Lesson 5

Christian Schwarz, Jakob Krebs

25.11.2019

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

Typedef

typedefs

In previous lessons we said that `size_t` or `bool` are not native to the C type system, but library types built upon it. But how do you define a type?

```
1  typedef int foo_t;
2
3  // we can now use foo_t just like short:
4  void foo(){
5      foo_t x = 3;
6      short y = 4;
7      foo_t z = x + y;
8  }
```

Using typedefs we can abstract away plattform differences, shorten long type names (function pointers) etc. But don't go out typedef'ing every integer you use :).

typedef with structs

In many other programming languages, (including `C++`). We can use structs and enums just like the predefined types. So why do we have to write `struct foo` in `C`, instead of just `foo`?. Well we don't **have** to. We can typedef our struct definition to be it's own type: `typedef struct foo foo;`. And we can combine this with the definition:

```
1  // struct point is the struct name, can be used as normal
2  // the struct and typedef name can also be different
3  typedef struct point{
4      int x, y;
5  }point; //point now means the same as struct point
6
7  point add(point p1, point p2){
8      point sum = {p1.x + p2.x, p1.y + p2.y};
9      return sum;
10 }
```

legacy code with unnamed structs

In old code the struct name is also left out:

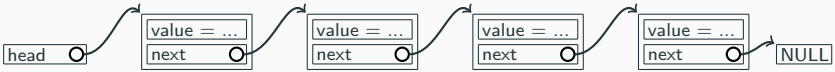
```
1 typedef struct{
2     //point* p; //this wouldn't work
3     int x, y;
4 }point;
5 // C++ compilers would complain about a type conflict here
6 // struct point;
```

This can cause problems in forward declarations and when you want to have pointers to the struct inside itself, since the typedef isn't available until the end.

Therefore we don't do that.

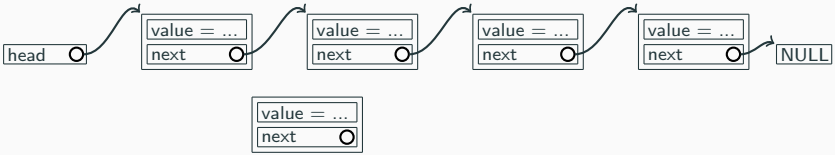
data structures

linked lists



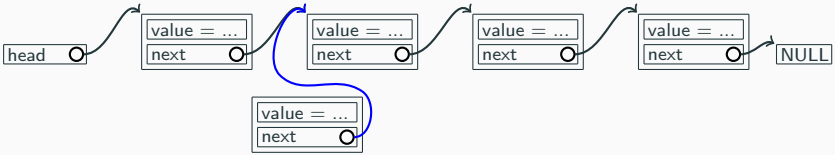
- Now we can build lists of this structures.

linked lists



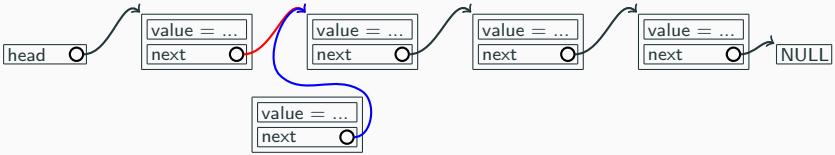
- Now we can build lists of this structures.
- If we want to insert a value at a specific position...

linked lists



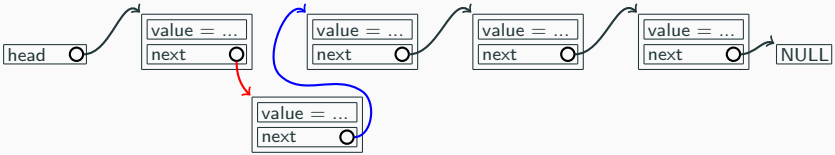
- Now we can build lists of this structures.
- If we want to insert a value at a specific position...
- ... we just have to change the pointers.

linked lists



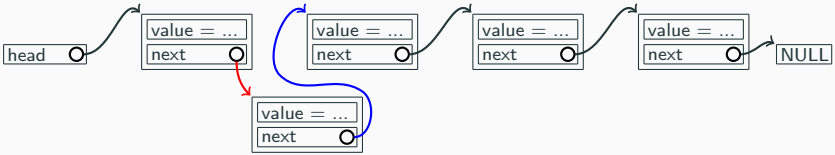
- Now we can build lists of this structures.
- If we want to insert a value at a specific position...
- ... we just have to change the pointers.

linked lists



- Now we can build lists of this structures.
- If we want to insert a value at a specific position...
- ... we just have to change the pointers.

linked lists



- Now we can build lists of this structures.
- If we want to insert a value at a specific position...
- ... we just have to change the pointers.

- Additionally we can easily iterate through this list.
- But only in one direction.

Intermediate Task: Linked List Insertion

No we will try to implement in code what we just saw. You can start out with the code you can find at

`https://jkrbs.github.io/c_lessons/tasks/intermediate_5_linked_list_insertion.c`

Or just click on
`Lesson 5 Intermediate Task Starting Point: Linked List Insertion`

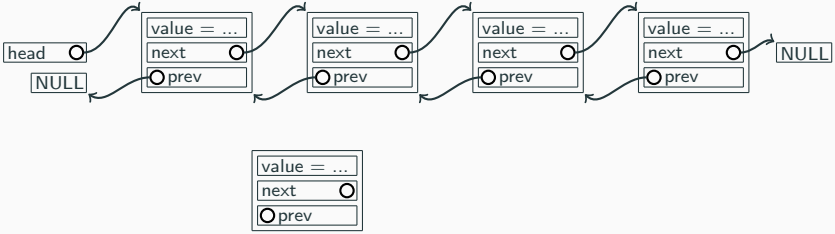
on our website (`https://jkrbs.github.io/c_lessons`).

Task: Write the function `list_insert` and test if your code works!

Double-linked lists

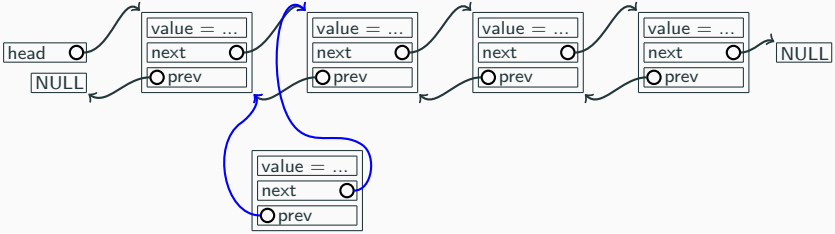
- With a pointer in each direction, we can iterate forwards and backwards.

Double-linked lists



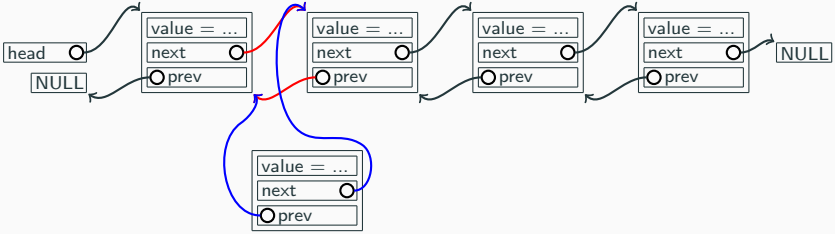
- With a pointer in each direction, we can iterate forwards and backwards.
- When inserting a value we now have to change a little more.

Double-linked lists



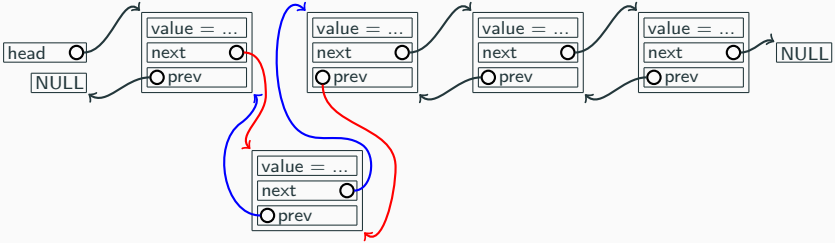
- With a pointer in each direction, we can iterate forwards and backwards.
- When inserting a value we now have to change a little more.

Double-linked lists



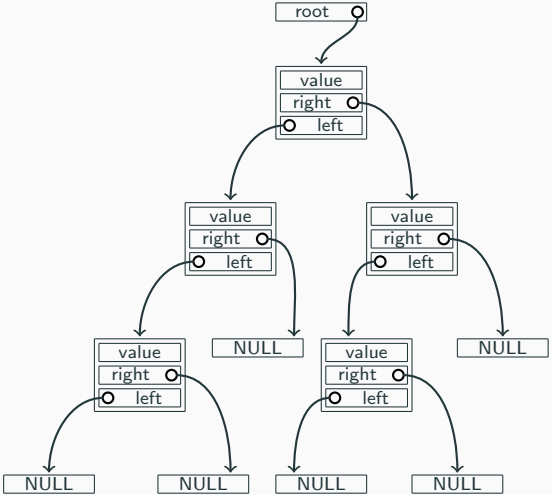
- With a pointer in each direction, we can iterate forwards and backwards.
- When inserting a value we now have to change a little more.

Double-linked lists



- With a pointer in each direction, we can iterate forwards and backwards.
- When inserting a value we now have to change a little more.

Binary trees



- To improve the performance of certain algorithms over lists, we can usse binary trees instead.
- Typical such algorithms are lookup, insertion and removal of elements while maintaining an ordering.

Macros

define

We have talked about `#include <foo.h>`, which copies the content of foo.h into the file. This is done using the C preprocessor (as indicated by the `#`).

Using the preprocessor we can also map macro identifiers to strings. The occurences of the macro will then be replaced in our code during compilation / preprocessing.

```
1 #define SIZE 10
2 #define PLAYER_CHAR '@'
3 #define PI 3.14159265
```

Everywhere we use `SIZE`, it will be replaced with `10` by the preprocessor.

ifdef

There are many macros provided by our compiler, for example identifying the current os. We can check for these using another preprocessor feature: `#if`

```
1 #ifdef __linux__
2 #   include "your_linux_lib.h"
3     int some_linux_only_variable = 5;
4 #elif defined(_WIN32)
5 #   include "your_windows_lib.h"
6 #else
7 #error your operating system is not supported
8 #endif
```

Here we check, if the macros `__linux__` or `_WIN32` are defined in our file. The blocks inside the if statements are completely ignored if the condition is false.

Libraries and Header Files

header files

When our projects become large, we want to split our code into multiple files. But still our main file needs to know about all the functions. So we write the signatures into a header file (extension is `.h` by convention)

```
1 //foo.h
2 typedef struct foo{
3     int x;
4 }foo;
5
6 void some_function(foo* s, int x);
```

We can include this file by writing

```
1 #include "foo.h"
```

in our C files, where we want to use those functions.

other things in header files

Usually the follwing things are done in header files

- including other headers
- struct forward declarations (if we want to hide the implementation)
- macros and typedefs

But: Headers should only include the headers they really need for the forward declarations. Headers only used for the implementation should be put in the C file!

Intermediate Task: Linked List Library

No we will try to factor out our Linked List into a library. If you haven't already, the code is still at.

`https://jkrbs.github.io/c_lessons/tasks/intermediate_5_linked_list_insertion.c`

Or just click on `Lesson 5 Intermediate Task Starting Point: Linked List Insertion`

on our website (`https://jkrbs.github.io/c_lessons`).

The Task:

- Move our code into a little library:
  - `list.c` which contains our implementation
  - `list.h` which forward declares our list interface.
- Add a `EMPTY_LIST` macro that can be used instead of `NULL`!
- Try to use the library from `main.c`! You can compile both c files at once using `gcc list.c main.c`, no need to specify header files.
- Try and see what happens if you include the `list.h` header twice!

Multiple inclusions

Sometimes with libraries we end up in a situation like this:

```
1 // matrix.h
2 #include "math.h"
3
4 // vector.h
5 #include "math.h"
6
7 // main.c
8 #include "vector.h"
9 #include "matrix.h"
```

What will happen during compilation?

Multiple inclusions

Sometimes with libraries we end up in a situation like this:

```
1 // matrix.h
2 #include "math.h"
3
4 // vector.h
5 #include "math.h"
6
7 // main.c
8 #include "vector.h"
9 #include "matrix.h"
```

What will happen during compilation?

- The math header will be included twice.
- This might cause compilation errors if structs are redeclared.

Include Guards

How can we fix this?

```
1 // math.h
2
3 // some math definitions...
```

Include Guards

How can we fix this?

- First We add a macro, indicating that the header has been included.

```
1 // math.h
2 #define MATH_H
3
4 // some math definitions...
```

Include Guards

How can we fix this?

- First We add a macro, indicating that the header has been included.
- Then we check whether that macro has already been defined.

```
1 // math.h
2 #ifndef MATH_H
3 #define MATH_H
4
5 // some math definitions...
6
7 #endif
```

Include Guards

How can we fix this?

- First We add a macro, indicating that the header has been included.
- Then we check whether that macro has already been defined.
- This concept is called include guards or header guards, and should be used on virtually all headers

```
1 // math.h
2 #ifndef MATH_H
3 #define MATH_H
4
5 // some math definitions...
6
7 #endif
```

Pragma once

Alternatively, you can use `#pragma once`.

This is nonstandard, but supported by all major compilers. For this course, we recommend using it.

```
1 // math.h
2 #pragma once
3
4 // some math definitions...
```

Cyclic include dependencies

Sometimes with libraries we end up in a situation like this:

```
1 //matrix.h
2 #include "vector.h"
3
4 //vector.h
5 #include "matrix.h"
```

What is the compiler supposed to do?

Cyclic include dependencies

Sometimes with libraries we end up in a situation like this:

```
1 //matrix.h
2 #include "vector.h"
3
4 //vector.h
5 #include "matrix.h"
```

What is the compiler supposed to do?

- This code will just refuse to compile, since both files expect the other but somebody has to be looked at first.
- Sometimes you can factor out the shared portion into a third header that both files include
- Sometimes you can just forward declare the one struct that you need from the other header.
- Remember: Only include the headers that are necessary for the declarations! Everything else belongs in the C file!