Alignment

- For a memory address we say it is n(-byte) aligned if it is a multiple of n.
- Many processors impose requirement for data alignment in order to fully exploit its capabilities.
- Therefor, access to unaligned data is possible, but it is not as efficient.
- Example: memory width is 8 bits, and there are instructions for accessing 4 consecutive bytes at once (to read on 32 bit value) – but only if the address which is being accessed in such a manner is 4 aligned.

Common alignments

- The usual alignments for 32bit architecture x86:
  - **char** (1 byte) aligned to **1 byte**.
  - **short** (2 bytes) aligned to **2 bytes**.
  - **int** (4 bytes) aligned to **4 bytes**.
  - **float** (4 bytes) aligned to **4 bytes**.
  - **double** (8 bytes) aligned to **8 bytes** in Windows, aligned to **4 bytes** on Linux (you can set alignment to 8 with a compiler switch).
  - **показивач** (4 bytes) aligned to **4 бајта**

```
char a;
int b;
char c;
short d;
```

What if data is not properly aligned

- Compiler usually makes sure that all the data is properly (optimally) aligned, and therefor it assumes proper aligning when generating code. But, careless programmer can, in curtain cases, cause access to unaligned data.
- In case of unaligned access, several things can happen:
  - Program will give wrong result
  - Processor exception will be raised
  - The result will be correct, but you will get it less efficiently

```
int8_t buffer[4];
int32_t x = *(int32_t*)buffer;
```

Alignment of structure fields

```
char a;
int b;
char c;
short d;
```

```
struct S
{
   char a;
   int b;
   char c;
   short d;
};
```

```
x = sizeof(struct S);

y =
sizeof(a) + sizeof(b) +
sizeof(c) + sizeof(d);

y ≤ x
```

- Structure fields have to be stored in memory in the same order in which they are defined.
- Size of struct S variable can be larger than sum of sizes of each of its fields.
- That is because some bytes can be inserted by compiler, in-between fields in order to assure optimal alignment. They are called padding bytes.

Another example

```
struct MixedData          struct MixedData
{                         {
    char Data1;             char Data1;
                            char Pad1[1];
    short Data2;            short Data2;
    int Data3;              int Data3;
    char Data4;             char Data4;
                            char Pad2[3];
};                        };
```

- Although only Pad1 is necessary to ensure optimal alignment of all fields, Pad2 is still added at the end. It is done to ensure optimal alignment for each element in arrays of these structures.
- Therefore, structure's size will always be multiple of biggest optimal alignment of its members.

Structure equality

```
struct MixedData s1 = {a, b, c};
struct MixedData s2;
s2.x = a; s2.y = b; s2.z = c;
```

✗
```
if (memcmp(&s1, &s2, sizeof(struct MixedData)) == 0)
```

```
s1 = s2;
```

✗
```
if (memcmp(&s1, &s2, sizeof(struct MixedData)) == 0)
```

```
memcpy(&s1, &s2, sizeof(struct MixedData));
```

✓
```
if (memcmp(&s1, &s2, sizeof(struct MixedData)) == 0)
```

Structure packing

- Working with aligned structures is faster, but padding bytes increase memory usage.
- Number of padding bytes can be reduced without also reducing speed, if we rearrange structure fields.
- Order of structure fields has to be guaranteed, so compiler can not do the rearrangement automatically. But programmer can.
- However, such solution can reduce number of padding bits just up to some point. If we want further reduction we can do it only if we reduce speed.
- Most of the compilers for platforms that have optimal alignments it is possible to order custom alignment, e.g. **`pack(2)`** means that compiler should align data to maximally 2, which then means that mostly one padding byte is necessary between any two fields.
- Structure packing is mostly used to reduce memory usage, but can also be used when preparing data for network transfer etc.

Reordering structure fields

Example of structure `MixedData.` Left it is with rearranged elements, and right is the original structure:

```
struct MixedData
{
    char Data1;
    char Data4;
    short Data2;
    int Data3;
};
```

```
struct MixedData
{
    char Data1;
    short Data2;
    int Data3;
    char Data4;
};
```

With these rearrangements there is no need for any padding byte.

Reminder: char - 1 byte, short - 2 bytes, int - 4 bytes on x86

What is the size of each structure?

Structure packing

- Let us imagine that structure MixedData does not have field Data 2. In that case, even for the best arrangement, we would have 2 padding bytes.
- Then we can use feature that many compilers offer (but which is not part of the standard), and that is structure packing. Most compilers (Microsoft, Borland, GNU...) use **`#pragma`** directives as a mechanism of specifying desired packing.

```
#pragma pack(push)   /* push current alignment to stack */
#pragma pack(1)      /* set alignment to 1 byte boundary */
struct MixedData
{
    char Data1;
    int Data3;
    char Data4;
};
#pragma pack(pop)    /* restore original alignment from stack */
```

GNU __attribute__

- In GCC the is a new, non-standard, keyword **__attribute__** which enables attaching of certain attributes to variables and functions.
- Here is the code that, when compiler with GCC, give the same result as the code on the previous slide:

```
struct MixedData
{
    char Data1;
    int Data3;
    char Data4;
}__attribute__ ((packed));
```

- In case that you need some special alignment, in GCC you can use “aligned” attribute. With it you order compiler to align that variable to specified number.

```
int data __attribute__ ((aligned (16))) = 0;
```

Compiler will place variable “data” on some address that is multiple of 16.

New things in C11 related to alignment

- New standard header is introduced: <stdalign.h>
  - Nicer writing of new keywords _Alignof и _Alignas
- alignof(type)

```
size_t x = alignof(int);
size_t y = alignof(struct {char c; int n;});
```

- alignas(type) or alignas(expression) (only for overalignment)

```
alignas(16) int data = 0;
alignas(int) char array[57];
struct alignas(2 * alignof(int)) _s {
  char Data1;
  int Data3;
  char Data4;
}
```

- aligned_alloc();
  - You can specify alignment