C’s Memory Model

C0

C

Balance Sheet … so far

Lost	***Gained***
• Contracts • Safety • Garbage collection • Memory initialization	• Preprocessor • Whimsical execution • Explicit memory management • Separate compilation

Arrays in C

Creating an Array

- Here's how we create a 5-element int array

```
int *A = malloc(sizeof(int) * 5);
```

The type is int*,
not int[]

We use malloc like for pointers,
not a special array-only instruction

- In C **arrays and pointers are the same thing***
  - No special array type
  - No special allocation instruction

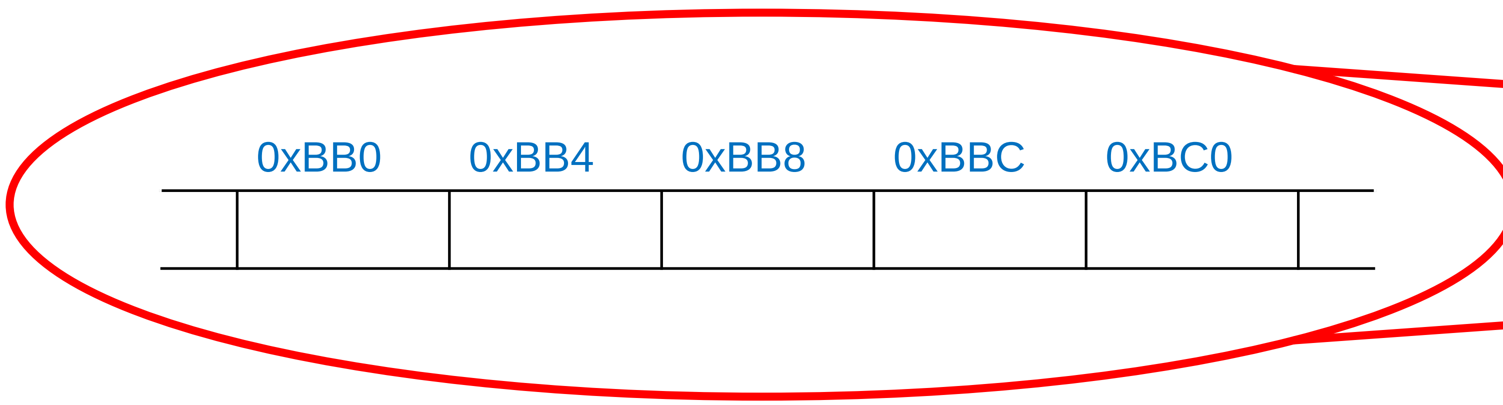
    - malloc returns NULL when we have run out of memory
      - we use xmalloc instead

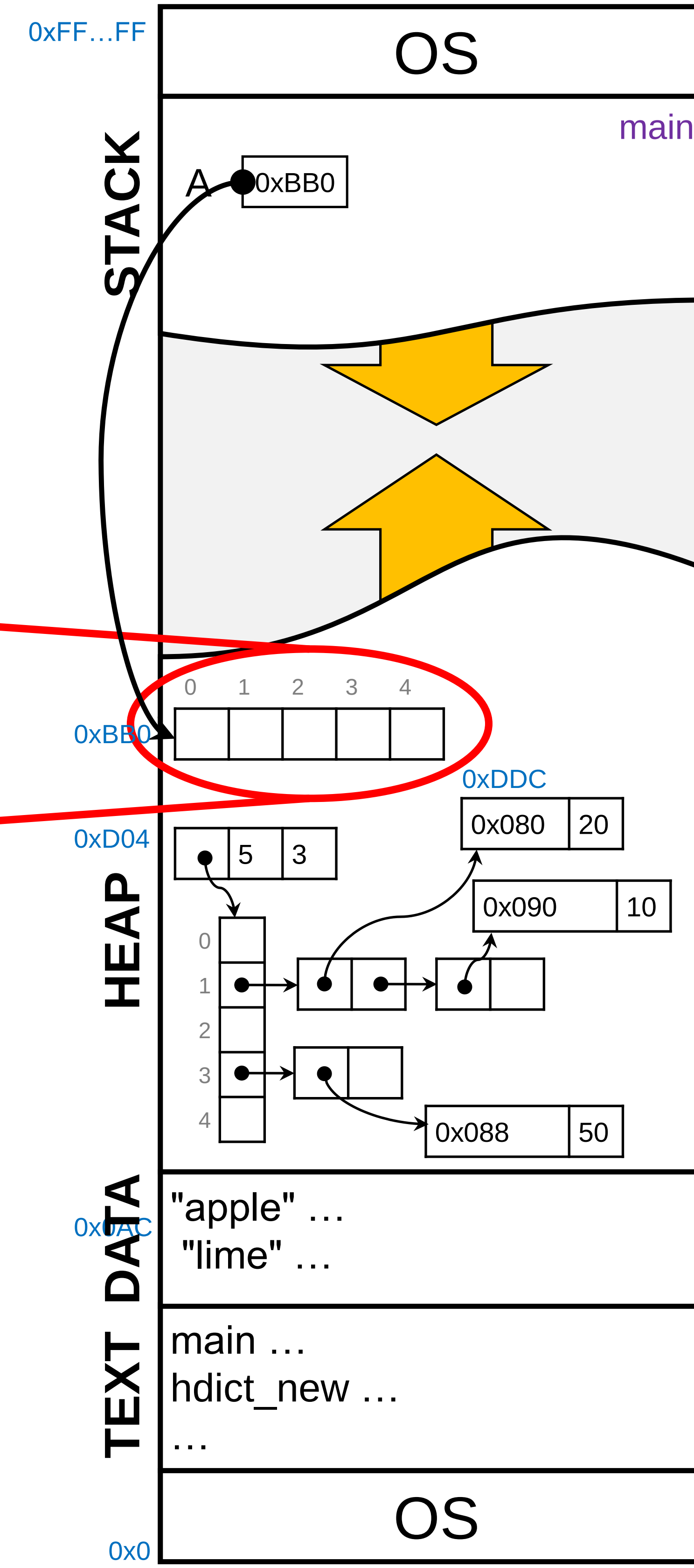
*on the heap

Creating an Array

int *A = xmalloc(sizeof(int) * 5);

- But what does it do?





  - It allocates contiguous space that can contain 5 ints on the heap
  - and returns its address

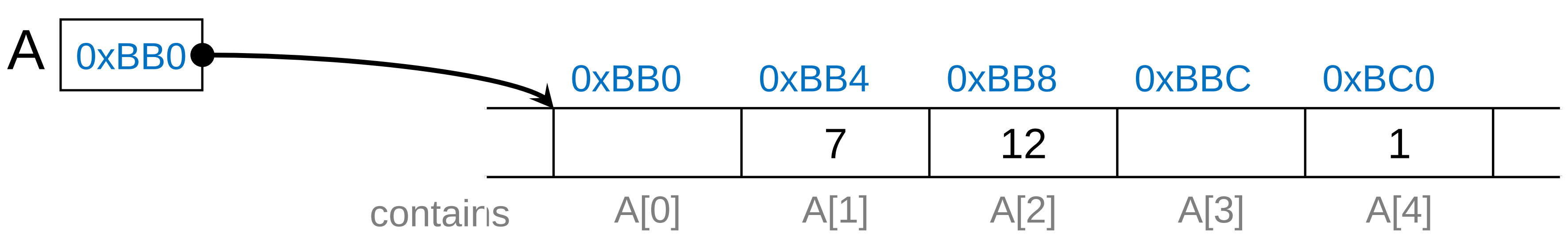
```
int main() {
  int *A = xmalloc(sizeof(int) * 5);
  ...
}
```

Using an Array

- Arrays are accessed like in C0

```
A[1] = 7;
A[2] = A[1] + 5;
A[4] = 1;
```

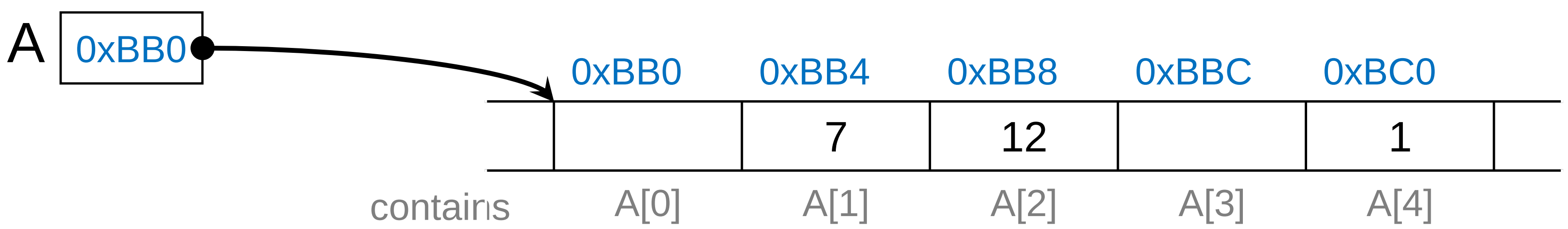
A[0] refers to the 1st int pointed to by A,
A[1] to the 2nd int pointed to by A,
…
A[4] to the 5th int pointed to by A



  - Like in C0, C arrays are 0-indexed

```
int main() {
  int *A = xmalloc(sizeof(int) * 5);
  A[1] = 7;
  A[2] = A[1] + 5;
  A[4] = 1;
  ...
}
```

Pointer Arithmetic



- If A is a pointer, then *A is a valid expression
  - What is it?
- A is an int*, so *A is an int
  - it refers to the first element of the array
  - ***A is the same as A[0]**

```
*A = 42;
```

sets A[0] to 42

Pointer Arithmetic

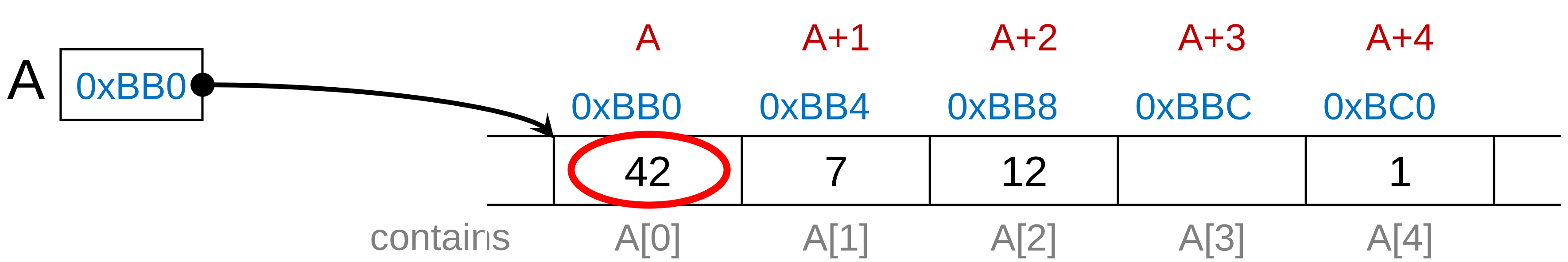
```
int main() {
  int *A = xmalloc(sizeof(int) * 5);
  A[1] = 7;
  A[2] = A[1] + 5;
  A[4] = 1;
  *A = 42;
  ...
}
```

- A is the address of the first element of the array

- What is the address of the next element?
  - It’s A + one int over: A+1
  - In general the address of the i-th element of A is A+i

A plus i **elements** over

Not A plus i *bytes* over



- This is called **pointer arithmetic**

Pointer Arithmetic

```
int main() {
  int *A = xmalloc(sizeof(int) * 5);
  A[1] = 7;
  A[2] = A[1] + 5;
  A[4] = 1;
  *A = 42;
  ...
}
```

- A+i is the **address** of A[i]
  - so *(A+i) is A[i]
    - ➢ the **value** of the element A[i]
  - so

    printf("A[1] is %d\n", *(A+1));

    prints 7

A	A+1	A+2	A+3	A+4
0xBB0	0xBB4	0xBB8	0xBBC	0xBC0
42	7	12		1
A[0]	A[1]	A[2]	A[3]	A[4]
*A	*(A+1)	*(A+2)	*(A+3)	*(A+4)

- In fact, A[i] is just convenience syntax for *(A+i)

In the same way that p->next is just convenience syntax for (*p).next

Pointer Arithmetic

A	A+1	A+2	A+3	A+4
0xBB0	0xBB4	0xBB8	0xBBC	0xBC0
42	7	12		1
A[0]	A[1]	A[2]	A[3]	A[4]
*A	*(A+1)	*(A+2)	*(A+3)	*(A+4)

- Pointer arithmetic is one of the **most error-prone features** of C
- But no C program needs to use it
  - Every piece of C code can be rewritten without
    - change *(A+i) to A[i]
    - change A+i to … *(later)*
- Code that doesn't use pointer arithmetic
  - is more readable
  - has fewer bugs

Initializing Memory

```
int main() {
  int *A = xmalloc(sizeof(int) * 5);
  A[1] = 7;
  A[2] = A[1] + 5;
  A[4] = 1;
  *A = 42;
  ...
}
```

- (x)malloc does not initialize memory to default value
  - A[3] could contain any value

0xBB0	0xBB4	0xBB8	0xBBC	0xBC0
42	7	12		1
A[0]	A[1]	A[2]	A[3]	A[4]

- To allocate memory and initialize it to all zeros, use the function calloc

```
int *A = calloc(5, sizeof(int));
```

calloc takes two arguments, while malloc takes only one

Number of elements

Size of each element

  - calloc returns NULL if there is no memory available
    - lib/xalloc.h provides xcalloc that aborts execution instead

0xBB0	0xBB4	0xBB8	0xBBC	0xBC0
42	7	12	0	1
A[0]	A[1]	A[2]	A[3]	A[4]

Now A[3] contains 0

Freeing Arrays

```
int main() {
  int *A = xcalloc(5, sizeof(int));
  A[1] = 7;
  A[2] = A[1] + 5;
  A[4] = 1;
  *A = 42;
  free(A);
}
```

- A was created in allocated memory
  - on the heap
- Therefore we must free it before the program exits
  - otherwise there is a memory leak

```
free(A);
```

- The C motto

If you allocate it, you free it

The Length of an Array

```
int main() {
  int *A = xcalloc(5, sizeof(int));
  A[1] = 7;
  A[2] = A[1] + 5;
  A[4] = 1;
  *A = 42;
  free(A);
}
```

- In C0, we can know the length of an array only in contracts

C0 stores it secretly

- In C, there is **no way** to find out the length of an array
  - We need to keep track of it meticulously

It is written nowhere

- But free knows how much memory to give back to the OS
  - The memory management part of the run-time keeps track of the starting address and size of every piece of allocated memory …
  - … but none of this is accessible to the program

Arrays Summary

Arrays in C

- Arrays are pointers
- Created with (x)malloc
  - *does not initialize elements*

  or with (x)calloc
  - *does initialize elements*
- Must be freed
- No way to find the length

Arrays in C0

- Arrays have a special type
- Created with alloc_array
  - *Initializes the elements to 0*
- Garbage collected
- Length available in contracts

Undefined Behavior

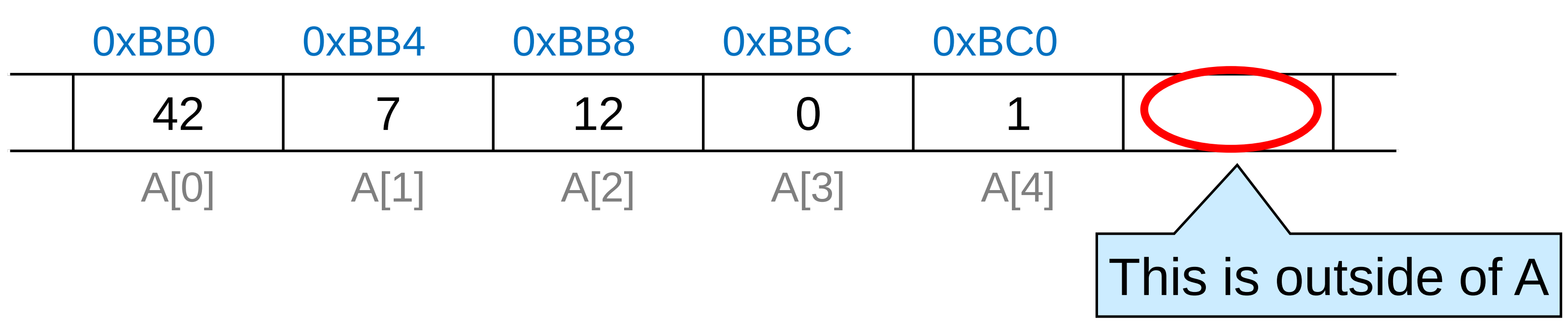
Out-of-bound Accesses

```
int main() {
  int *A = xcalloc(5, sizeof(int));
  A[1] = 7;
  A[2] = A[1] + 5;
  A[4] = 1;
  *A = 42;
}
```

- What if we try to access A[5]?

```
printf("A[5] is %d\n", A[5]);
```

- In C0, this is a **safety violation**
  - array access out of bounds
- In C, that's *(A+5)
  - the value of the 6th int starting from the address in A



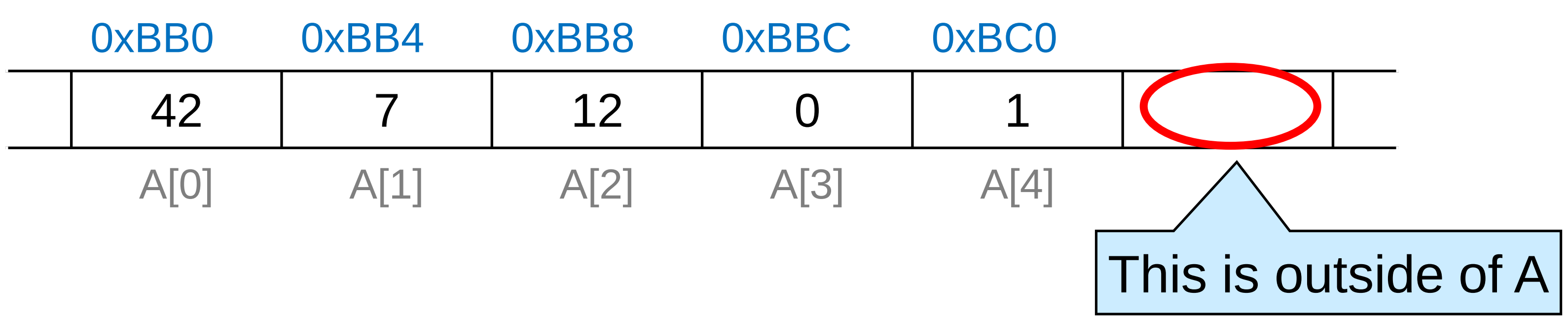
- *What will happen?*

Out-of-bound Accesses

```
int main() {
  int *A = xcalloc(5, sizeof(int));
  A[1] = 7;
  A[2] = A[1] + 5;
  A[4] = 1;
  *A = 42;
}
```

- *What will happen?*

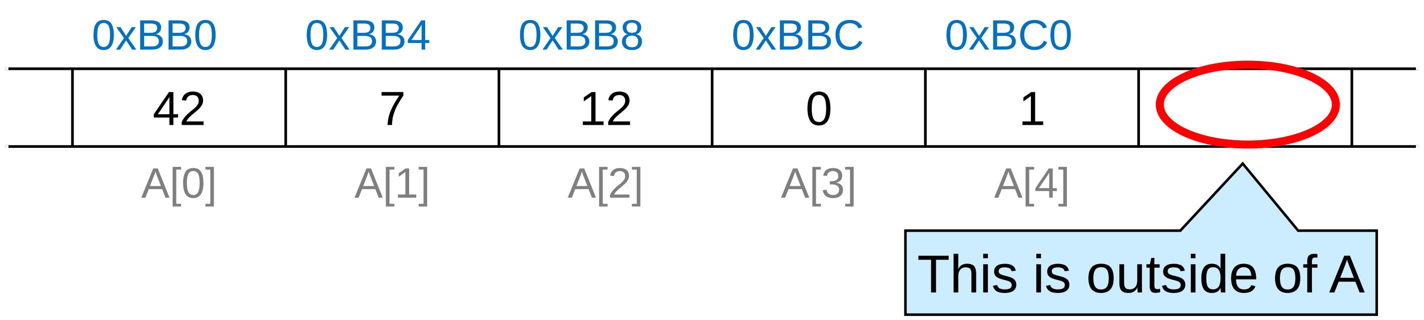
printf("A[5] is %d\n", A[5]);



- It could
  - print some int and continue execution
  - abort the program
  - crash the computer
  - do weirder things
    (within the laws of physics)

Google joke:
order pizza for the whole team

Out-of-bound Accesses



printf("A[5] is %d\n", A[5]);

could do different things on different runs

- it could work as expected most of the times but not always
  - corrupt the data and crash in mysterious ways later

Linux Terminal

```
# gcc -Wall …
# ./a.out
A[5] is 1879048222
A[1000] is -837332876
A[-1] is 1073741854
Segmentation fault (core dumped)
```

- Same thing with

  printf("A[-1] is %d\n", A[-1]);

  printf("A[1000] is %d\n", A[1000]);

- But

  printf("A[10000000] is %d\n", A[10000000]);

  will consistently crash the program
  - with a **segmentation fault**

Debugging Out-of-bound Accesses

- The code could work as expected most of the times but not always
  - Extremely hard to debug
- Valgrind will often point out out-of-bound accesses

```c
printf("A[5] is %d\n", A[5]);
```

Linux Terminal

```
# valgrind ./a.out
==14980== Invalid read of size 4
==14980==    at 0x1089C2: main (test.c:40)
==14980==  Address 0x522d054 is 0 bytes after a block of size 20 alloc'd
==14980==    at 0x4C31B25: calloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
   linux.so)
==14980==    by 0x108878: xcalloc (xalloc.c:16)
==14980==    by 0x108965: main (test.c:29)
…
```

In this code, ints are 4 bytes

Line where the bad access occurred

A contains 5 ints,
so it's 20 bytes long

Line where it was allocated

Debugging Out-of-bound Accesses

- Valgrind will often point out out-of-bound accesses

```
A[5] = 15122;
```

Here we are *writing* to A[5]

Linux Terminal

In this code, ints are 4 bytes

Line where the bad access occurred

```
# valgrind ./a.out
==15847== Invalid write of size 4
==15847==    at 0x108982: main (test.c:46)
==15847==  Address 0x522d054 is 0 bytes after a block of size 20 alloc'd
==15847==    at 0x4C31B25: calloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
   linux.so)
==15847==    by 0x108838: xcalloc (xalloc.c:16)
==15847==    by 0x108925: main (test.c:29)
…
```

Line where it was allocated

Debugging Out-of-bound Accesses

- Valgrind will often point out out-of-bound accesses

```
printf("A[-1] is %d\n", A[-1]);
```

Linux Terminal

```
# valgrind ./a.out
==15091== Invalid read of size 4
==15091==    at 0x1089C2: main (test.c:42)
==15091==  Address 0x522d03c is 4 bytes before a block of size 20 alloc'd
==15091==    at 0x4C31B25: calloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==15091==    by 0x108878: xcalloc (xalloc.c:16)
==15091==    by 0x108965: main (test.c:29)
…
```

In this code, ints are 4 bytes

Line where the bad access occurred

A contains 5 ints, so it's 20 bytes long

Line where it was allocated

Debugging Out-of-bound Accesses

- Valgrind will often point out out-of-bound accesses

```
printf("A[1000] is %d\n", A[1000]);
```

Linux Terminal

In this code, ints are 4 bytes

Line where the bad access occurred

```
# valgrind ./a.out
==15063== Invalid read of size 4
==15063==    at 0x1089C4: main (test.c:41)
==15063==  Address 0x522dfe0 is 3,904 bytes inside an unallocated block of size 4,194,112
   in arena "client"
…
```

- It doesn’t give as much information further away from the array

Debugging Out-of-bound Accesses

- Valgrind will often point out out-of-bound accesses

```
printf("A[10000000] is %d\n", A[10000000]);
```

Linux Terminal

In this code, ints are 4 bytes

Line where the bad access occurred

```
# valgrind ./a.out
==15113== Invalid read of size 4
==15113==    at 0x1089C4: main (test.c:44)
==15113==  Address 0x7852a40 is not stack'd, malloc'd or (recently) free'd
==15113==
==15113==
==15113== Process terminating with default action of signal 11 (SIGSEGV)
==15113==  Access not within mapped region at address 0x7852A40
==15113==    at 0x1089C4: main (test.c:44)
…
Segmentation fault (core dumped)
```

  - What does this mean?

Out-of-bound Accesses

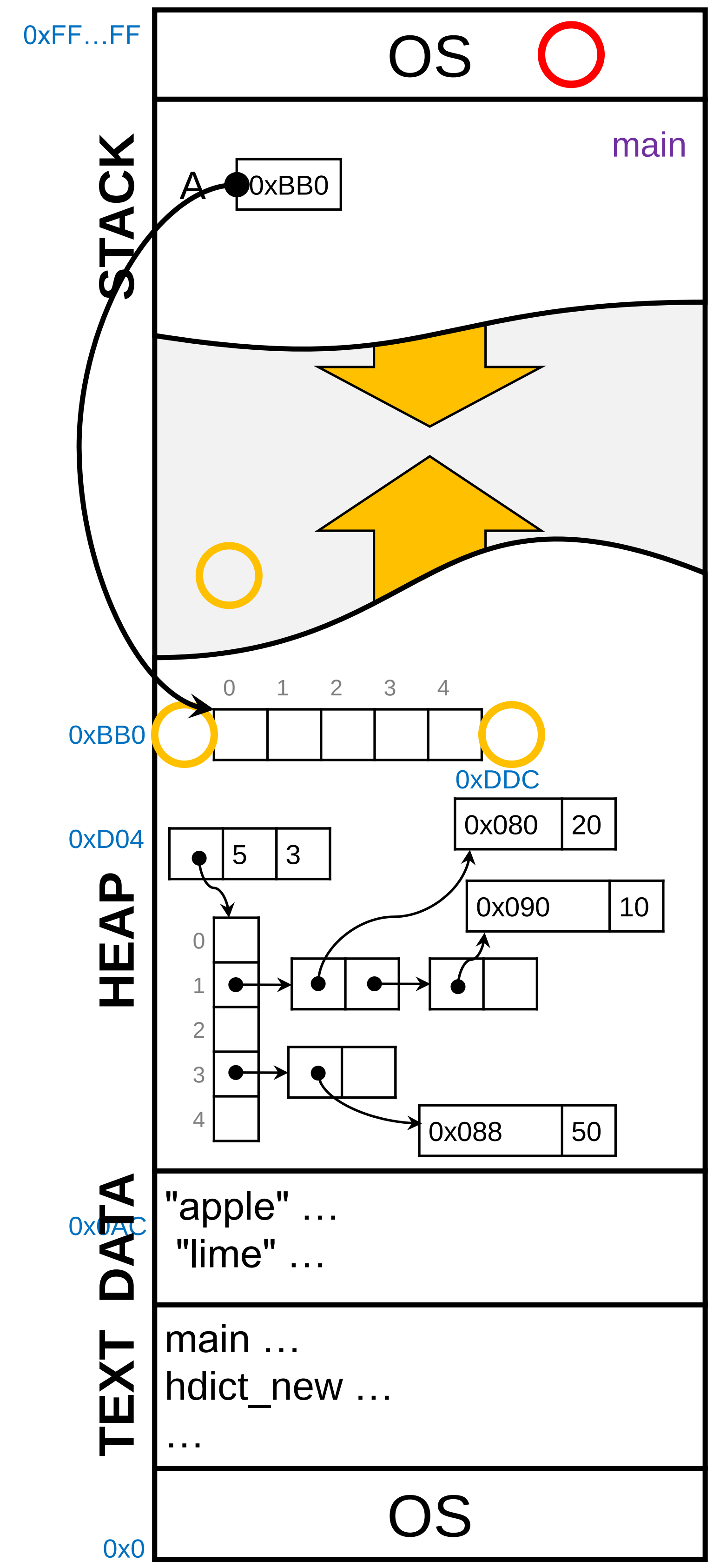
- printf("A[5] is %d\n", A[5]);
- printf("A[-1] is %d\n", A[-1]);
- printf("A[1000] is %d\n", A[1000]);

all access memory in the heap, near A

- printf("A[10000000] is %d\n", A[10000000]);

accesses memory outside in the heap

- in a different segment of memory
- That's why the program crashes with a **segmentation fault**



Debugging Out-of-bound Accesses

- Valgrind cannot catch all out-of-bound accesses

A[-1000] = 42;

Linux Terminal

```
# valgrind ./a.out
==16357==
==16357==
…
```

No error reported!

  - Valgrind keeps track of likely locations where programmers make mistakes
    - e.g., off-by-one errors
  - it does not monitor the whole memory

Undefined Behavior

Out-of-bound accesses may do different things on different runs

- Why?
- Because the C99 standard does not specify what should happen
- Out-of-bound accesses are **undefined behavior**
  - different compilers do different things
  - often just carry on
    - read or write other program data
    - unless accessing a restricted segment

That's what will make the code run fastest

But debugging is a nightmare

Undefined Behavior

- **Every safety violation in C0 is undefined behavior in C**
  - accessing an array out-of-bound
  - dereferencing NULL
  - (plus other violations we will examine later)

C0 was engineered this way on purpose:
- everything that could happen during execution is defined
- bad thing that could happen abort the program

- But there is more in C than in C0
- Almost anything else slightly weird is undefined behavior in C
  - reading uninitialized memory
    - even if correctly allocated
  - using memory that has been freed
  - double free
  - …

More later

Undefined Behavior

- What's so bad about them?
  - Security vulnerabilities
    - Heartbleed, Stuxnet
  - Software bugs
    - buffer overflow

- Why does C have undefined behaviors?
  - These were the early days of programming language research

- Why haven't they been fixed?
  - Some legacy code relies on the behavior of a specific compiler on a specific OS to do its job
    - Fixing it would break this code

Aliasing

Aliasing into an Array

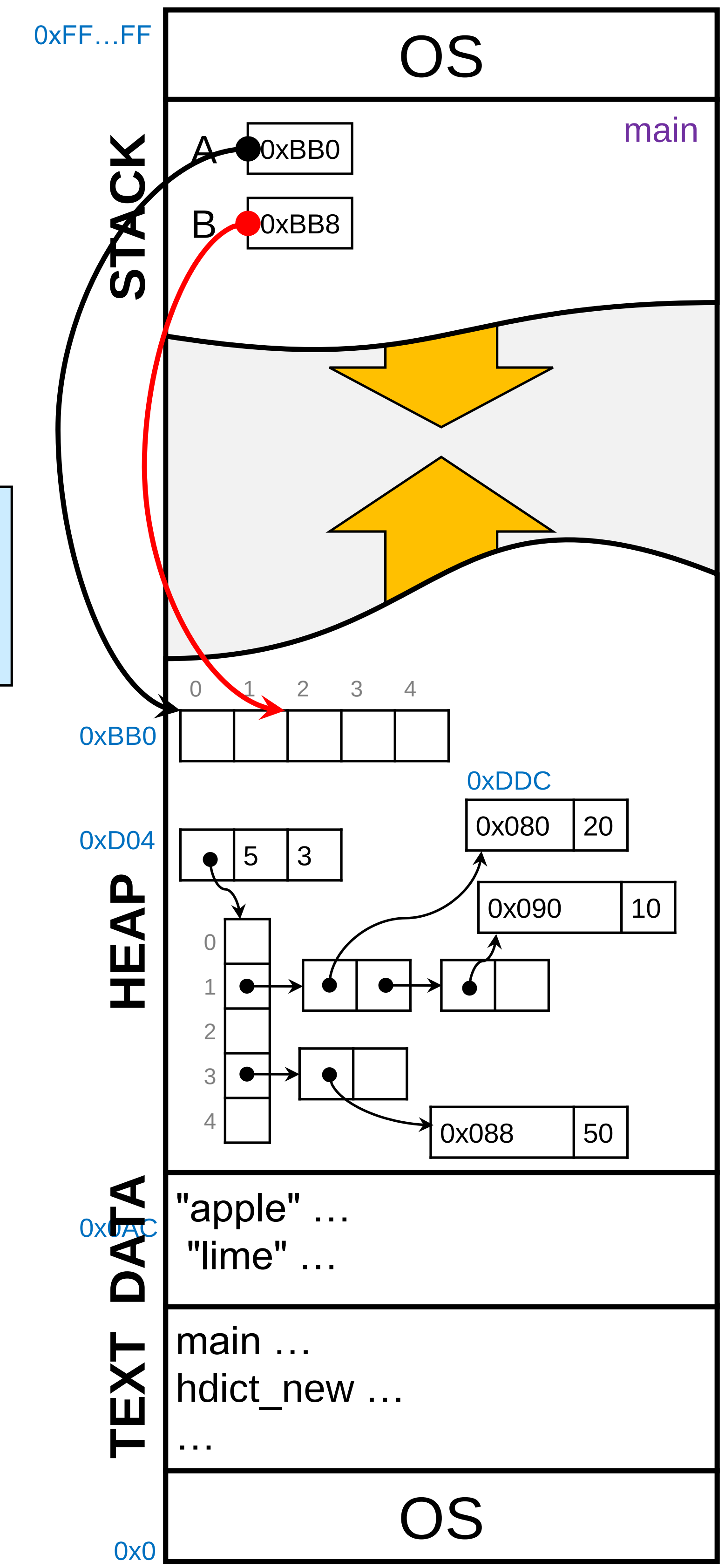
int *B = A+2;

- B contains the address of the third element of A

Pointer arithmetic lets us grab the address of an element in the middle of an array

- But B has type int*
  - an array of ints
    - B[0] is A[2]
    - B[1] is A[3], …

		B	B+1	B+2
A	A+1	A+2	A+3	A+4
0xBB0	0xBB4	0xBB8	0xBBC	0xBC0
42	7	12		1
A[0]	A[1]	A[2]	A[3]	A[4]
		B[0]	B[1]	B[2]



Aliasing into an Array

```
int *B = A+2;
assert(B[0] == A[2]);
assert (B[1] == A[3]);
assert(*(B+2) == A[4]);
```

		B	B+1	B+2
A	A+1	A+2	A+3	A+4
0xBB0	0xBB4	0xBB8	0xBBC	0xBC0
42	7	12		1
A[0]	A[1]	A[2]	A[3]	A[4]
		B[0]	B[1]	B[2]

B[0] is A[2],
B[1] is A[3], …

- We have a **new form of aliasing**

```
B[1] = 35;
assert(A[3] == 35);
```

		B	B+1	B+2
A	A+1	A+2	A+3	A+4
0xBB0	0xBB4	0xBB8	0xBBC	0xBC0
42	7	12	35	1
A[0]	A[1]	A[2]	A[3]	A[4]
		B[0]	B[1]	B[2]

Aliasing into an Array

```
int *B = A+2;
B[1] = 35;
```

		B	B+1	B+2
A	A+1	A+2	A+3	A+4
0xBB0	0xBB4	0xBB8	0xBBC	0xBC0
42	7	12	35	1
A[0]	A[1]	A[2]	A[3]	A[4]
		B[0]	B[1]	B[2]

- We are not allowed to free B
  - It was not returned by (x)malloc or (x)calloc
  - Doing so is **undefined behavior**

Casting Pointers in C

Casting Pointers

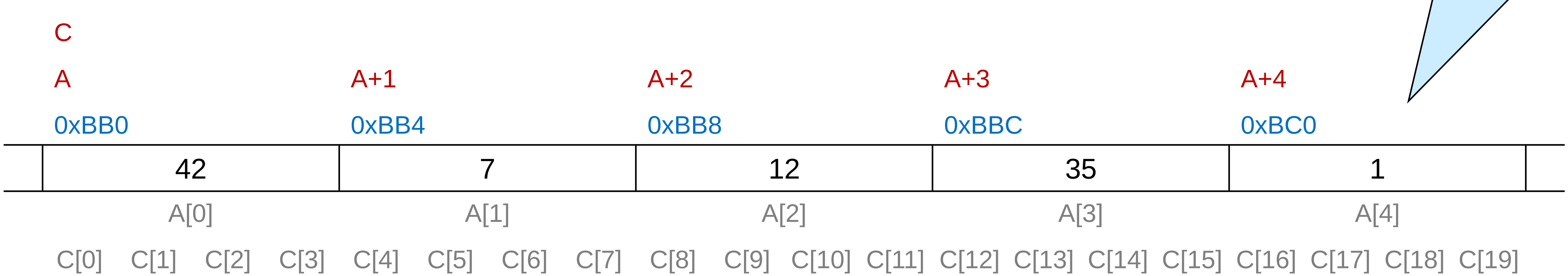
- In C1, we can
  - cast any pointer to void*
  - cast void* only to the original pointer type

- In C, we can cast any pointer to any pointer type
  - this never triggers an error

char *C = (char*)A;

- As C, it views the space occupied by A as a char array

A char is 1 byte, so each int is 4 chars



Casting Pointers

C				
A	A+1	A+2	A+3	A+4
0xBB0	0xBB4	0xBB8	0xBBC	0xBC0
42	7	12	35	1
A[0]	A[1]	A[2]	A[3]	A[4]
C[0] C[1] C[2] C[3]	C[4] C[5] C[6] C[7]	C[8] C[9] C[10] C[11]	C[12] C[13] C[14] C[15]	C[16] C[17] C[18] C[19]

- C[16] is the 17th character in C
  - i.e., the first byte of A[4]

- Since A[4] is 1 == 0x00000001
  - we expect C[16] to be 0

Casting Pointers

C				
A	A+1	A+2	A+3	A+4
0xBB0	0xBB4	0xBB8	0xBBC	0xBC0
42	7	12	35	1
A[0]	A[1]	A[2]	A[3]	A[4]

C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]	C[7]	C[8]	C[9]	C[10]	C[11]	C[12]	C[13]	C[14]	C[15]	C[16]	C[17]	C[18]	C[19]

```
printf("The 16th char in C is %d\n", C[16]);
```

- We expect C[16] to be 0

Linux Terminal

```
# gcc -Wall …
# ./a.out
The 16th char in C is 1
```

Why?

- Integers can be represented in various way over 4 bytes
  - gcc uses **little-endian** format

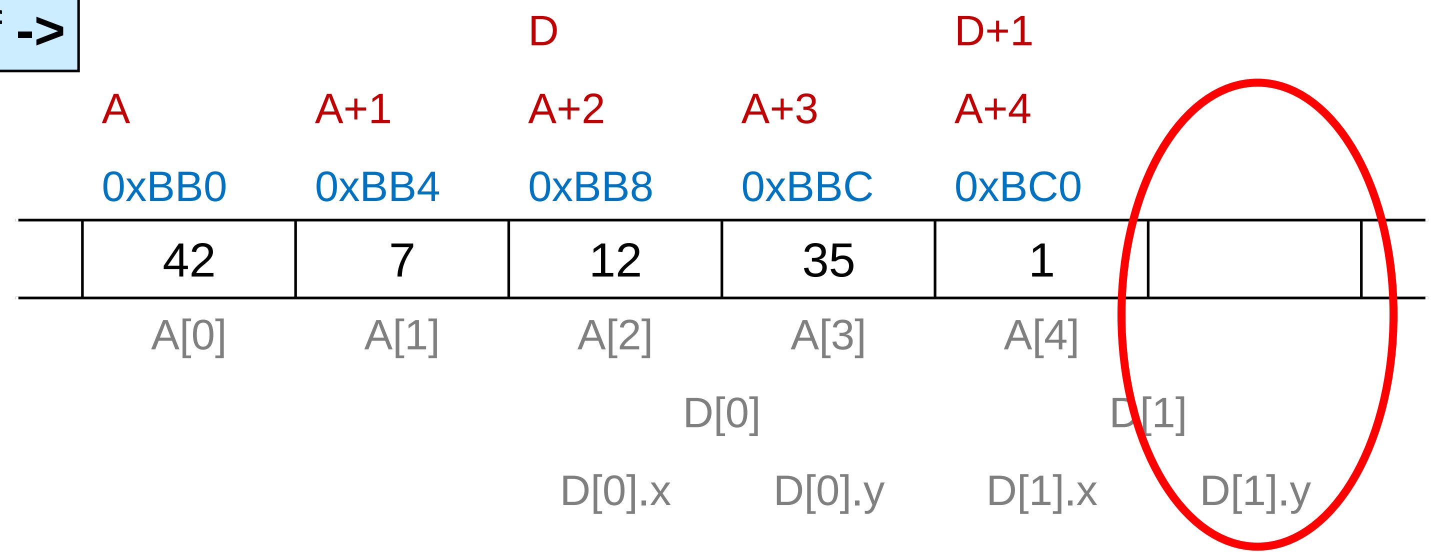
The most significant byte has the highest address

Casting Pointers

D[0] and D[1] are *not pointers*, so we need to use **.** instead of **->**

```
struct point {
   int x;
   int y;
 };
…

 struct point *D = (struct point *)(A + 2);
 printf("(x0,y0) = (%d, %d)\n", D[0].x, D[0].y);
 printf("(x1,y1) = (%d, %d)\n", D[1].x, D[1].y);
```



- As an an array, each element of D is two ints
  - accessing D[1].y is the same as accessing A[5]
    - out of bounds
    - undefined behavior
- When casting pointers, we must be mindful of alignment

Casting Pointers

```
struct thermonuclear_device_controller {
   …
};
…

  struct thermonuclear_device_controller *danger = (struct thermonuclear_device_controller*)(A + 2);
  activate(danger[17].warhead);
```

- Careless casting can be outright dangerous

In practice,

- cast a pointer of arbitrary type to void* or char* only
  - accessing pointers cast to other types is undefined behavior

Casting to void*

- In C1, void* stands for a pointer of any type
  - this is the basis for building **generic data structures**
    - as long as the elements are pointers

- In C, void* is also the type of an array of … void
    - but **void is not a type** in C
  - void* can be viewed as **the address of the first element of any array**
    - there is no way to infer the size of the elements
    - nor the number of elements

- With this, we can write generic operations on arrays with arbitrary elements
  - not just pointers

Generic Array Operations

- We can write generic operations on arbitrary arrays by
  - casting their address to void*
  - specifying the element size
  - specifying the number of elements

- Example: a generic sort function

```
void sort(void *A, int elem_size, int num_elem, compare_fn *cmp);
```

The array to be sorted, as a void*

The number of bytes of the elements of A

The number of elements of A

A function to compare elements

Stack Allocation

Stack-allocated Arrays

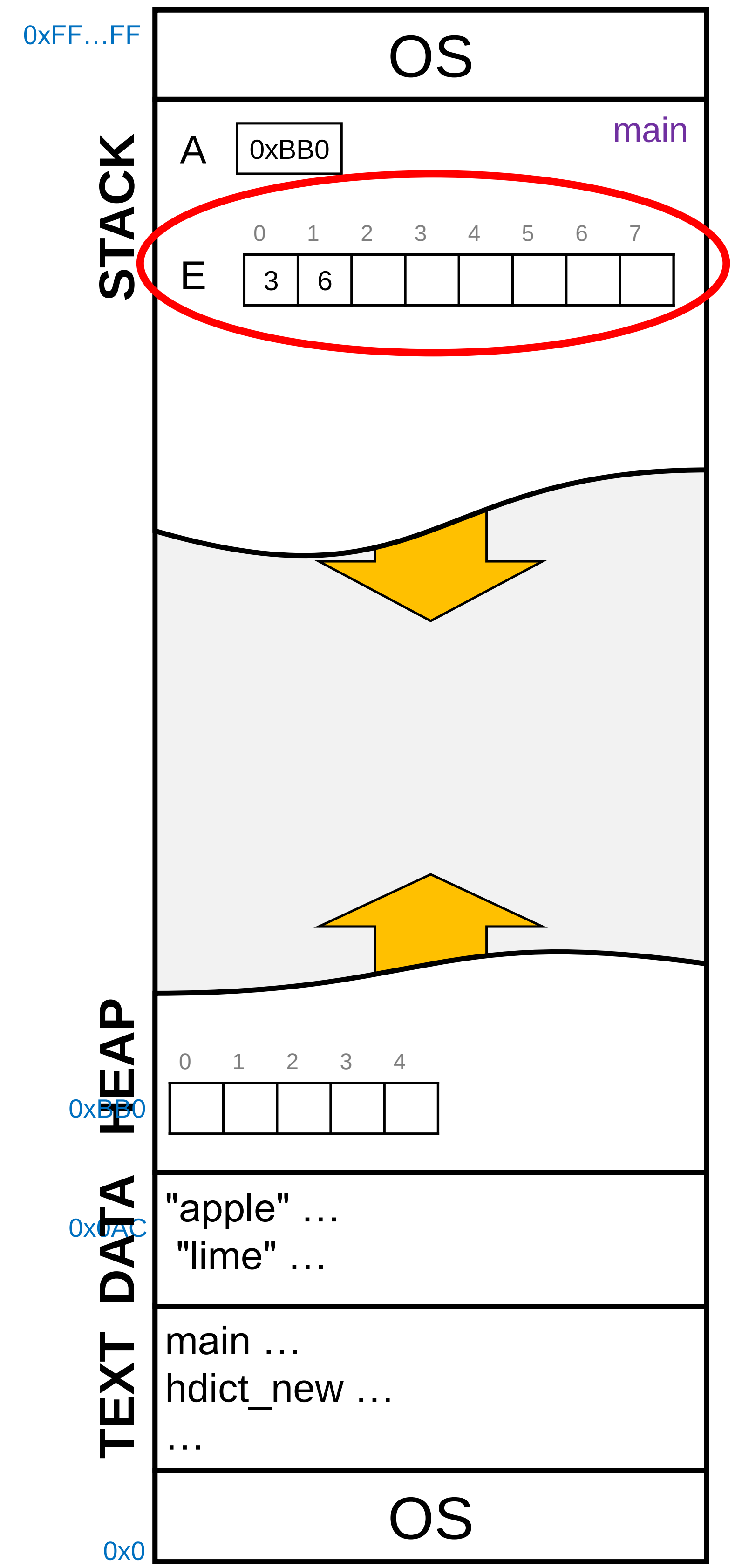
- In C0, arrays can only live on the heap
- C allows creating arrays on the stack
  - these are **stack-allocated arrays**
- The instruction

  int E[8];

  allocates an 8-element int array on the stack
  - It is accessed using the normal array notation

    E[0] = 3;

    E[1] = 2 * E[0];



Stack-allocated Arrays

- Stack-allocated arrays can be initialized to **array literals**

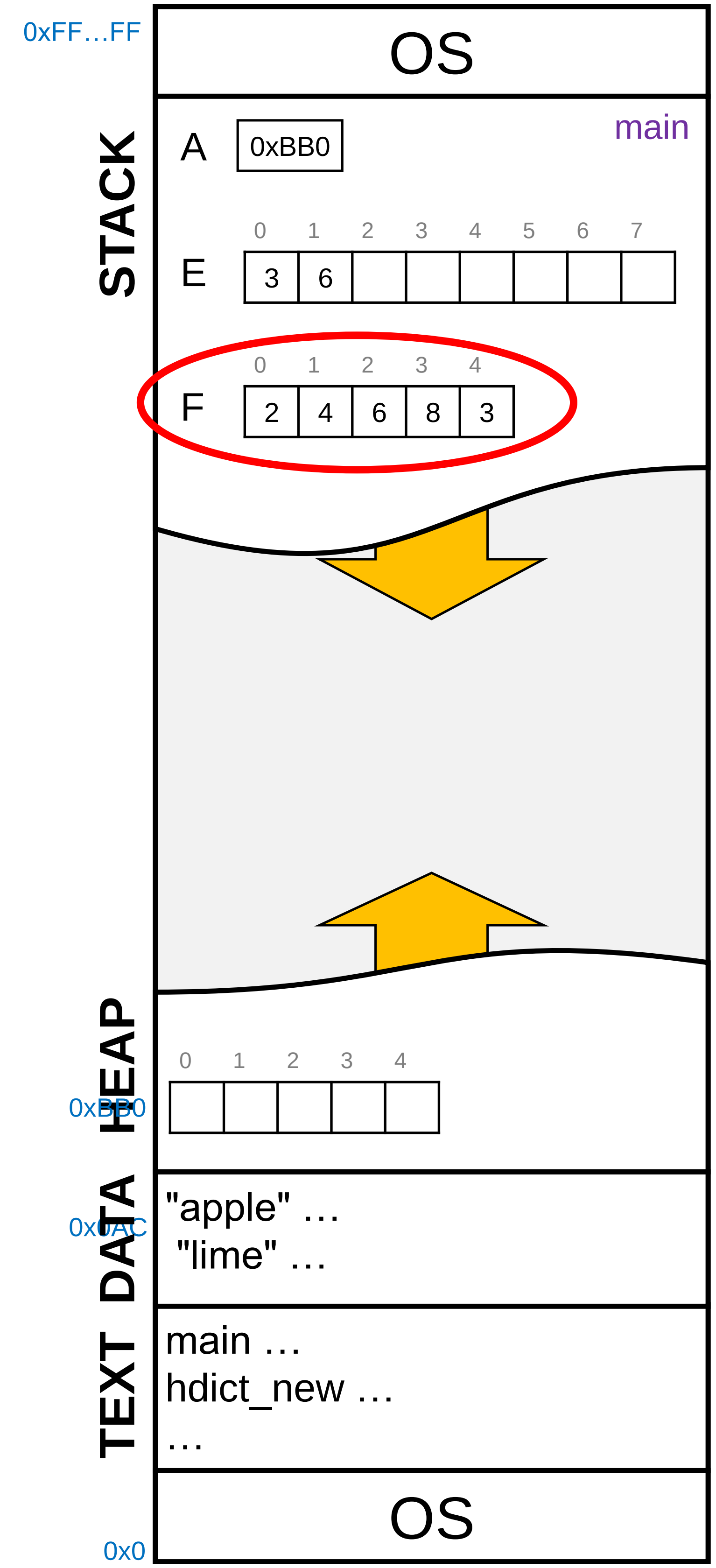
  `int F[] = {2, 4, 6, 8, 3};`

  The compiler will figure out the size of the array

  The initial elements of F

  allocates a 5-element int array on the stack and initializes with the given values

- Array literals are really useful to write test cases
  - but they cannot be very big



Stack-allocated Structs

- Similarly, C allows allocating structs on the stack

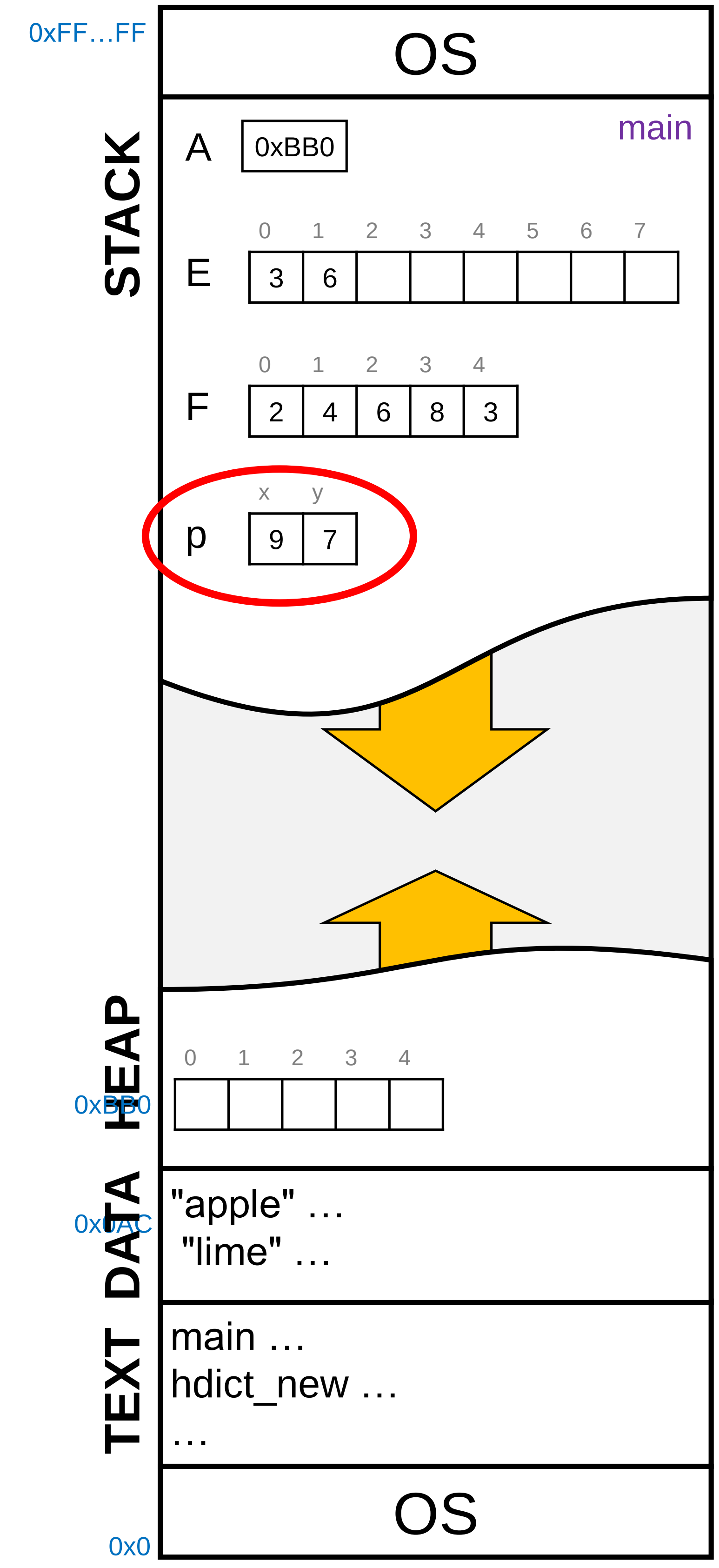
    struct point p;

    - and we can conveniently initialize them

    struct point q = { .x = 15, .y=122 };

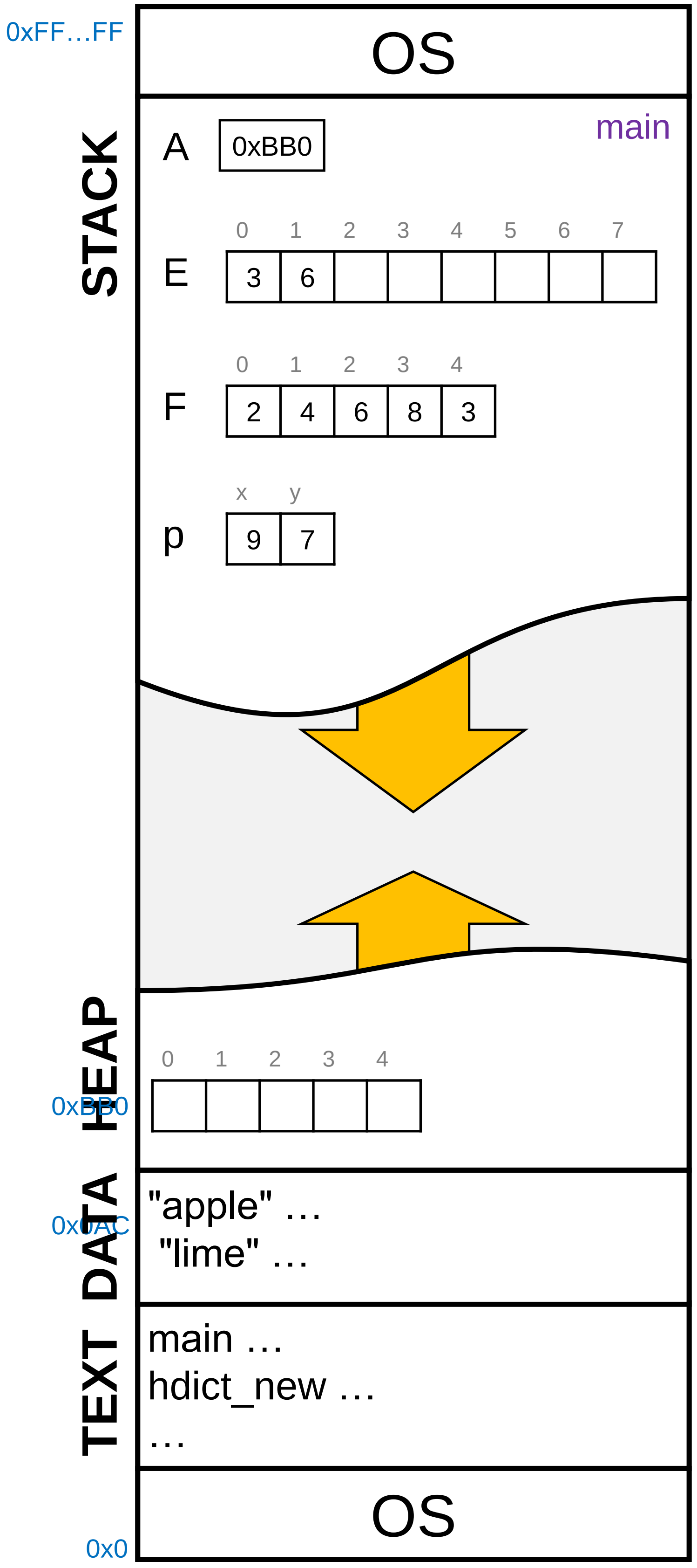
- Stack-allocated structs are **not pointers**
    - their fields must be accessed using the **dot notation**

```
p.x = 9;
p.y = 7;
printf("p is (%d, %d)\n", p.x, p.y);
```



Disposing of Stack-allocated Data

- The space for stack-allocated arrays and structs is reclaimed when exiting the function that declared them
  - **No need to free them**
  - In fact, this is undefined behavior!
- Because of this they cannot be used for traditional data structures
  - if queue_new were to allocate a queue on the stack, other queue functions wouldn't be able to use it when it returns
    - ➢ Traditional queues must be heap-allocated



Address-of

Capturing Memory Addresses

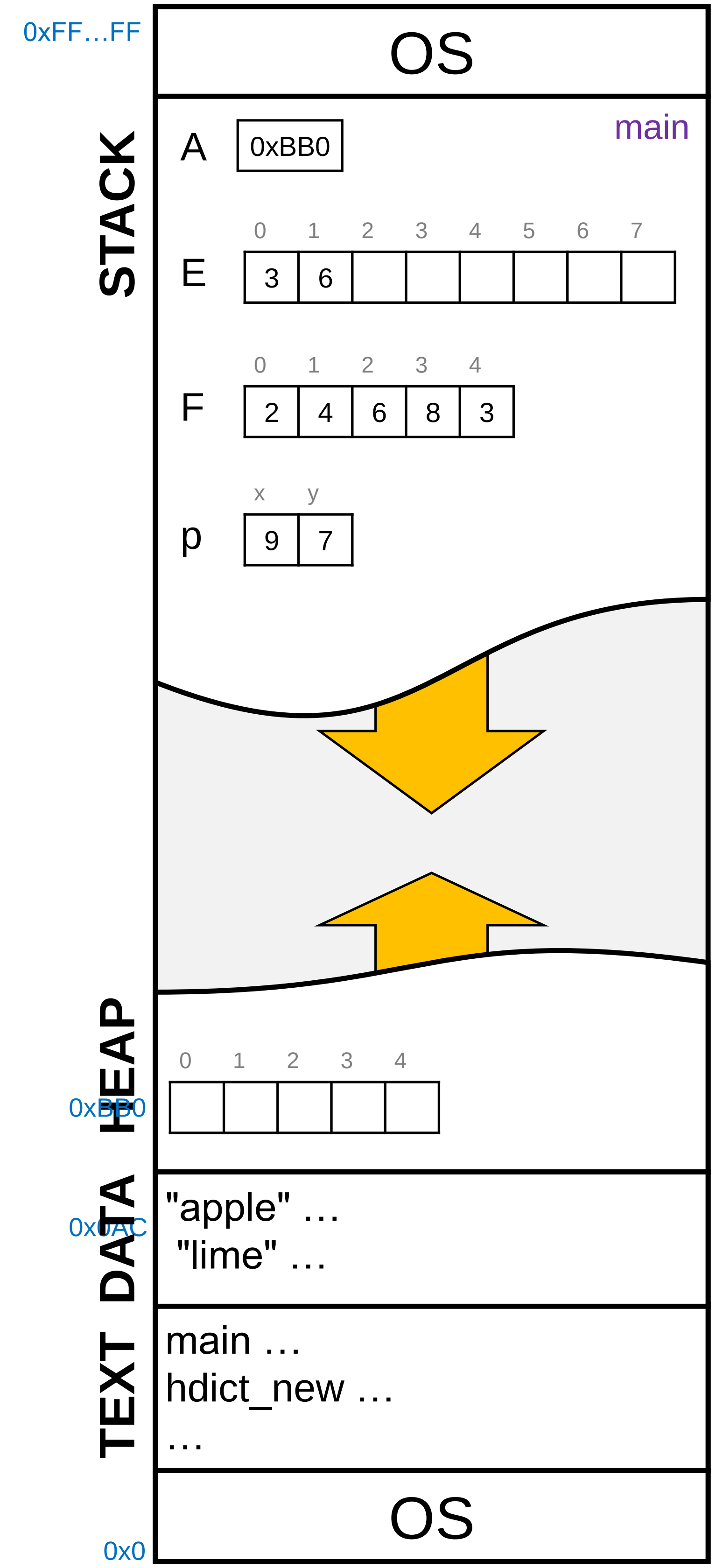
- In C1, & can **only** be used on function names
- In C, & can get the address of **anything that has a memory address**
  - functions
  - local variables
  - fields of structs
  - array elements
- In general, for any exp for which

  exp = …

  is syntactically valid, we can write

  &exp

Such exp are called **l-values**



Capturing Memory Addresses

Increments an int* by 1

```
void increment(int *p) {
  REQUIRES(p != NULL);
  *p = *p + 1;
}
```

- local variables

  ```
  int i = 11;
  increment(&i);
  ```
  i is now 12

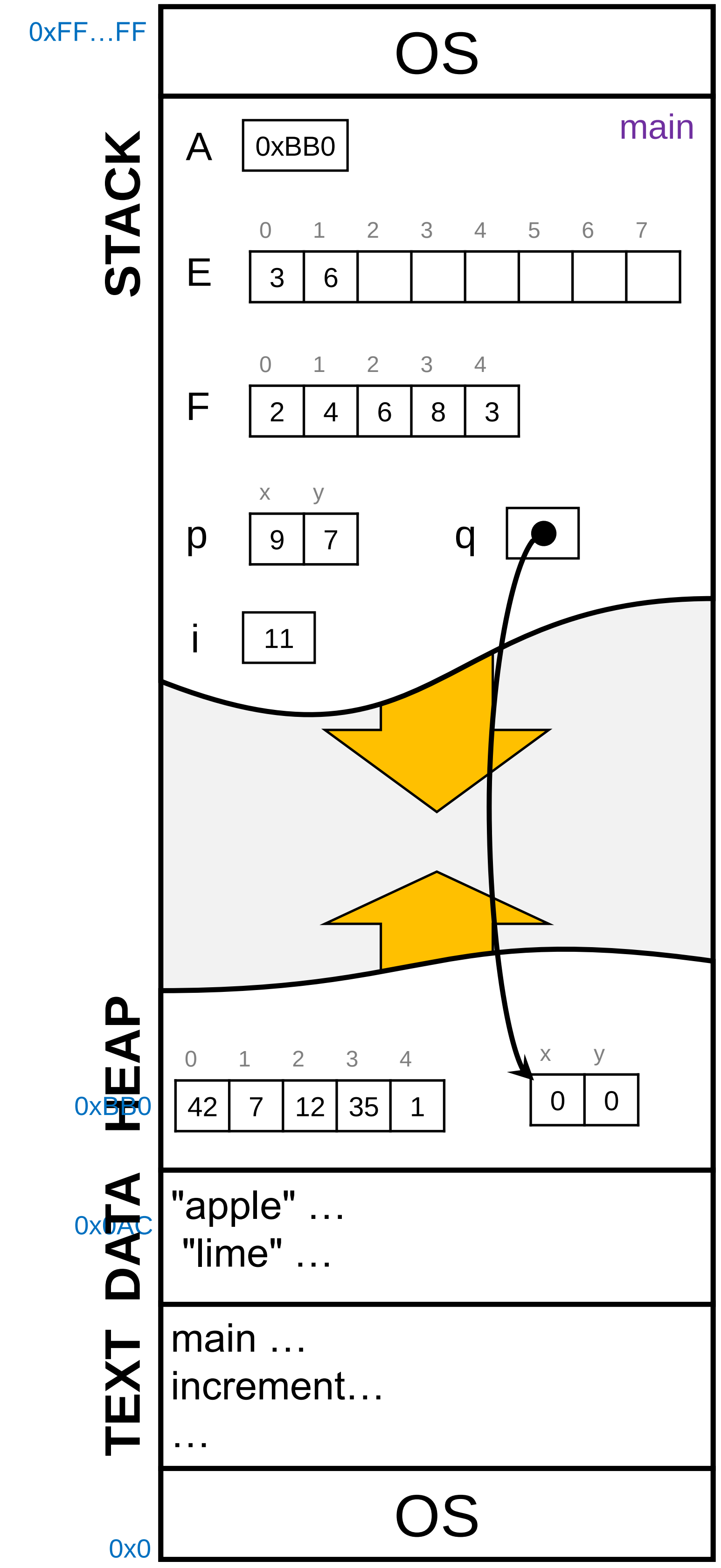
- fields of structs

  ```
  increment(&p.y);
  struct point *q = calloc(1, sizeof(struct point));
  increment(&(q->y));
  ```
  p.y is now 8

  Initializes q to (0,0)

  q->y is now 1

- array elements
  - increment(&A[3]); — A[3] is now 36
  - increment(&F[2]); — F[2] is now 7



Pointer Arithmetic

- All code using pointer arithmetic can be rewritten without
  - Code is more readable
  - and has fewer bugs

- Change
  - `*(A + i)` to `A[i]`
  - `A + i` to `&A[i]`

Bad Uses of Address-of

- *In general, for any exp for which*

  exp = …

  is syntactically valid, we can write

  &exp

  - &(i+2) ✘
    - i+2 = 7; is not legal
  - &(A+3) ✘
    - A+3 = xcalloc(4, sizeof(int)); is not legal
  - &&i ✘
    - &i = xmalloc(sizeof(int)); is not legal

Really Bad Uses of Address-of

```
int* bad() {
  int a = 1;
  return &a;
}
```

- Returns the address of a stack value that will be deallocated upon return!
  - The next function call will overwrite it
- This is a huge security vulnerability

Recent versions of gcc stopped allowing it

Strings in C

Strings

- There is no type string in C

- Strings are just **arrays of characters**
  - of type char*
  - The string syntax

    "hello"

    is just convenience syntax for an array containing 'h', 'e', …

- Given

  ```
  char *s1 = "hello";
  ```

  the statements

  ```
  printf("%c%c%c%c%c\n", s1[0], s1[1], s1[2], s1[3], s1[4]);
  printf("%s\n", s1);
  ```

  produce the exact same output

NUL

```
char *s1 = "hello";
printf("%s\n", s1);
```

- How does printf know when to stop printing characters?
  - the length of an array is recorded nowhere
- The end of a string is indicated by the **NUL character**
  - written '\0'
  - whose value is 0
- Thus, s1 is an array of **six** characters and s1[5] == '\0'

The <string> Library

- The <string> library contains lots of useful functions to work with strings
  - strlen returns the number of characters in a string
    - up to the first NUL character, excluded

```
char *s1 = "hello";
assert(strlen(s1) == 5);
```

    - s1 is an array of 6 characters but it has length 5

This is an endless source of bugs

  - strcpy(dst, src) copies all the characters of string src to dst
    - up to the NUL character, included
    - dst must be big enough to store all the characters in src **plus NUL**

This is an endless source of bugs

  - and many more utility functions

Strings

- Strings can live in three places
  - in the DATA segment

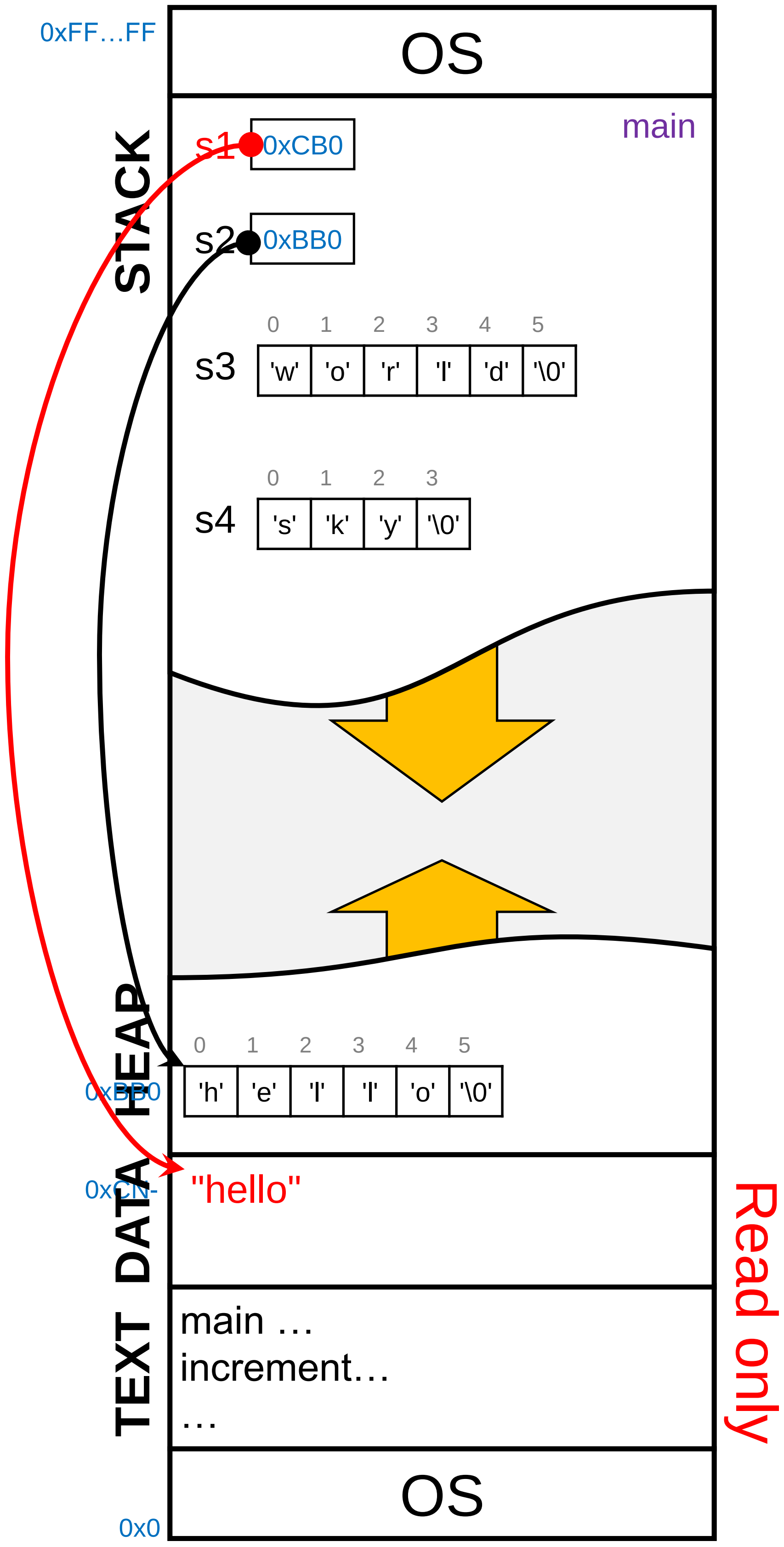
    `char *s1 = "hello";`

    - these strings are **read-only**

      `s1[0] = 'm';` ✘

      is undefined behavior
    - no need to free them

      in fact, that's undefined behavior
  - in the heap
  - on the stack



Strings

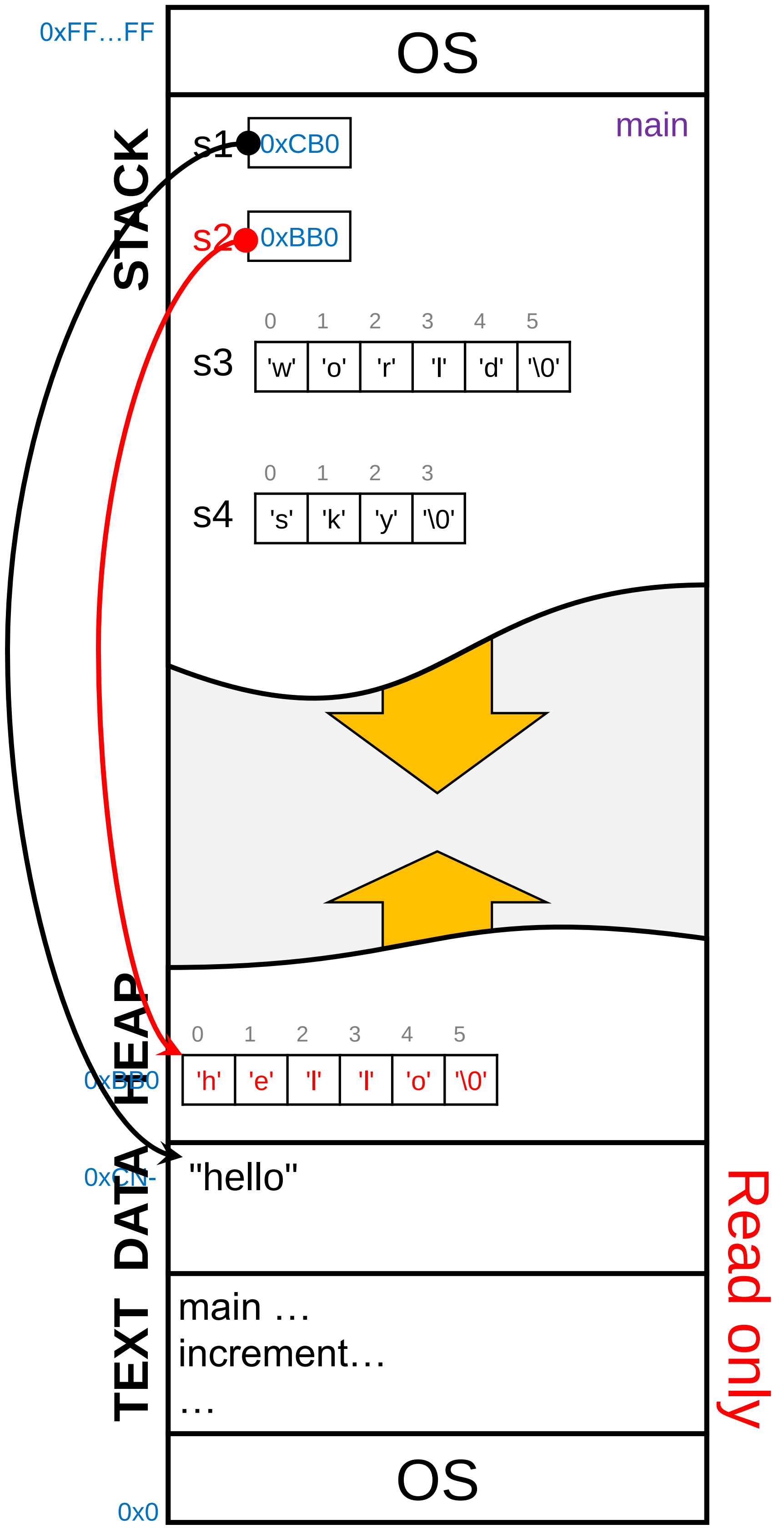
- Strings can live in three places
  - in the DATA segment
  - in the heap


```
char *s2 = xmalloc(strlen(s1) + 1);
strcpy(s2, s1)
s2[0] = 'Y';
free(s2);
```

    - we need to allocate one extra character for the NUL terminator
    - we need to free them

This is an endless source of bugs

  - on the stack



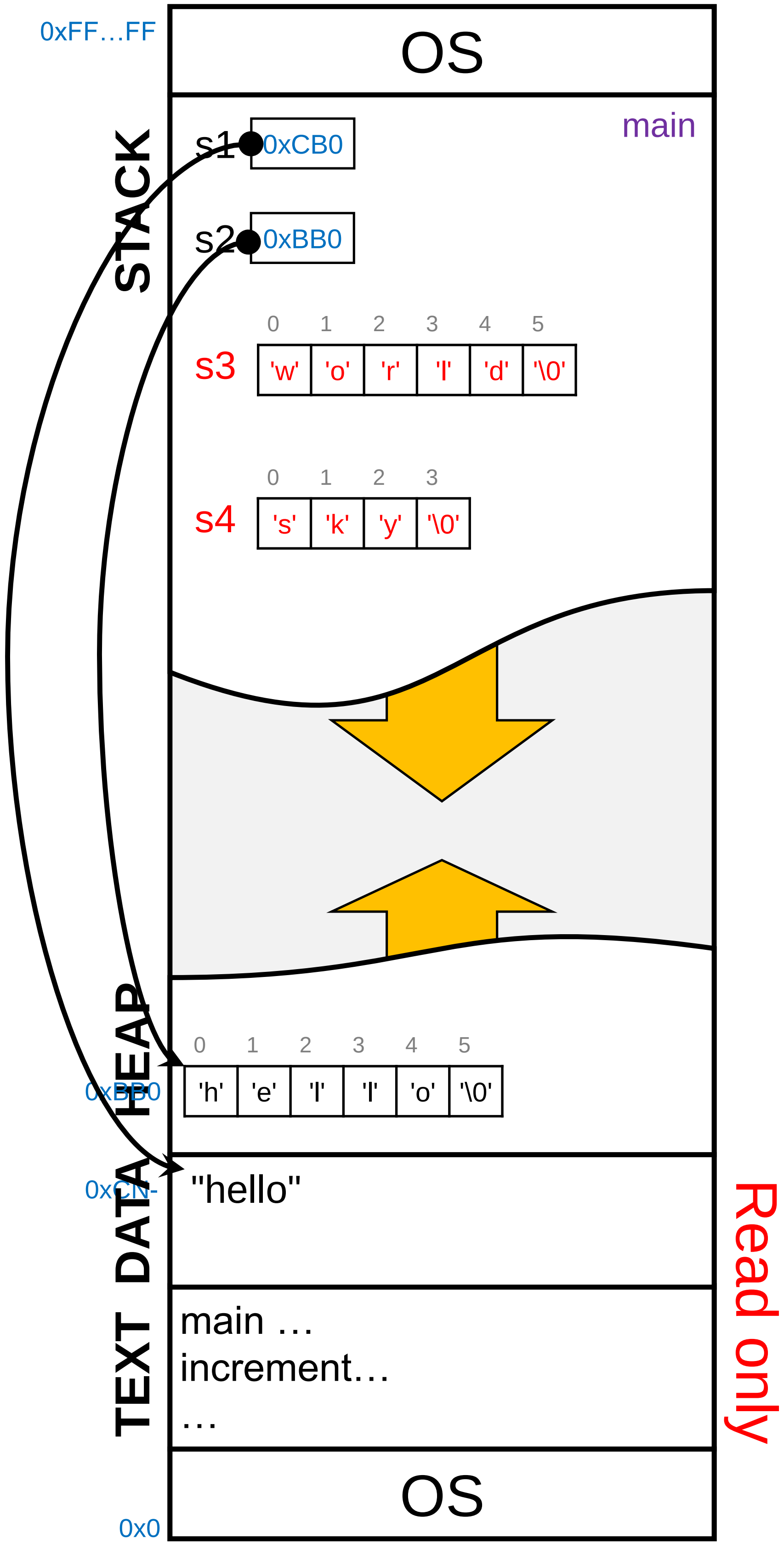
Strings

- Strings can live in three places
  - in the DATA segment
  - in the heap
  - on the stack

    char s3[] = "world";

    char s4[] = {'s', 'k', 'y', '\0'};

    - if using array literals, we often need to include the NUL terminator
    - no need to free them



Strings in Summary

- Strings can live in three places
  - in the DATA segment

    char *s1 = "hello";

  - in the heap

    char *s2 = xmalloc(strlen(s1) + 1);

    strcpy(s2, s1)

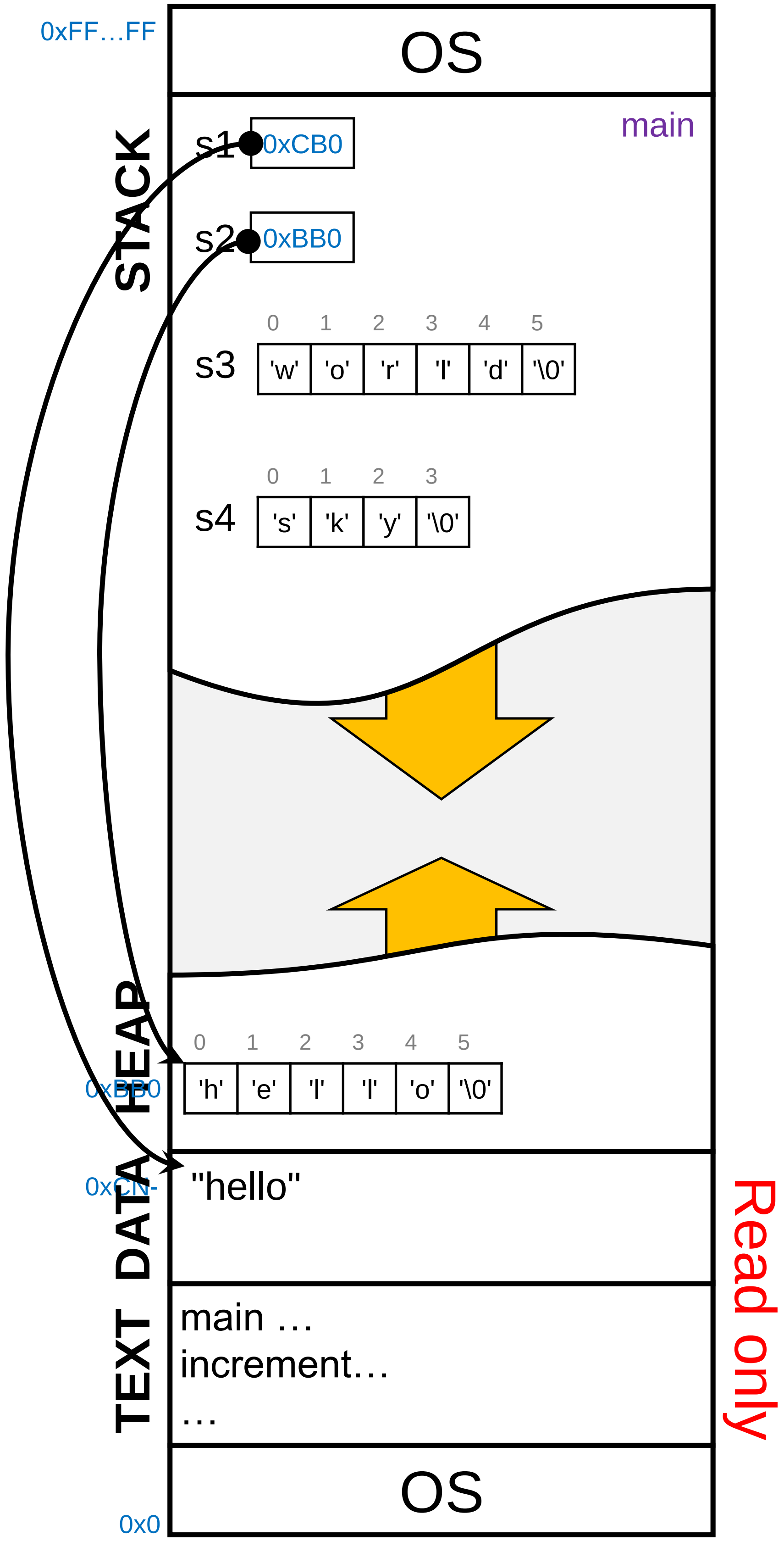
    s2[0] = 'Y';

    free(s2);

  - on the stack

    char s3[] = "world";

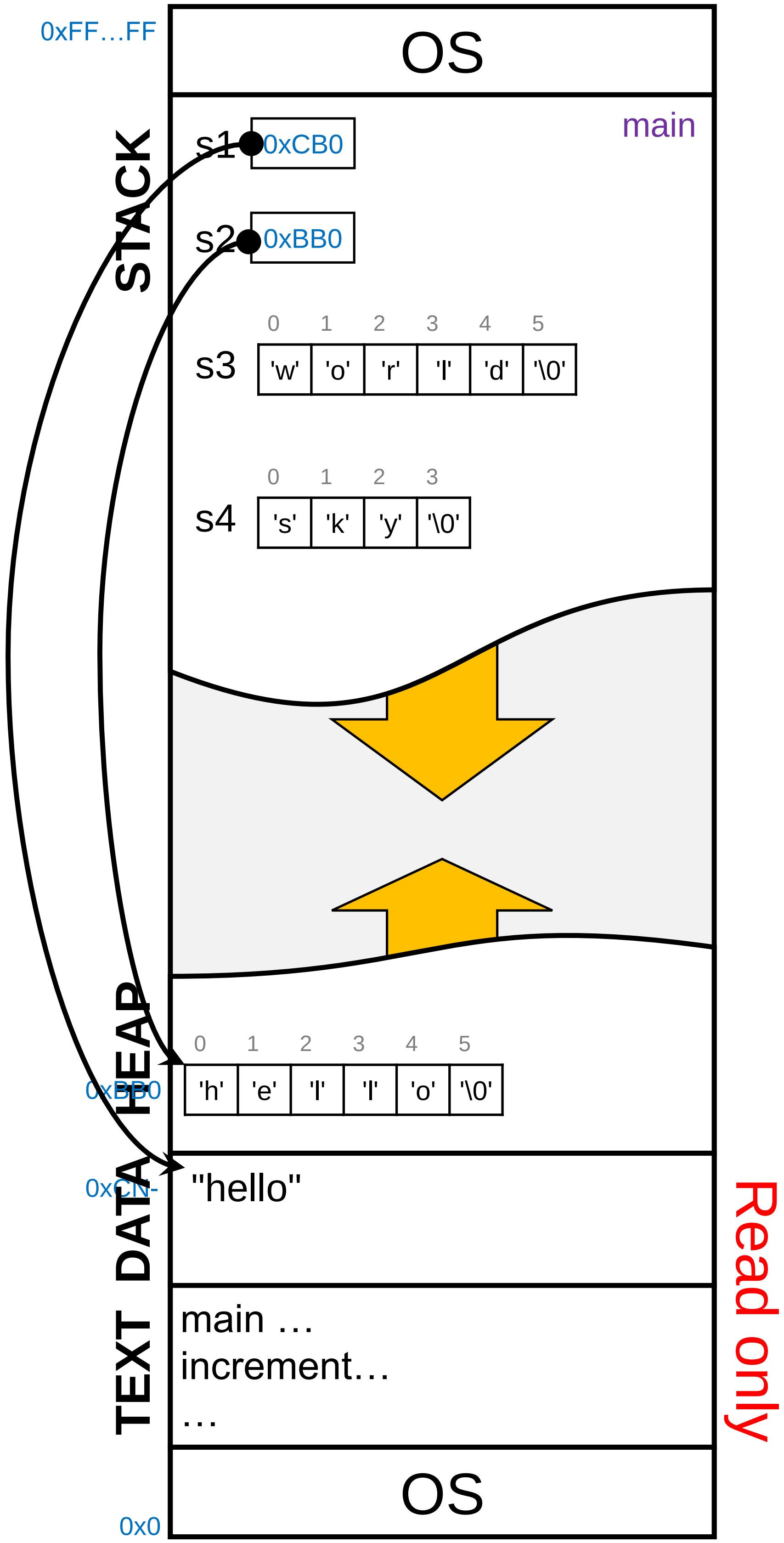
    char s4[] = {'s', 'k', 'y', '\0'};



Strings in Summary

- Strings can live in three places

	Writable?	*Allocation*	*Deallocation*
DATA	No	Automatic (when execution starts)	N/A
Stack	Yes	Automatic (when function is called)	Automatic (when function returns)
Heap	Yes	Manual (with malloc)	Manual (with free)



Summary

Undefined Behavior

- Reading/writing to non-allocated memory
- Reading uninitialized memory
  - even if correctly allocated
- Use after free
- Double free
- Freeing memory not returned by malloc/calloc
- Writing to read-only memory

Balance Sheet

Lost	*Gained*
• Contracts • Safety • Garbage collection • Memory initialization • Well-behaved arrays • Fully-defined language • Strings	• Preprocessor • Undefined behavior (?) • Explicit memory management • Separate compilation • Pointer arithmetic (?) • Stack-allocated arrays and structs • Generalized address-of