Dynamic Memory & Memory Management

Rob Hackman

Winter 2026

University of Alberta

Table of Contents

Arrays of Arbitrary Length

Let's write a function that reads in all the integers from `stdin` and prints them all back out in sorted order.

We must keep track of each `int` we've seen — but how? We don't know how many integers the user is going to type.

We can allocate an array on our stack that we *hope* is big enough, but we can't know for certain.

```
int main() {
  const int size = 1000;
  int arr[size];
  int len = 0;
  while (!feof(stdin)) {
    int rc = scanf("%d", &x);
    if (1 == rc) {
      insert(arr, len, size, x);
    }
  }
  printArr(arr, len);
}
```

Sample insert implementation

```
1  int insert(int *arr, int l, int s, int val) {
2    if (l == s) {
3      printf("Array already at capacity");
4      return -1;
5    }
6    int i = 0;
7    for (; i < l && arr[i] <= val; ++i);
8    for (int j = l; j > i; --j) arr[j] = arr[j-1];
9    arr[i] = val;
10 }
```

The Problem

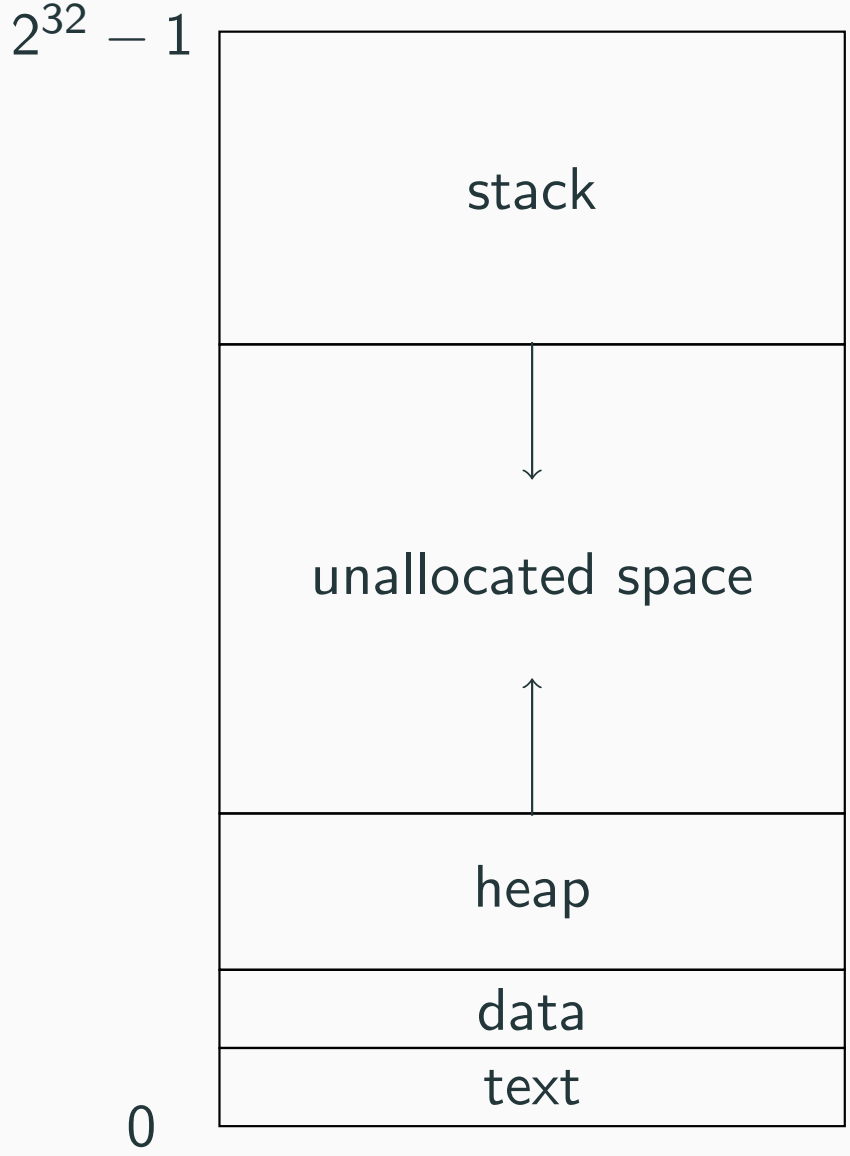
What if the input contains more than 1000 integers?

Program fails. What can we do?

```
int main() {
  const int size = 1000;
  int arr[size];
  int len = 0;
  while (!feof(stdin)) {
    int rc = scanf("%d", &x);
    if (1 == rc) {
      insert(arr, len, size, x);
    }
  }
  printArr(arr, len);
}
```

Recall - the Heap

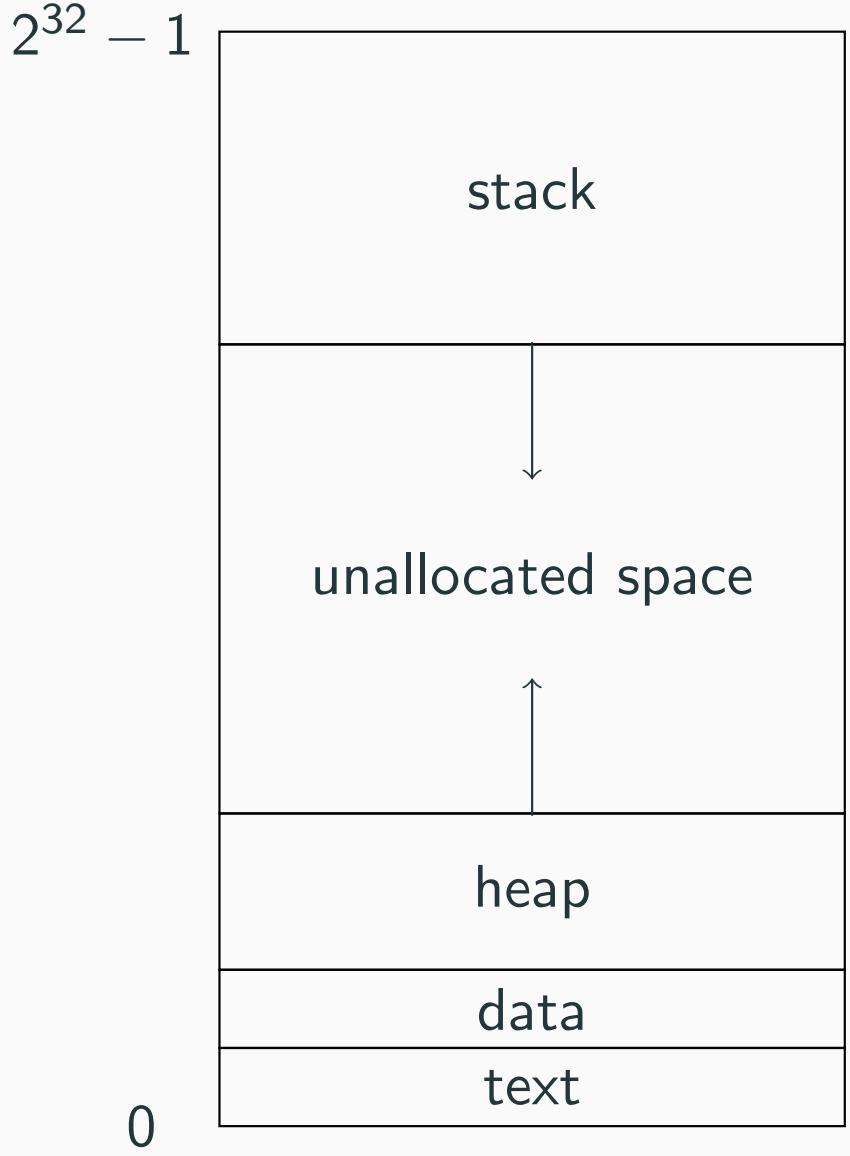
- data — stores data whose lifetime is the entirety of your program. **Known** at compile time.
- heap — stores data whose lifetime is dictated by the programmer. **Not known** at compile time.
- stack — stores data whose lifetime is tied to their enclosing scope. **Known** at compile time.



The Heap — Memory under your control

Unlike the stack which is automatically managed by the code generated by the compiler, the heap is the programmer's responsibility to manage.

The programmer may ask for bytes to be allocated onto the heap, but they are responsbile for deallocating (freeing) that memory when finished with it.



`malloc` — with great power...

The function `malloc` defined in the `stdlib` library should be used in C to allocate heap memory.

`malloc` takes one parameter: the number of contiguous bytes you would like allocated. The type of this parameter is a `size_t`.

`malloc` allocates the bytes asked for (if able) and returns a pointer to the first byte allocated. Returns `NULL` if unable to make the allocation.

Aside: the `size_t` type

`size_t` is a type defined in the `stddef` library, but is also typically accessible through several other libraries.

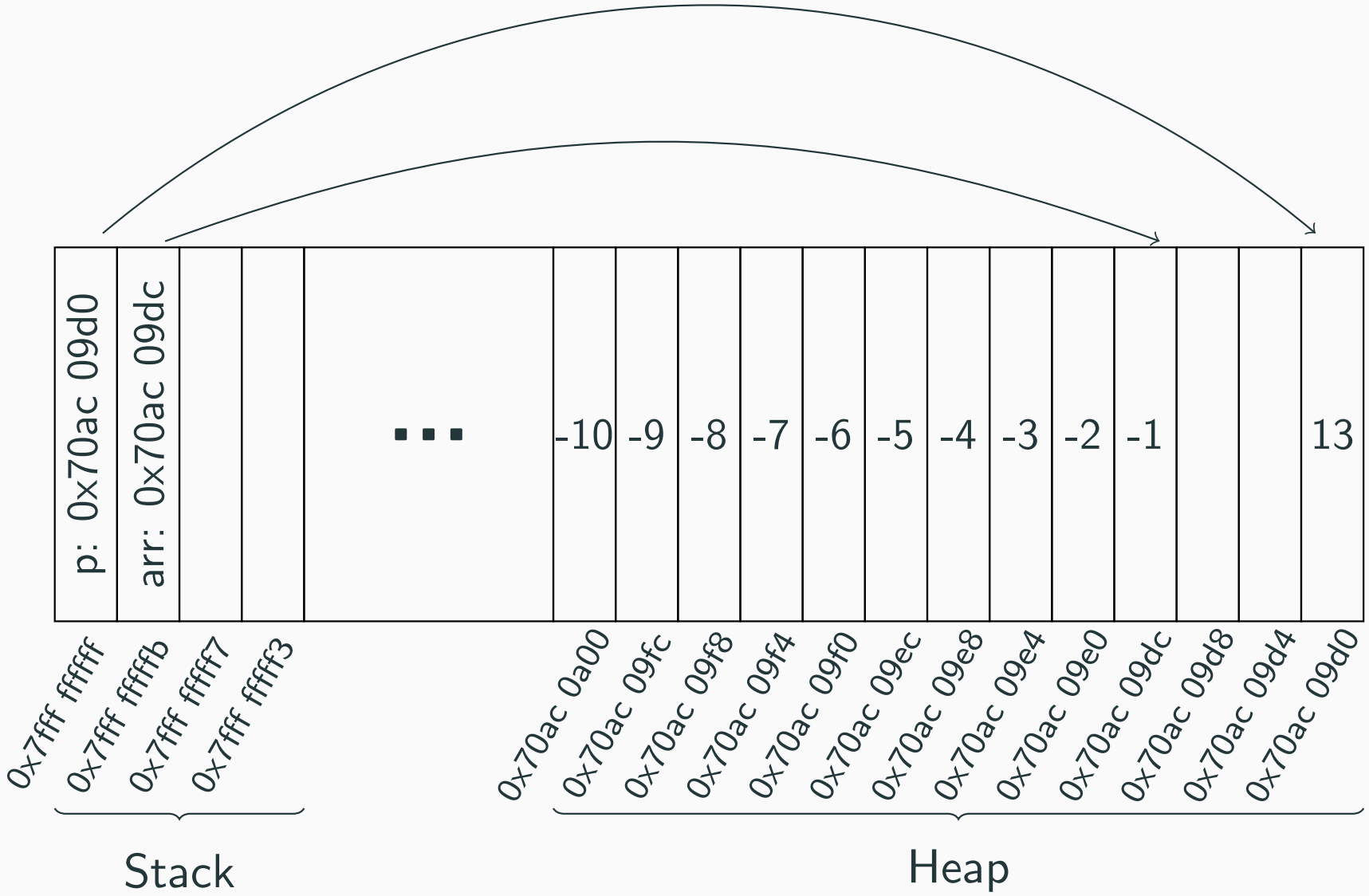
`size_t` is an unsigned integral type that is guaranteed to be large enough to store the size of any possible data type. As such the size of `size_t` itself is system dependant.

You should use `size_t` for data related to the size of things (e.g. length of arrays, indices of arrays)

malloc example

```
1   int main() {
2     // Allocate enough bytes for 1 integer.
3     // Store pointer to the first integer in p.
4     int *p = malloc(sizeof(int)*1);
5     // Allocate enough bytes for 10 integers.
6     // Store pointer to first integer in arr.
7     int *arr = malloc(sizeof(int)*10);
8     *p = 13;
9     for (size_t i = 0; i < 10; ++i) {
10      arr[i] = -1-i;
11    }
12  }
```

`malloc` example — memory diagram



`free` — comes great responsibility

The function `free` is also provided by the `stdlib` library, it is used to free memory that was previously allocated with `malloc`.

`free` has one parameter, a pointer to the heap memory that was previously allocated with `malloc`. Has a `void` return type, returns nothing.

Whenever you are finished with heap-allocated memory you must free it — failure to do so is called a *memory leak*. A program with memory leaks that runs long enough will eventually crash[1].

[1] Assuming the leak doesn't occur only a constant amount.

Fixing `malloc` example

```
1   int main() {
2     // Allocate enough bytes for 1 integer.
3     // Store pointer to the first integer in p.
4     int *p = malloc(sizeof(int)*1);
5     // Allocate enough bytes for 10 integers.
6     // Store pointer to first integer in arr.
7     int *arr = malloc(sizeof(int)*10);
8     *p = 13;
9     for (size_t i = 0; i < 10; ++i) {
10      arr[i] = -1-i;
11    }
12    free(p);
13    free(arr);
14  }
```

Dangling Pointers

Once you have freed a block of memory you should no longer read or write to it.

As such, it is dangerous to maintain pointers to freed memory as you may accidentally dereference them.

Pointers that store an address of memory that has already been freed are said to be *dangling*. To avoid having dangling pointers, anytime you free a pointer you should set it to `NULL`.

```
free(p);
p = NULL;
```

Printing sorted integers

Let's try to write our program again that reads every integer from stream and prints them out in sorted order.

```
int main() {
  int size = 16;
  int *arr = malloc(size*sizeof(int));
  int len = 0;
  while (!feof(stdin)) {
    int rc = scanf("%d", &x);
    if (1 == rc) {
      insert(arr, len, size, x);
    }
  }
  printArr(arr, len);
  free(arr);
}
```

Fixing sorted integers

We haven't really solved anything with this solution though, it still is only allocating an array of fixed size — this time it is just on the heap instead of the stack.

Fixing sorted integers

We haven't really solved anything with this solution though, it still is only allocating an array of fixed size — this time it is just on the heap instead of the stack.

With the heap we can always choose to allocate more memory when we realize we need it though, unlike the stack which is predetermined at compile time!

Fixing sorted integers

We haven't really solved anything with this solution though, it still is only allocating an array of fixed size — this time it is just on the heap instead of the stack.

With the heap we can always choose to allocate more memory when we realize we need it though, unlike the stack which is predetermined at compile time!

We can't just extend our existing array though[1]. What can we do?

[1] Okay, `realloc` exists but user's often make mistakes, and no analagous function exists in C++ so we'll avoid getting in the habit.

Growing Array

We can grow our array by allocating a new and larger copy each time we reach our capacity.

How much should we grow by?

Intuition might say to allocate exactly as much as we need, so in our example one more `int` worth of space each time. That would be foolish.

While growing by only as much as you need is memory efficient, it's very time inefficient.

Steps to “growing” an array

1. Allocate a larger array
2. Copy over all items from original array to larger array
3. Free original array
4. Update relevant data (array pointer, capacity, etc)

Growing Array — Growing by 1

```
int size = 1;
int *arr = malloc(size*sizeof(int));
int len = 0;
while (...) {
  ... // Things happen, need to grow our array.
  if (len == size) {
    int *newArr = malloc(sizeof(int)*(size+1));
    for (int i = 0; i < len; ++i) {
      newArr[i] = arr[i];
    }
    ++size;
    free(arr);
    arr = newArr;
  }
}
```

Growing by 1 — slow

Why is growing by one element each time so poor an idea? Consider what we must do each time we grow our array.

Each time our array grows we have to copy every single item currently in our array to our new array. If we consider each of those copies to be one unit of time then how many units of time do we take if we end up reading in n integers?

The first time we grow we must copy 1 `int`, the second time we must copy 2 `ints`, the third time we must copy 3 `ints`...

Growing by 1 — Analysis

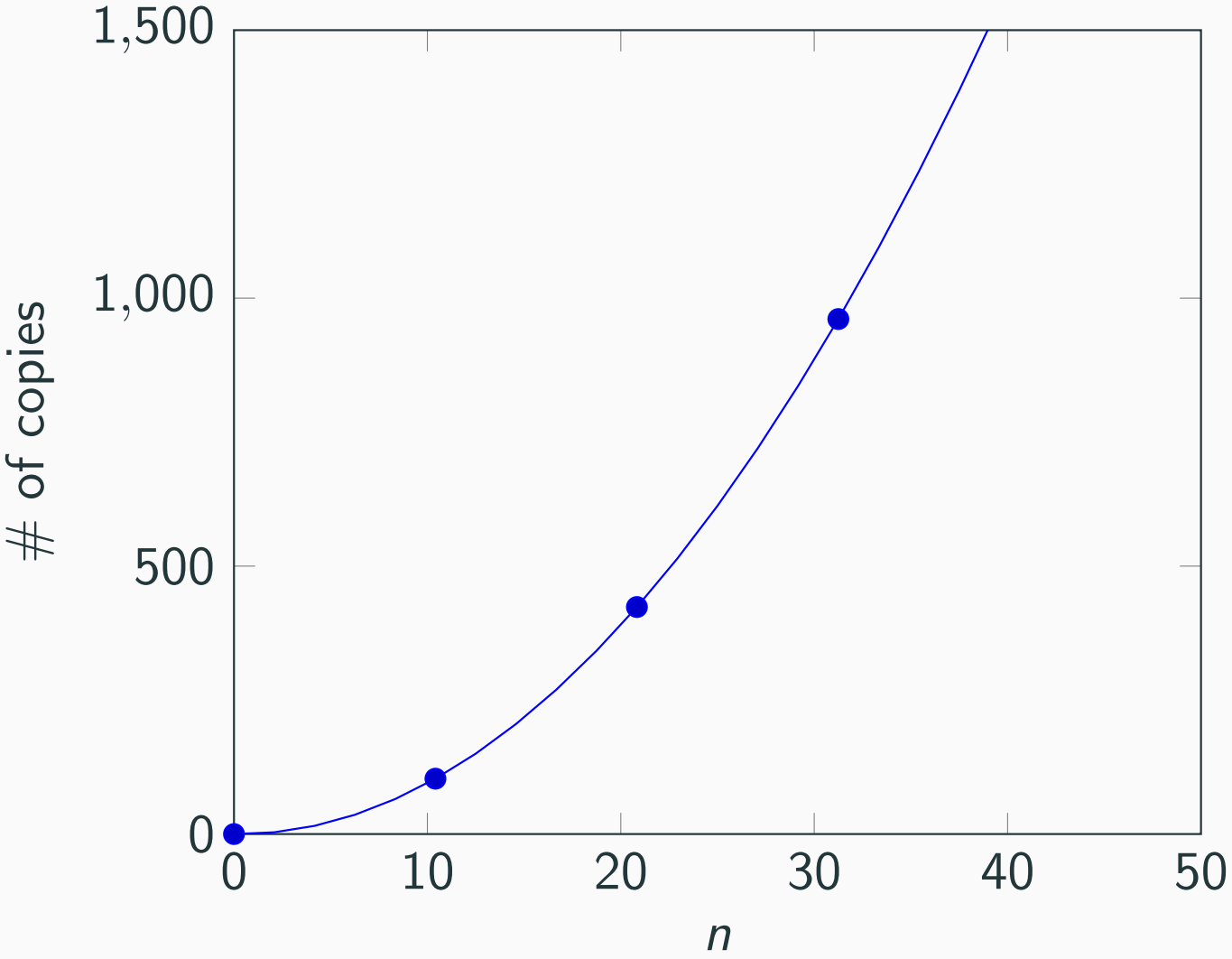
If we start with space for 1 `int`, and read n `ints` total, only allocating space for one more `int` each time then the amount of copies we must do is

$$\begin{aligned} & 1 + 2 + 3 + \ldots + n - 1 \\ &= \frac{n(n-1)}{2} \\ &= \frac{n^2 - n}{2} \end{aligned}$$

If n is the amount of `ints` the user types in, then the total amount of copies we need to do is growing relative to n^2 — which is not ideal!

Growing by 1 — Analysis

Here is a plot of how many times we must copy an int relative to our input size if we grow by 1 each time



Growing by 1 — too slow

If we only ever increase our array size by one, then the time complexity of appending an item to the end of our array becomes $O(n)$, since worst case we have to copy the entire existing array.

Adding an item to the end of an array should be constant time!

Solution: every time our array requires more space, double the capacity.

Doubling Strategy — Code

```
int size = 4;
int *arr = malloc(size*sizeof(int));
int len = 0;
while (...) {
  ... // Things happen, need to grow our array.
  if (len == size) {
    int *newArr = malloc(sizeof(int)*(size*2));
    for (int i = 0; i < len; ++i) {
      newArr[i] = arr[i];
    }
    size = size*2;
    free(arr);
    arr = newArr;
  }
}
```

Doubling Strategy — Analysis

Consider our array has a capacity of c elements, and is currently empty.

Appends	Ops	Ops/Append
c	c	$\frac{c}{c}$
$c+1$	$c+c+1$	$\frac{2c+1}{c+1}$
$2c$	$2c+c$	$\frac{3c}{2c}$
$2c+1$	$3c+2c+1$	$\frac{5c+1}{2c+1}$
$4c$	$5c+2c$	$\frac{7c}{4c}$
$4c+1$	$7c+4c+1$	$\frac{11c+1}{4c+1}$
$8c$	$11c+4c$	$\frac{15c}{8c}$
	...	
$2^i c$	$(2^i-1)c+2^i c$	$\frac{(2^{i+1}-1)c}{2^i c} \approx 2$

By doubling everytime we have an *amortized constant* append

Doubling Strategy — Tradeoff

Doubling every time is trading memory for time, however the memory wasted is only ever at most twice the memory we're actually using.

The time inefficiency of growing by a constant scalar gives linear insert. Certainly worth a constant scalar more amount of space used, unless otherwise constrained by memory.

Practice Questions

1. Write a function `void primes(int n)` that takes as a parameter one `int` and prints out (in order) all of the prime numbers smaller than that `int`.
2. Write a function `int str_to_int(const char *s)` that takes in a string texttts that is a numeric only string and returns the integer that string represents.

Table of Contents

Considering Lifetime

Until now, as we only used the stack, we haven't considered cases where the lifetime of data is relevant.

The C programmer must be ever aware of the lifetime of data they are working with.

Consider, we want to write a function that takes in an `int n` and returns an array of the first 4 multiples of `n`.

Life's too short

Since we know we always want to create an array of 4 elements we may try the following:

```
1       int *genMultiples(int n) {
2         int arr[4];
3         for (int i = 1; i < 5; ++i) arr[i-1] = i*n;
4         return arr;
5       }
```

This might *seem* fine — it is not fine at all.

Life's too short

We write the following `main` to test our `genMultiples` function:

```
1  int main() {
2    int *fives = genMultiples(5);
3    int *sevens = genMultiples(7);
4    for (int i = 0; i < 4; ++i) printf("%d ", fives[i]);
5    printf("\n");
6    for (int i = 0; i < 4; ++i) printf("%d ", sevens[i]);
7    printf("\n");
8  }
```

and when we run it... it crashes[1]

[1] When compiled with gcc, that is. This is undefined behaviour and as such using a different compiler may produce a different result (and does)

Returning pointers

The problem is we cannot return an array itself so we must return a pointer to an array's first element.

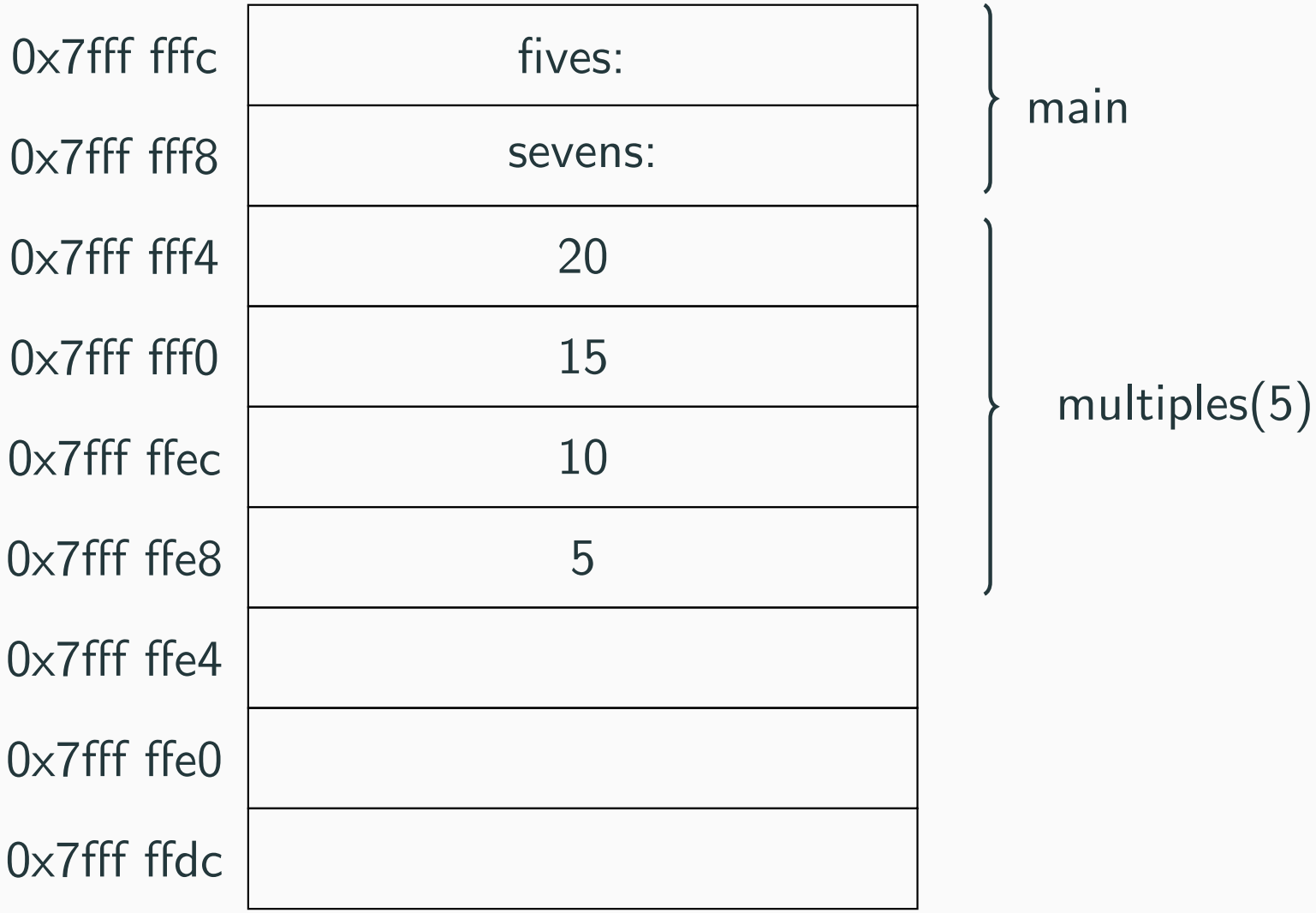
The array to which `genMultiples` is returning a pointer is a local varaible to `genMultiples`. As a local variable it is stored on the stackframe of the particular call to `genMultiples`.

When `genMultiples` returns its stack frame is deallocated... which means that the array it just returned a pointer to is also deallocated.

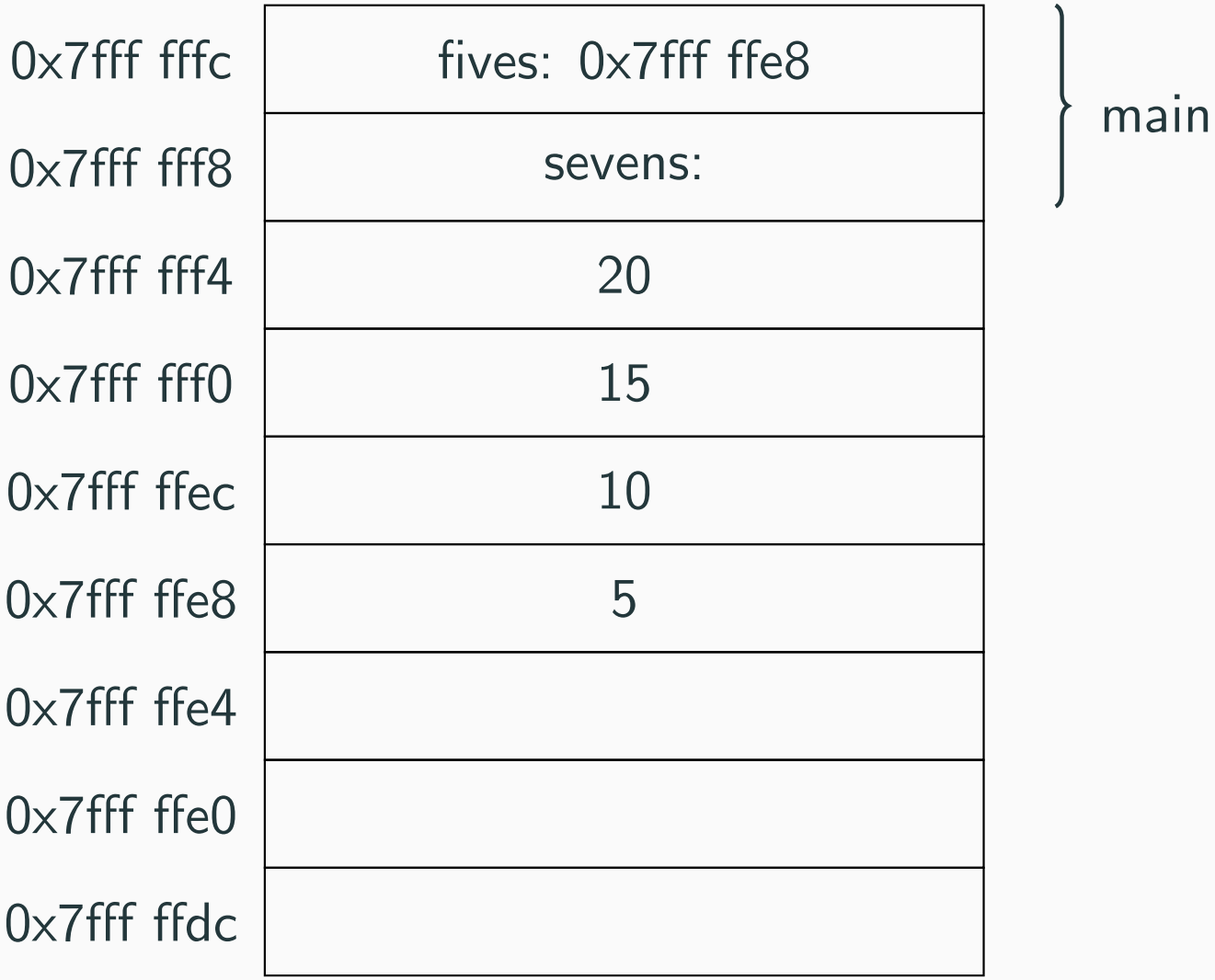
`genMultiples` is returning a dangling pointer!

Returning stack pointers — a diagram

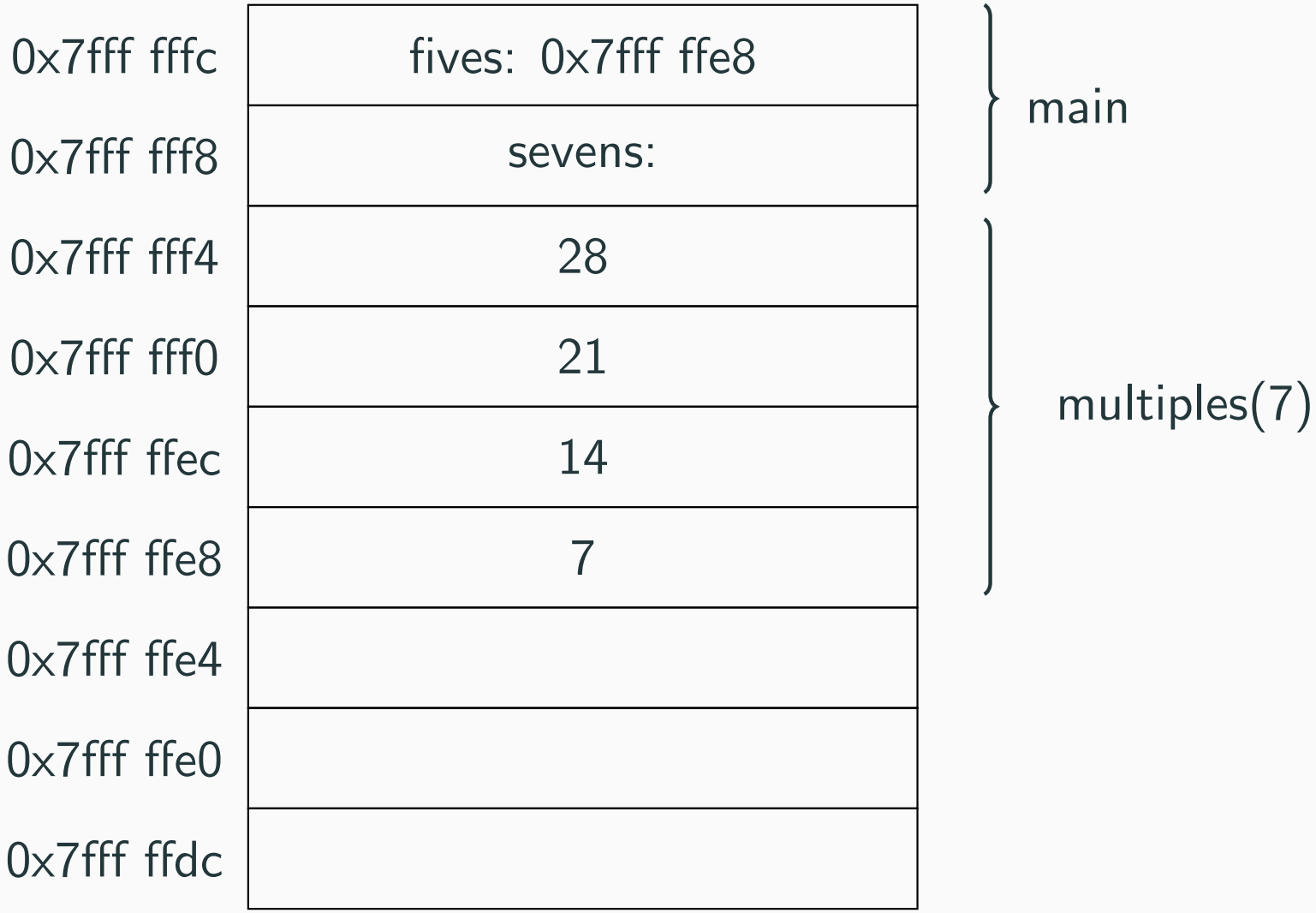
Returning stack pointers — a diagram



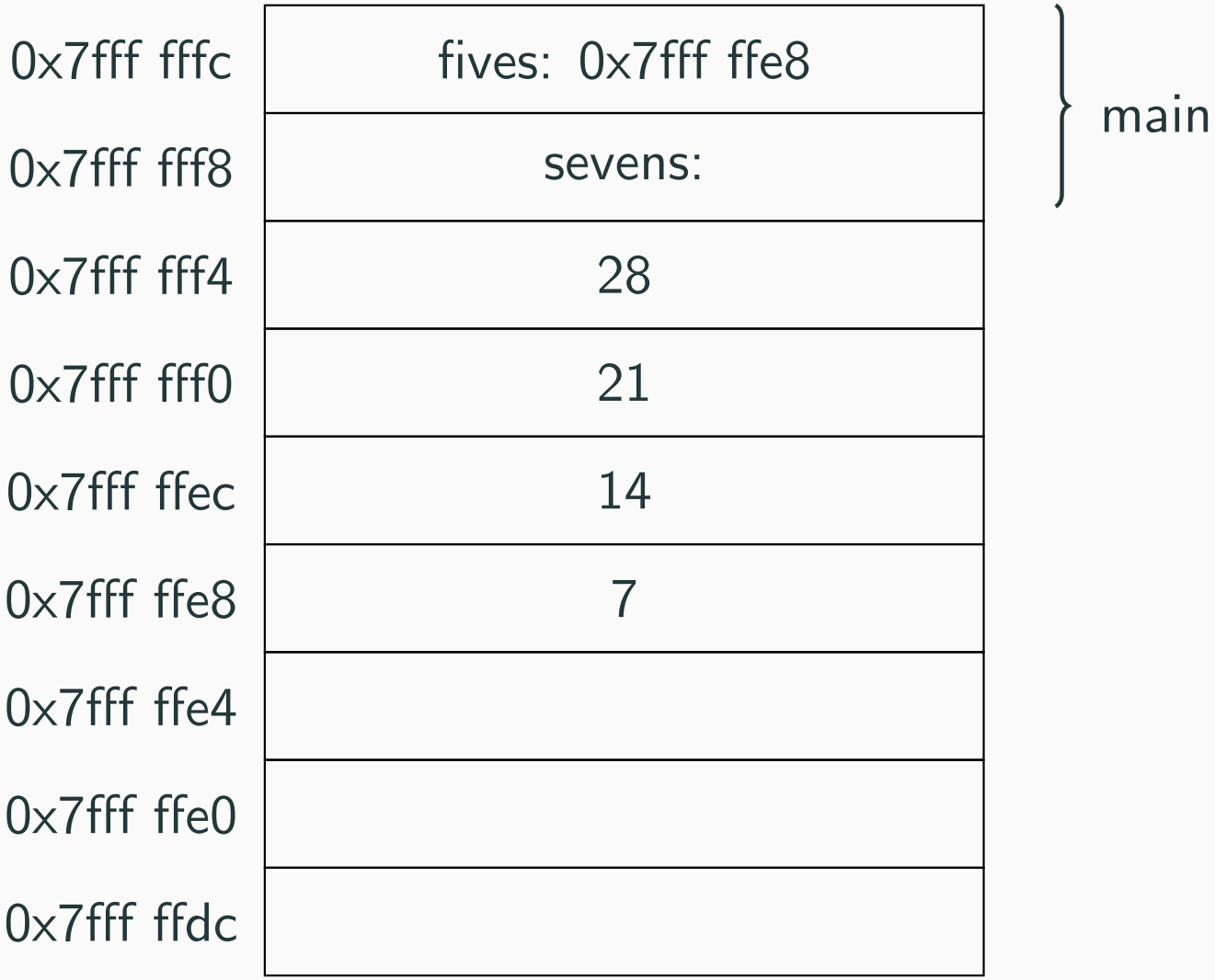
Returning stack pointers — a diagram



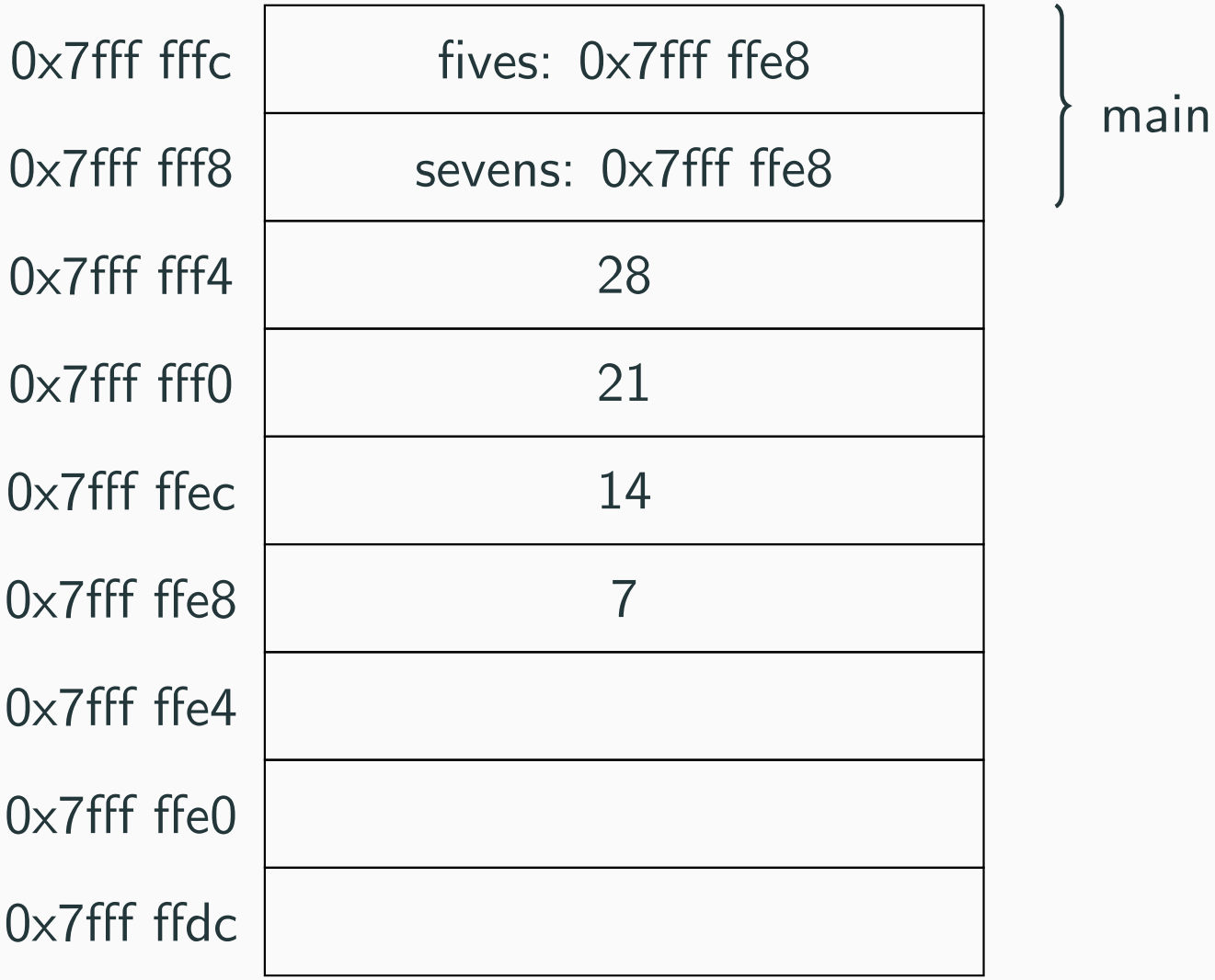
Returning stack pointers — a diagram



Returning stack pointers — a diagram



Returning stack pointers — a diagram



Returning pointers

A function that returns a pointer to its local stackframe is undefined behaviour. Should **never** be done!

Since it is undefined behaviour `gcc` actually chooses to just return `NULL`, this is why our program earlier crashed when trying to access the arrays returned by `multiples`.

`clang` does not choose to do this, and we can see the behaviour of our stack getting overwritten by compiling our program with `clang` and running it.

Again, there is no guarantee what happens in this case — that is the essence of undefined behaviour. It is not valid C to invoke undefined behaviour.

When can we return a pointer?

A pointer can be returned when the lifetime of the data it points at is longer than the function which was called. The three most likely possibilities for that are:

1. It is a pointer that was provided as one of the arguments, and thus must be alive at least for the caller of this function
2. It is a pointer to heap allocated memory that hasn't been freed, and thus is alive until it is freed.
3. It is a pointer to global or static data.

Returning a pointer to data passed in

Consider our own implementation here of `strchr` which searches a string for a specific character and if it finds it returns a pointer to the first occurrence of that character in the string.

We're returning a pointer, but it's not to our stack. It may be to the heap, but we don't know. The person who called us gave it to us though, so we know its lifetime is at least longer than just this function call.

```
1  char *strchr(const char *s, int c) {
2      for (; *s != '\0'; ++s) {
3        if (*s == c) return s;
4      }
5      return NULL;
6  }
```

Returning a pointer to heap allocated data

A function that is going to create an array to return, like `multiples`, either has to be given space to place it in as a parameter (or a global space to place it) or, more commonly, allocate it on the heap to return it.

```
1  int *multiples(int m, size_t size) {
2      int *ret = malloc(sizeof(int)*size);
3      for (size_t i = 0; i < size; ++i) {
4        ret[i] = i*m;
5      }
6      return ret;
7  }
```

Note: We haven't freed this memory! The caller must be aware this function returns a heap-allocated array and they, the caller, are responsible for freeing it when they are finished with it!

Using fixed `multiples`

```
1  int main() {
2    int *fives = multiples(5,10);
3    int *sevens = multiples(7,10);
4    for (size_t i = 0; i < 10; ++i) {
5      printf("%d ", fives[i]);
6    }
7    printf("\n");
8    for (size_t i = 0; i < 10; ++i) {
9      printf("%d ", sevens[i]);
10   }
11   printf("\n");
12   free(fives);
13   free(sevens);
14 }
```

Lifetimes — Important details!

Whenever using pointers you must be be aware of the lifetime of the data you assign a pointer to!

If you assign a pointer to a data and that data is deallocated (goes out of scope if on stack, freed if on heap) then that pointer is now dangling.

Dereferencing a dangling pointer is undefined behaviour and is a bug in your program! No matter what result it generates it is not what you *meant* to do.

Pay attention to lifetimes!

Practice Question

Write a function `digits` that takes in an `unsigned int` and returns a pointer to an array that stores the digits of the `int` in order from most significant to least significant.

For example `digits(257)` would return an array that contained 2, 5, and 7 in that order. `digits(72519)` would return an array that contained 7, 2, 5, 1, and 9 in that order.