C introduction

Dynamic memory management

Contents

Runtime conditions

Static C arrays are great if you already know at *compile time* how many elements you will need later at *runtime*.

Seriously, how often is this the case?

Runtime conditions

Static C arrays are great if you already know at *compile time* how many elements you will need later at *runtime*.

Seriously, how often is this the case?

Never. Unless your program does not take *any* user input, you cannot determine how much data you will have to store.

This is where dynamic memory management comes into play.

Explicit allocation

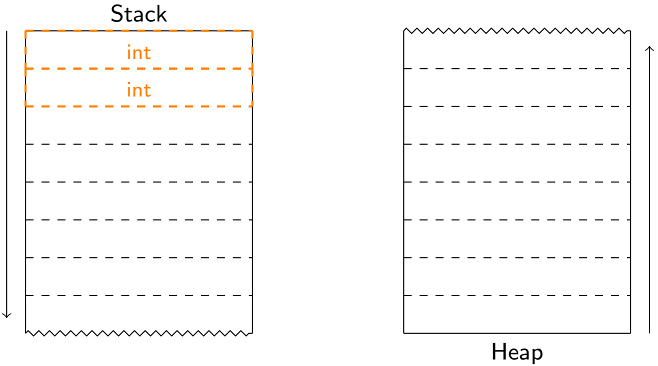
All the variables and arrays you have used so far were placed in memory automatically. In dynamic memory management, you have to *allocate* parts memory to identifiers on your own.

There are four functions in the standard library that do almost all the work for you:

- *malloc()*: Allocate a block of memory
- *calloc()*: Allocate a block of memory and initialize it
- *realloc()*: Alter the size of a block of memory
- *free()*: Release a block of memory

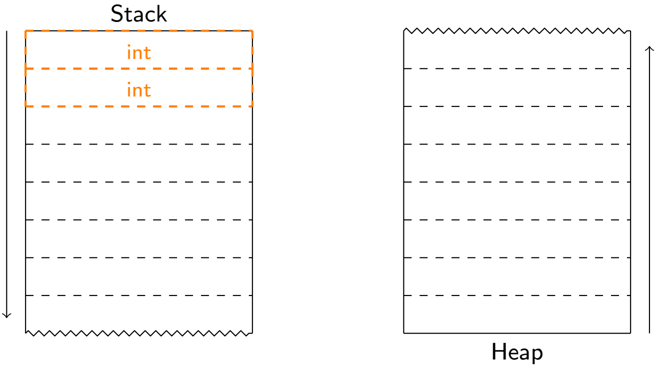
They are declared in *stdlib.h*.

A closer look at memory



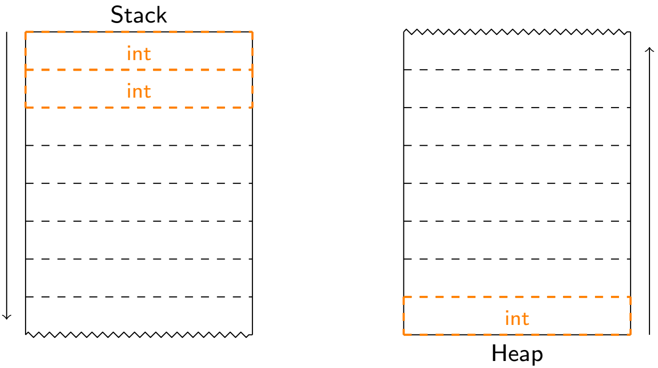
All local variables of functions are placed at the *stack*.
It grows and shrinks as variables are declared and functions return.

A closer look at memory



Dynamical memory is allocated on the *heap*.
The example shows a function with two local *int* variables.

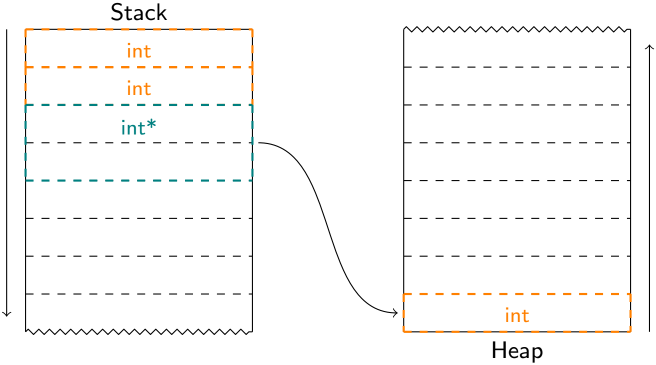
A closer look at memory




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

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

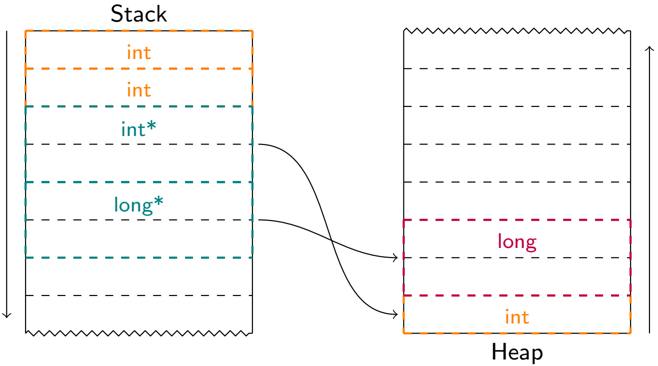
A closer look at memory




```
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_t* is an **unsigned integer** type.
  Any positive integer number (e.g. an *int* > 0) will do the job.
- *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.

Casting or not casting. . .

Some people claim you had to explicitly *cast* the return value of *malloc()*.

Casting or not casting...

Some people claim you had to explicitly *cast* the return value of *malloc()*.

This is outdated.

Casting or not casting...

Some people claim you had to explicitly *cast* the return value of *malloc()*.

This is outdated.

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

has the same result as

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

while the second one contains a redundant *cast* and if you want to change the type of *block* later, you will have to hit more keys. Consider:

```
int *block = malloc(sizeof *block); /* gold standard */
```

. . . that is the question

Confused?

. . . that is the question

Confused?

Do not typecast the result of malloc() & Co.

Tidying up

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

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

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. If you pass pointers on other things, undefined behaviour occurs (most likely program crashes).

Reserving large chunks

To get a dynamic array of a certain *type* and *length*, you have to

- Pass the block size *length* $*$ *sizeof*(*type*) to *malloc()*
- Assign the return value to a pointer to *type*

int array with 42 elements:

```c
int *field = malloc(42 * sizeof(*field));
```

Since the size of your dynamically allocated array is unknown at compile time, you cannot use *sizeof* to get its length. Save it in its own variable!

With the help of pointer arithmetic, you can use the dynamic array like a "normal" one.

The fancy alternative

```
void *calloc(size_t nmemb, size_t size);
```

- Allocates a block of *nmemb* ∗ *size* bytes, where *nmemb* is supposed to be the array's length and *size* the size of its type.
- The whole block is filled with *0*s

```
int field_length = 42;
int *field = malloc(field_length * sizeof(*field));
for (int i = 0; i < field_length; i++)
    field[i] = 0;
```

↓ Feel the difference ↓

```
int field_length = 42;
int *field = calloc(field_length, sizeof(*field));
```

Resizing arrays

Now we come to the point that motivated us to use dynamic arrays:

```
void *realloc(void *ptr, size_t size);
```

- *ptr* is a pointer to a dynamically allocated memory block
- *size* is the wanted new size of the memory block
- The return value is a pointer to the resized block

Note that the new *size* can be greater or smaller than the old one!

- If it's smaller, you may lose some data at the end of the block
- If it's greater, the block may be at a different location in the memory $\rightarrow$ *ptr* is freed then, also the additional bytes are not initialized

Clean up your code

Passing arrays between functions can be complicated if you store the pointer and the length seperately.

Do you remember a way to keep different things together?

Clean up your code

Passing arrays between functions can be complicated if you store the pointer and the length seperately.

Do you remember a way to keep different things together?

```
struct int_array {
    int *field;
    int length;
}
```

This allows you to use the *struct int_array* as a single argument or return value. Even better: pass a pointer on that structure.

Strings from pointers to *char*

By handling strings as dynamic *char* arrays you can alter their size which is needed for many operations on them.

- *strlen()* returns the actual length of a string (up to '\0' character)
- *strncpy()* copies a string into a dynamically allocated block

```
char conststr[] = "Hello";              /* not of much use */
int bufsize = strlen(conststr) + 1;     /* add '\0' char */
char *str = calloc(bufsize, sizeof(*str));
str = strncpy(str, conststr, bufsize); /* ready to go */
```

These functions and others are declared in *string.h*.

```
$ man string.h
```

Exercises

- You are now able to solve tasks 26, 27 and 28.
- You additionally are now able to visit the advanced C course.