

[Главная](#) » [Posts](#)

# Выравнивание И Заполнение Структур

мая 13, 2019 · 4 минуты · German Gorelkin



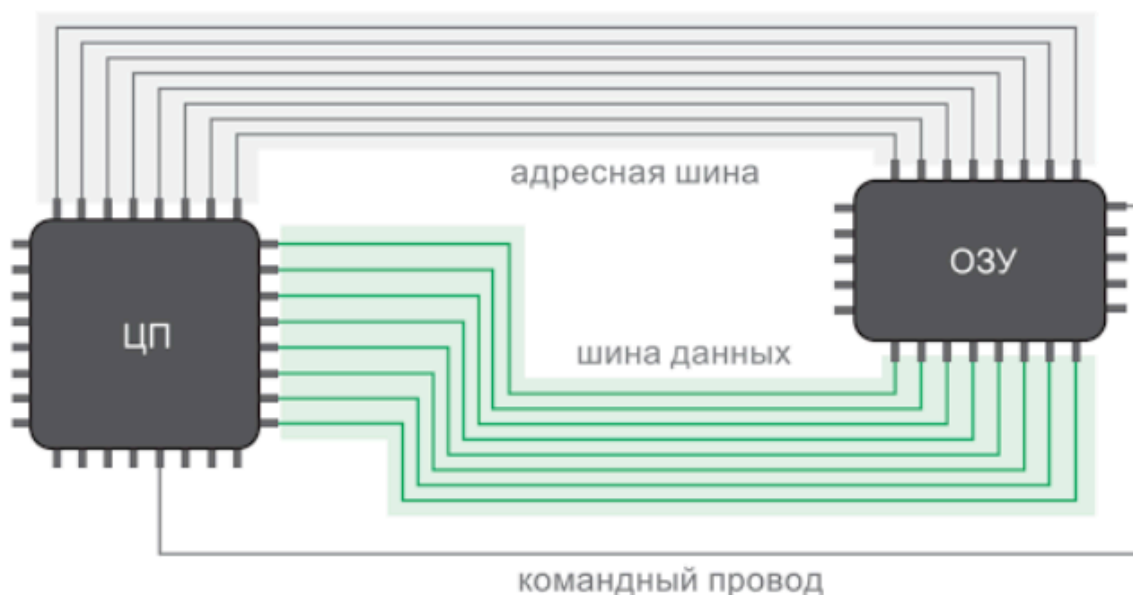
Разбираемся с такими понятиями, как: **type alignment guarantees**, **type sizes** и **structure padding**.

## Процессор и память

*Упрощенное представления взаимодействия процессора и памяти.* Память имеет адресную байтовую последовательность и расположена последовательно. Чтение или запись данных в памяти выполняется посредством операций, которые воздействуют на одну ячейку за раз. Чтобы прочитать ячейку памяти или произвести запись в нее, мы должны передать ее числовой адрес. Память способна выполнять с адресом ячейки две

операции: получить хранящееся в ней данные или записать новые. Память имеет специальный входной контакт для установки ее рабочего режима.

Группа проводов(линий), используемых для передачи одинаковых данных, называется **шиной**. Для передачи адреса используется **адресная шина**. А **шина данных** позволяет получать или записывать данные.



Основной характеристикой адресной шины является её **ширина в битах**, что, как правило, равно максимально допустимому числу разрядов адреса.

У 32-разрядной шины каждый байт имеет 32-битный адрес, что позволит использовать 4Гб адресное пространство( $2^{32}$ ). С 64-битным адресом можно использовать до  $2^{64}$  байт.

При 32-разрядной шине за раз можно получить до 4 байта данных, а 64-разрядная шина позволяет прочитать сразу 8 байт. Такие куски информации принято называть **машинным словом**.

Компьютеры наиболее эффективно загружают и сохраняют значения в памяти, когда эти значения **выравнены(alignment)**.

**Адрес 2-байтового типа(int16) должен быть кратен 2, адрес 4-байтового значения(int32) должен быть кратен 4, а 8-байтового соответственно кратен 8.**

**Пример.** У нас 64-разрядная архитектура, что позволяет за один раз прочитать до 8 байт. Сохраняем одно 1-байтовое значение и три 2-байтовых значения:

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15

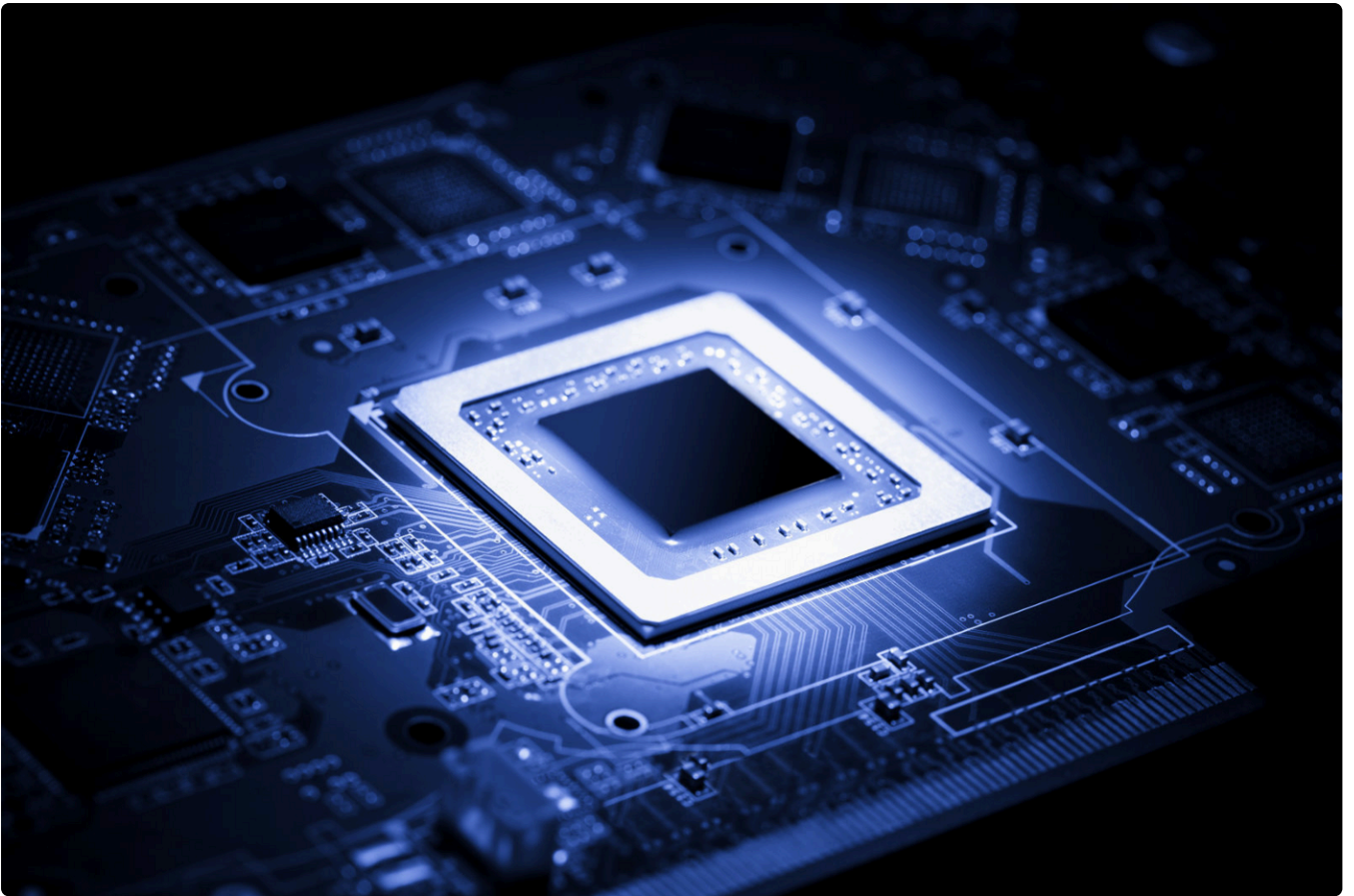
Один байт храниться по адресу 0, а 2-байтовые — 2, 4, 6 (кратные 2).

Благодаря выравниванию не будет ситуации, когда придется делать два цикла чтения для получения одного значения:

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15

**Гарантии выравнивания типов (type alignment guarantees)** также называют **гарантиями выравнивания адресов (value address alignment guarantees)**.

Если гарантией выравнивания типа  $T$  является  $N$ , то адрес каждого значения типа  $T$  должен быть кратным  $N$  во время выполнения. Можно также сказать, что адреса адресуемых значений типа  $T$  гарантированно выровнены по  $N$ -байтам.



## Размеры типов

Type	Size
bool	1 byte
intN, uintN, floatN	N/8 bytes
int, uint, uintptr	1 word
*T	1 word
string	2 words
[]T	3 words
map	1 word
func	1 word
chan	1 word
interface	2 words

К счастью, выравнивания и многое другое, гарантирует компилятор, и нам не нужно об этом беспокоиться. *Но полезно помнить, что выравнивания делает обращение к памяти более эффективное, но вот использование памяти может стать менее эффективным.*

Вспомним первый пример. В 8 байтах хранится одно 1-байтовое значение и три 2-байтовых. Из-за выравнивания (у 2-байтового значение должен быть четный адрес) ячейка памяти по адресу 1 осталась свободной. Но если для переменных с простым типов компилятор использует оптимизации, то **для структур выравнивание происходит по самому большому полю и может образоваться большое кол-во занятой, но не использованной памяти.**

Рассмотрим структуру:

```
type someData struct{
    a int8 // 1 byte
    b int64 // 8 byte
```

```
c int8 // 1 byte
}
```

Размер значения этой структуры должен составлять сумму ее полей.  $1 + 8 + 1 = 10$  байт. Проверяем это утверждение:

```
v := someData{}
typ := reflect.TypeOf(v)
fmt.Printf("Type someData is %d bytes long\n", typ.Size())
```

Результат совсем не тот, который ожидали. Структура занимает почти в 2.5 раза больше:

```
Type someData is 24 bytes long
```

Посмотрим подробнее, что происходит:

```
n := typ.NumField()
for i := 0; i < n; i++ {
    field := typ.Field(i)
    fmt.Printf("%s at offset %v, size=%d, align=%d\n",
        field.Name, field.Offset, field.Type.Size(),
        field.Type.Align())
}
fmt.Printf("someData align is %d\n", typ.Align())
```

**Offset** — это смещение адреса поля в значении структуры. **Size** — размер поля. **Align** — выравнивание для типа поля.






```
a at offset 0, size=1, align=1
b at offset 8, size=8, align=8
c at offset 16, size=1, align=1
someData align is 8
```

**Почему такой результат.** В общем случае экземпляр структуры будет выровнен по самому длинному элементу. Для компилятора это самый простой способ убедиться, что все поля структуры будут также выровнены для быстрого доступа.

**Неиспользованная память заполняется(padding) нулями.**

Поле `v.a` занимает всего один байт, но следующее поле начинается только через 8 байт (`b at offset 8`). 7 байт просто не используется.

# Struct alignment: 8

Fields	Alignment
a int8	
padding	
b int64	
c int8	
padding	

Посмотрим подробнее как это выглядит в памяти:

```
v = someData{a:1,b:2,c:3}
b := (*[24]byte)(unsafe.Pointer(&v))
fmt.Printf("Bytes are %#v\n", b)
```

Теперь поле `a=1`, `b=2`, `c=3`. Значение структуры представили как массив байт:

```
Bytes are &[24]uint8{
0x1, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x2, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x3, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}
```

**Компилятор не меняет порядок полей в структуре и не может оптимизировать такие случаи. А мы можем.**


*Смежные поля могут быть объединены, если их сумма не превышает выравнивания структуры.*

```
type someDataV2 struct{
    a int8 // 1 byte
    c int8 // 1 byte
    b int64 // 8 byte
}
```

**Переместив поля**, можно уменьшить размер структуры до 16 байт.

```
Type someDataV2 is 16 bytes long
a at offset 0, size=1, align=1
c at offset 1, size=1, align=1
b at offset 8, size=8, align=8
someDataV2 align is 8
Bytes are &[16]uint8{
0x1, 0x3, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x2, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}
```






## Struct alignment: 8

Fields	Alignment
a int8	
c int8	
padding	
b int64	

6 байт все еще не используются, но с этим уже ничего не поделаешь. Туда всегда можно добавить данных до 6 байт, не изменив размер структуры.



# Struct alignment: 8

Fields	Alignment
a int8	
c int8	
f int32	
e int16	
b int64	

**Почти всегда, гораздо важнее читаемость кода, чем такие оптимизации.** Нужно понимать по какой причине у значения типа именно такой размер и что вообще происходит, а уже в случае необходимости заниматься оптимизацией.

Линтеры, которые могут помочь:

- <https://gitlab.com/opennota/check>
- <https://github.com/mdempsky/maligned>

Код примеров [play.golang](#) или [github](#)

## Дополнительная информация

- <https://go101.org/article/memory-layout.html>
- [https://golang.org/ref/spec#Size\\_and\\_alignment\\_guarantees](https://golang.org/ref/spec#Size_and_alignment_guarantees)

- <https://www.geeksforgeeks.org/structure-member-alignment-padding-and-data-packing/>
- <http://golang-sizeof.tips/>
- <https://dave.cheney.net/2015/10/09/padding-is-hard>

Комментарии в [Telegram-группе!](#)

go

« ПРЕДЫДУЩАЯ

Race Condition и Data Race

СЛЕДУЮЩАЯ »

Паттерны Проектирования На Go.  
Functional Options.

