Wyrównanie (*alignment*) a C++17

Bartosz Szreder
bartosz.szreder@huuugegames.com

Ale o co chodzi?

- *Alignment (wyrównanie)* – adres jest wielokrotnością pewnej liczby $N = 2^k$
- *Natural alignment* – wyrównanie zgodne z rozmiarem typu (może poza `long double`)
- W typach złożonych: *padding*
- Wyrównanie w klasach przynajmniej takie, jak wyrównanie elementu o najwyższym wymogu tegoż

Czy muszę o tym myśleć?

```
1 struct T {
2     uint8_t    c;
3     uint32_t   i;
4     uint16_t   s;
5 };
```

```
1 struct T {
2     uint8_t    c;
3     //char padding_1[3];
4     uint32_t   i;
5     uint16_t   s;
6     //char padding_2[2];
7 };
```

```
1 struct T {
2     uint8_t    c;
3     uint16_t   s;
4     uint32_t   i;
5 };
```

```
1 struct T {
2     uint8_t    c;
3     //char padding[1];
4     uint16_t   s;
5     uint32_t   i;
6 };
```

Czy kompilator nie może tego zrobić za mnie?

- *Reordering* – NIE
- *Packing* – jak go ładnie poprosimy (ale na naszą odpowiedzialność): `__attribute((packed))__`, `pragma pack`

Packing narusza wymogi wyrównania

- Co jeśli `uint32_t` nie jest wyrównany do 4?
- . . . nie znajduje się w jednym wierszu cache?
- . . . nie znajduje się w jednej stronie pamięci operacyjnej?

Bonusowe UB przy rzutowaniu wskaźników z typów mniej do bardziej wyrównanych, jeśli rzutowany adres nie spełnia silniejszego wymogu wyrównania.

C++11 na ratunek

```
1 struct alignas(16) sse_t {
2     float data[4];
3 };
4
5 alignas(128) char cacheline[128];
```

- `alignas`(2^k)
- Można nakładać na całe typy, pojedyncze zmienne, pojedyncze pola w typach złożonych
- Nie można osłabić wyrównania.

signal 7 (SIGBUS),
code 1 (BUS_ADRALN)

A *fundamental alignment* is represented by an alignment less than or equal to the greatest alignment supported by the implementation in all contexts, which is equal to `alignof(std::max_align_t)`

`[...]`

An extended alignment is represented by an alignment greater than `alignof(std::max_align_t)`. It is **implementation-defined** whether any extended alignments are supported and the contexts in which they are supported. A type having an extended alignment requirement is an *over-aligned type*.

Co sie stao

Implementation-defined zwykle znaczy „not supported” i powoduje kupę śmiechu (z przewagą kupy).

- Kompilator widzi specyfikator `alignas()` i mocno w niego wierzy
- W oparciu o tę wiarę generuje instrukcje wyrównanych odczytów

Ale ja bardzo muszę mieć moje wyrównanie!

- Miłej zabawy w przeciążaniu `operator new` i `delete` w klasach
- Pomocnicze funcje: `_aligned_malloc()`, `posix_memalign()`, `aligned_alloc()` (**C**++17)
- *Intrinsic functions*
- *STL ist verboten*

Ale ja bardzo muszę mieć moje wyrównanie **oraz STL**!

- Miłej zabawy w przeciążaniu **globalnego** `operator new` i `delete`
- oh wait, to nie zadziała (chyba, że wszystkie alokacje zaczniemy wyrównywać) – nie wiemy kiedy na co alokujemy pamięć
- Specjalizacja szablonu `std::allocator`.
- `std::make_shared` oszukuje!

C++17 – jeszcze więcej operatorów new i delete

```
1 void* operator new(std::size_t size, std::align_val_t al);
2 void* operator new[](std::size_t size, std::align_val_t al);
3 void operator delete(void* ptr, std::align_val_t al);
4 void operator delete[](void* ptr, std::align_val_t al);
```

Do tego warianty z `std::nothrow_t`, *placement*, overloady w klasach...

```
1  void* operator new(std::size_t size, std::align_val_t al)
2  {
3      const auto align = static_cast<size_t>(al);
4
5      void* result = aligned_alloc(align,  size);
6      if (!result)
7          throw std::bad_alloc{};
8
9      return result;
10 }
11
12 void operator delete(void* ptr, std::align_val_t al)
13 {
14     free(ptr);
15 }
16
```