INF113: C Crash Course

Kirill Simonov

22.08.2025

Orga

- **Updates to group sessions:**
  1 - Monday 08:15-10:00, TM51 Room A
  2 - Tuesday 10:15-12:00, RFB Grupperom 4
  4 - Thursday 10:15-12:00, Høyteksenteret, Grupperom 209M1
  5 - Friday 10:15-12:00, TM51 Room C
  6 - Friday 12:15-14:00, **Biologen Blokk B, K3+K4**

- Next homework is out—Programming in C

C—the language

- In use from the 1970s, designed as successor to the *B* language
- C programs are *compiled* to binary executables

```
gcc -o source source.c
./source
```

- Statically typed, but weakly enforced
- All executable code is contained in *functions*
- Basic syntax like in Java: {} delimit blocks, statements end with ;

or, rather, Java has C-like syntax

- The execution starts from the `main` function

```
int main() {
  //TODO code here
  return 0;
}
```

Data types

- Basic (numerical) types
  - integers of various size
  - characters
  - floating-point numbers
- Arrays
  - static arrays—size is fixed at compilation
  - dynamic arrays—with memory allocation
- Pointers
  - special datatype for addresses
  - C distinguishes between the value and its address
- Structs
  - combined datatype

Basic types

- `int` is your default integer type
  - Almost always* 32-bit, so values from -2^{31} to $2^{31}-1$, or roughly $2 \cdot 10^9$
  - (*) standard only guarantees 16 bits, so formally this is up to the compiler
  - Use `%d` for standard library I/O

- modifiers:
  - `unsigned int` (`%u`)—same size, but only positive values, i.e., 0 to $2^{32}-1$
  - `short int` (`%hd`)—16-bit integer, $-2^{15}..2^{15}-1$
  - `long int` (`%ld`)—32-bit integer, same* as int
  - `long long int` (`%lld`)—64-bit integer

Control

- Familiar `for`, `while`, `if`

```c
for (int i = 0; i < 10; ++i) {
  ...
}
```

```c
while (true) {
  ...
}
```

```c
if (a < b) {
  ...
} else {
  ...
}
```

- Functions:

```c
int add(int a, int b) {
  return a + b;
}

add(1, 2)
```

Input/Output

- Need to include the header to call I/O functions:

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

- Formatted

```c
int h, m;
scanf("%d:%d", &h, &m);
printf("%d:%d", h, m);
```

- Files

```c
freopen("out.txt", "w", stdout);
freopen("a.log", "a", stderr);
fprintf(stderr, "ERROR %d", code);
```

```c
FILE* inp_file = fopen("a.in", "r");
int n;
fscanf(inp_file, "&d", &n);
fclose(inp_file);
```

- Unformatted

```c
char c[1000];
for (int i = 0;; ++i) {
  int nxt = getc();
  if (nxt == EOF) {
    break;
  }
  c[i] = nxt;
}
```

- standard streams:
  stdin, stdout, stderr
- "r" for reading
  "w" for writing
  "a" for appending

Arrays

- Static array:

```c
int a[10]; //fixed size
```

- allocated on stack or program address space (for global arrays)

```c
int a[] = {1, 1, 3}; //can initialize right away
int a[5] = {0}; //sets 0 to all elements
```

- a is just a pointer to the first element
  - cannot compare or initialize other arrays directly

```c
memcpy(b, a, sizeof(a)); //copies a to b
```

```c
memcmp(a, b, sizeof(a)); //lex. compares a to b
```

```c
memset(a, val, sizeof(a)); //fills bytes of a with val
```

Strings

- Text strings are just `char` arrays

```c
char c[] = "abc";
```

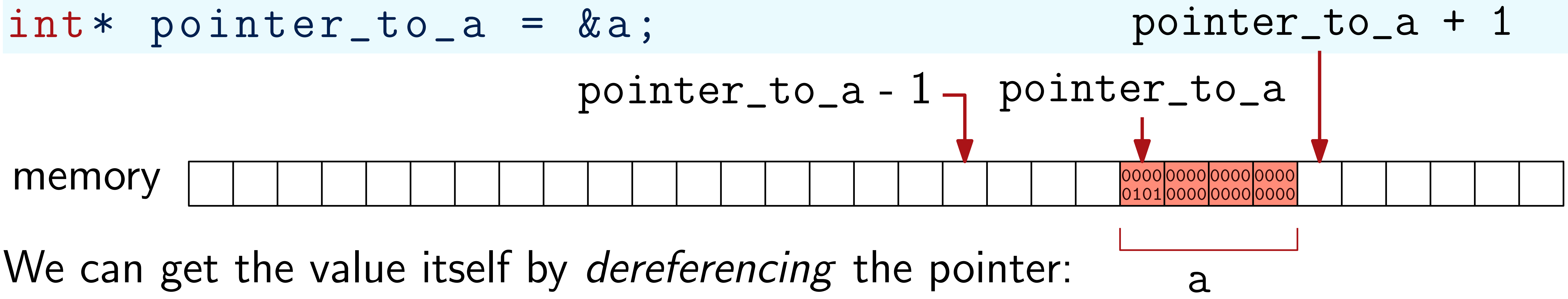
- String functions assume that 0 element is the end of the string

```c
#include <string.h>
...
strlen(s); //return actual length of the string
strcat(s1, s2); //concatenates s2 to s1
strcpy(s1, s2); //copies s2 to s1
strcmp(s1, s2); //lex. compares s1 and s2
strstr(s1, s2); //returns a pointer to the first
   occurrence of s2 in s1
strtok(s, " "); //splits the string into tokens
```

Pointers

- A pointer points to an address in memory—just a number that tells the position of the byte
- From a variable, we can get the pointer to where it is stored

```c
int a = 5;
int* pointer_to_a = &a;
```



- We can get the value itself by *dereferencing* the pointer:

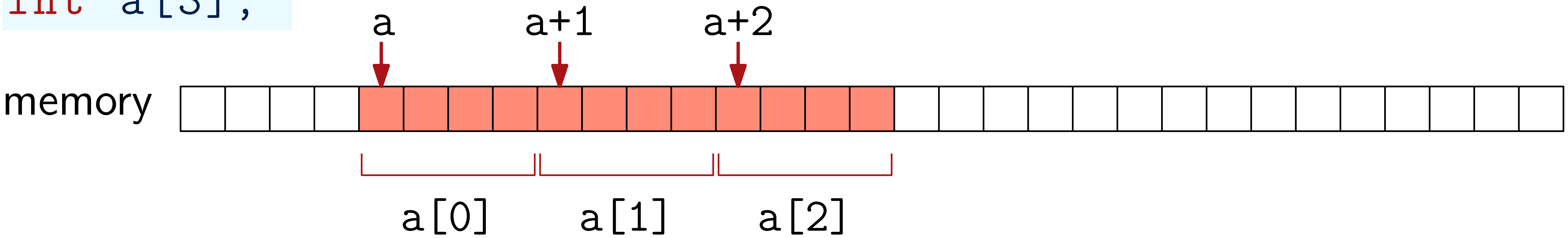
```c
printf("%d", *pointer_to_a);
```

- We can do arithmetics on pointers

Pointers

- The array variable is really just a pointer
- And [] notation is just shifting pointers: `a[i]` is the same as `*(a + i)`

```
int a[3];
```



- Array functions are defined on pointers too

```
memcpy(b, a, sizeof(a)); //copies a to b
memcpy(b + 10, a, sizeof(a)); //now starting from b[10]
```

Dynamic arrays

```c
int* a = malloc(n * sizeof(int)); //n is any variable
```

- allocated on heap
- memory should be deallocated, otherwise a memory leak:

  ```c
  free(a);
  ```

- `calloc` is like `malloc` but initializes to 0
- `realloc` can change the size of the previously allocated memory

Lifetime of a variable

- A stack variable is only kept while its scope is executing

```
for (int i = 0; i < 5; ++i) {
  int x = 2;
}
//x no longer exists here
```

- Often leads to mistakes with pointers

```
int* read_int() {
  int x;
  scanf("%d", &x);
  return &x;
}
//x no longer exists here
```

Struct

- struct allows to combine any other types into a composite type

```
struct point {
  int x, y;
};
```

```
struct student {
  char* name;
  int grade;
};
```

```
struct list {
  int value;
  struct list* name;
};
```

```
struct point p = {1, 2};
printf("%d%d", p.x, p.y);
```

- view struct as a way of bundling together data, not a full-fledged object
- member functions can be defined, but are not doing much
- usually one has external functions that work with the struct

Struct

- more convenient with the typedef hack:

```c
typedef struct {
  int x, y;
} point;
```

```c
point p = {1, 2};
printf("%d%d", p.x, p.y);
```

- passing a struct to a function will be by value, i.e., copying
  - to modify, pass by pointer

```c
void set_x(point p, int x) {
  p.x = x;
}
```

✗

```c
void set_x(point* p, int x) {
  p->x = x;
}
```

✓

if point were large, this would also copy lots of memory!

- always keep in mind the difference between a value variable and a pointer variable

If still not enough

- Troubleshooting:
  - Use warning flags, e.g., `-Wall`, `-Wextra`, to let compiler help
  - Use debug output to check what the values are at any time

```
fprintf(stderr, "LOG: %d, %d, %d\n", all, my, variables);
```

- Documentation: `https://en.cppreference.com/w/c.html`
  - Also available from command-line in Ubuntu

    ```
    man malloc
    ```

  - The language is determined by the C standard: current is C23
  - Compilers are supposed to follow the standard
    - Some things are not strictly defined, like the size of `int`