C RECAP

WHY C

- **The** system level programming language
- Many OS components still written in C (kernel, subsystems, …)

HELLO WORLD

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    puts("hello World");
    return EXIT_SUCCESS;
}
```

```
cc -std=c11 -Wall -Wextra hello_world.c -o hello_world
```

VARIABLES

```
int counter = 0;
double deltaTime = 1.0 / 60.0;
```

- **Always initialize**
- Type (`int`, `float`, …) defines value range / precision and thus the size of the variable

SIZE & SIGNEDNESS

```
sizeof(int) => 4 Byte
  Min: -2^31       = - 2 147 483 648
  Max:  2^31 - 1   =   2 147 483 647
```

```
sizeof(unsigned int) => 4 Byte
  Min: 0
  Max: 2^32 - 1    = 4 294 967 296
```

- Actual size depends on your platform
- Consider type aliases from `stdint.h` (e.g. `uint32_t`)

- Only `unsigned` over- / underflow well defined
- Conversion between `signed` / `unsigned` and different sizes is a common source of errors
  - Consider `-Wconversion` in addition to `-Wall -Wextra`

BOOLEAN

```c
#include <stdbool.h>

bool isAlive = true;
bool hasFood = false;

bool doingOk = isAlive && hasFood;
```

- Added in C99, but just an integers in disguise
- Prefer `bool` over `int` where it makes sense (cleaner code)

USER-DEFINED TYPES

```
enum Color {
    COLOR_RED,
    COLOR_BLUE,
};

struct Point {
    double x;
    double y;
};
```

- enums are just `int`s, like `bool` using them often results in cleaner code

```c
enum Color color = COLOR_RED;

struct Point p1 = {
    .x = 1.2,
    .y = 2.3,
    // other fields initialized to zero
};
```

- **Always initialize**

- `enum`, `struct`, (and `union`) often accompanied with a `typedef`

```c
struct Point {
    double x;
    double y;
};
typedef struct Point Point;
```

```c
Point p2 = { .x = 1.2, .y = 2.3 };
```

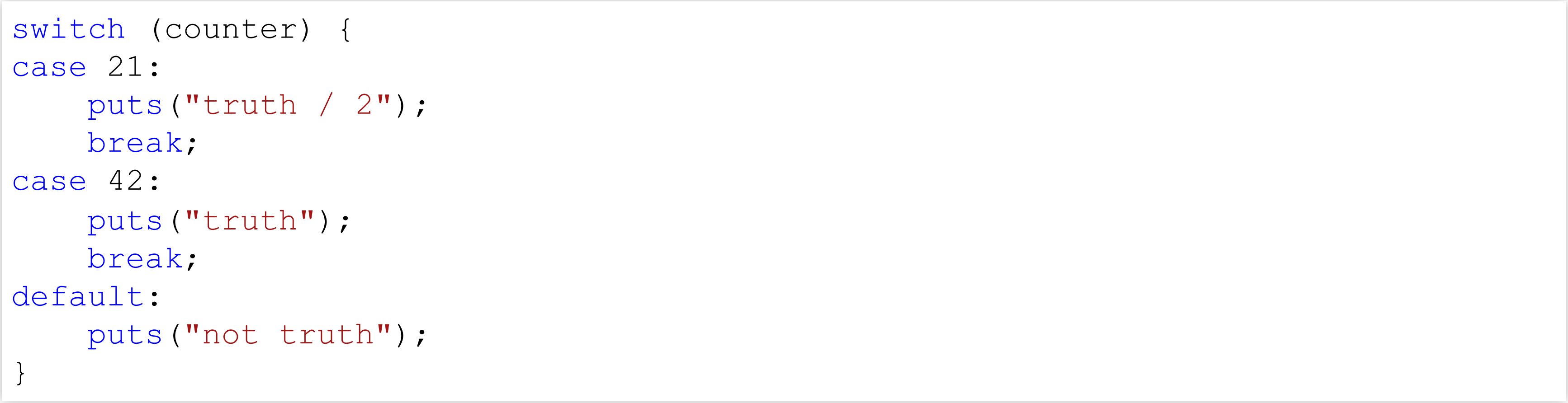
CONTROL-FLOW STATEMENTS

```
if (condition) {
    puts("Condition is true");
} else {
    puts("Condition is false");
}

while (condition) {
    puts("looping while condition is true");
}

do {
    // ...
} while (condition);
```

- Please do **not** omit braces { }

```c
for (int i = 0; i < size; ++i) {
    // ...
}
```

- Since C99 we can declare variables inside `for`, use this!
- Avoid complex `continue` / `break` behaviors



```
switch (counter) {
case 21:
    puts("truth / 2");
    break;
case 42:
    puts("truth");
    break;
default:
    puts("not truth");
}
```

- `break` super important here, otherwise **fall-through**
- Use `// fall-through` to suppress warnings if actually needed

FUNCTIONS

- Your bread ‘n’ butter in C
- Should be **self-contained**: result depends only on arguments, not some global variable
- **Pick a descriptive name!**

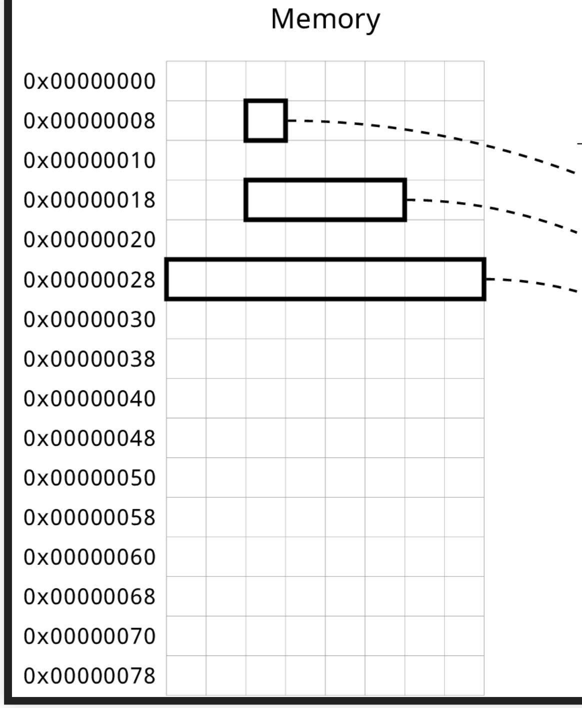
```
double square(double n);

bool isPrime(int n);

double xfb(int a, int b, double pre); // 🤔
```

POINTERS

	Memory							
0x00000000								
0x00000008								
0x00000010								
0x00000018								
0x00000020								
0x00000028								
0x00000030								
0x00000038								
0x00000040								

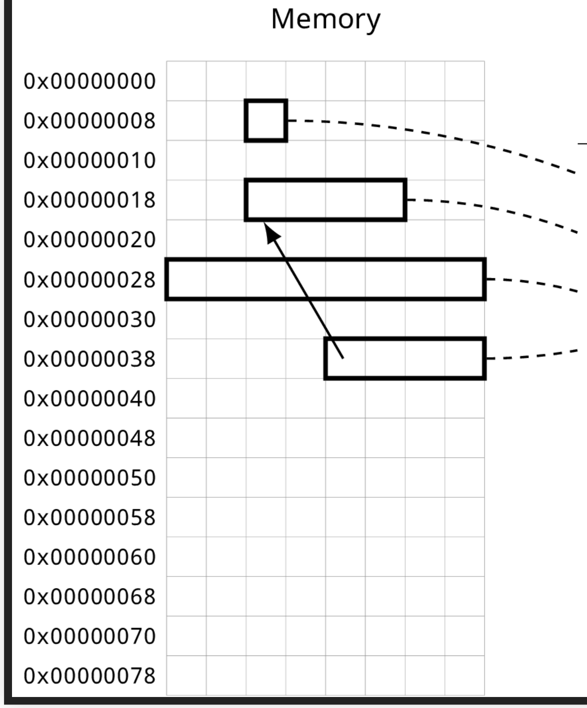
0x00000048								
0x00000050								
0x00000058								
0x00000060								
0x00000068								
0x00000070								
0x00000078								

Memory 1



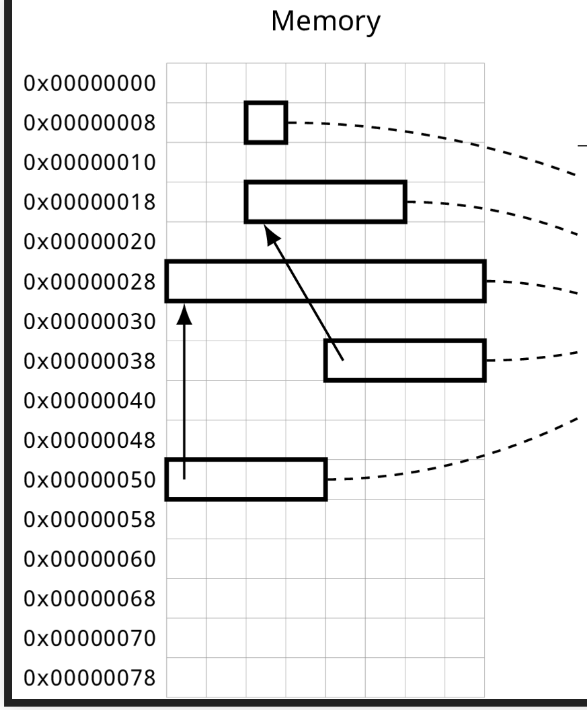
Type	Name	Value	Size	Address
char	`var1`	`'A'`	1 B	`0x0000000a`
int	`var2`	`12`	4 B	`0x0000001a`
double	`var3`	`3.14...`	8 B	`0x00000028`

Memory 2



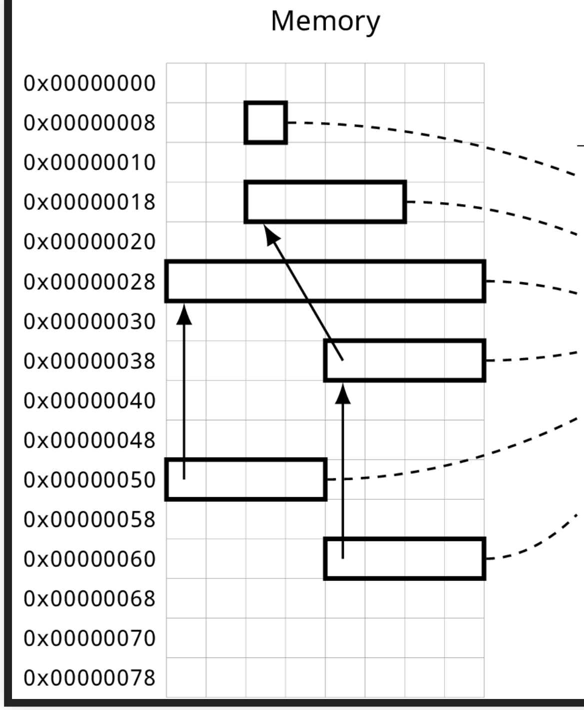
Type	Name	Value	Size	Address
char	`var1`	`'A'`	1 B	`0x0000000a`
int	`var2`	12	4 B	`0x0000001a`
double	`var3`	3.14...	8 B	`0x00000028`
int*	`ptr1`	`0x0000001a`	4 B	`0x0000003c`

Memory 3



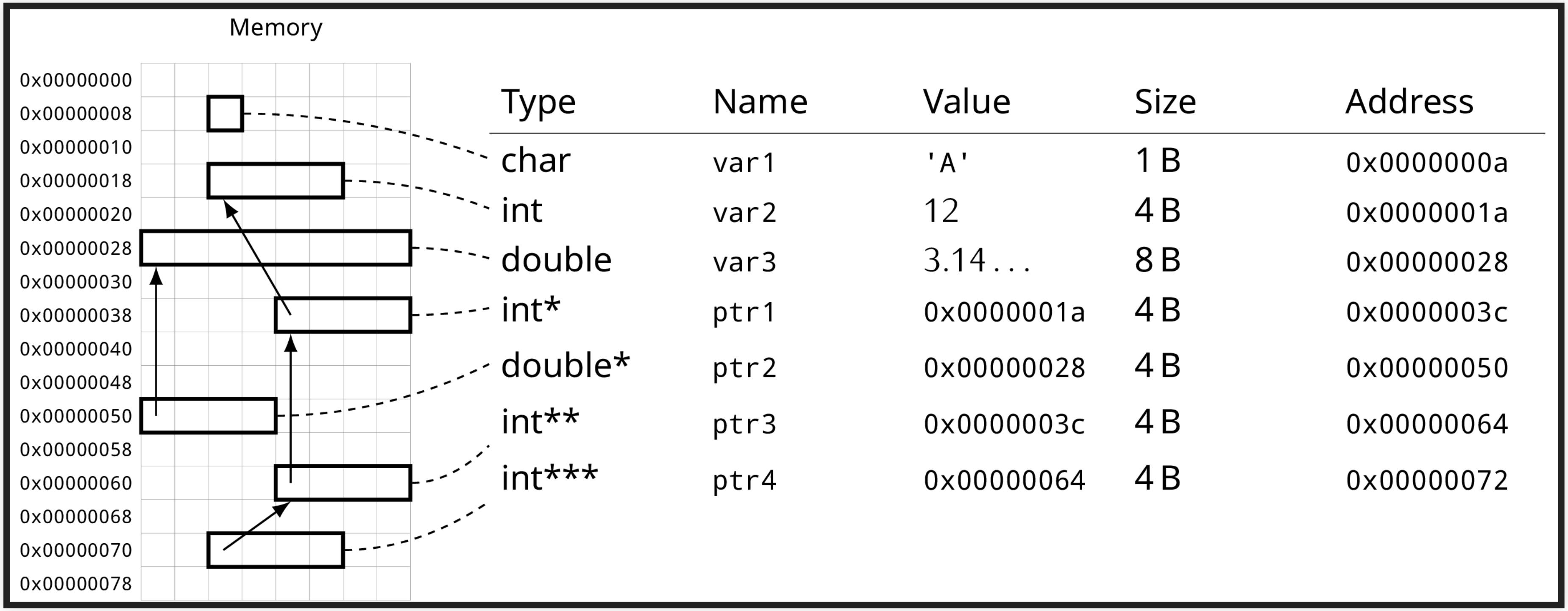
Type	Name	Value	Size	Address
char	`var1`	`'A'`	1 B	`0x0000000a`
int	`var2`	12	4 B	`0x0000001a`
double	`var3`	3.14 . . .	8 B	`0x00000028`
int*	`ptr1`	`0x0000001a`	4 B	`0x0000003c`
double*	`ptr2`	`0x00000028`	4 B	`0x00000050`

Memory 4



Type	Name	Value	Size	Address
char	var1	'A'	1 B	0x0000000a
int	var2	12	4 B	0x0000001a
double	var3	3.14 . . .	8 B	0x00000028
int*	ptr1	0x0000001a	4 B	0x0000003c
double*	ptr2	0x00000028	4 B	0x00000050
int**	ptr3	0x0000003c	4 B	0x00000064

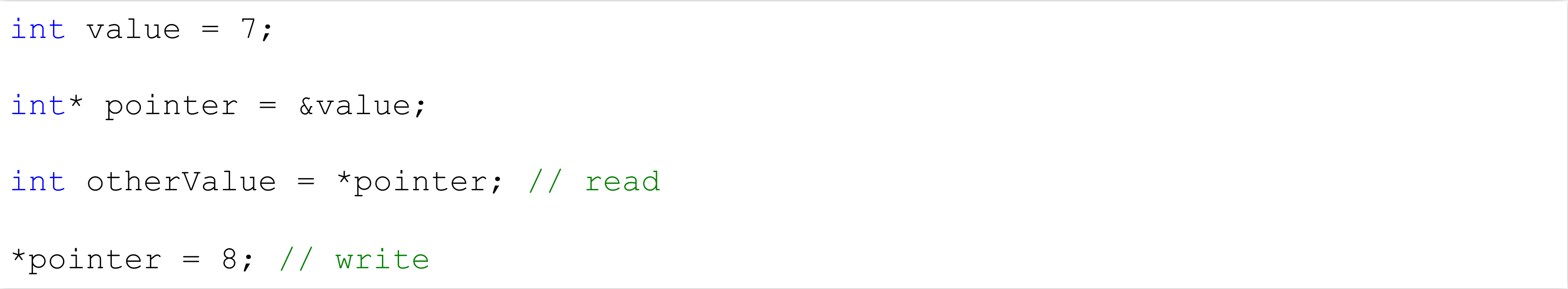
Memory 5



Type	Name	Value	Size	Address
char	`var1`	`'A'`	1 B	0x0000000a
int	`var2`	12	4 B	0x0000001a
double	`var3`	3.14 . . .	8 B	0x00000028
int*	`ptr1`	0x0000001a	4 B	0x0000003c
double*	`ptr2`	0x00000028	4 B	0x00000050
int**	`ptr3`	0x0000003c	4 B	0x00000064
int***	`ptr4`	0x00000064	4 B	0x00000072

Memory 6

- Note that this example uses 32 bit, you’ll commonly encounter 64 bit addresses



```
int value = 7;

int* pointer = &value;

int otherValue = *pointer; // read

*pointer = 8; // write
```

- Pointers are an essential language feature in C
- Underlying building block for arrays
- Lots of different applications
  - Working with large objects
  - Output parameters
  - Building lists and trees
  - …
- Special value `NULL` often used to indicate absence or error

ARRAYS

```
int field[100] = {0};

int value = field[32]; // read

field[32] = 21; // write

field[100] = 42; // out-of-bounds error
```

- Array ranges from `field[0]` – `field[99]`
- Array out-of-bounds access is a **very** common source of errors
- Need to keep track of the size

`myArray[5]` $\Longleftrightarrow$ `*(myArray + 5)`

STRINGS

- String literals (e.g. `"foo"`) are immutable
  - Typically handled as `const char *`
- Strings are `'\0'`-terminated!
  - Be careful when using string related functions!
  - Pay close attention to the terminator (see `strncpy` for a negative example)
  - **Very** common source of errors
- Consider using `asprintf` (GNU extension) for building strings
  - Alternatively use `snprintf`

- Strings **cannot** be compared with ==

DYNAMIC ALLOCATION

```
char* buffer = malloc(someSize);

// ...

free(buffer);
```

- Request memory from the OS at runtime
- Always *free* requested memory when you no longer need it
- Use `realloc` when you need to grow/shrink your allocation

DON’T FORGET ABOUT `const`

- `const` can save you from accidentally modifying a value
- Define variables `const` by default and only remove the `const` when you have to
- Very handy when using pointers

```
bool getCell(const Field* field, int x, int y) {
    // ...
}

void setCell(Field* field, int x, int y, bool alive) {
    // ...
}
```

MUCH, MUCH MORE!

- Function pointers
- Lifetime & ownership
- Short-circuit evaluation
- `goto`
- Preprocessor
- Bit fields
- …

READING MATERIAL

- Modern C by Jens Gustedt
- Pay close attention to man pages when using standard library functions