Dynamic Memory

2024 Winter APS 105: Computer Fundamentals

Jon Eyolfson

Lecture 20

1.0.1

Recall: Local Variables Only Exist While the Function Runs

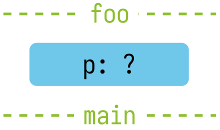
```
#include <stdio.h>
#include <stdlib.h>

int *foo(void) {
    int x = 1;
    return &x;
}

int main(void) {
    int *p = foo();
    printf("*p: %d\n", *p);
    return EXIT_SUCCESS;
}
```

----- main -----

Stack

Recall: Local Variables Only Exist While the Function Runs

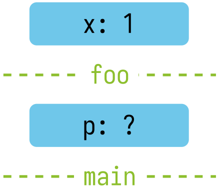
```
#include <stdio.h>
#include <stdlib.h>

int *foo(void) {
    int x = 1;
    return &x;
}

int main(void) {
    int *p = foo();
    printf("*p: %d\n", *p);
    return EXIT_SUCCESS;
}
```



Stack

Recall: Local Variables Only Exist While the Function Runs

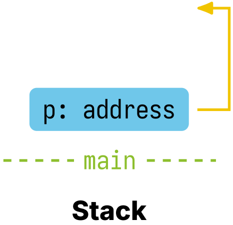
```c
#include <stdio.h>
#include <stdlib.h>

int *foo(void) {
    int x = 1;
    return &x;
}

int main(void) {
    int *p = foo();
    printf("*p: %d\n", *p);
    return EXIT_SUCCESS;
}
```



Stack

Recall: Local Variables Only Exist While the Function Runs

```c
#include <stdio.h>
#include <stdlib.h>

int *foo(void) {
    int x = 1;
    return &x;
}

int main(void) {
    int *p = foo();
→   printf("*p: %d\n", *p);
    return EXIT_SUCCESS;
}
```



Can We Return a Pointer to Memory Created in a Function?

Any variables in a function are allocated (created in memory) on the "stack"

So previously, `int x` only exists in memory as long as we're running `foo`

We Can Explicitly Allocate Memory

There's another region of memory C can use that is unrelated to the current running function

This region of memory is called the "heap"

It comes from the literal word heap: "a large amount or number of"

We Can Request Memory Using `malloc`

Its function prototype in the C standard library is:

```
void* malloc(size_t size);
```

`size_t` is basically a positive integer type
(the `sizeof(size_t)` depends on your machine)

We Can Request Memory Using `malloc`

Its function prototype in the C standard library is:

```
void* malloc(size_t size);
```

`size_t` is basically a positive integer type
(the `sizeof(size_t)` depends on your machine)

The `size` argument is how many contiguous bytes to allocate

`malloc` returns a pointer to a starting address,
you may then use `size` contiguous bytes

You Will See the Term API in Software

API stands for Application Programming Interface, and it tells you how to use a library (functions, types, etc.)

You may say “what’s the `malloc` API?”

We Can Create a Pointer to an `int` Using `malloc`

For example:

```
int *p = malloc(sizeof(int));
```

This creates a pointer to an `int`, pointing to 4 contiguous bytes on the heap

The value it’s pointing to is undefined, meaning it could be anything

We Can Create a Pointer to an `int` Using `malloc`

For example:

```
int *p = malloc(sizeof(int));
```

This creates a pointer to an `int`, pointing to 4 contiguous bytes on the heap

The value it's pointing to is undefined, meaning it could be anything

However, we can initialize the value by dereferencing it

We Can Return Pointers That are Still Valid

```c
#include <stdio.h>
#include <stdlib.h>

int *foo(void) {
    int *p = malloc(sizeof(int));
    *p = 1;
    return p;
}

int main(void) {
    int *p = foo();
    printf("*p: %d\n", *p);
    return EXIT_SUCCESS;
}
```

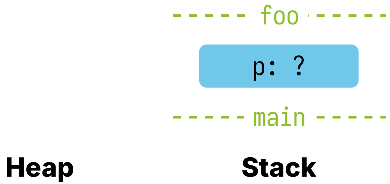
----- main -----

Heap

Stack

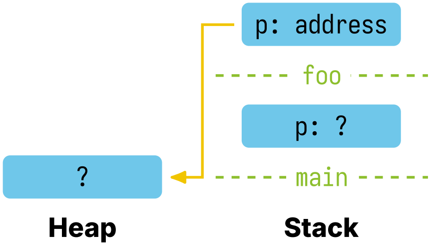
We Can Return Pointers That are Still Valid

```c
#include <stdio.h>
#include <stdlib.h>

int *foo(void) {
    int *p = malloc(sizeof(int));
    *p = 1;
    return p;
}

int main(void) {
    int *p = foo();
    printf("*p: %d\n", *p);
    return EXIT_SUCCESS;
}
```



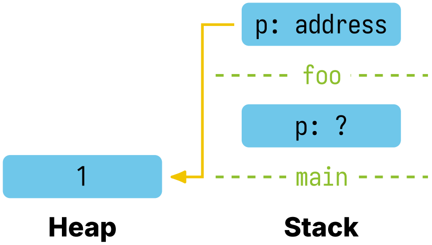
We Can Return Pointers That are Still Valid

```
#include <stdio.h>
#include <stdlib.h>

int *foo(void) {
    int *p = malloc(sizeof(int));
    *p = 1;
    return p;
}

int main(void) {
    int *p = foo();
    printf("*p: %d\n", *p);
    return EXIT_SUCCESS;
}
```



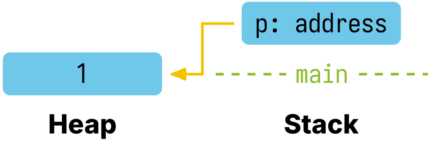
We Can Return Pointers That are Still Valid

```c
#include <stdio.h>
#include <stdlib.h>

int *foo(void) {
    int *p = malloc(sizeof(int));
    *p = 1;
    return p;
}

int main(void) {
    int *p = foo();
    printf("*p: %d\n", *p);
    return EXIT_SUCCESS;
}
```



We Can Return Pointers That are Still Valid

```c
#include <stdio.h>
#include <stdlib.h>

int *foo(void) {
    int *p = malloc(sizeof(int));
    *p = 1;
    return p;
}

int main(void) {
    int *p = foo();
    printf("*p: %d\n", *p);
    return EXIT_SUCCESS;
}
```



`malloc` Can Run Out of Memory and Return an Error

If there's no more room in the heap, `malloc` returns `NULL`

You should check the return value of `malloc`
- Otherwise, you may dereference `NULL`!

We Have a New Responsibility, Deallocating

Previously, when the function returns, all the variables disappear

- They're no longer valid, and your computer can re-use the memory

We say the memory is deallocated, meaning you're done using it

For memory allocated with `malloc`, we have to deallocate it

We Can Deallocate Using the `free` Function

Its function prototype in the C standard library is:

```c
void free(void *ptr);
```

The pointer argument, `ptr`, needs to be the address returned from `malloc`

- Afterwards you cannot use the memory pointed to by `ptr`

We Can Deallocate Using the `free` Function

Its function prototype in the C standard library is:

```
void free(void *ptr);
```

The pointer argument, `ptr`, needs to be the address returned from `malloc`

Afterwards you cannot use the memory pointed to by `ptr`

If the value of `ptr` is `NULL` then `free` does nothing

We Should Deallocate Memory When We No Longer Use It

```c
#include <stdio.h>
#include <stdlib.h>

int *foo(void) {
    int *p = malloc(sizeof(int));
    *p = 1;
    return p;
}

int main(void) {
    int *p = foo();
    printf("*p: %d\n", *p);
    free(p);
    return EXIT_SUCCESS;
}
```

Forgetting to Deallocate Memory is Called a Memory Leak

Your program would be using more memory than it actually needs to function

This is a big problem when your program runs for a long time!

- You may actually run out of memory, and slow down other programs

You Can Use a Tool Called `valgrind` to Detect Memory Issues

If you run your program normally in the terminal using: `build/valid-pointer`
  you can use: `valgrind build/valid-pointer`

If you forget to `free` in the previous example, you'll see:

```
==int== LEAK SUMMARY:
==int==    definitely lost: 4 bytes in 1 blocks
==int==    indirectly lost: 0 bytes in 0 blocks
==int==      possibly lost: 0 bytes in 0 blocks
==int==    still reachable: 0 bytes in 0 blocks
==int==         suppressed: 0 bytes in 0 blocks
==int== Rerun with --leak-check=full to see details of leaked memory
==int==
==int== For lists of detected and suppressed errors, rerun with: -s
==int== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

You Can Follow `valgrind`'s Directions to Get More Information

You can run it again with: `valgrind --leak-check=full build/valid-pointer`

You'll see the same as before but also with:

```
==int== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
==int==    at 0x48850C8: malloc (vg_replace_malloc.c:381)
==int==    by 0x1087E3: foo (valid-pointer.c:5)
==int==    by 0x10880B: main (valid-pointer.c:11)
```

You Can Follow `valgrind`'s Directions to Get More Information

You can run it again with: `valgrind --leak-check=full build/valid-pointer`

You'll see the same as before but also with:

```
==int== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
==int==    at 0x48850C8: malloc (vg_replace_malloc.c:381)
==int==    by 0x1087E3: foo (valid-pointer.c:5)
==int==    by 0x10880B: main (valid-pointer.c:11)
```

This tells you what `malloc` you forgot to `free`

- `main` called `foo`, then `foo` called `malloc`, therefore the `malloc` on line 5 leaked

Our Full Example Should Make Sure We're Not Out of Memory

```c
#include <stdio.h>
#include <stdlib.h>

int *foo(void) {
    int *p = malloc(sizeof(int));
    if (p == NULL) {
        exit(EXIT_FAILURE);
    }
    *p = 1;
    return p;
}

int main(void) {
    int *p = foo();
    printf("*p: %d\n", *p);
    free(p);
    return EXIT_SUCCESS;
}
```

The `exit` Function Immediately Ends Your Program

Its function prototype in the C standard library is:

```
void exit(int status);
```

This behaves the same as returning from `main`

- The `status` tells the OS whether there was an issue with your program

For this course we can just use `EXIT_SUCCESS` or `EXIT_FAILURE`

We Should Not Use the Memory After We `free`

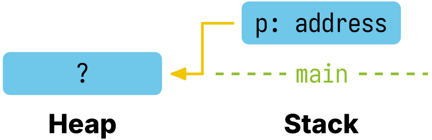
```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int *p = malloc(sizeof(int));
    *p = 14;
    free(p);
    printf("*p: %d\n", *p);
    return EXIT_SUCCESS;
}
```

----- main -----

Heap **Stack**

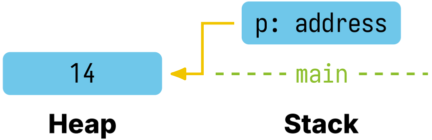
We Should Not Use the Memory After We free

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int *p = malloc(sizeof(int));
    *p = 14;
    free(p);
    printf("*p: %d\n", *p);
    return EXIT_SUCCESS;
}
```



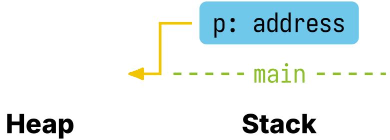
We Should Not Use the Memory After We `free`

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int *p = malloc(sizeof(int));
    *p = 14;
    free(p);
    printf("*p: %d\n", *p);
    return EXIT_SUCCESS;
}
```



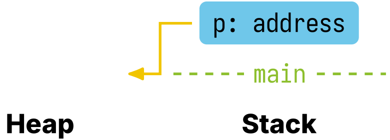
We Should Not Use the Memory After We `free`

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int *p = malloc(sizeof(int));
    *p = 14;
    free(p);
    printf("*p: %d\n", *p);
    return EXIT_SUCCESS;
}
```



We Should Not Use the Memory After We `free`

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int *p = malloc(sizeof(int));
    *p = 14;
    free(p);
    printf("*p: %d\n", *p);
    return EXIT_SUCCESS;
}
```



The Value We Read from Memory is Undefined After the `free`

The issue in the previous slide is called use after free

- You may also hear that `p` is a dangling pointer

It is good practice to set the pointer to `NULL` after freeing:

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

The Value We Read from Memory is Undefined After the `free`

The issue in the previous slide is called use after free

  You may also hear that `p` is a dangling pointer

It is good practice to set the pointer to `NULL` after freeing:

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

Now we'll see a segmentation fault immediately instead of an undefined value

You Also Cannot Call `free` Twice on the Same Pointer

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int *p = malloc(sizeof(int));
    *p = 14;
    free(p);
    free(p);
    printf("*p: %d\n", *p);
    return EXIT_SUCCESS;
}
```

You'll get a run-time error like:

```
free(): double free detected in tcache 2
```

`valgrind` Can Help You Debug the Two Previous Issues

`valgrind` will let you know if you're accessing invalid memory
(likely because of a use-after-free or dangling pointer)

`valgrind` will also let you know which two lines called `free`
(this helps you debug the double free)

Let’s Define Some Functions That Work on Arrays

```c
#include <stdio.h>
#include <stdlib.h>

void randomizeArray(int array[], int arrayLength) {
    for (int i = 0; i < arrayLength; ++i) {
        array[i] = rand() % 100 + 1;
    }
}

void printArray(int array[], int arrayLength) {
    printf("array:");
    for (int i = 0; i < arrayLength; ++i) {
        printf(" %d", array[i]);
    }
    printf("\n");
}
```

We Can Dynamically Allocate an Array!

```c
int main(void) {
    int arrayLength = 0;
    do {
        printf("Enter the length of the array: ");
        scanf("%d", &arrayLength);
    } while (arrayLength <= 0);

    int *array = malloc(sizeof(int) * arrayLength);
    if (array == NULL) { return EXIT_FAILURE; }

    randomizeArray(array, arrayLength);
    printArray(array, arrayLength);

    free(array);
    array = NULL;

    return EXIT_SUCCESS;
}
```

Use Dynamic Memory Only When Needed

Dynamic memory is tricky to get correct, you need to:

- Remember to `free` when you're done using the memory
- Don't try to use the memory after you `free` (use-after-free)
- Don't call `free` twice on the same pointer (double free)

You should only use it when:

- Your function needs to return a pointer to valid memory
- You do not know the amount of memory you need at compile-time