Lesson 4

Christian Schwarz, Jakob Krebs

18.11.2019

Contents

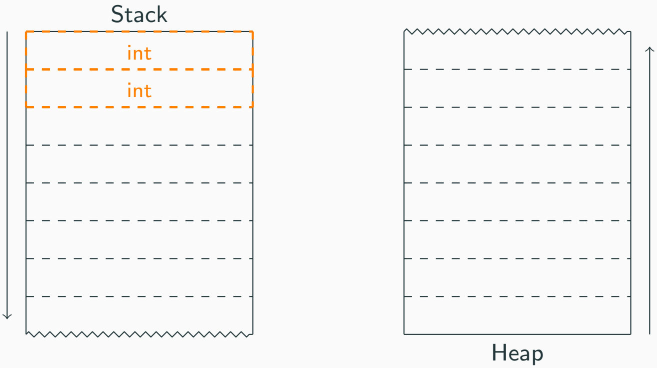
Source Code and Solutions

Sources and Solutions

- we publish all code written in this course at `https://github.com/jkrbs/c_lessons`
- we will publish example solutions of the tasks on same site
- send us questions or your solutions to c-lessons@deutschland.gmbh

dynamic memory

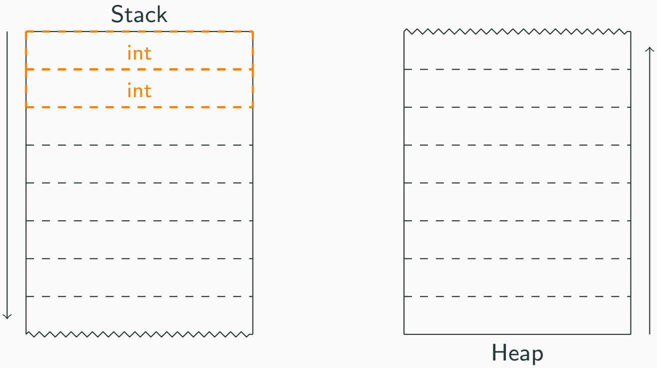
A closer look at memory



All local variables of functions are placed at the *stack*.

It grows and shrinks as variables are declared and functions return.

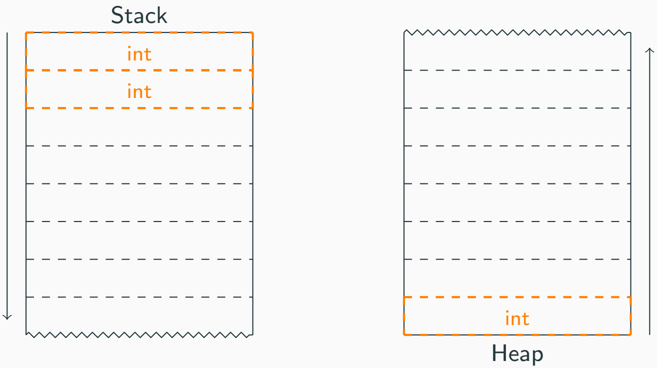
A closer look at memory



Dynamical memory is allocated on the *heap*.

The example shows a function with two local *int* variables.

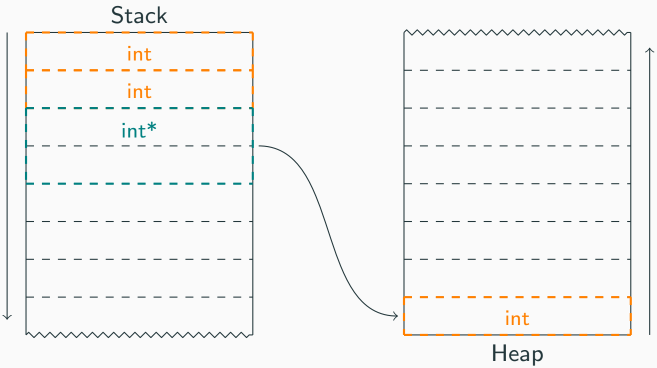
A closer look at memory




```
malloc(sizeof(int));
```

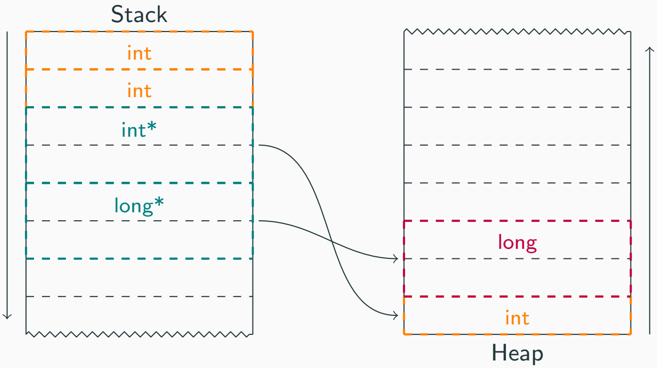
Reserve exactly the amount of memory an *int* variable takes.

A closer look at memory



`int *new_block = malloc(sizeof(int));` The adress of that memory block is stored in an *int* pointer.

A closer look at memory



malloc() just needs to know the size of the block it reserves.

Let us allocate a *long* variable as well.

malloc() in detail

The function declaration might be a little bit confusing:

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

- *size* is the size of the reserved block in **bytes**.
  If you want to use that block *seriously*, pass the size of an actual type (e.g. *sizeof(int)*).
- A *void* pointer is returned since *malloc()* does not know how you want to use the reserved block. By assigning it to a regular pointer variable it is automatically converted to that type.

Tidying up

Unlike normally declared variables, dynamically allocated storage is not automatically released when the function returns.

```
void foo(void) {
        int *bar = malloc(sizeof *bar);
}
```

With the pointer *bar* being removed from the stack, we havo no reference on its allocated memory and those four bytes are blocked forever!

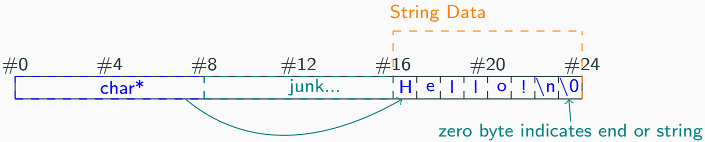
```
free(void *ptr);
```

Pass any pointer to previously allocated memory to *free()* and it gets realeased.

Slightly Better User Input

The length of Strings

Last time we learned that we always need to pass the size of an array together with the pointer to it. So if strings are just `char` arrays, why do `puts` etc not require a size?



- Strings avoid the length problem by always storing a zero byte after them.
- For this reason C Strings are also called "Zero terminated strings".
- Zero means the literal byte value 0 here, not the textual number (which is ascii index 48).
- We can represent this character value using `'\0'`.
- Case study: `puts("Foo!\0 Bar!");` Will only output `Foo!`.
- For this reason we use arrays for arbitrary byte sequences instead.

string utility functions 1

Here are some libc functions for handling strings. They all reside in the `<strings.h>` header file. For a full list, see http://www.cplusplus.com/reference/clibrary/

- `size_t strlen(char * str);`
  Returns the length of the string (not including the null terminator).
- `size_t strcmp(char * str1, char * str2);`
  Compares the two strings. returns 0 when equal, otherwise a value with the sign of of str1[n] - str2[n], where n is the index of the first differing character

string utility functions 2

Oftentimes one searches for a sequence in a string. These functions can help.

- `char * strchr(char * str, char c);`
  Returns a pointer to the first occurence of `c` in `str`. If none is found, returns `NULL`.
- `char * strstr (char* str, char* substr);`
  Returns a pointer to the first occurence of `substr` in `str`. If none is found, returns `NULL`.
- `size_t strcspn(char *str, char *charset);`
  Returns the first index of any character out of `charset` in `str`. If none is found, returns the index of str's zero byte.

fgets

`fgets` is a function from the C standard library that allows us to read in a string. It reads characters into a buffer until it's after a newline or the buffer is full.

On success, `fgets` always leaves space and puts in a trailing zero byte afterwards. The function signature as follows:

```
char* fgets(char *str, int count, FILE *stream );
```

- `str` wants a pointer to a char buffer to store the read in data in
- `count` wants the size of the `str` buffer to avoid overflow. Because of the zero byte this means that we are reading in at most `count - 1` characters.
- `stream` wants the byte stream to read from. In our case, this will be `stdin`, which is then input stream from the terminal, but it could also be a handle for a file stored on disk
- On success, fgets returns `str`, on error it returns `NULL` A full buffer is still a success, error means the stream closed etc.

using fgets

```c
#include <stdio.h>
int main(int argc, char** argv){
    char buffer[32];
    if(fgets(buffer, sizeof(buffer), stdin) != NULL){
        // For consistency, we remove
        // the potential trailing newline
        buffer[strcspn(buffer, "\n")] = 0;
        printf("We received: '%s'\n", buffer);
    }
    else{
        puts("input error");
    }
}
```

sscanf

`int sscanf (char* str, char* format, ...);` Works exactly like `scanf`, but scans `str` instead of `stdin`. Can be used together with fgets for sane user input:

```c
#include <stdio.h>
int main(int argc, char** argv){
    char buffer[32];
    int res;
    if(fgets(buffer, sizeof(buffer), stdin) != NULL){
        if(sscanf(buffer, "%i", &res) == 1){
            printf("we parsed %i!\n", res);
            return 0;
        }
    }
    puts("input error");
    return 1;
}
```

memcpy, memmove and memset

Like with strings, c also as a few useful functions for dealing with raw arrays. Counterintuitively, these are also found in `<string.h>`.

- `void* memcpy (void* destination, void* source, size_t num);`
  Copys `num` bytes from source to destination.
  No Zero Termination. There is no error condition, since the only way this can fail is causing a Segfault.
  Returns destination (usually ignored).
- `void* memmove (void* destination, void* source, size_t num);`
  Like memcopy, but can deal with overlapping source and destination.
- `void * memset (void* buffer, char value, size_t num);`
  Sets `num` bytes of `buffer` to `value`