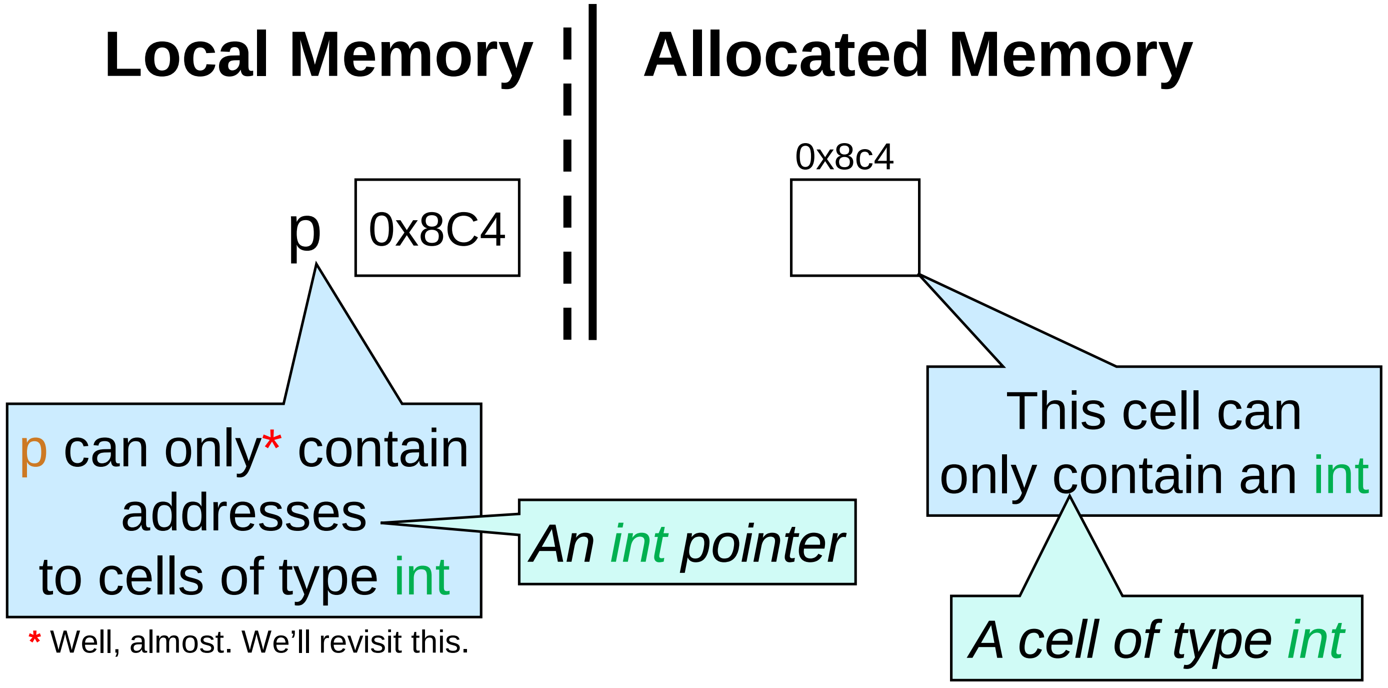
```
int* p = alloc(int)
```

The type of pointers to a cell that can contain an int

Memory Cells and Pointers

`int* p = alloc(int)`

- o creates a new cell
- o the returned address is stored in p



Local Memory | **Allocated Memory**

p can only* contain addresses to cells of type int

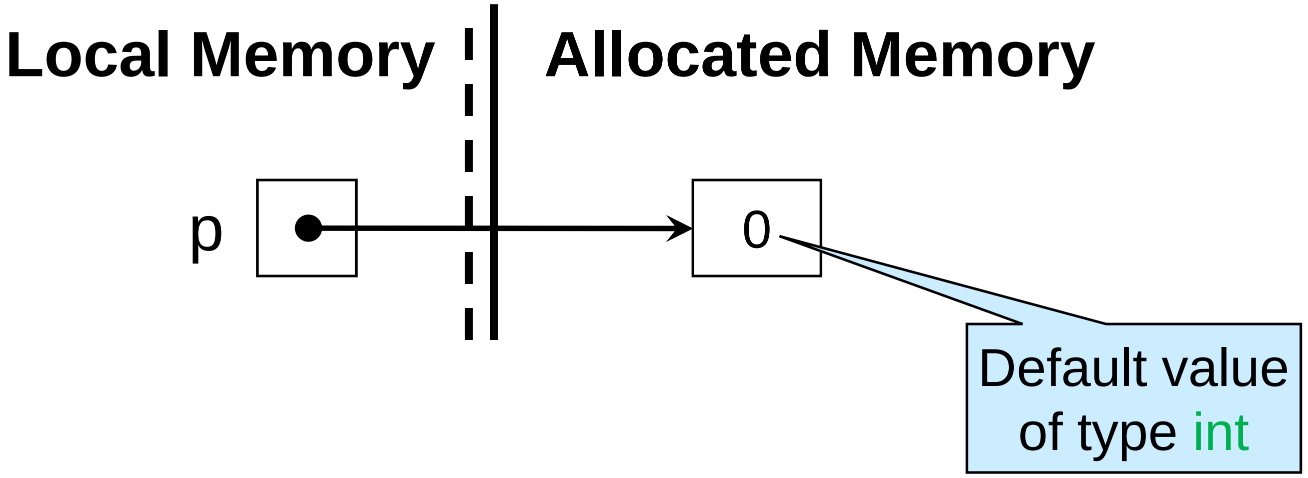
An int pointer

This cell can only contain an int

A cell of type int

* Well, almost. We'll revisit this.

- Similar to arrays
  - o Specific addresses are not visible within the program
    - ➢ We write arrows
  - o Memory cells are initialized to default value for their type



Local Memory | **Allocated Memory**

Default value of type int

Working with Pointers

- We read and write to a memory cell through a pointer to it

*p

Follow the pointer in p and return the value in the cell

…
or write a new value in the cell

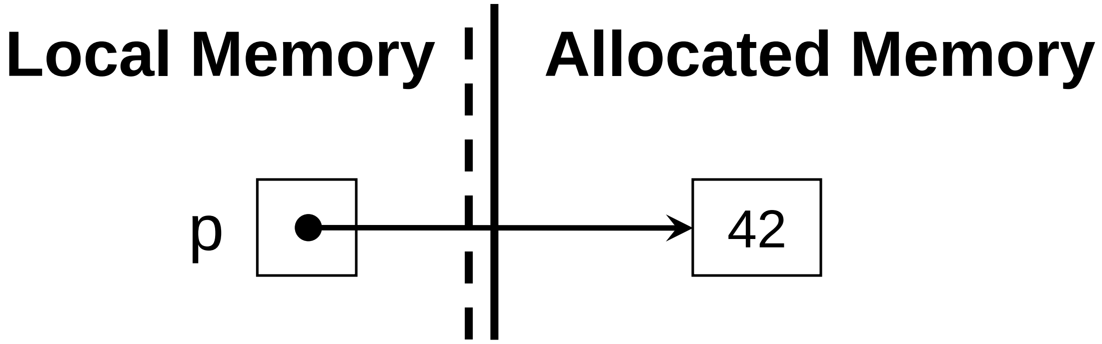
  - This is called **dereferencing** p

```
printint(*p);
*p = 42;
printint(*p);
```

Prints 0

Puts 42 in the cell pointed by p

Prints 42



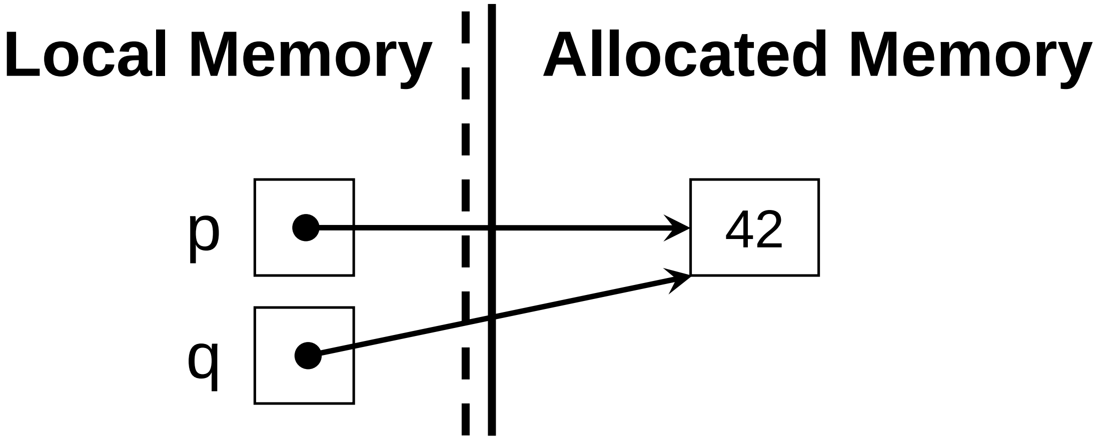
Aliasing

- Pointers are subject to aliasing …

```
int* q = p;
printint(*q);
```

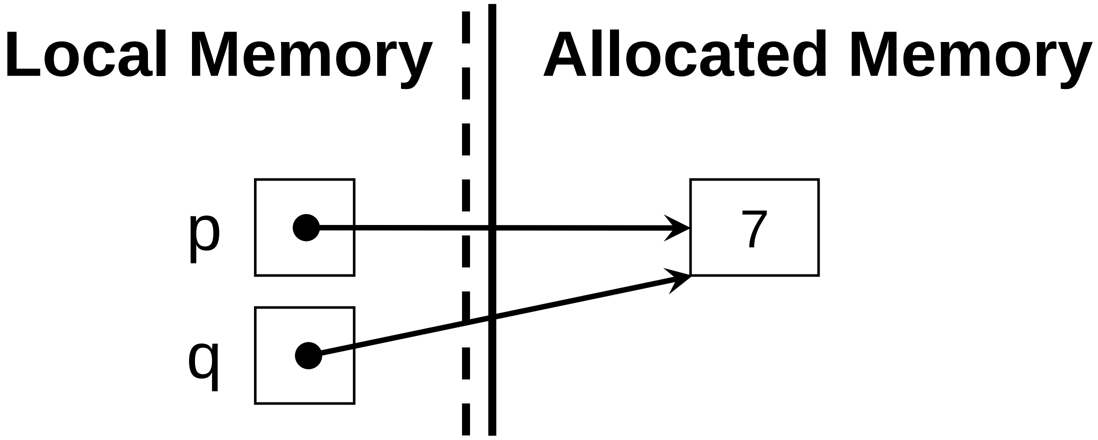
q and p point to the **same** cell

Prints 42



```
*q = 7;
printint(*p);
```

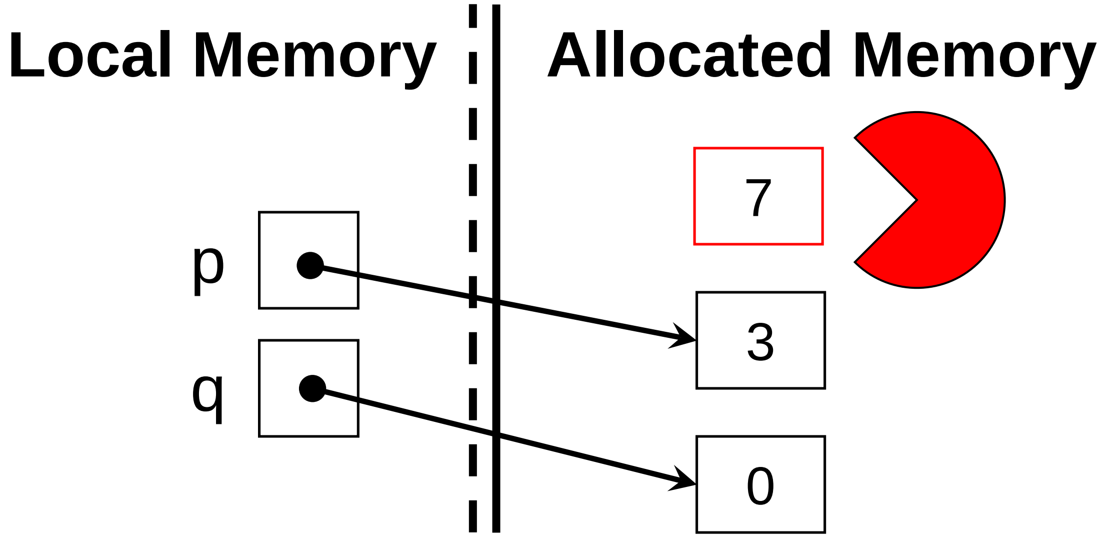
Prints 7



Garbage Collection

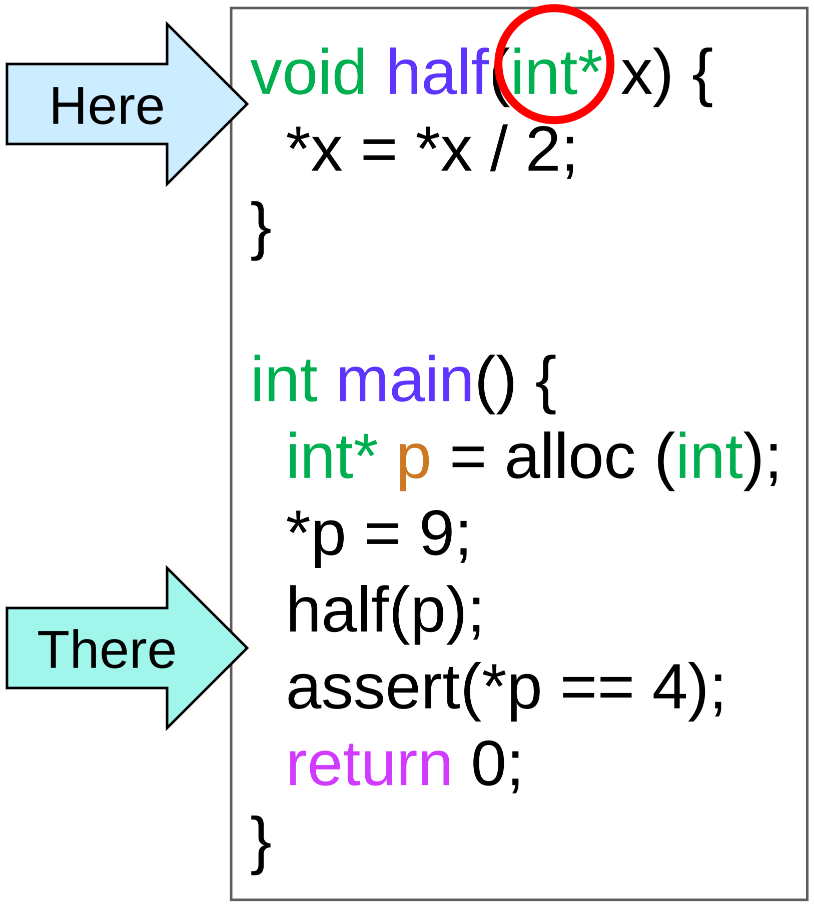
- … and memory cell are subject to garbage collection
  - when there is no way to access them

```
p = alloc(int);
*p = 3;
q = alloc(int);
```



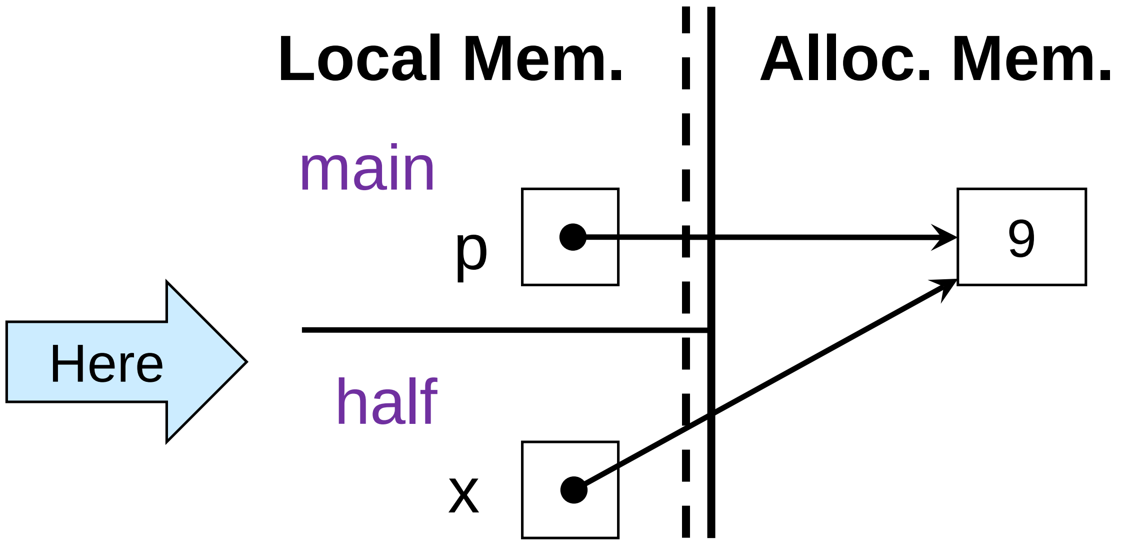
Functions on Pointers

- A function that halves the content of an int cell

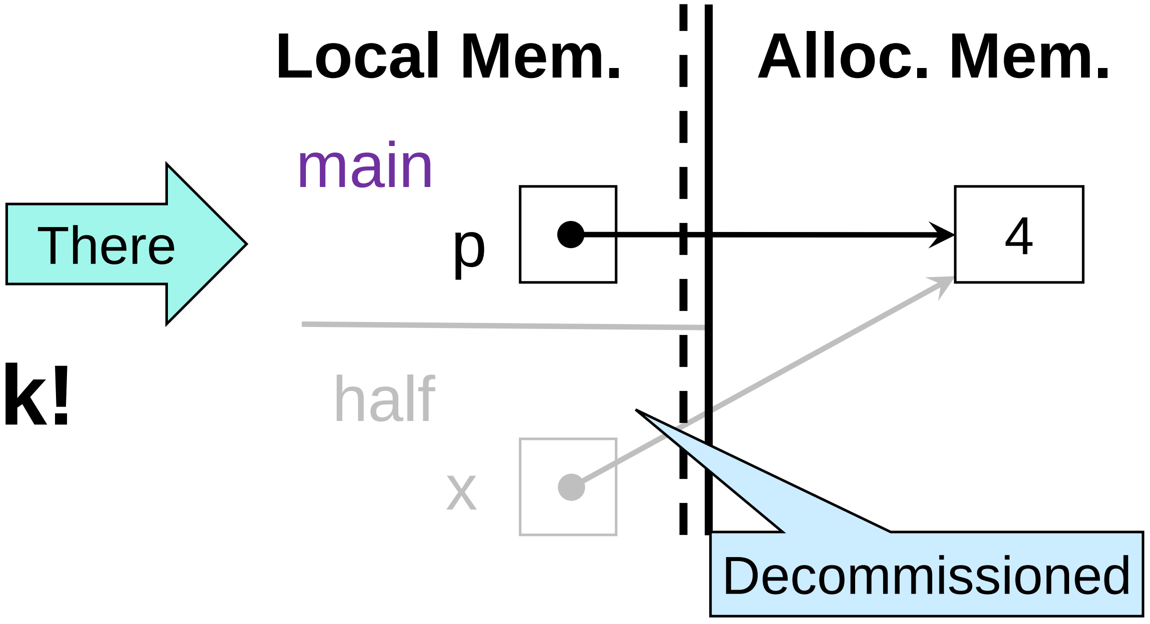


```
void half(int* x) {
  *x = *x / 2;
}

int main() {
  int* p = alloc (int);
  *p = 9;
  half(p);
  assert(*p == 4);
  return 0;
}
```

- half is passed the value of p
  - an *address*



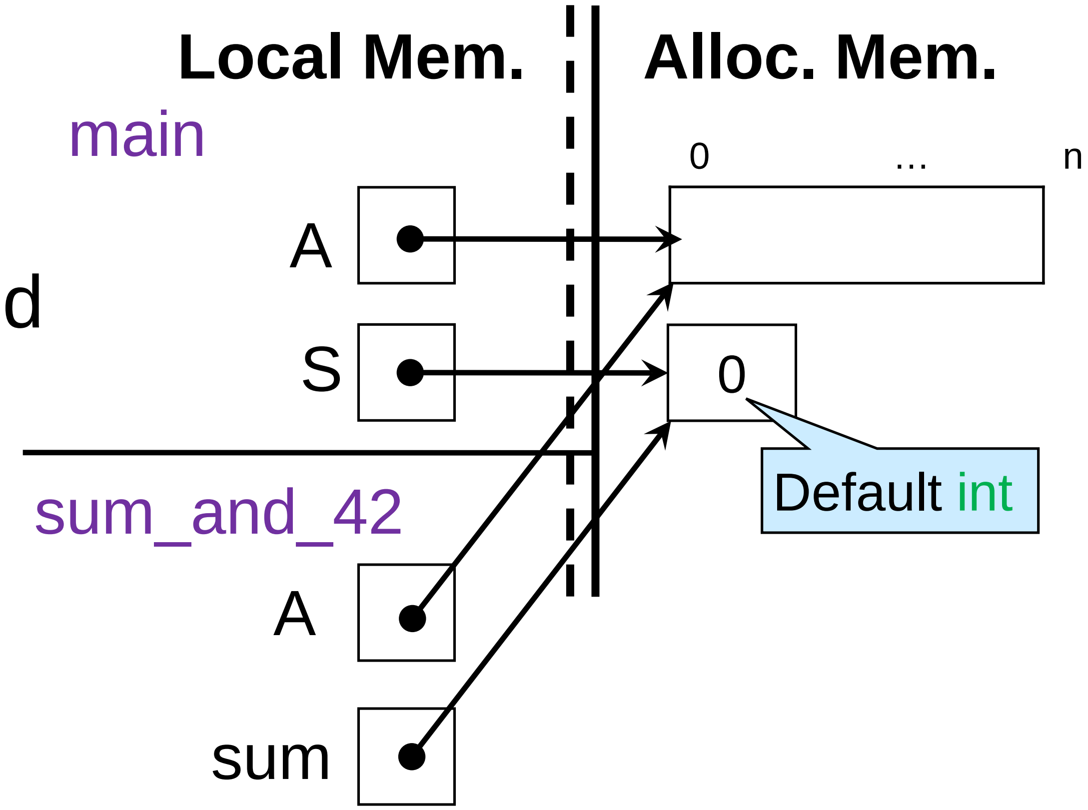
- It modifies the *same* cell p points to
  - upon returning, the cell pointed by p contains 4



- **Aliasing at work!**

Returning two Values from a Function

- This is how we solve our problem using pointers
  - caller pass an int* to store the sum and
  - function return a bool

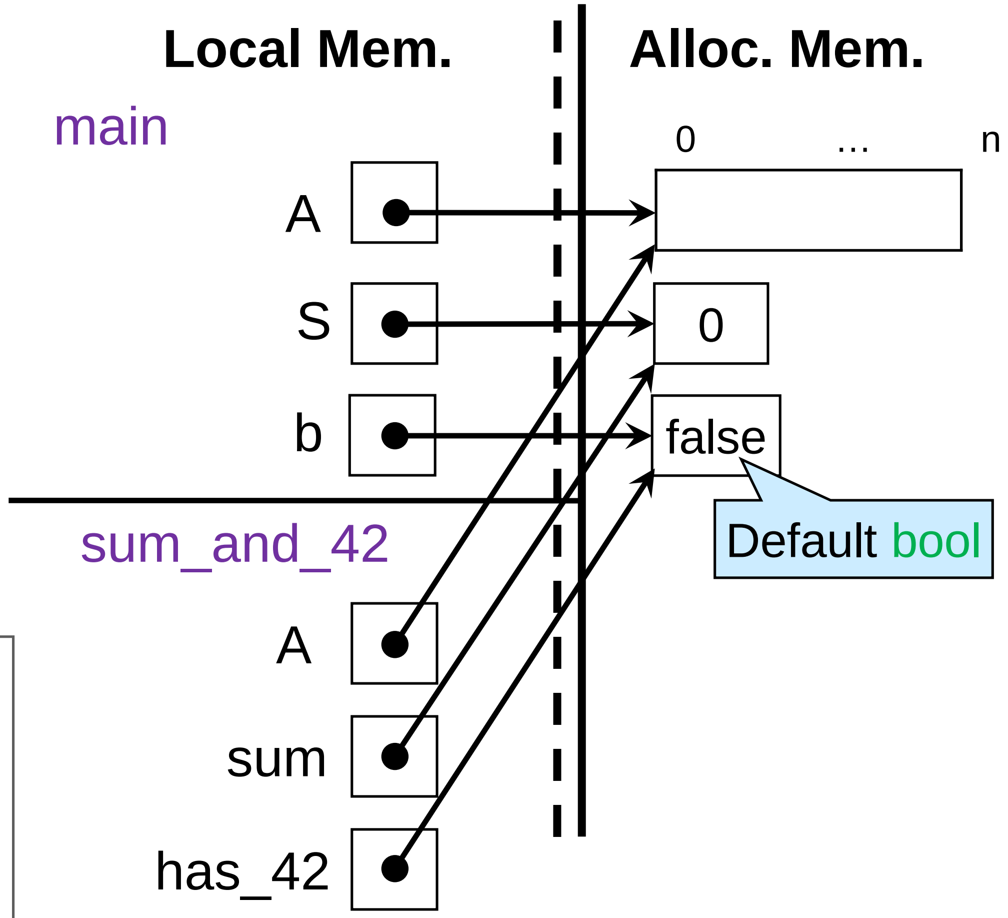


```
bool sum_and_42(int[] A, int n, int* sum)
//@requires n == \length(A);
{
  *sum = 0;
  bool has_42 = false;
  for (int i = 0; i < n; i++) {
    *sum += A[i];
    if (A[i] == 42) has_42 = true;
  }
 return has_42;
}
```

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

  int* S = alloc(int);
  bool b = sum_and_42(A, 10, S);
  return 0;
}
```

Returning two Values from a Function

- We can even share *both* via allocated memory
  - caller pass an int* to store the sum
  - **and** a bool* to store whether 42 is in the array

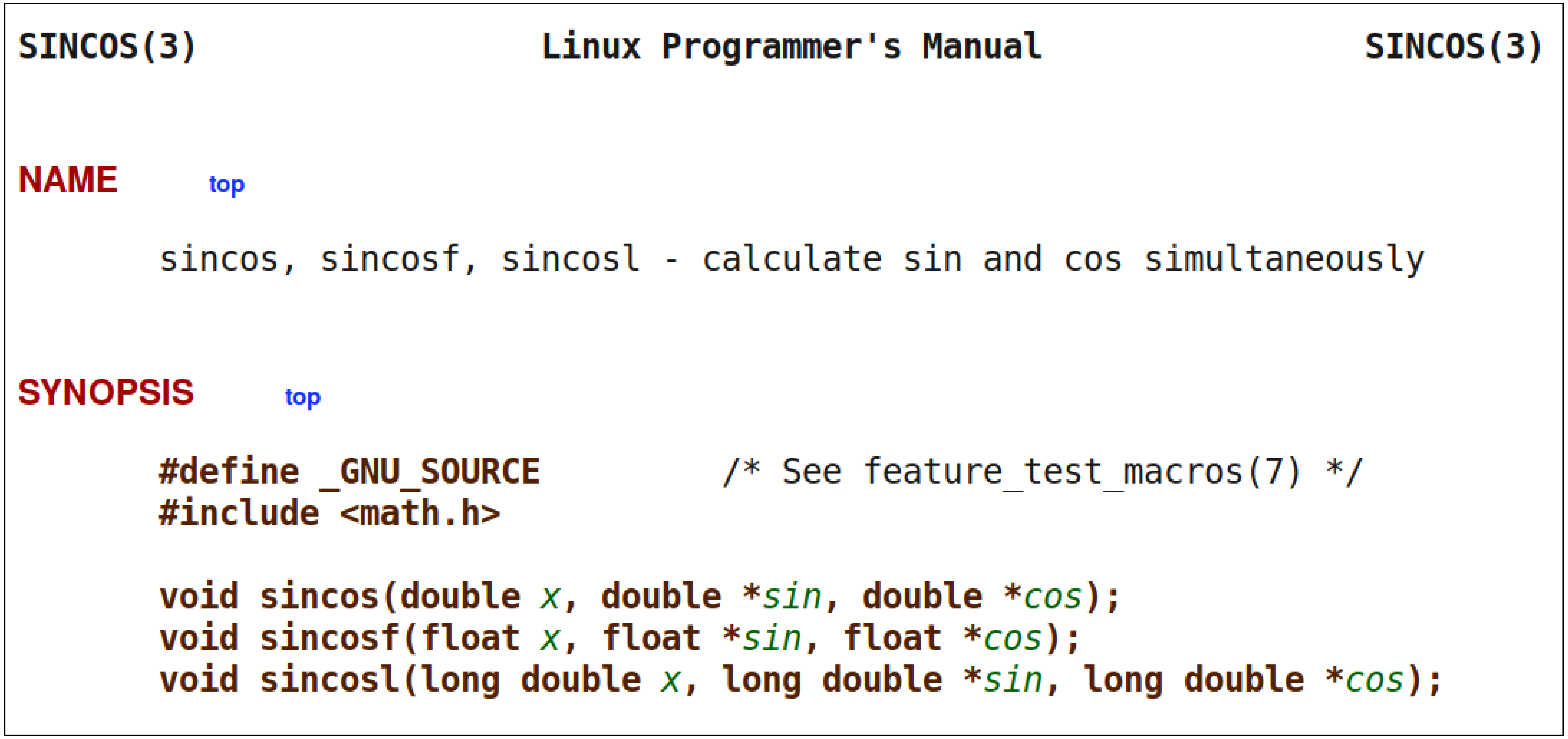


```
void sum_and_42(int[] A, int n, int* sum, bool* has_42)
//@requires n == \length(A);
{
 *sum = 0;
 *has_42 = false;
  for (int i = 0; i < n; i++) {
   *sum += A[i];
   if (A[i] == 42) *has_42 = true;
  }
}
```

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

  int* S = alloc(int);
  bool* b  = alloc(bool);
  sum_and_42(A, 10, S, b);
  return 0;
}
```

Returning two Values from a Function

- Real world example



SINCOS(3) Linux Programmer's Manual SINCOS(3)

NAME top

sincos, sincosf, sincosl - calculate sin and cos simultaneously

SYNOPSIS top

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <math.h>

void sincos(double x, double *sin, double *cos);
void sincosf(float x, float *sin, float *cos);
void sincosl(long double x, long double *sin, long double *cos);
```

http://man7.org/linux/man-pages/man3/sincos.3.html

Summary

- Memory cells are kind of like 1-element arrays
  - Live in allocated memory
  - Subject to aliasing
  - Garbage collected

- But they are not array!

Linux Terminal

```
--> int* p = alloc_array(int, 1);
<stdio>:1.10-1.29:error:type mismatch
expected: int*
   found: int[]
--> int[] A = alloc(int);
<stdio>:1.11-1.21:error:type mismatch
expected: int[]
   found: int*
```

Type error!

- int* and int[] are distinct type
  - ➢ Not interchangeable!

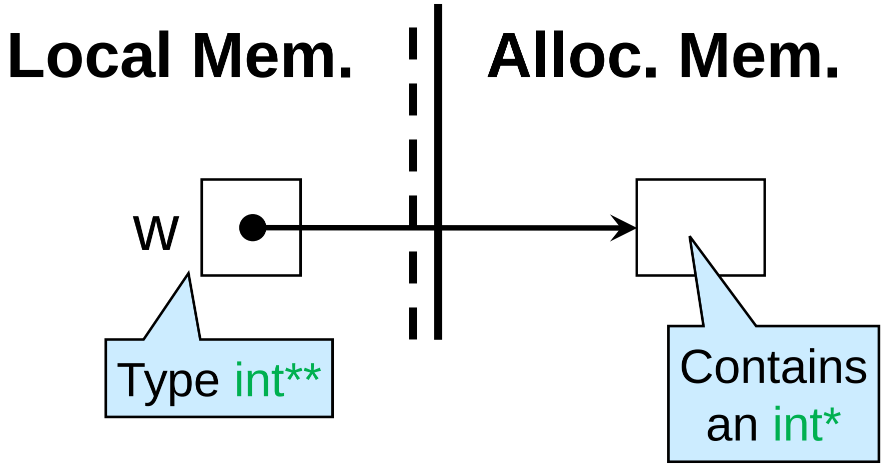
NULL

Double Pointers

- What does this do?

  int** w = alloc(int*);

  - Create a cell that can contain an int*



Local Mem.

Alloc. Mem.

w

Type int**

Contains an int*

- What is the default value of type int*?
  - Let's ask coin

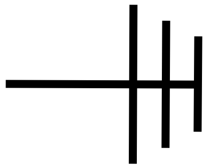
Linux Terminal

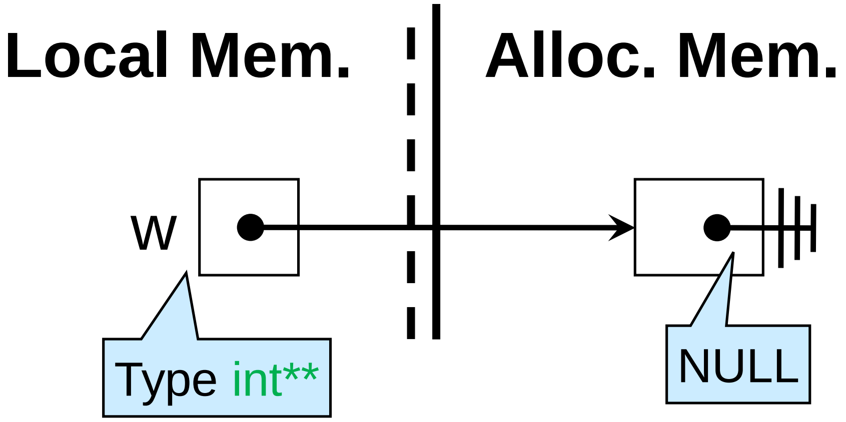
```
--> int** w = alloc(int*);
w is 0x1D75260 (int**)
--> *w;
NULL (int*)
```

- What is NULL?

NULL

- What is NULL?
  - The default value of *any* pointer type
  - Drawn as





- A value of pointer type can be either
  - an address to a cell in allocated memory, or
  - NULL

- We can check if a pointer is NULL

Linux Terminal

```
--> w == NULL;
true (false)
--> *w == NULL;
true (bool)
```

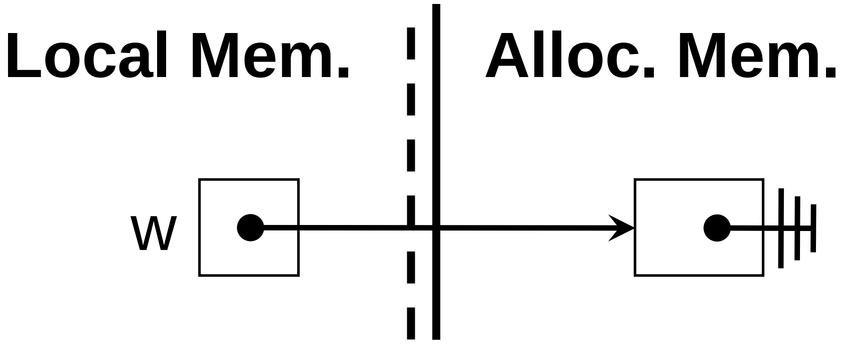
NULL

- What is NULL good for?

Linux Terminal

```
--> int** w = alloc(int*);
w is 0x1D75260 (int**)
--> *w;
NULL (int*)
--> **w;
Error: null pointer was accessed
```

We are accessing the value contained in *w, i.e., we are dereferencing NULL

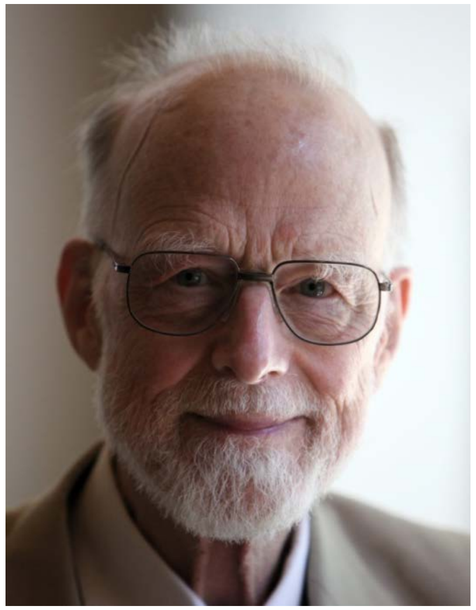


  - NULL is **not** the address of a memory cell
    - We can dereference addresses to memory cells
    - But, we are getting an error instead

- Dereferencing NULL is a **safety violation**

This is bad!

The Billion Dollar Mistake



- Tony Hoare introduced the NULL pointer in Algol W in 1965
- Part of most imperative programming languages ever since
  - C, C++, Python, Javascript, PHP, …
- One of the most error-prone programming constructs!

> *This led me to suggest that the **null** value is a member of every type, and a null check is required on every use of that reference variable, and it may be perhaps **a billion dollar mistake**.*
>
> -- Tony Hoare (InfoQ 2009 -- minute 27:40)

Trillions by now

  - Every time we dereference a pointer, we need to know it is not NULL
    - Many programmers forget
    - Endless source of bugs

Linux Terminal

```
# ./a.out
attempt to dereference null pointer
Segmentation fault (core dumped)
```

Pointer Safety

Dereferencing NULL is a **safety violation**

- *p has the *precondition*

  //@requires p != NULL;

  - Every time we dereference a pointer, we need to have a reason to believe it is not NULL
    - point-to reasoning!

- alloc(tp) has the postcondition

  //@ensures \result != NULL;

Linux Terminal

```
--> int** w = alloc(int*);
w is 0x1D75260 (int**)
--> *w;
NULL (int*)
```

Is this safe?

YES: w != NULL by postcondition of alloc

Pointer Safety

- Is our earlier code safe?
  - We are dereferencing sum, but we don't know it's not NULL
  - Add a precondition to ensure safety

```
//@requires sum != NULL;
```

A common contract
when working with pointers

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

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

  int* S = alloc(int);
  bool b = sum_and_42(A, 10, S);
  return 0;
}
```

Pointer Safety

- Is our earlier code safe now?

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

Is this safe?
YES: sum != NULL by new precondition

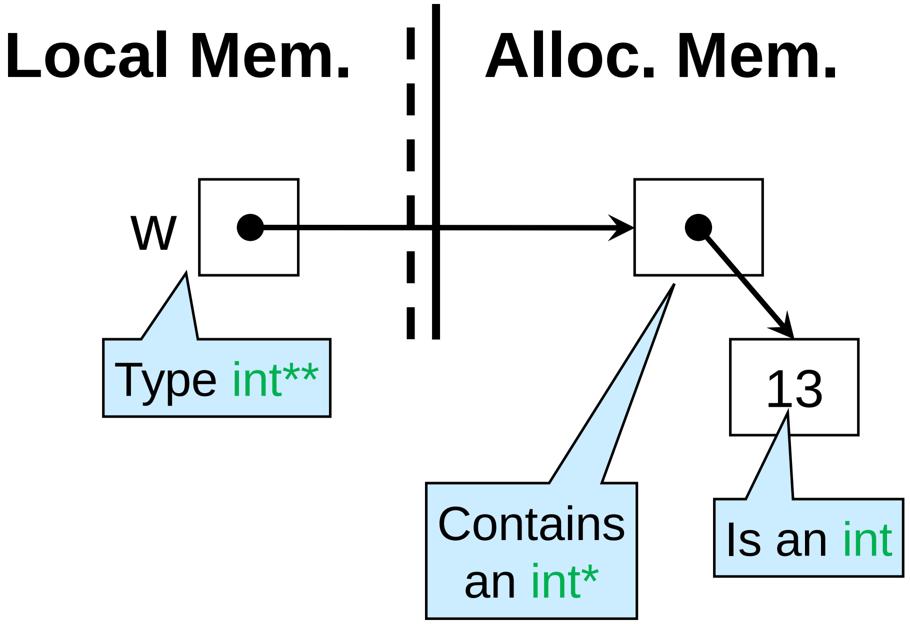
```
int main() {
  int[] A = alloc_array(int, 10);
  for (int i = 0; i < 10; i++) A[i] = i - 5;

  int* S = alloc(int);
  bool b = sum_and_42(A, 10, S);
  return 0;
}
```

Is this safe?
YES: S != NULL by postcondition of alloc

More about Double Pointers

- Let’s put something other than NULL in *w

```
int** w = alloc(int*);
*w = alloc(int);
**w = 13
```



  - w has type int** and points to a cell of type int*
  - *w has type int* points to a cell of type int
    - Why is this dereference safe?
      - by postcondition of alloc(int*)
  - **w is an int
    - Why is this dereference safe?
      - by postcondition of alloc(int)

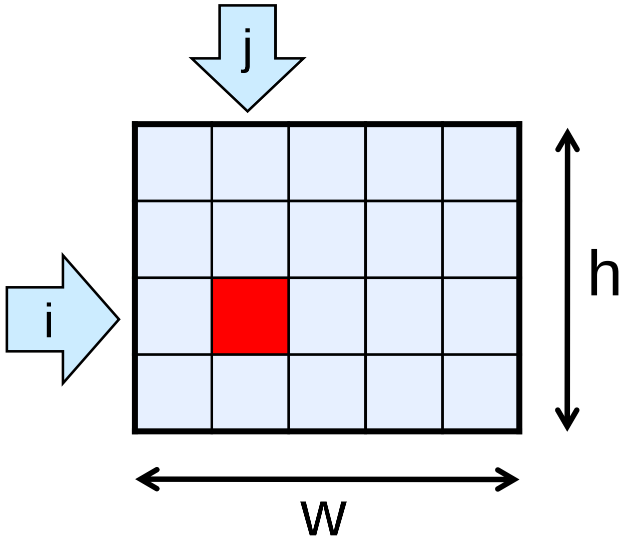
Summary: Pointers vs. Arrays

	Pointers	**Arrays**
Type	tp*	tp[]
Creation	alloc(tp) /*@ensures \result != NULL; @*/	alloc_array(tp, size) /*@requires size >= 0; @*/ /*@ensures \length(\result) == size; @*/
Reading and writing	*p	A[i] /@requires 0 <= i && i < \length(A); @*/
Contract-only operations		\length(A) /@ensures \result >= 0 @*/

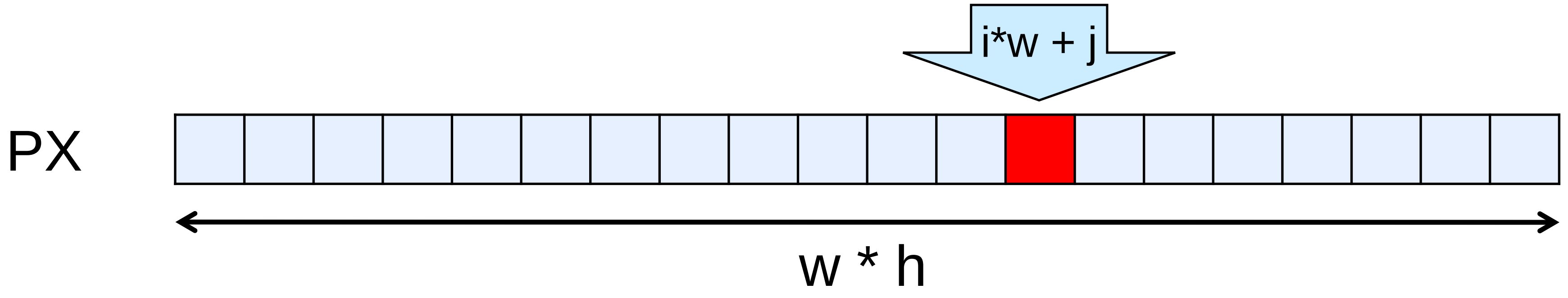
Structs

Representing Images

- We can represent an image
  of width *w*
  and height *h*
  by means of a *w*h* array of pixels, PX



  - Pixel on row i and column j is PX[i*w + j]



  - For simplicity, let’s say a pixel is an int

Manipulating Images

- A function that returns the first quadrant of an image
  - returns the pixel array of the output image
  - passes pointers to width and height of the output image

Exactly what we did earlier

```
int[] first_quadrant(int[] PX, int w, int h,        // input image
                     int* w_out, int* h_out)        // output image
//@requires w_out != NULL && h_out != NULL;
{
  *w_out = w/2;
  *h_out = h/2;
  int[] PX_out = alloc_array(int, (*w_out)*(*h_out));
  for (int i=0; i < *w_out; i++)
    for (int j=0; j < *h_out; j++)
      PX_out[i * (*w_out) + j] = PX[i*w + j];

  return PX_out;
}
```

This is to ensure the safety of dereferencing these pointers

What is going on here is not very important

Manipulating Images

```
int[] first_quadrant(int[] PX, int w, int h,       // input image
                     int* w_out, int* h_out)       // output image
//@requires w_out != NULL && h_out != NULL;
{ …}
```

Yuck!

- This looks clumsy
  - We like to think of an image as a single entity
    - Not a list of parts

- Furthermore
  - Caller has to create `int*` cells to hold width and height of output image
  - Easy to make mistakes by swapping width and height

Structs

- All modern programming language provide a way to view **a collection of parts as a single entity**

- In C0 (and C), this is a **struct**

A new way to create a type

```
struct image_header {
   int width;
   int height;
   int[] data;  // pixels in the image
};
```

  - This defines a new type called struct image_header
  - It has 3 parts: width, height and data
    - These are the **fields** of the struct

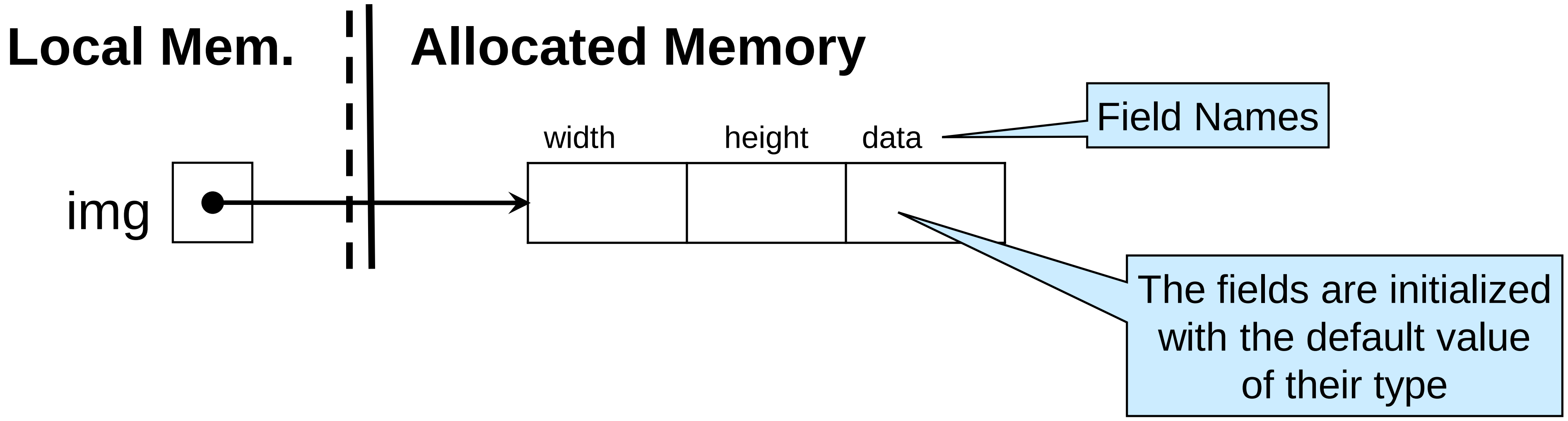
Using structs

```
struct image_header {
  int width;
  int height;
  int[] data;  // pixels in the image
};
```

- In C0, structs can only exist in allocated memory
  - We cannot have variables of type struct image_header
- They must be accessed via pointers
  - We can only have variables of type struct image_header*

- We create an image by allocating a struct in allocated memory

```
struct image_header* img = alloc(struct image_header);
```



Field Names

The fields are initialized with the default value of their type

Using structs

```
struct image_header {
   int width;
   int height;
   int[] data;  // pixels in the image
};
typedef struct image_header image;
```

```
struct image_header* img = alloc(struct image_header);
```

Yuck!

- *Seriously??*

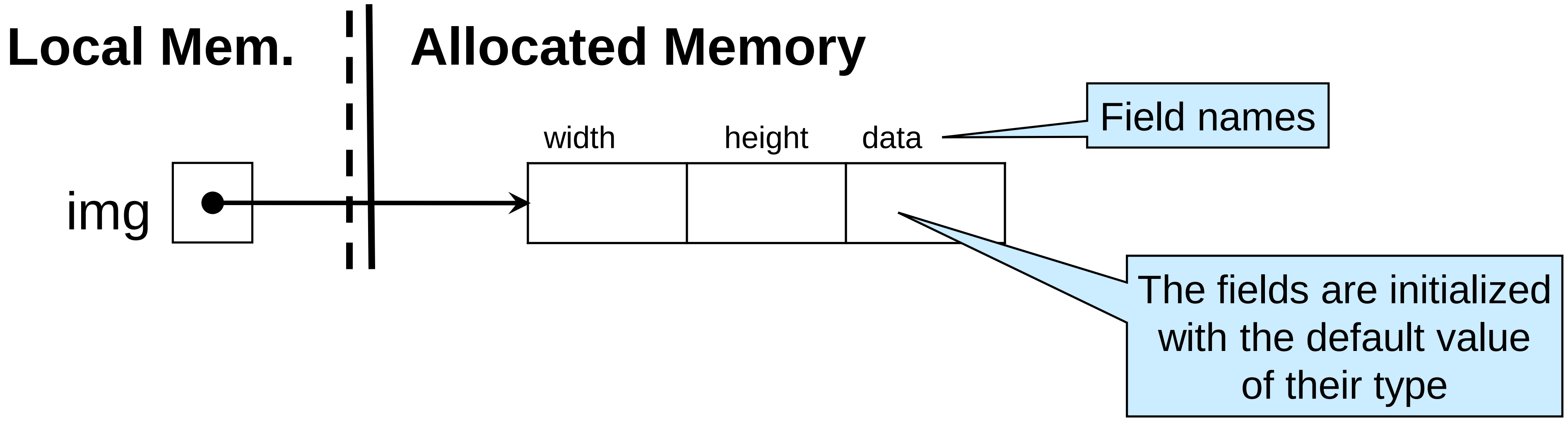
- Struct types are long and tedious to write
- We almost always give them a nickname with a typedef

```
typedef struct image_header image;
```

Now, we can write image anywhere we had struct image_header

  - Now

```
image* img = alloc(image);
```


Local Mem.

Allocated Memory

Field names

The fields are initialized with the default value of their type

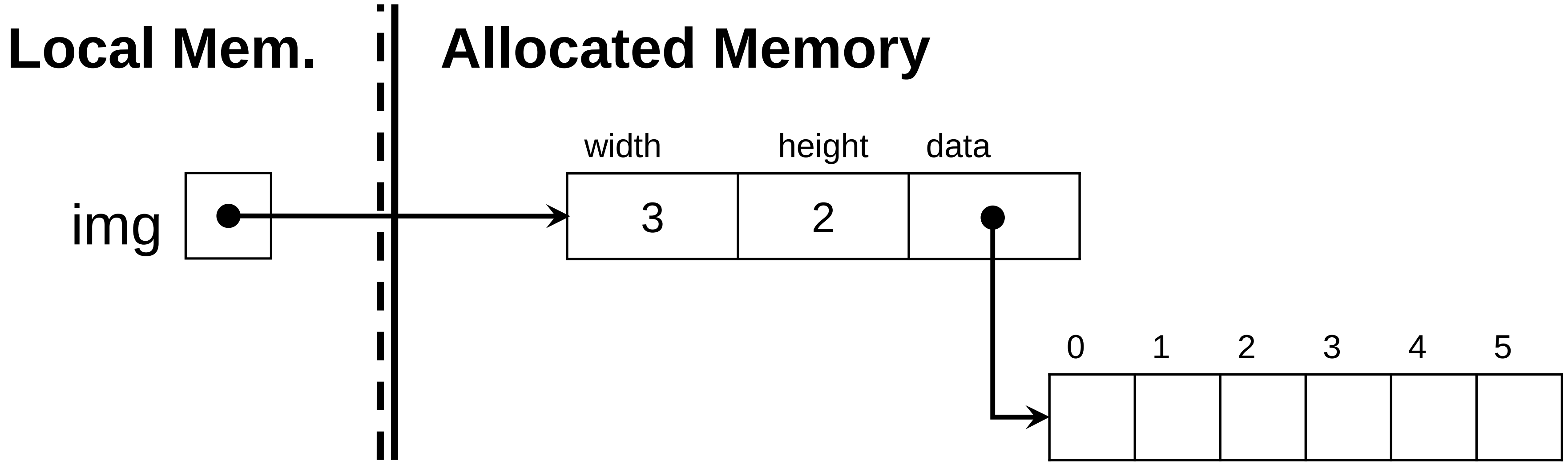
Using Structs

```
struct image_header {
   int width;
   int height;
   int[] data;  // pixels in the image
};
typedef struct image_header image;
```

- We manipulate a field of a struct using the *field access* operator: ->

```
image* img = alloc(image);
img->width = 3;
img->height = 2;
img->data = alloc_array(int, 6);
```

- follows the pointer in img
- goes to the width field
- writes 3 there



Safety

- img->width dereferences the pointer img
  - We must be sure this is safe
  - img must not be NULL

- ptr->field has the precondition

  ```
  //@requires ptr != NULL;
  ```

  - just like *ptr

- The compiler will issue an error if the field name is wrong

Safety

- ptr->field has the precondition

  //@requires pointer != NULL;

  - just like *ptr

- So, there are two ways to dereference a pointer depending on its type?
  - Kind of
    - img->width is shorthand for (*img).width

Normal pointer dereference

Field access within a struct

When we are **not** following a pointer

  - In C0, we *never* have a reason to use the "." operator
    - ❑ We will **always** write img->width
    - C is a different story, however

Returning Multiple Values

```
struct image_header {
   int width;
   int height;
   int[] data;  // pixels in the image
};
typedef struct image_header image;
```

- A function that returns the first quadrant of an image
  - takes an image* as input
  - returns an image* as output

No funny business!

```
image* first_quadrant(image* img)
//@requires img != NULL;
//@ensures \result != NULL;
{
  image* out = alloc(image);
  out->width = img->width/2;
  out->height = img->height/2;
  out->data = alloc_array(int, out->width * out->height);
  for (int i=0; i < out->width; i++)
    for (int j=0; j < out->height; j++)
      out->data[i * out->width + j] = img->data[i*img->width + j];

  return out;
}
```

Supports safety of pointer dereferences

Supports safety of caller code

What is going on here is not very important
But a lot more readable!

Returning Multiple Values

- Should we always return multiple values using a struct?
  - If the right struct is already defined, by any means!
    - E.g., image
  - If we need to define the struct just for this purpose, don’t bother
    - E.g., sum_and_42
    - Other programming languages give a way to define things like structs on the fly

A Collection of Parts as a Single Entity

- All modern languages provide a way to view a collection of parts as a single entity
  - structs in C0 (and C)

- This is the basis for an **extraordinary** form of **abstraction**
  - Allows manipulating complex entities as a whole
    - through well-defined, abstract operations
    - without a need to know the details
  - This underlies the concept of **data structures**
    - The major topic of the rest of this course