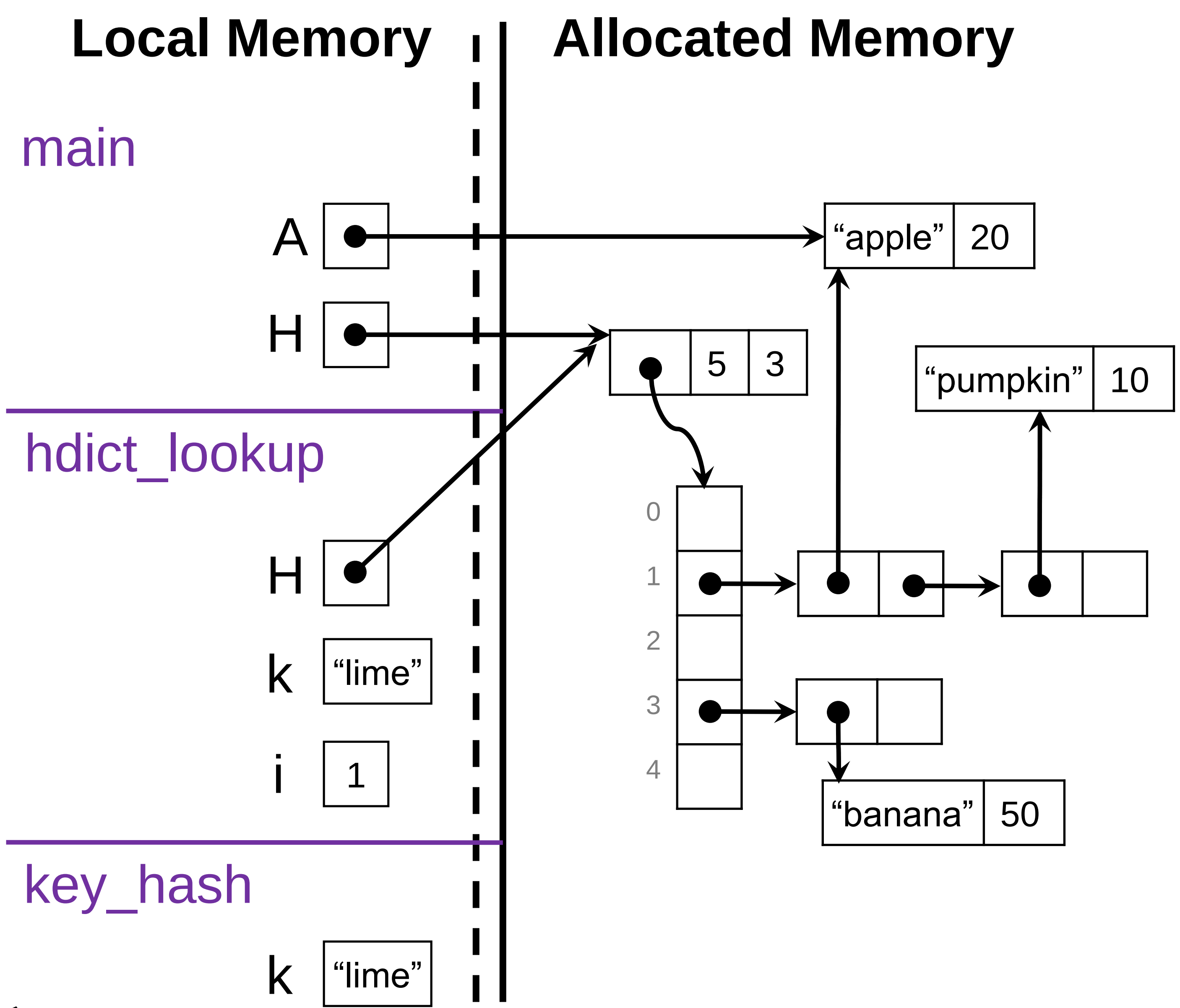
Function Pointers

Refined Memory Model

The C0 Memory Model … so far

Two memories

- **Local memory**
  - one frame per function call
  - frame contains
    - its parameters
    - its local variables
- **Allocated memory**
  - arrays
  - pointer targets

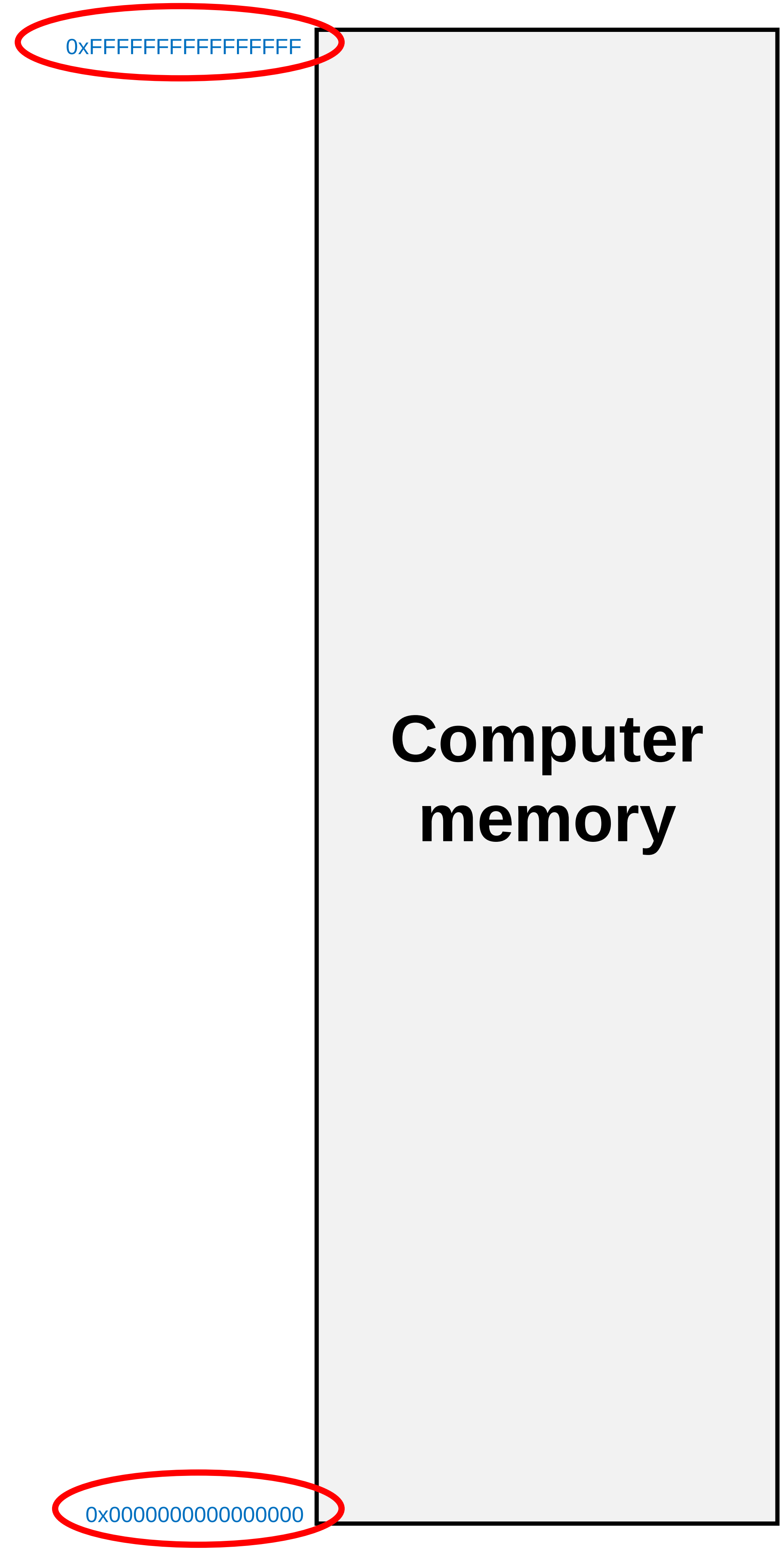


Sample memory snapshot
during the execution of an application
that uses a hash dictionary

A More Realistic Model

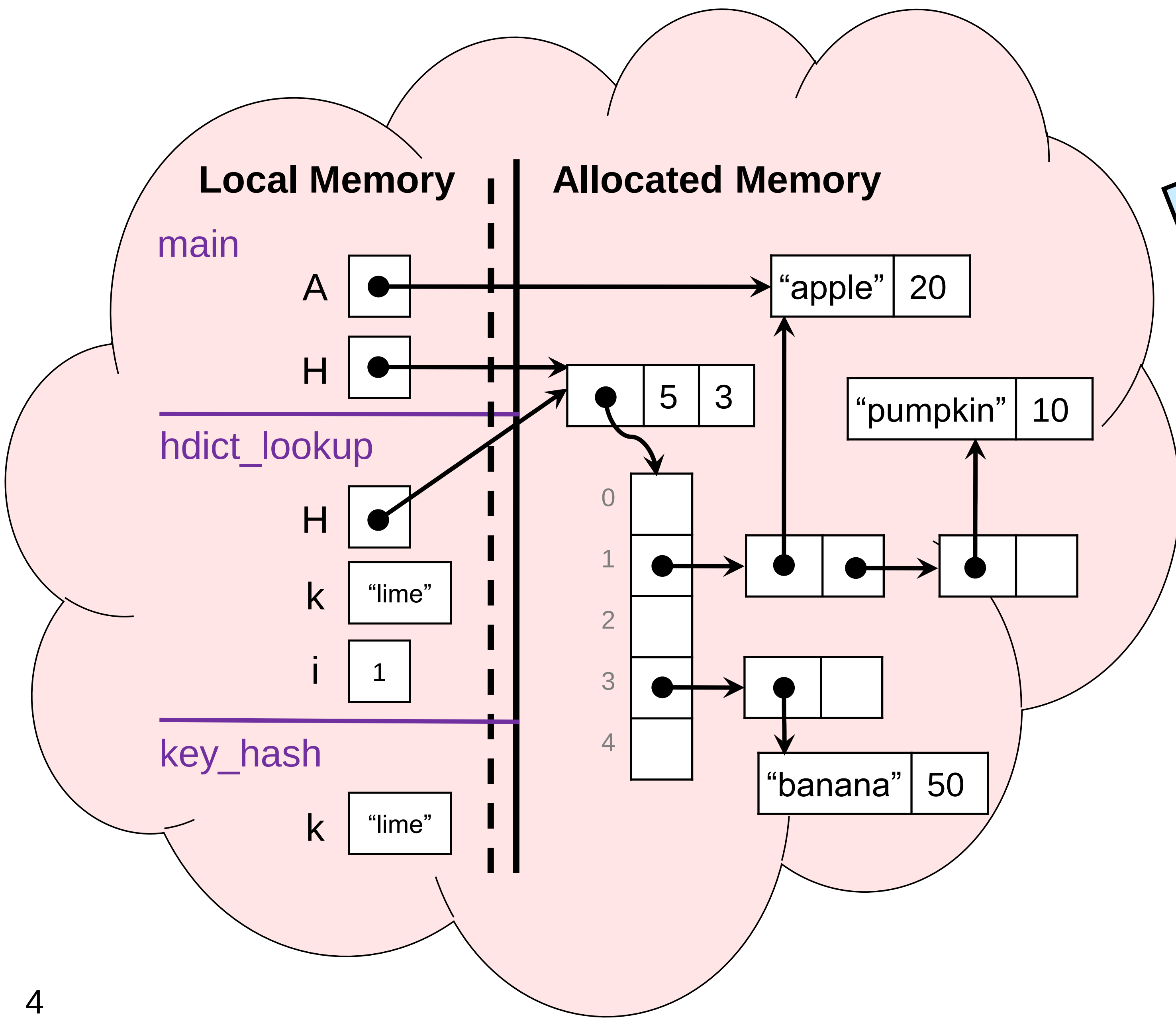
- *Two distinct memories?*
  - local memory and allocated memory
- But a computer has **one** memory
  - a large array of bytes indexed by **addresses**
- C0 addresses are 64 bit long
  - 2^{64} bytes
  - the smallest byte has address 0x0000000000000000
  - the largest byte has address 0xFFFFFFFFFFFFFFFF

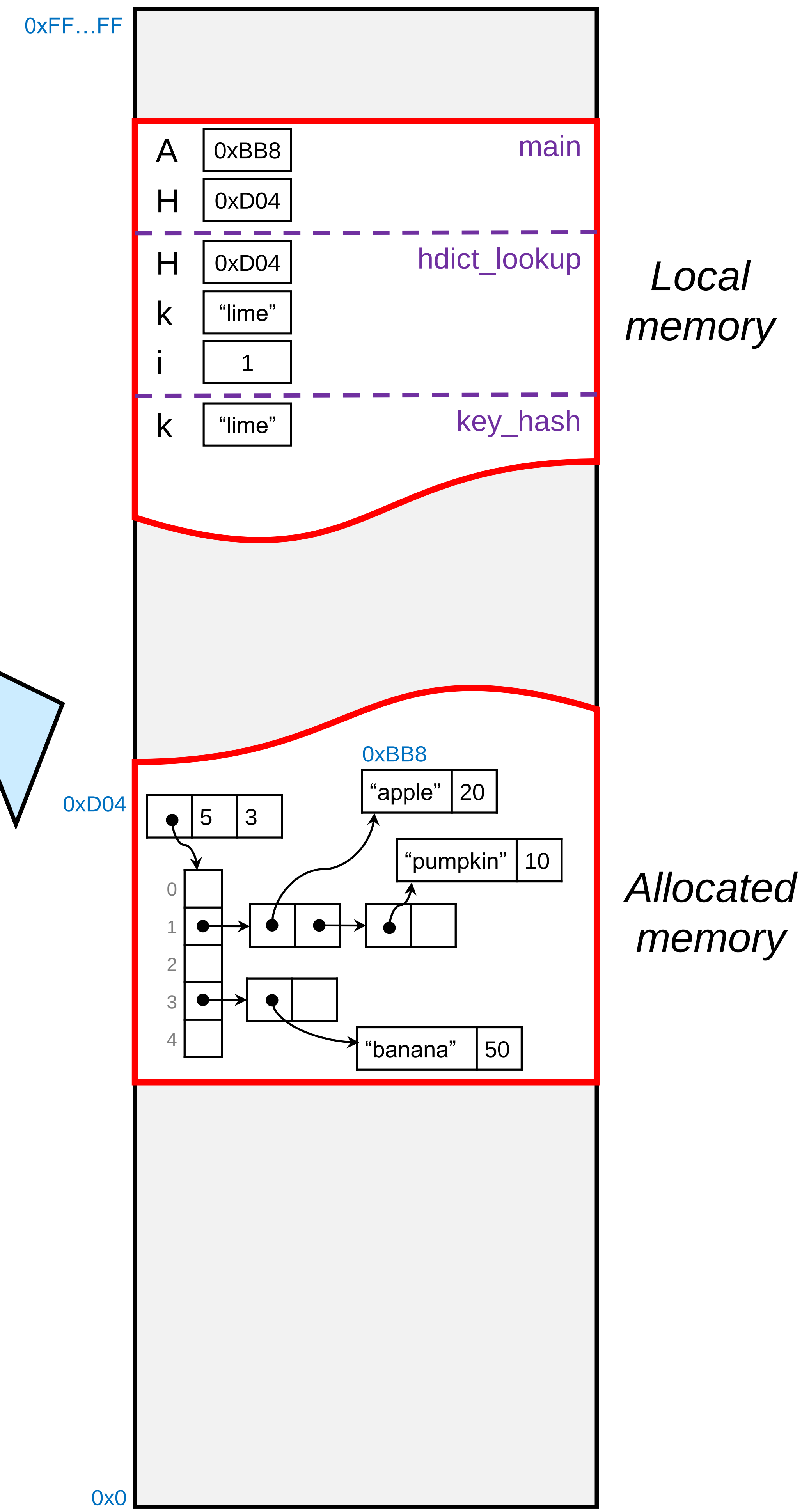
This is 2^{64}-1



A More Realistic Model

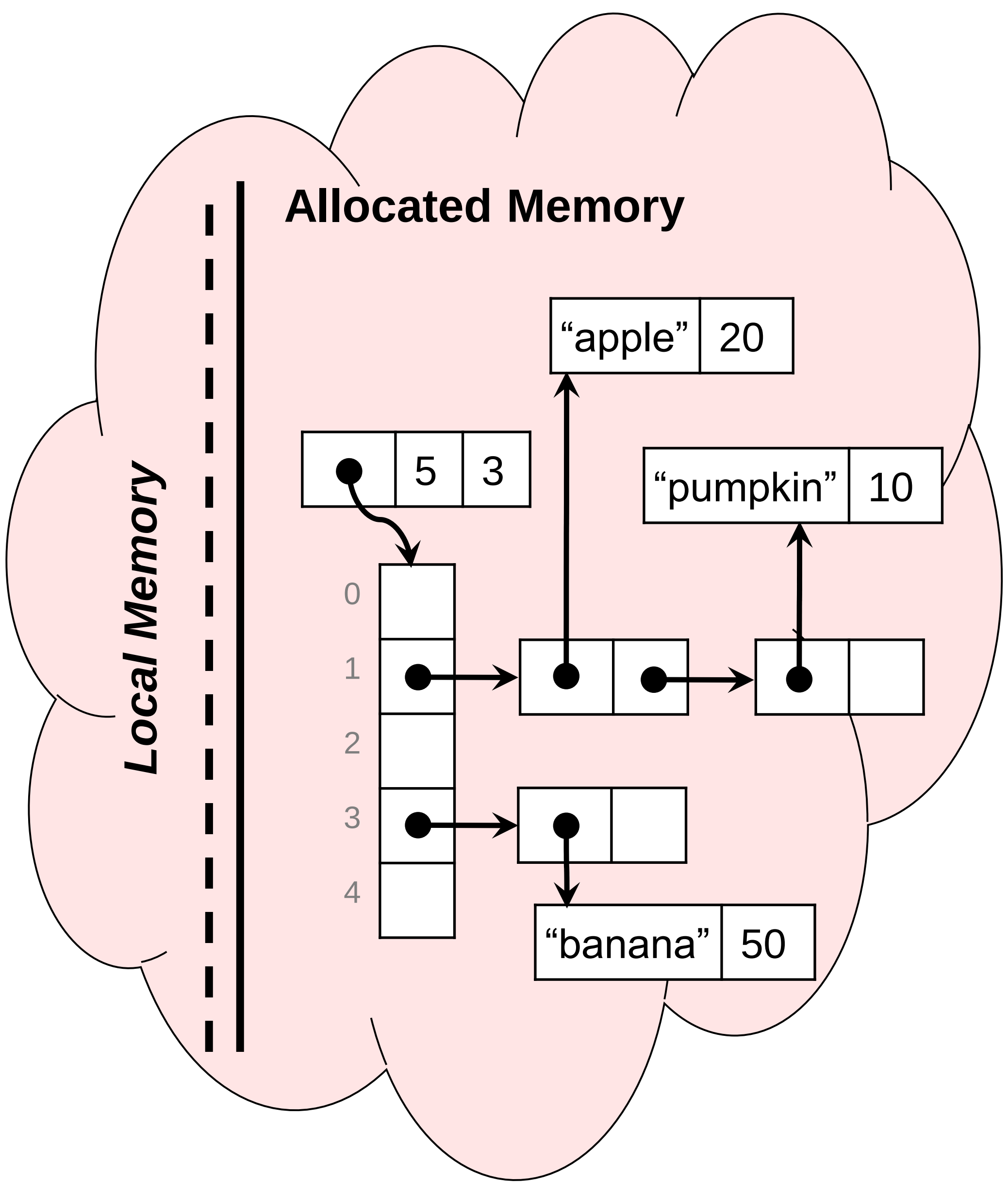
- Local and allocated memory are two **segments** in this memory

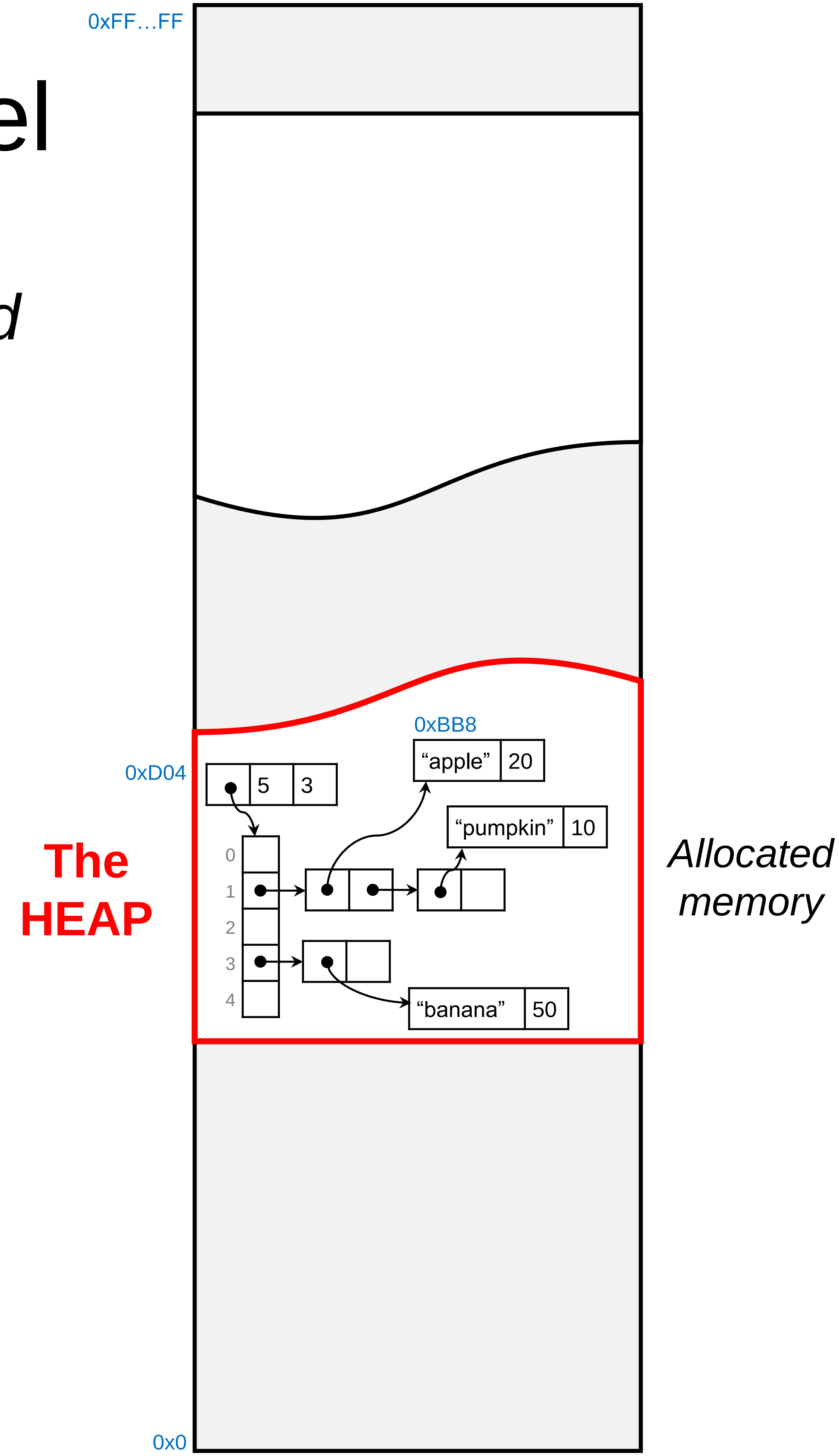




A More Realistic Model

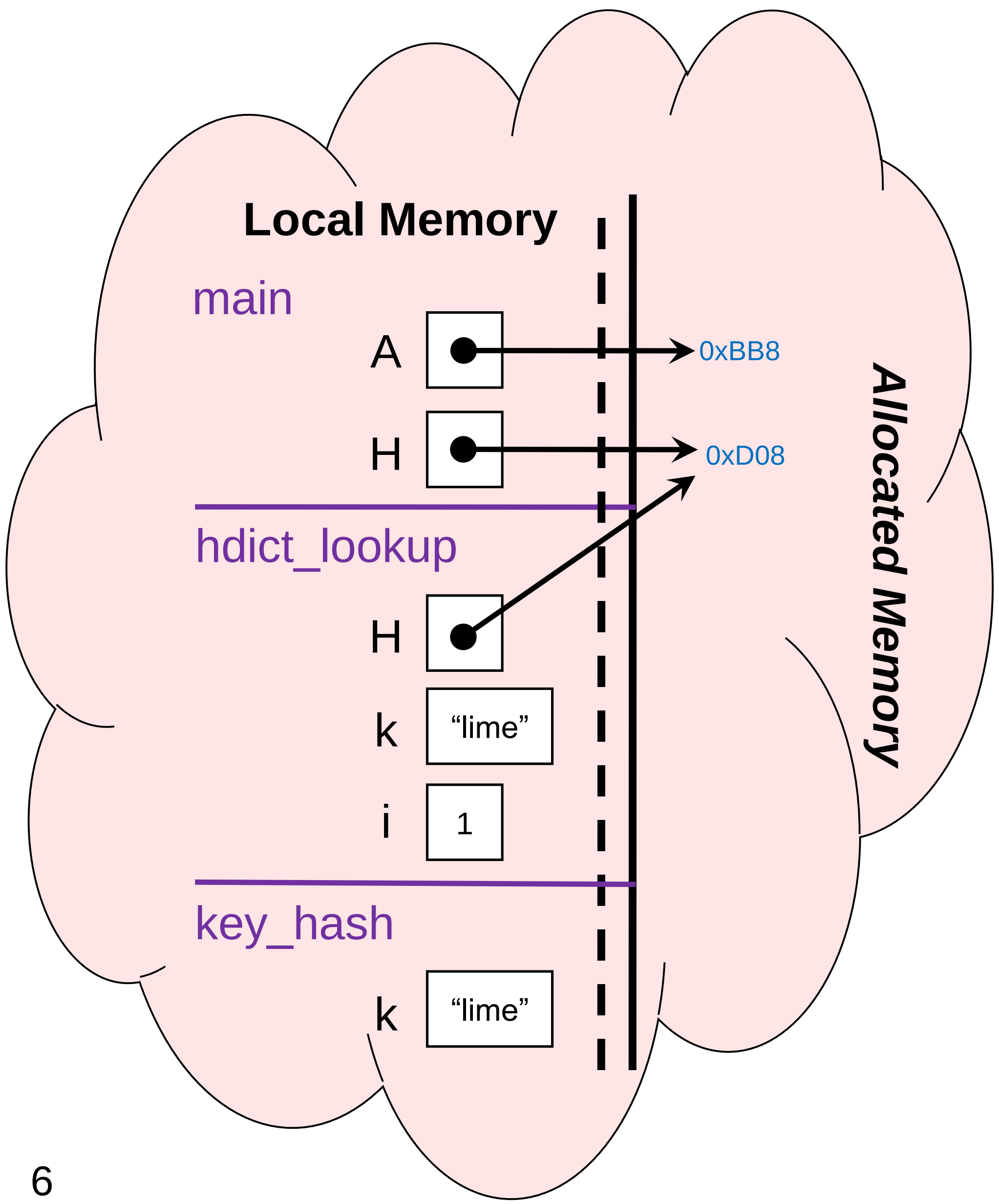
- The segment where the *allocated memory* lives is called **the heap**
  - it contains a pile of data structures

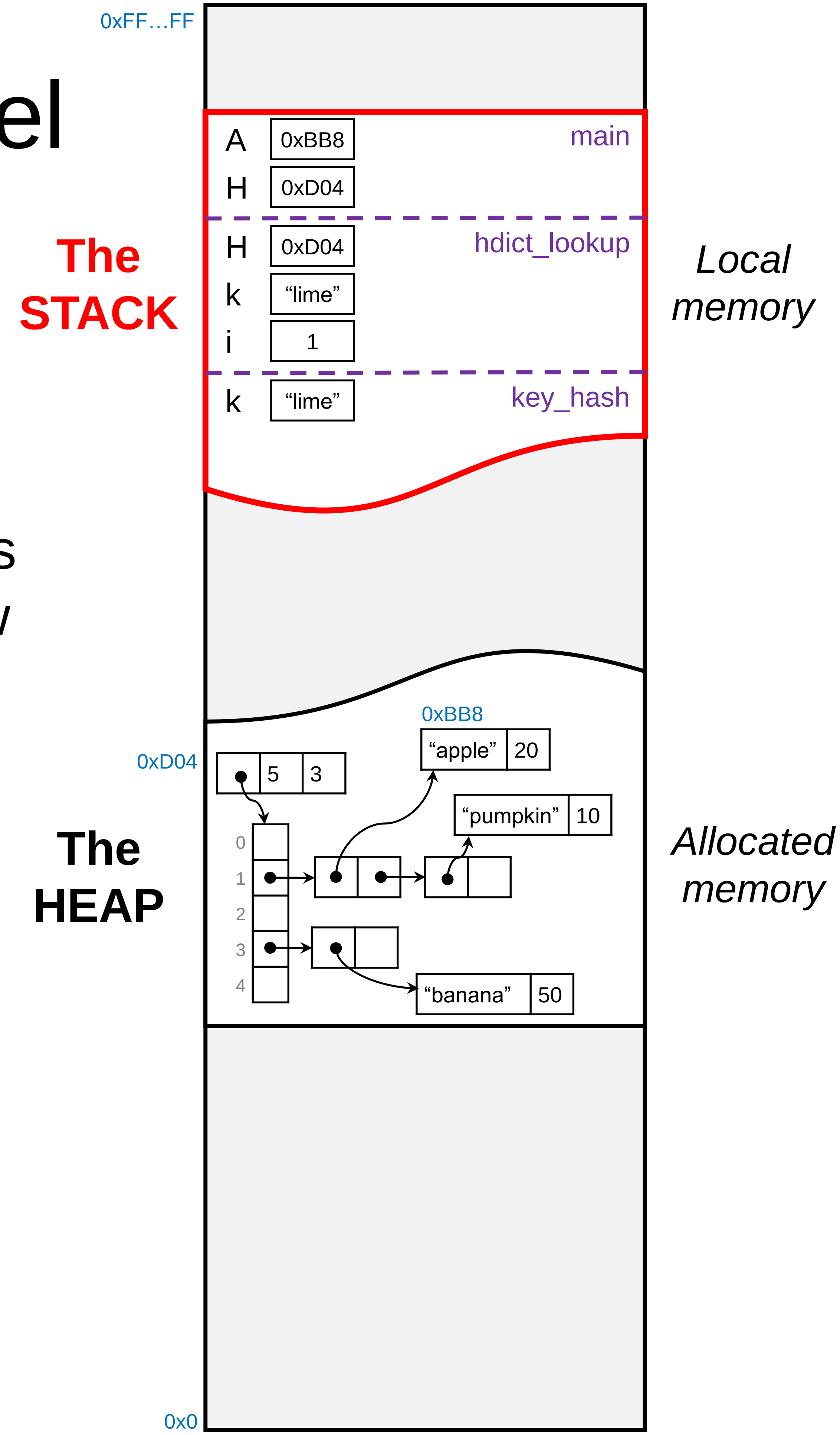




A More Realistic Model

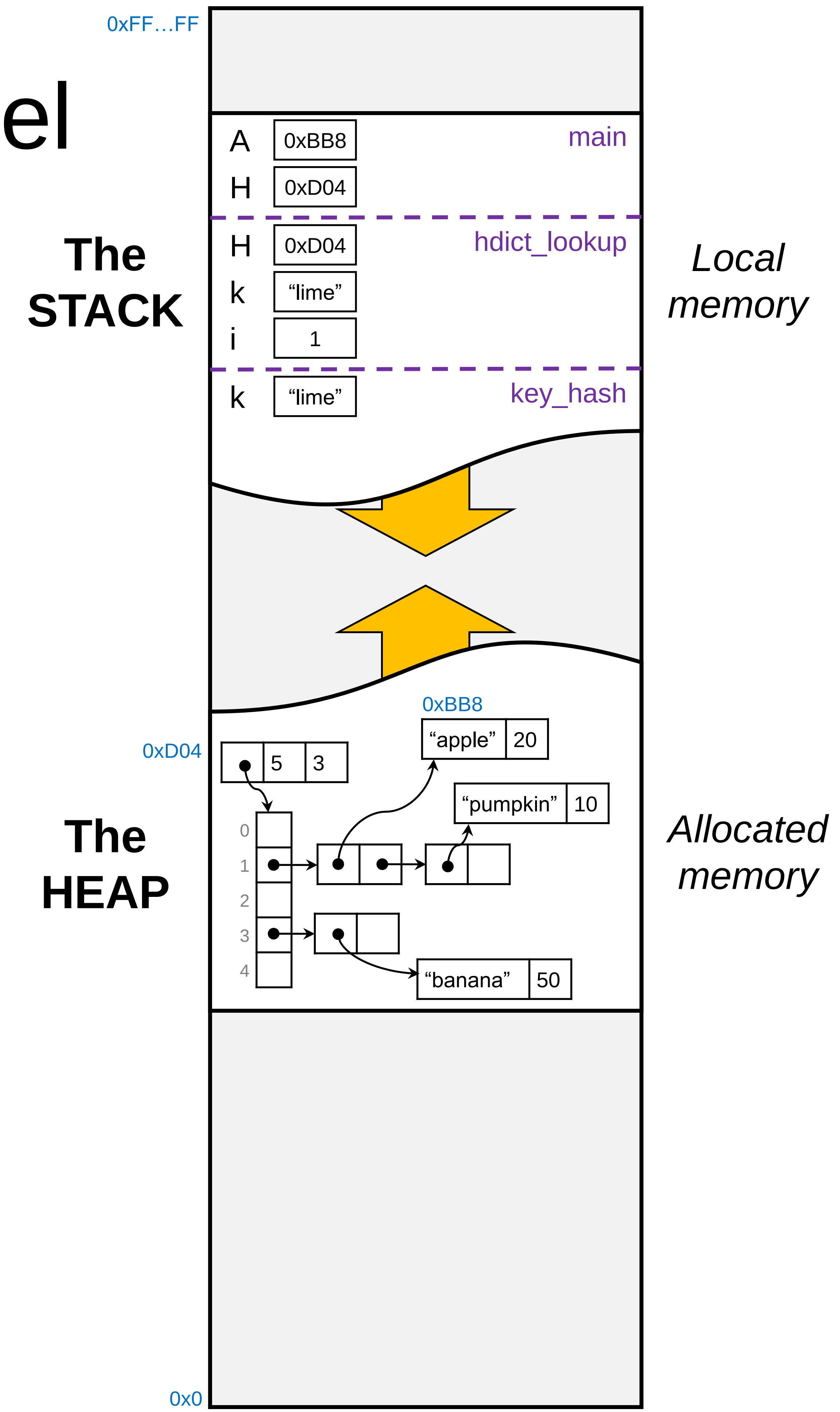
- The segment where the *local memory* lives is called **the stack**
  - function calls make it grow and shrink like a stack





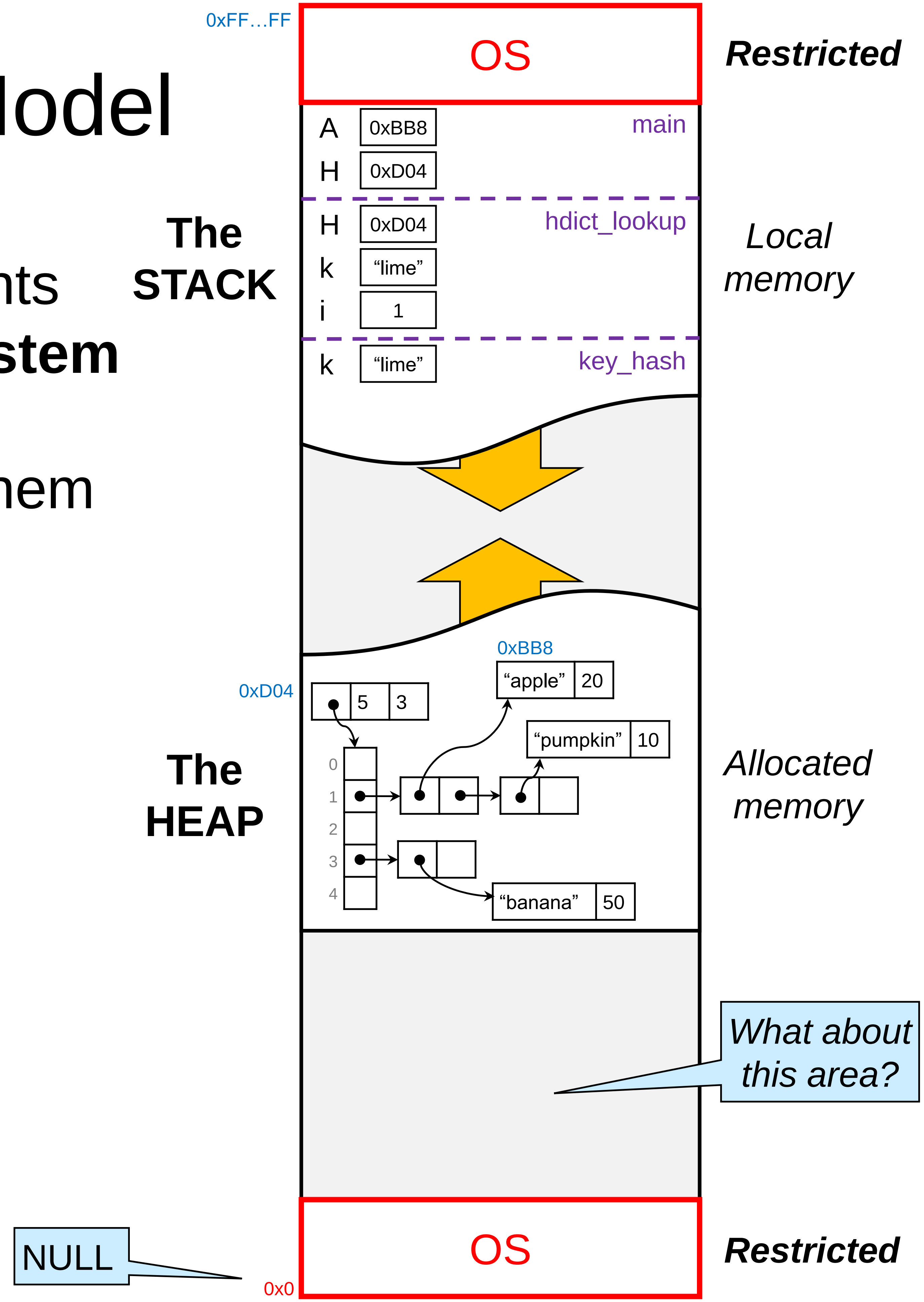
A More Realistic Model

- The stack grows downward
  - toward smaller addresses
- The heap grows upward
  - toward larger addresses
    - ➢ unless garbage collection has given back existing heap space
- If they grow so much that they run into each other, we have a **stack overflow**
    - ➢ very rare with modern hardware
  - *What about the rest of memory?*



A More Realistic Model

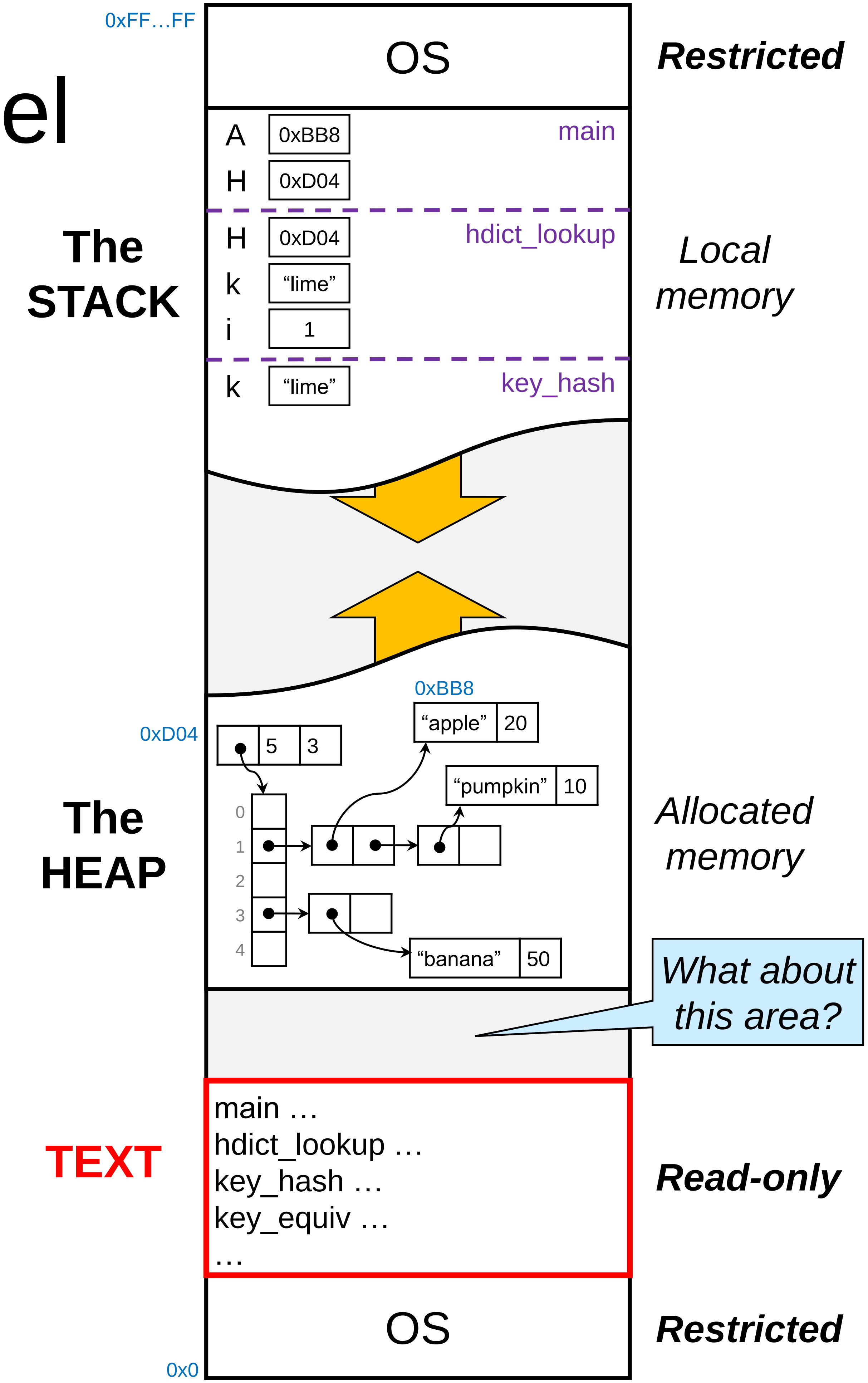
- The top and bottom segments belong to the **operating system**
- A C0 program cannot use them
  - ➢ it cannot read or write there
  - ○ This is **restricted memory**
  - ○ accessing it causes a **segmentation fault**
- **NULL** is address 0x0000000000000000
  - ○ a valid address that doesn't belong to the program
    - ➢ This is why dereferencing NULL causes a segmentation fault



A More Realistic Model

- The **TEXT** segment contains the compiled code of the program
  - every function has an address in memory
    - ➢ the beginning of its binary
- This segment is read-only
  - writing to it causes a segmentation fault

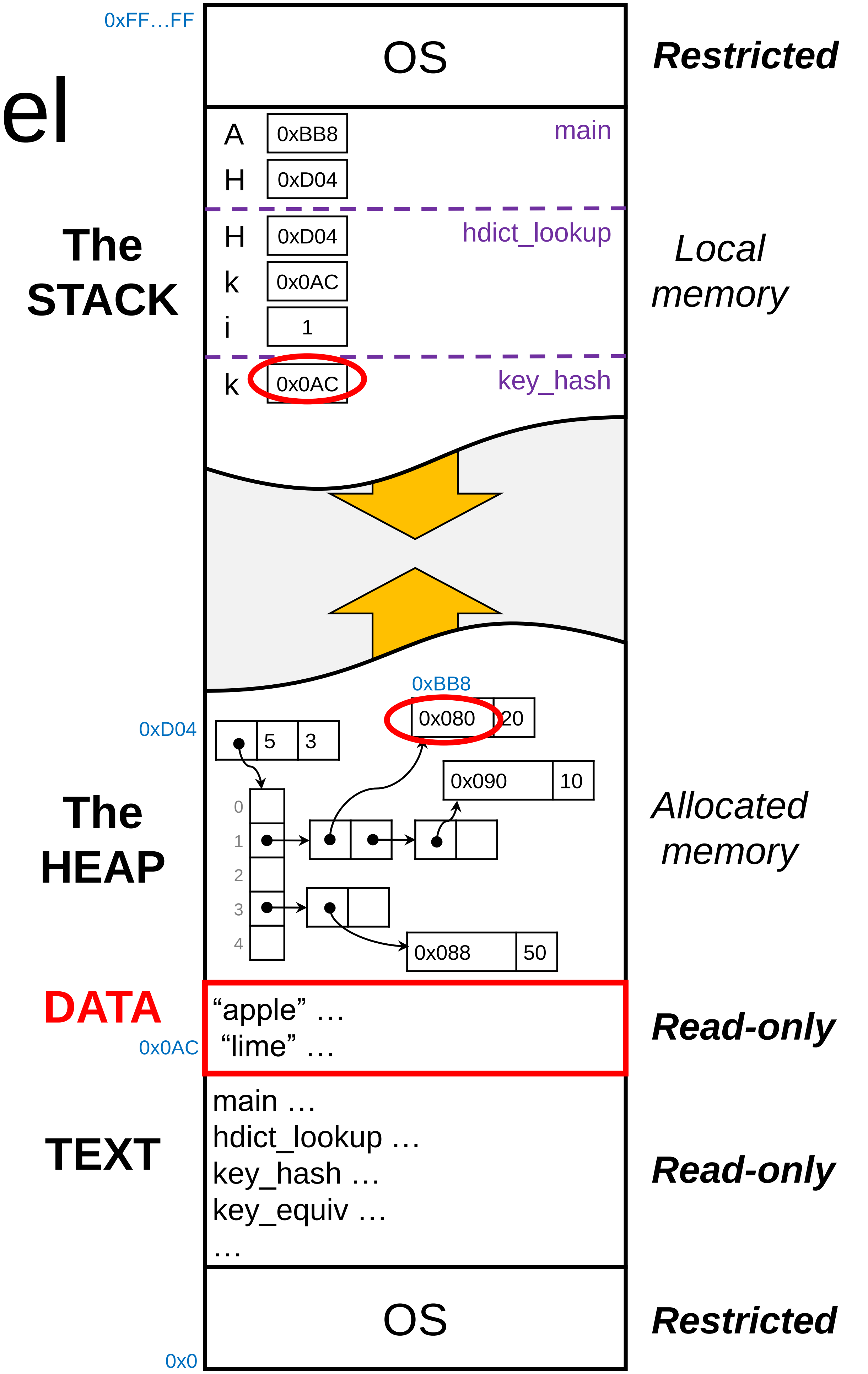
Well, it's got to live somewhere!



A More Realistic Model

- The **DATA** segment contains all the string literals present in the program
  - not the strings constructed by functions like string_join
    - those are hidden in the heap
  - every string has an address in memory
    - the address of its first character
    - variables and fields of type string contain this address
- This segment is read-only

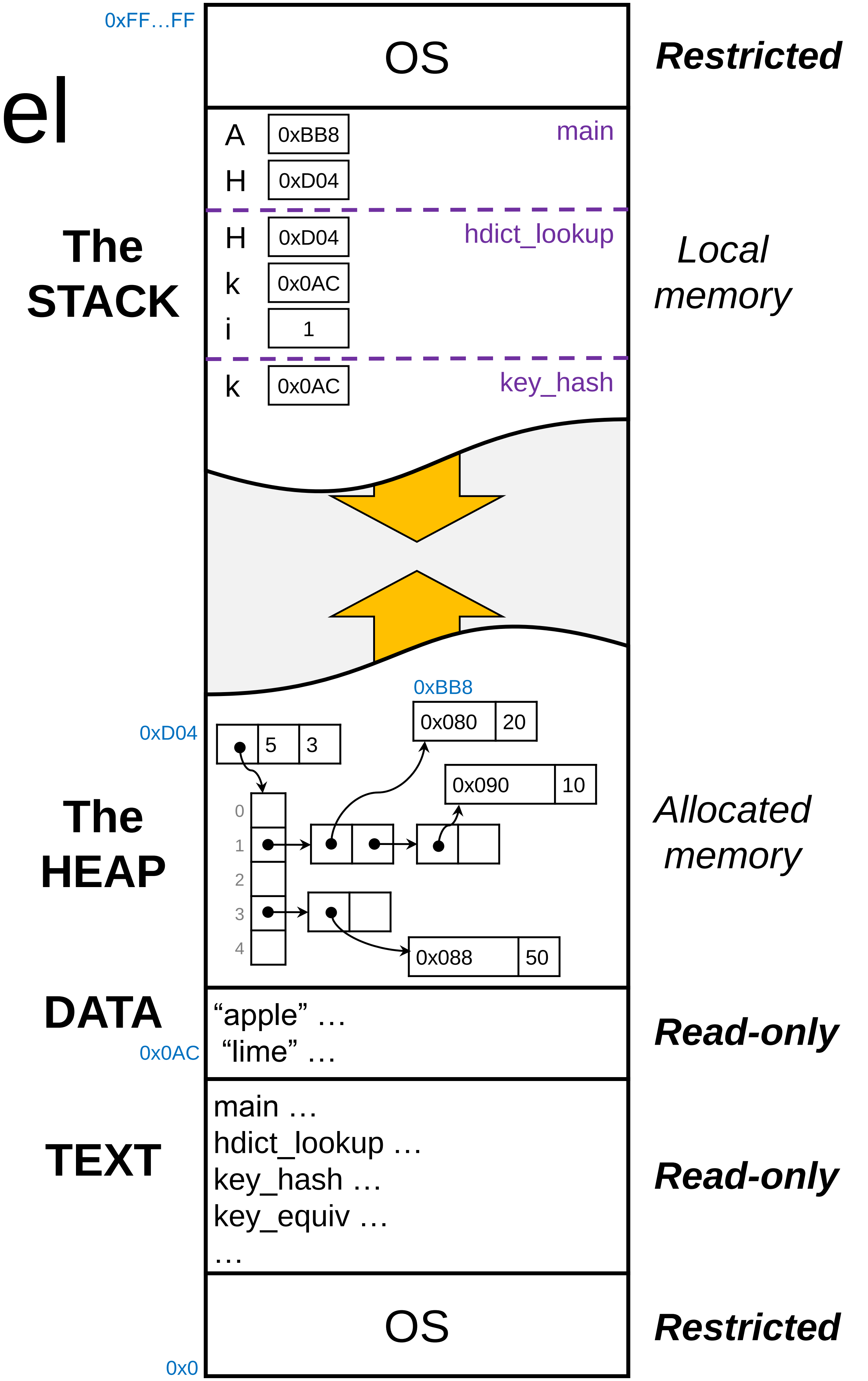
Writing to it causes a segmentation fault



A More Realistic Model

- This is not the end of the story!
- Actual memory is much more complicated
  - This model will be significantly refined in future classes

Hint: no computer in existence comes even close to having 2^{64} bytes of memory!



Function Pointers

Addresses a C0 Program can Use

- The address of an array
  - returned by alloc_array
- The address of a memory cell
  - returned by alloc
- NULL
  - that's just address 0x00000000
  - but we can't dereference it
- The address of a string
  - but C0 hides that they are even addresses

… and that's it

Addresses a C1 Program can Use

- Everything a C0 program can use
  - the address of an array
  - the address of a memory cell
  - NULL
  - the address of a string

- The address of a function
  - this is called a **function pointer**

The language C1

- C1 is an **extension** of C0
  - Every C0 program is a C1 program

- C1 provides two additional mechanisms
  - Generic pointers
  - Function pointers

  Both help with genericity

Rest of this unit

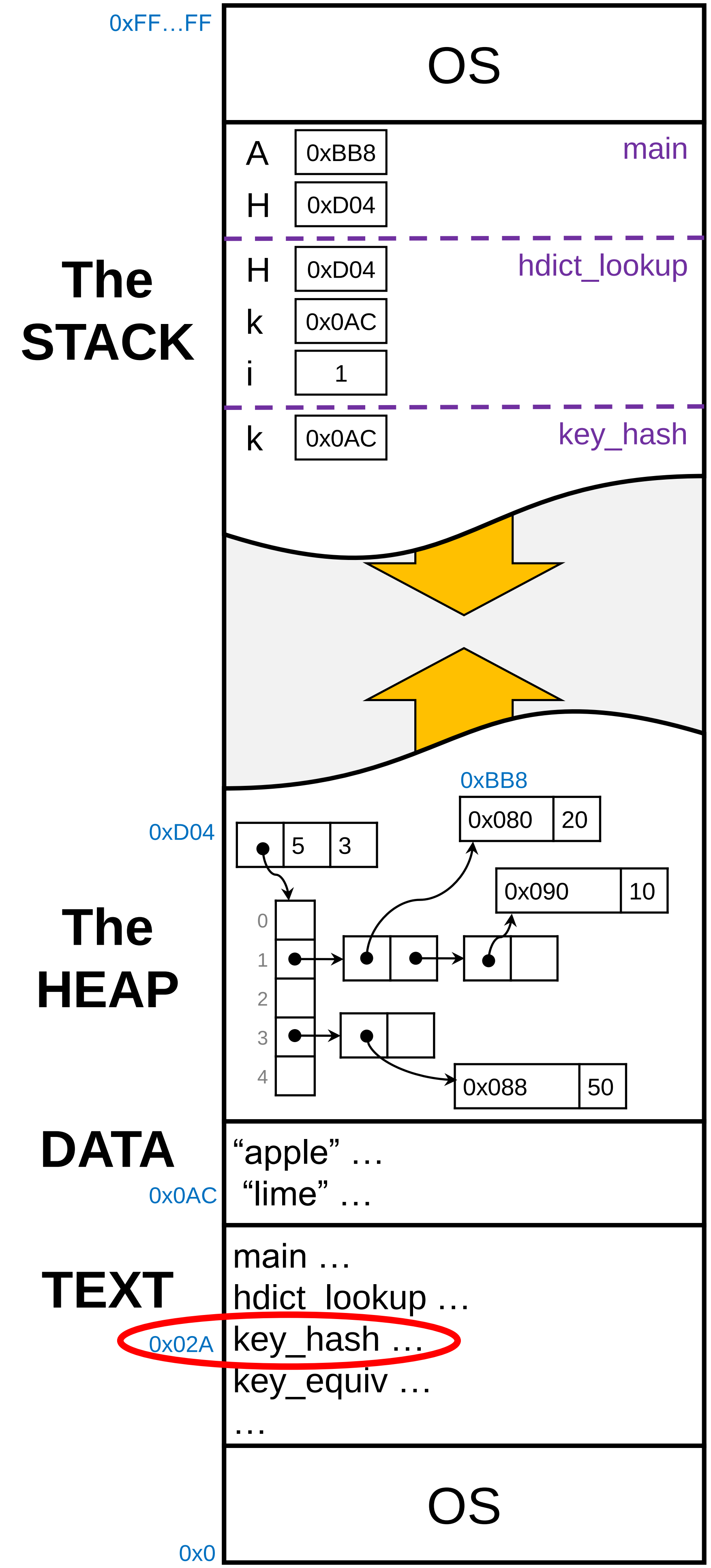
The Address of a Function

- C1 provides the **address-of operator** to grab the address of a function
  - written “&”, prefix
- If key_hash starts at address 0x02A in the TEXT segment, then

  &key_hash

  returns the address 0x02A
  - & can only be applied to the name of a function in a C1 program

C and other languages have many more uses for &



What to do with a Function Pointer?

- Eventually, we want to **apply** the function it points to to some arguments

- But first, we generally store a function pointer
  - in a variable

    F = &key_hash;

    - F is a variable containing a function pointer
    - what is its type?
      - all C0/C1 variables have a type

  - but also in a data structure

    p->hash = &key_hash;

    - p->hash is a field containing a function pointer
    - what is its type?
      - struct fields, array elements, memory cells have a type in C0/C1

Function Types

- key_hash is a function that takes a string as input and returns an int

- To give the name string_to_int_fn to the type of the functions that take a string as input and return an int, we write

```
int key_hash(string s) {
  int len = string_length(s);
  int h = 0;
  for (int i = 0; i < len; i++) {
    h = h + char_ord(string_charat(s, i));
    h = 1664525 * h + 1013904223;
  }
  return h;
}
```

typedef int **string_to_int_fn**(string s);

Return type

Name chosen for the function type

Type of the parameter

  - thus, key_hash has type string_to_int_fn
  - and so does the **<string>** library function string_length

Function Types

```
int key_hash(string s) {
  int len = string_length(s);
  int h = 0;
  for (int i = 0; i < len; i++) {
    h = h + char_ord(string_charat(s, i));
    h = 1664525 * h + 1013904223;
  }
  return h;
}
```

```
typedef int string_to_int_fn(string s);
```

Return type

Name chosen for the function type

Type of the parameter

- by convention, function types end in _fn
  - like types exported by a library interface end in _t
- This is a different use of typedef from what we had in the past

- Function types are **not** functions
  - we cannot write string_to_int_fn("hello")

- We can give a function type any name we want

```
typedef int string_hash_fn(string s);
```

Function Types

```
int POW(int x, int y)
//@requires y >= 0;
{
  if (y == 0) return 1;
  return x * POW(x, y-1);
}
```

- Any function can be given a type
  - the type of POW is defined as

    `typedef int binop_fn(int x, int y);`

  - Steps to defining a function type
    1. Write down the prototype of the function

       `int POW(int x, int y);`

    2. Write typedef in front of it

       `typedef int POW(int x, int y);`

    3. Replace the function name with a type name of your choice

       `typedef int binop_fn(int x, int y);`

  - Contracts can be included in the function type definition

```
typedef int binop_with_pos2_fn(int x, int y)
          /*@requires y >= 0; @*/ ;
```

```
int key_hash(string s) {
  int len = string_length(s);
  int h = 0;
  for (int i = 0; i < len; i++) {
    h = h + char_ord(string_charat(s, i));
    h = 1664525 * h + 1013904223;
  }
  return h;
}
```

Storing Function Pointers

```
typedef int string_to_int_fn(string s);

string_to_int_fn* F = &key_hash;
```

- F needs to be a **pointer** to a function that takes a string and returns an int

  string_to_int_fn* F

  - ➢ a pointer to a string_to_int_fn
  - ○ This is because &key_hash returns the **address** of key_hash
  - ○ and this address is stored in F

- string_to_int_fn F = &key_hash;   // no *

  is invalid C1 code

```
int key_hash(string s) {
  int len = string_length(s);
  int h = 0;
  for (int i = 0; i < len; i++) {
    h = h + char_ord(string_charat(s, i));
    h = 1664525 * h + 1013904223;
  }
  return h;
}
```

Using Function Pointers

```
typedef int string_to_int_fn(string s);
string_to_int_fn* F = &key_hash;
```

- F contains the address of key_hash

- To call F on an input, we first need to dereference it

```
int h = (*F)("hello");
```

Applying a function pointer

- Writing *F("hello") is incorrect
  - C1 interprets it as *(F("hello"))
    - which doesn't type check
- Other languages have better syntax

Safety of Function Pointers

- A function pointer is a pointer!

```
int h = (*F)("hello");
```

  is **safe** only if F != NULL

- The address of the functions in a program are never NULL
  - Thus

    ```
    string_to_int_fn* F = &key_hash;
    int h = (*F)("hello");
    ```

    is safe because F contains the address of key_hash
  - but

    ```
    string_to_int_fn* F = NULL;
    int h = (*F)("hello");
    ```

    is unsafe

Function Pointer Contracts

- *The address of the functions in a program are never NULL*

- Function pointer operators have their own contracts
  - & always returns a non-NULL pointer

```
&f
//@ensures \result != NULL;
```

    - where f is a function declared in the program

  This is a new way to justify that a pointer is non-NULL