Part VII

Pointers, Arrays & Memory Management

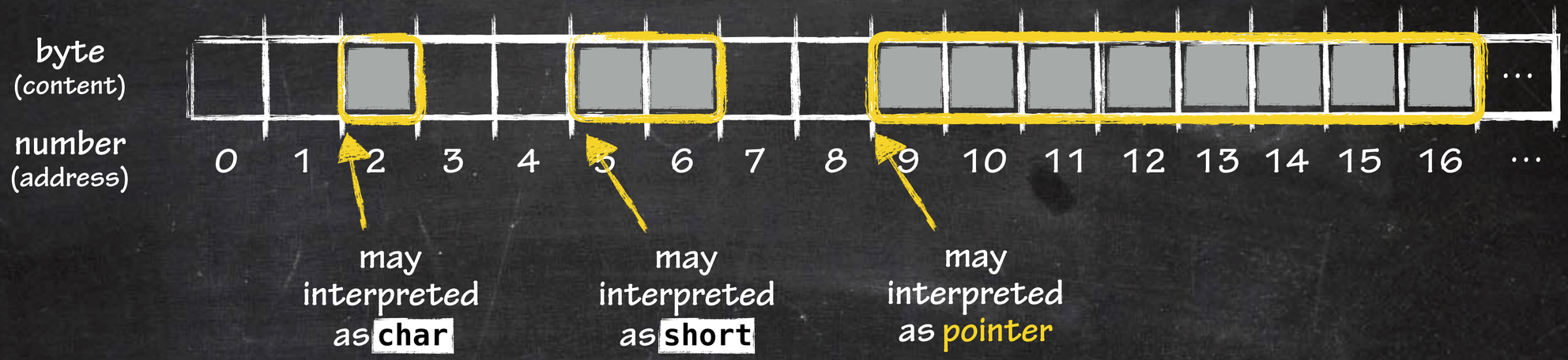
- Pointers
- Dynamic Memory Management
- Call-by-Reference
- Arrays
- Variable-Length Arrays
- Address Arithmetic
- Pointers to Arrays
- Pointer to Functions & Lambda Functions
- `void` Pointers and Casting

Pointers

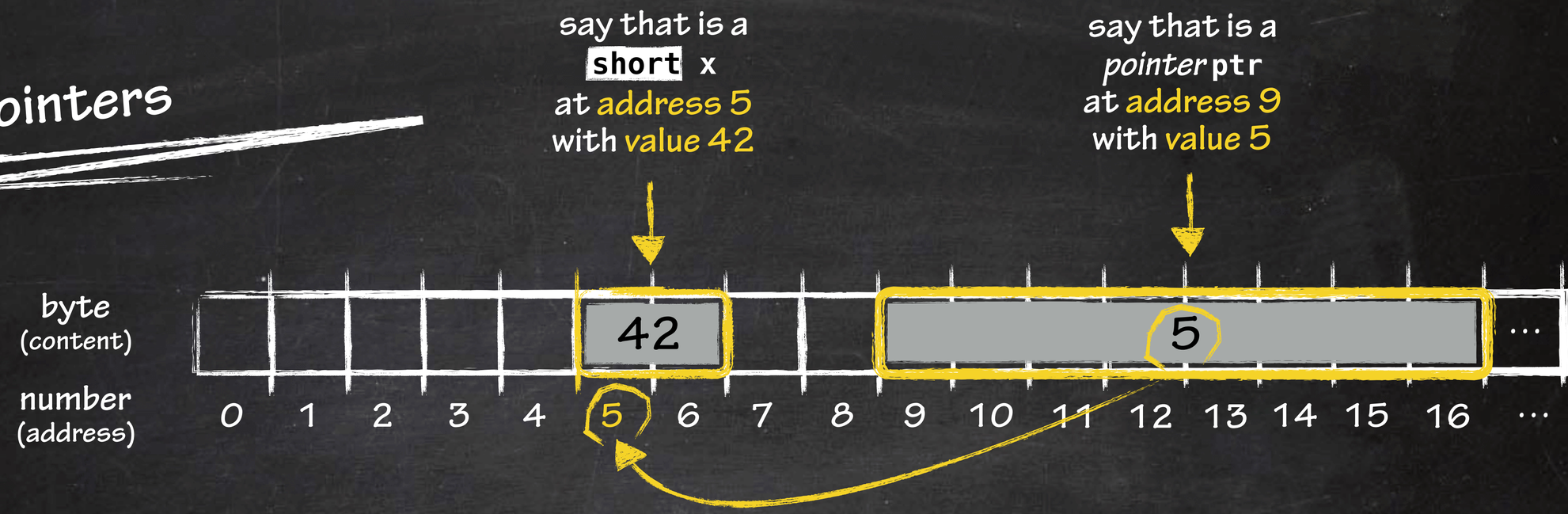
A pointer is a variable that contains the memory address of variable.

- typically of size 2, 4 or 8 bytes depending on machine architecture

Memory is organized as consecutive addressed memory cells (bytes)



Pointers



- The pointer `ptr` "points to" the variable `x` since the value of `ptr` (`5`) is the address at which `x` is stored.
- C supports member access operators to deal with addresses and pointers
  - Address operator: the unary operator `&x` gives the address of an object `x` (e.g., 5)
  - Dereferencing operator: the unary operator `*ptr` access the value to which `ptr` points to (e.g., 42)

(1) a pointer `z` that points to a variable of a particular *type* is declared as `type *z`

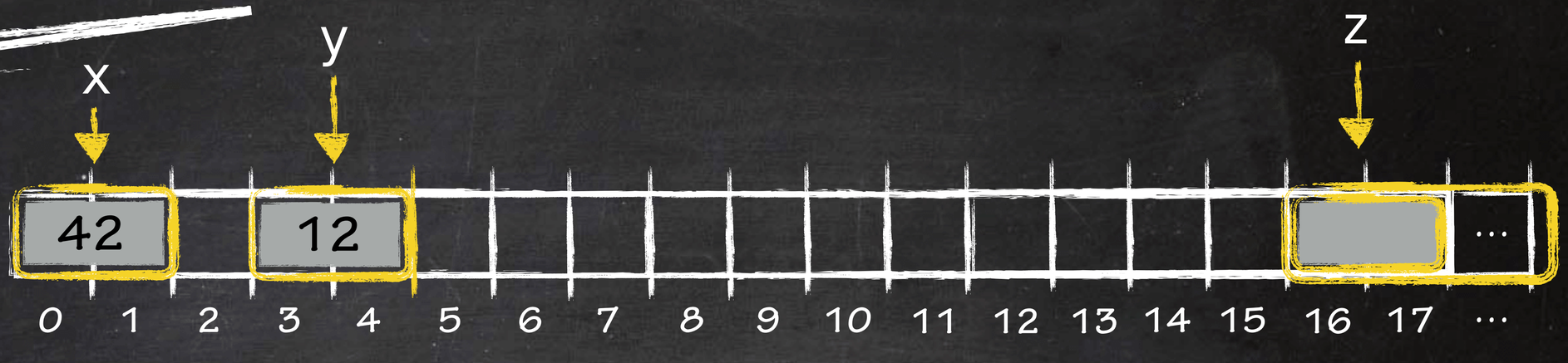
(2) a pointer `z` that points to a variable of *some* (unstated) type is a generic pointer declared as `void *z`

(3) generic pointers *cannot* be dereferenced

(4) alternative: pointer `z` to some character declared as `char *z`. Dereferencing `z` yields character (byte) value at address stored in `z`.

based on: The C Programming Language

Pointers



don't let the overloading of * confuses you

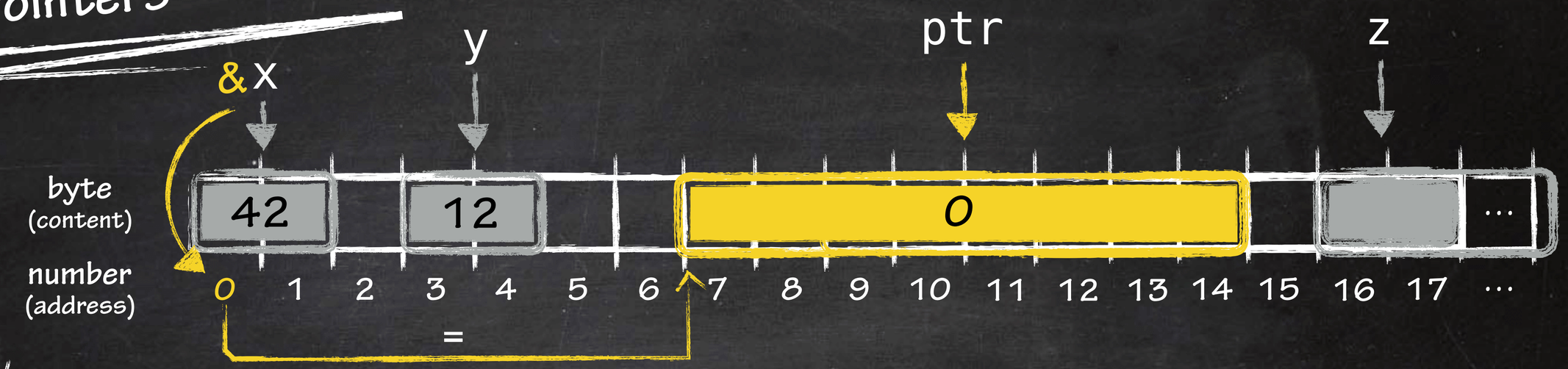
`type *ptr` declares a pointer to a variable of type *type*.

`*ptr` accesses the value of the memory address stored in `ptr`.

```
short  x   = 42,
       y   = 23,
       z[5];
short *ptr = &x;  // ptr is a pointer to short and points to x
y    = *ptr;      // y gets the value ptr points to, i.e., y is 42
*ptr =  15;       // x is now 15, since 15 is stored at address to which ptr points to (x)
ptr  = &z[0];     // ptr points now to the first element of the array z
```

based on: The C Programming Language

Pointers



don't let the overloading of * confuses you

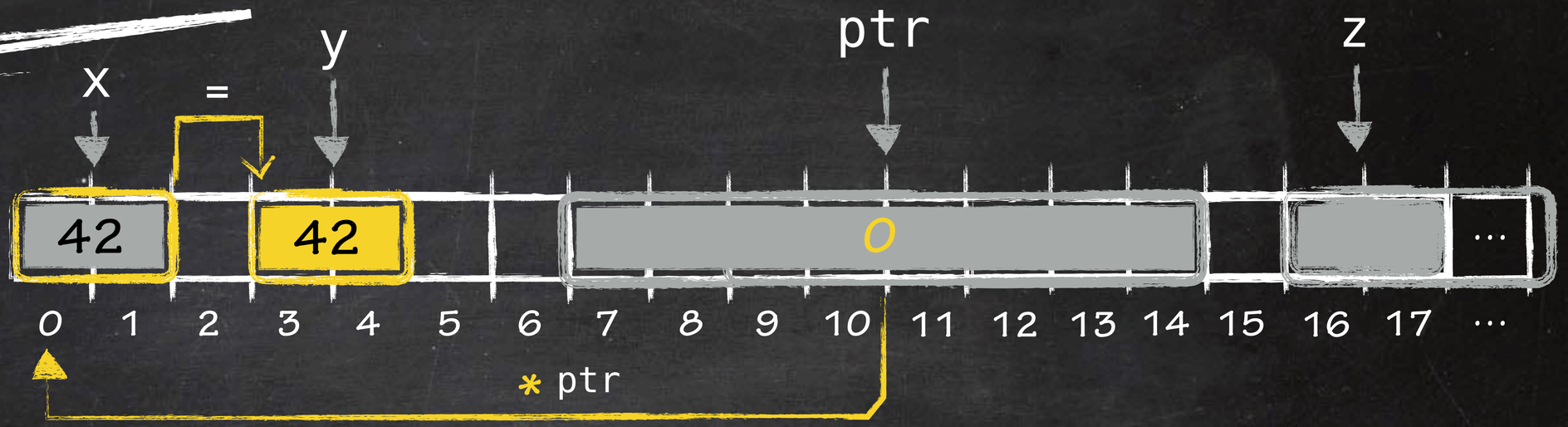
`type *ptr` declares a pointer to a variable of type *type*.

`*ptr` accesses the value of the memory address stored in `ptr`.

```
short  x   = 42,
       y   = 23,
       z[5];
short *ptr = &x;  // ptr is a pointer to short and points to x
y    = *ptr;      // y gets the value ptr points to, i.e., y is 42
*ptr =  15;       // x is now 15, since 15 is stored at address to which ptr points to (x)
ptr  = &z[0];     // ptr points now to the first element of the array z
```

based on: The C Programming Language

Pointers



don't let the overloading of * confuses you

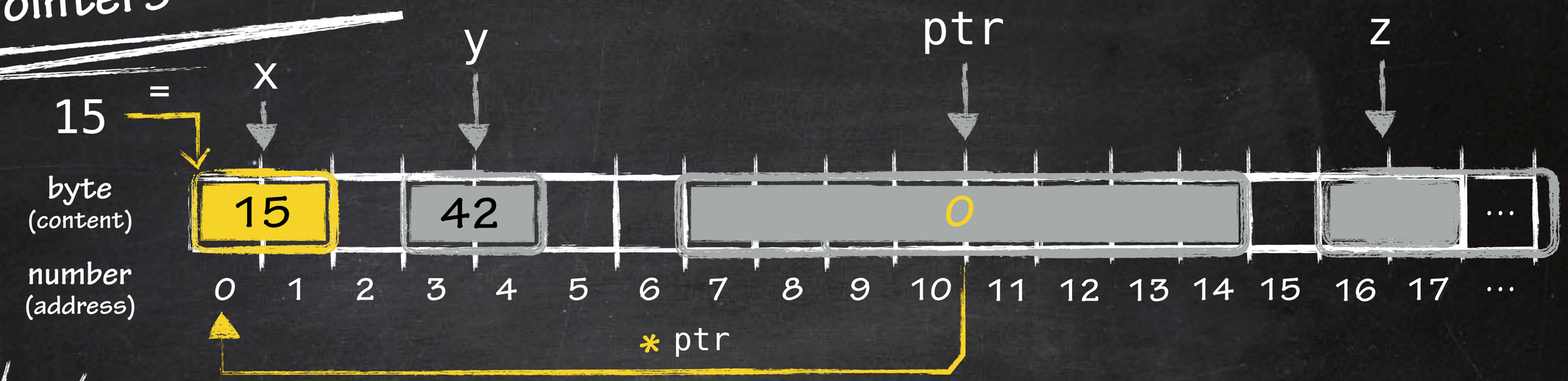
`type *ptr` declares a pointer to a variable of type *type*.

`*ptr` accesses the value of the memory address stored in `ptr`.

```
short  x   = 42,
       y   = 23,
       z[5];
short *ptr = &x;  // ptr is a pointer to short and points to x
y    = *ptr;      // y gets the value ptr points to, i.e., y is 42
*ptr =  15;       // x is now 15, since 15 is stored at address to which ptr points to (x)
ptr  = &z[0];     // ptr points now to the first element of the array z
```

based on: The C Programming Language

Pointers



don't let the overloading of * confuses you

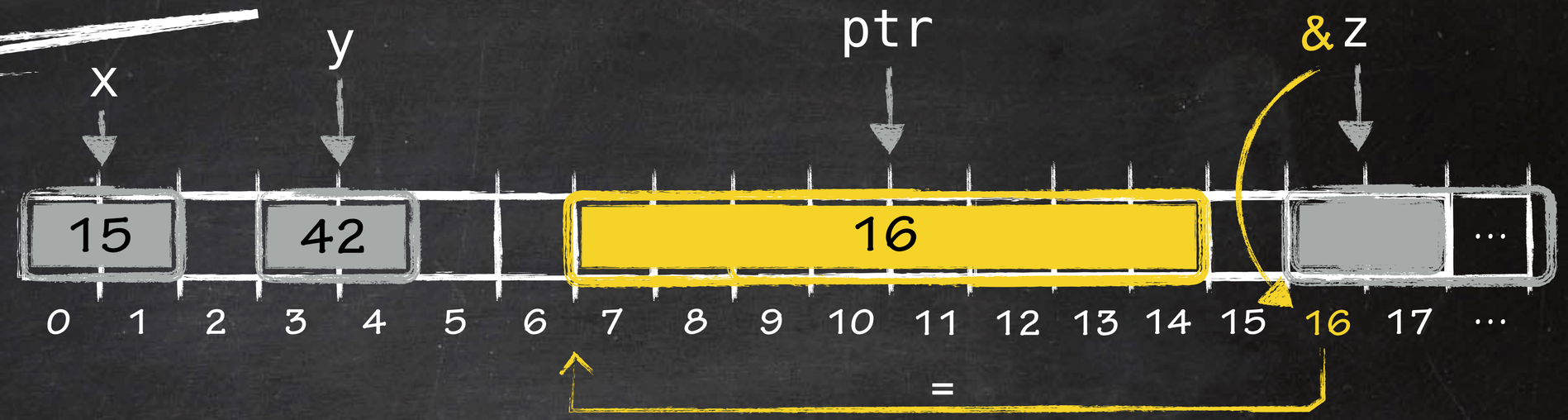
`type *ptr` declares a pointer to a variable of type *type*.

`*ptr` accesses the value of the memory address stored in `ptr`.

```
short  x   = 42,
       y   = 23,
       z[5];
short *ptr = &x;  // ptr is a pointer to short and points to x
y    = *ptr;      // y gets the value ptr points to, i.e., y is 42
*ptr =  15;       // x is now 15, since 15 is stored at address to which ptr points to (x)
ptr  = &z[0];     // ptr points now to the first element of the array z
```

based on: The C Programming Language

Pointers



don't let the overloading of * confuses you

`type *ptr` declares a pointer to a variable of type *type*.

`*ptr` accesses the value of the memory address stored in `ptr`.

```
short  x   = 42,
       y   = 23,
       z[5];
short *ptr = &x;  // ptr is a pointer to short and points to x
y    = *ptr;      // y gets the value ptr points to, i.e., y is 42
*ptr =  15;       // x is now 15, since 15 is stored at address to which ptr points to (x)
ptr  = &z[0];     // ptr points now to the first element of the array z
```

based on: The C Programming Language

Pointers & Virtual Memory

Avoid assignments of addresses by hand (unless you are in kernel-mode)

```
void *ptr = 5; // attention: leads to an segmentation fault (and crash of your program) in user-mode
```

Reason: the operating system provides virtual memory facilities to programs in user-mode (i.e., typically your program).

With virtual memory the operating system fakes that you can access more memory addresses as physical available (and more). In order to get this work, you must take addresses:

- from non-allocated storage class objects (i.e., auto, static, or thread local), or
- from allocated storage class objects where the operating system manages memory addresses to these objects

Exception: you access *physical memory* directly in user-space, or you write code for the kernel itself

Pointer Arithmetic

In principle, *pointers are normal unsigned number values*

- but capable to hold any memory address
- target for the dereferencing operator
- values are memory addresses
- assignment of values (value of other pointers, constants, or garbage) is possible

Applying *arithmetic operations on pointers* is called *pointer arithmetic*.

- `ptr1 - ptr1` used to compute the distance between two addresses `ptr1` and `ptr1`
- `ptr + off` used get the address starting at pointer `ptr` plus an integer offset `off`
- `ptr++` used to move pointer `ptr` to the next memory address (next char)
- `ptr--` used to move the pointer `ptr` to the previous memory address (previous char)
- … and all other arithmetic operators that are supported by C (not all reasonable in user-mode)

Pointer Casting

Explicit type cast can be applied to pointers

```
int *ptr_a = /* ... */;
char *ptr = (char *) ptr_a;
```

Consequence is, that dereferencing interprets the new type (i.e., `char` in the example) at the given memory (i.e., `*ptr` is first character of memory address `ptr_a` points to)

Null Pointer

```
#include <stddef.h>
int *ptr = NULL; // ptr is now a null pointer
```

The *null pointer* points to an special address indicating that the pointer does not point to a valid object. In C this value is the integer constant `0` resp. a macro `NULL` that expands to `0`.
Often used to

- indicate end of a list of unknown length
- abnormal condition or uninitialization