C Praktikum

Advanced Pointers

Eugen Betke, Nathanael Hübbe,
Michael Kuhn, Jakob Lüttgau, Jannek Squar

2018-11-26

Warning

This is a dive under the hood.
We will see, and hopefully understand many details
which still elude some seasoned programmers.

Do not expect this presentation to align with what you expect,
but expect to have some fun with the unexpected.

The Five Pointer Rules

1. any object in C can be pointed to
2. pointer declarations are read inside out
3. a[b] $\Leftrightarrow$ *(a + b) and a->b $\Leftrightarrow$ (*a).b
4. value of ptr + a is ptr + a***sizeof**(*ptr)
5. arrays are not pointers, they decay to pointers

Rule 1

any object in C can be pointed to

Both, the type of any object and its associated pointer, can be written down in C.
To turn a variable declaration into a pointer declaration, just add a * in front of the variable name.
(An extra set of parentheses () may be needed.)

Example time

Rule 2

pointer declarations are read inside out

- start at the variable name
  (type name for **typedef**)
- follow operator precedence
  - array subscript ([]) and function call (())
    take precedence over pointer dereference (*)

Rule 2

pointer declarations are read inside out

Example 1:

```
struct foo *(*bar)[5]; //bar is
```

Rule 2

pointer declarations are read inside out

Example 1:

```
struct foo *(*bar)[5]; //bar is
struct foo *(*bar)[5]; //... a pointer to an
```

Rule 2

pointer declarations are read inside out

Example 1:

```
struct foo *(*bar)[5]; //bar is
struct foo *(*bar)[5]; //... a pointer to an
struct foo *(*bar)[5]; //... array of size 5 (precedence!),
```

Rule 2

pointer declarations are read inside out

Example 1:

```
struct foo *(*bar)[5]; //bar is
struct foo *(*bar)[5]; //... a pointer to an
struct foo *(*bar)[5]; //... array of size 5 (precedence!),
struct foo *(*bar)[5]; //... elements are pointers to
```

Rule 2

pointer declarations are read inside out

Example 1:

```
struct foo *(*bar)[5]; //bar is
struct foo *(*bar)[5]; //... a pointer to an
struct foo *(*bar)[5]; //... array of size 5 (precedence!),
struct foo *(*bar)[5]; //... elements are pointers to
struct foo *(*bar)[5]; //... structs of type foo
```

Rule 2

pointer declarations are read inside out

Example 2:

```
void*(*foo[3])(int) //foo is an
```

Rule 2

pointer declarations are read inside out

Example 2:
`void*(*foo[3])(int)` //`foo` is an
`void*(*foo[3])(int)` //... array of size 3 (precedence!),

Rule 2

pointer declarations are read inside out

Example 2:

```
void*(*foo[3])(int) //foo is an
void*(*foo[3])(int) //... array of size 3 (precedence!),
void*(*foo[3])(int) //... which are pointers to
```

Rule 2

pointer declarations are read inside out

Example 2:

`void*(*foo[3])(int)` //`foo` is an

`void*(*foo[3])(int)` //... array of size 3 (precedence!),

`void*(*foo[3])(int)` //... which are pointers to

`void*(*foo[3])(int)` //... functions (precedence!),

Rule 2

pointer declarations are read inside out

Example 2:

```
void*(*foo[3])(int) //foo is an
void*(*foo[3])(int) //... array of size 3 (precedence!),
void*(*foo[3])(int) //... which are pointers to
void*(*foo[3])(int) //... functions (precedence!),
void*(*foo[3])(int) //... which take an int argument
```

Rule 2

pointer declarations are read inside out

Example 2:

```
void*(*foo[3])(int) //foo is an
void*(*foo[3])(int) //... array of size 3 (precedence!),
void*(*foo[3])(int) //... which are pointers to
void*(*foo[3])(int) //... functions (precedence!),
void*(*foo[3])(int) //... which take an int argument
void*(*foo[3])(int) //... and return a void*
```

Rule 3

$a[b] \Leftrightarrow *(a + b)$ and $a->b \Leftrightarrow (*a).b$

Example time

Rule 4

value of ptr + a is ptr + a∗**sizeof**(∗ptr)

Example time

Rule 5

arrays are not pointers, they decay to pointers

- only & and **sizeof** operators do not trigger decay
- decay happens even in function declarations (because argument passing is a use)

Example time

The Five Pointer Rules

1. any object in C can be pointed to
2. pointer declarations are read inside out
3. a[b] $\Leftrightarrow$ *(a + b) and a->b $\Leftrightarrow$ (*a).b
4. value of ptr + a is ptr + a***sizeof**(*ptr)
5. arrays are not pointers, they decay to pointers

Warning: Only Rule 2 and Rule 5 hold true in C++.

Passing Multidimensional Arrays to Functions

Three methods available - what are the differences?

Example time

Storing Multidimensional Arrays in Objects

Pitfall: Can't store pointer to array of dynamic size in **struct**.

$\Rightarrow$ Must use untyped pointer and casts.

Example time

Function Pointers as Callbacks

Whenever a callback comes in handy...

Example time

Function Pointers for Customizable Behavior

Idea: Store function pointer in **struct** to make function call runtime decision $\Rightarrow$ polymorphic objects!

But that's for another day...

Summary

Summary

- only 5 simple pointer rules ...
- ... that allow us to do complex stuff
- real dynamic 2D arrays (envious, C++?)
- function pointers make function calls runtime decisions (callbacks and polymorphism)