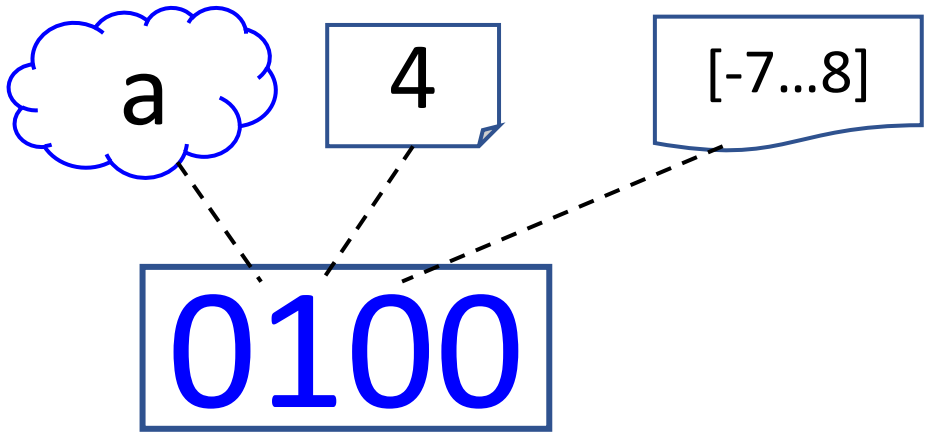
УКАЗАТЕЛИ И ССЫЛКИ

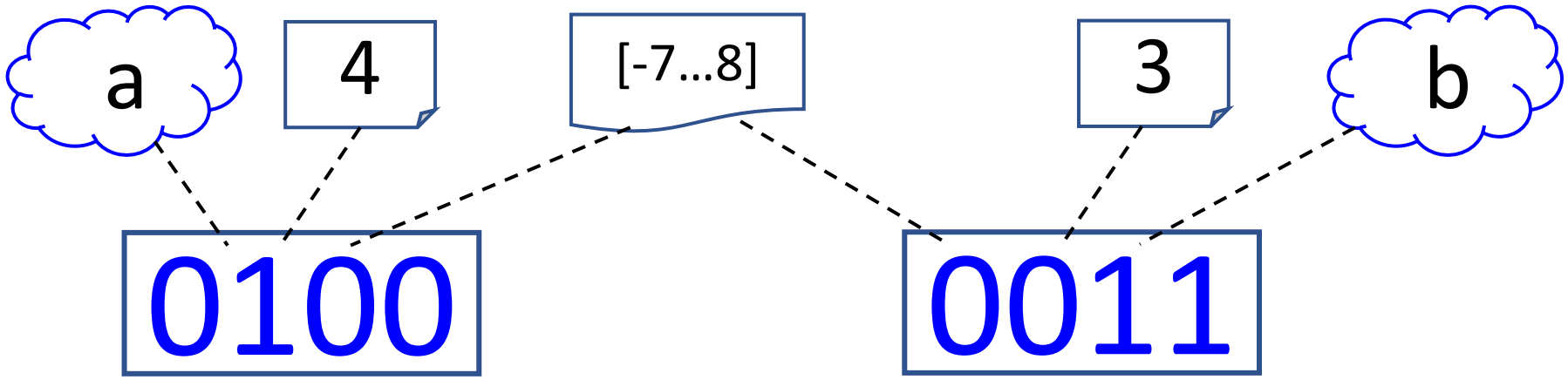
Genesis, проблема N Queens и при чем здесь инкапсуляция

Владимир Агаджанов, КА «Тор», 2023

- Genesis: имена и объекты
- Queens Problem
- Инкапсуляция
- Консистентность

0100


a
4
[-7...8]
0100


a
4
[-7...8]
3
b
0100
0011

Обсуждение

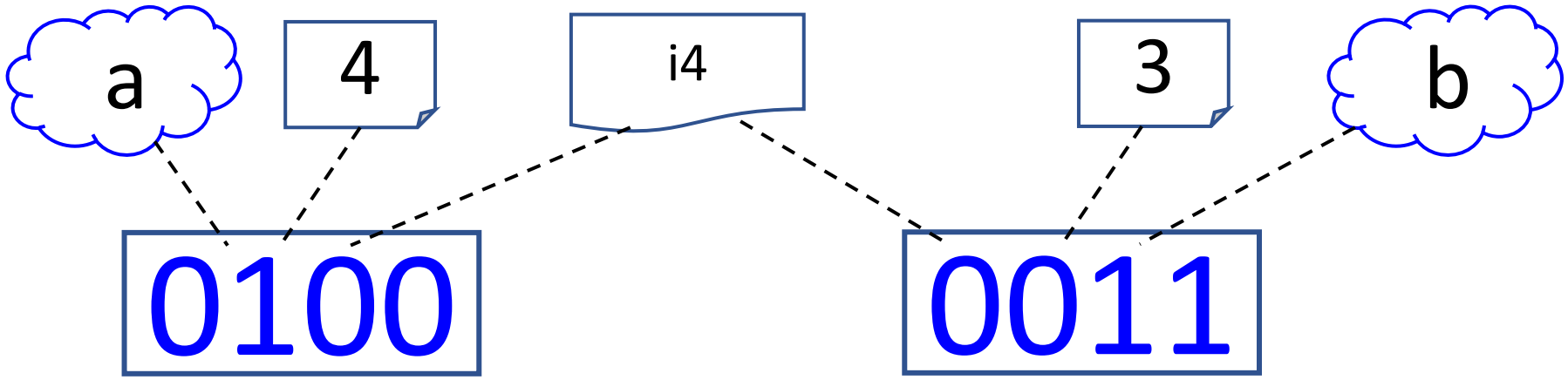
- Что такое тип?
- Достаточно ли для типа знать диапазон значений, например [-7…8]?

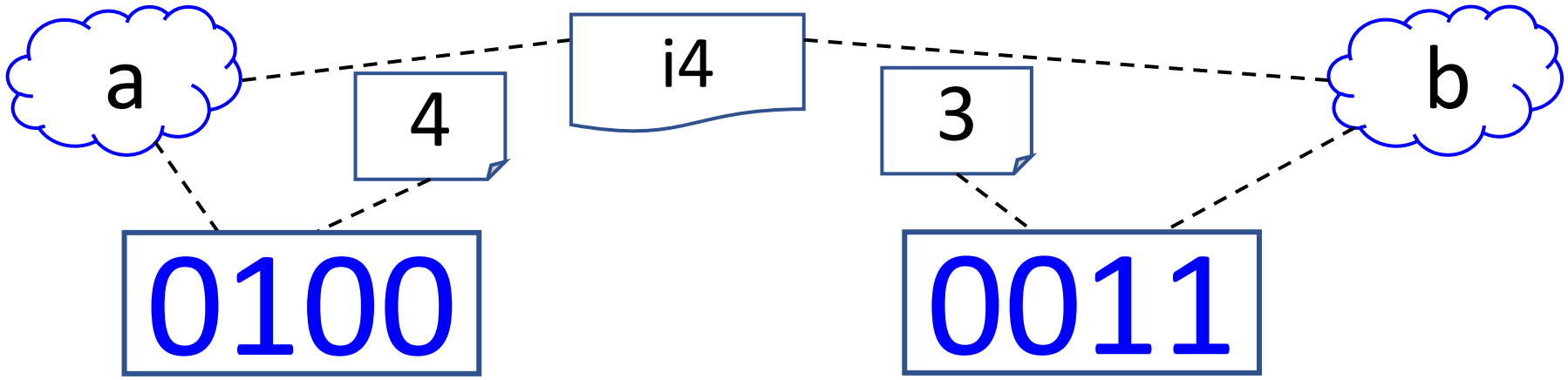
Типы: value types & object types

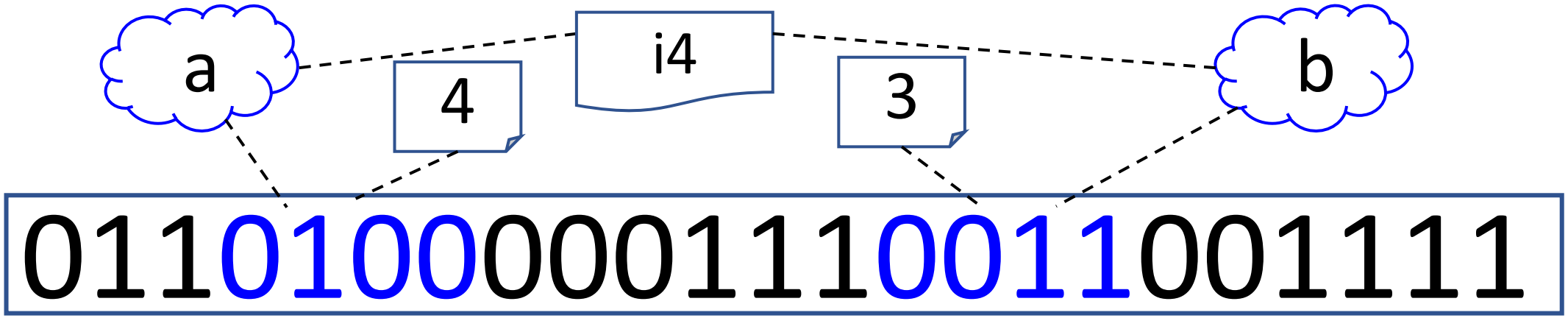
- Что такое тип?
- value type: диапазон возможных значений объекта
- object type: совокупность операций над объектом
  - Например: 5/2 даст 2 для типа int, но 2.5 для double
  - 0-1 даст -1 для char, но 255 для unsigned char

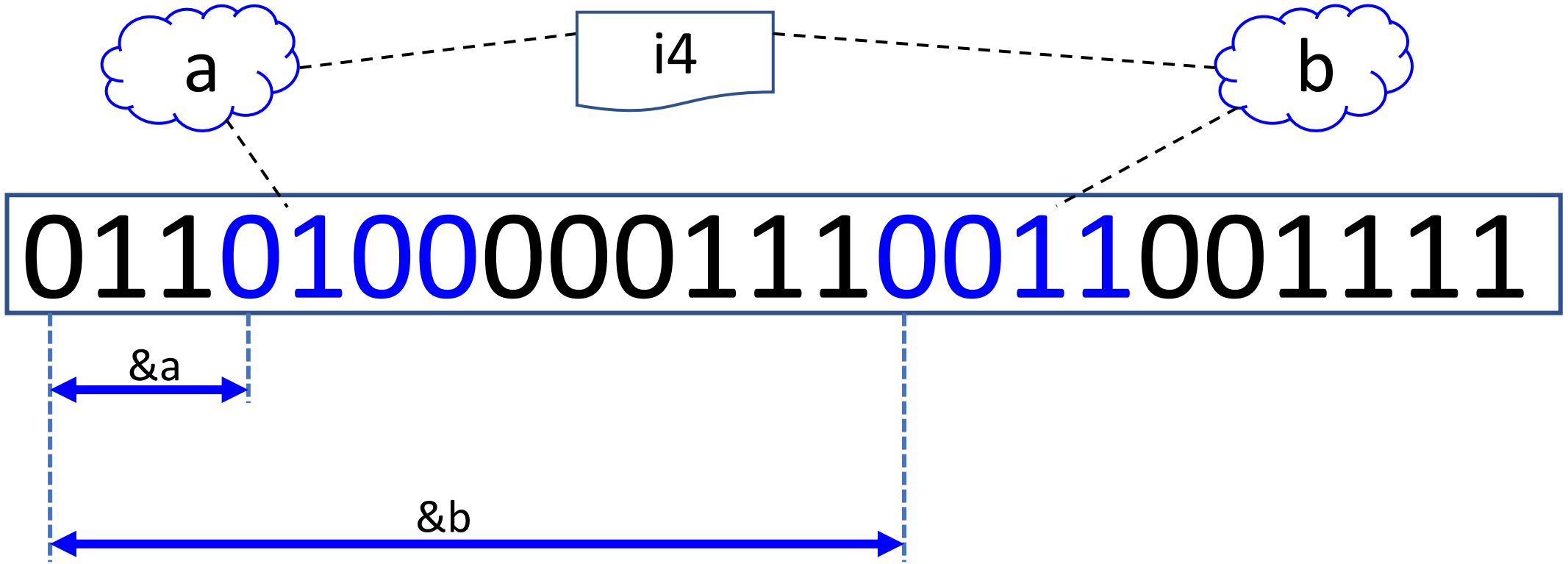
  Есть и неопределенные операции:
  127 + 5 для char является возможной, но неопределенной операцией.
  UB для таких операций – норма

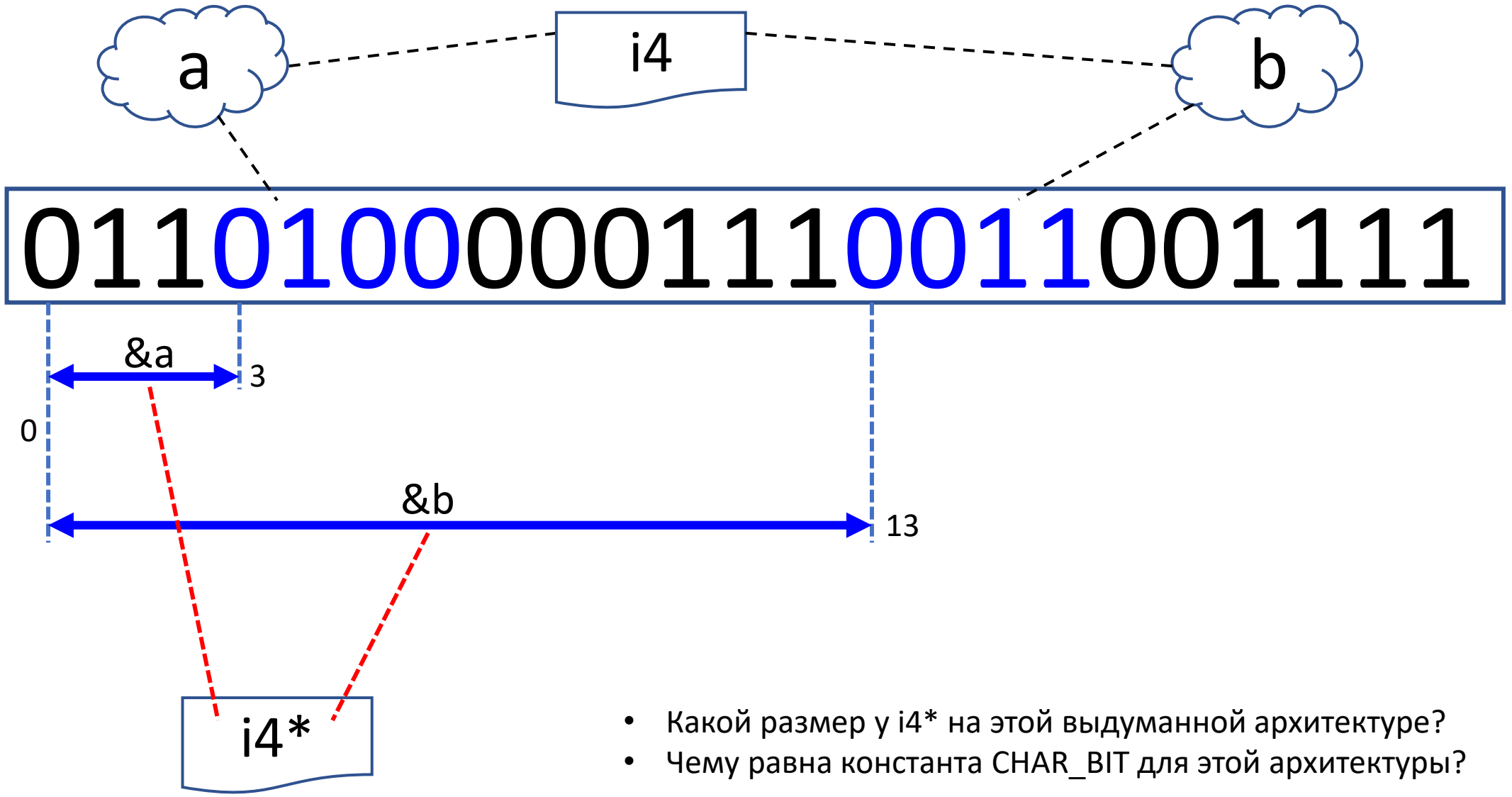
Назовем целочисленный четырехбитный арифметический тип i4


a
4
i4
3
b
0100
0011

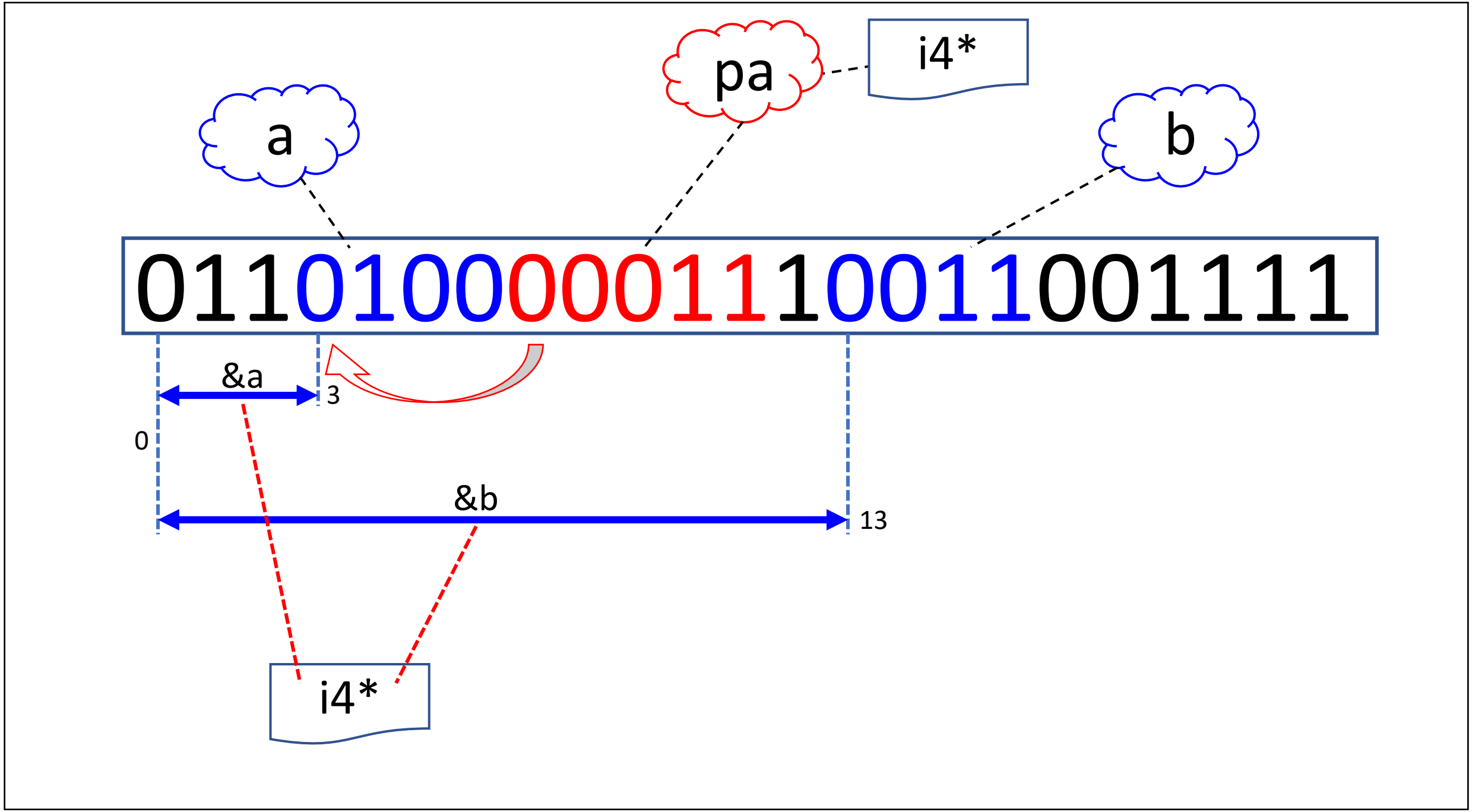

a
i4
b
4
3
0100
0011

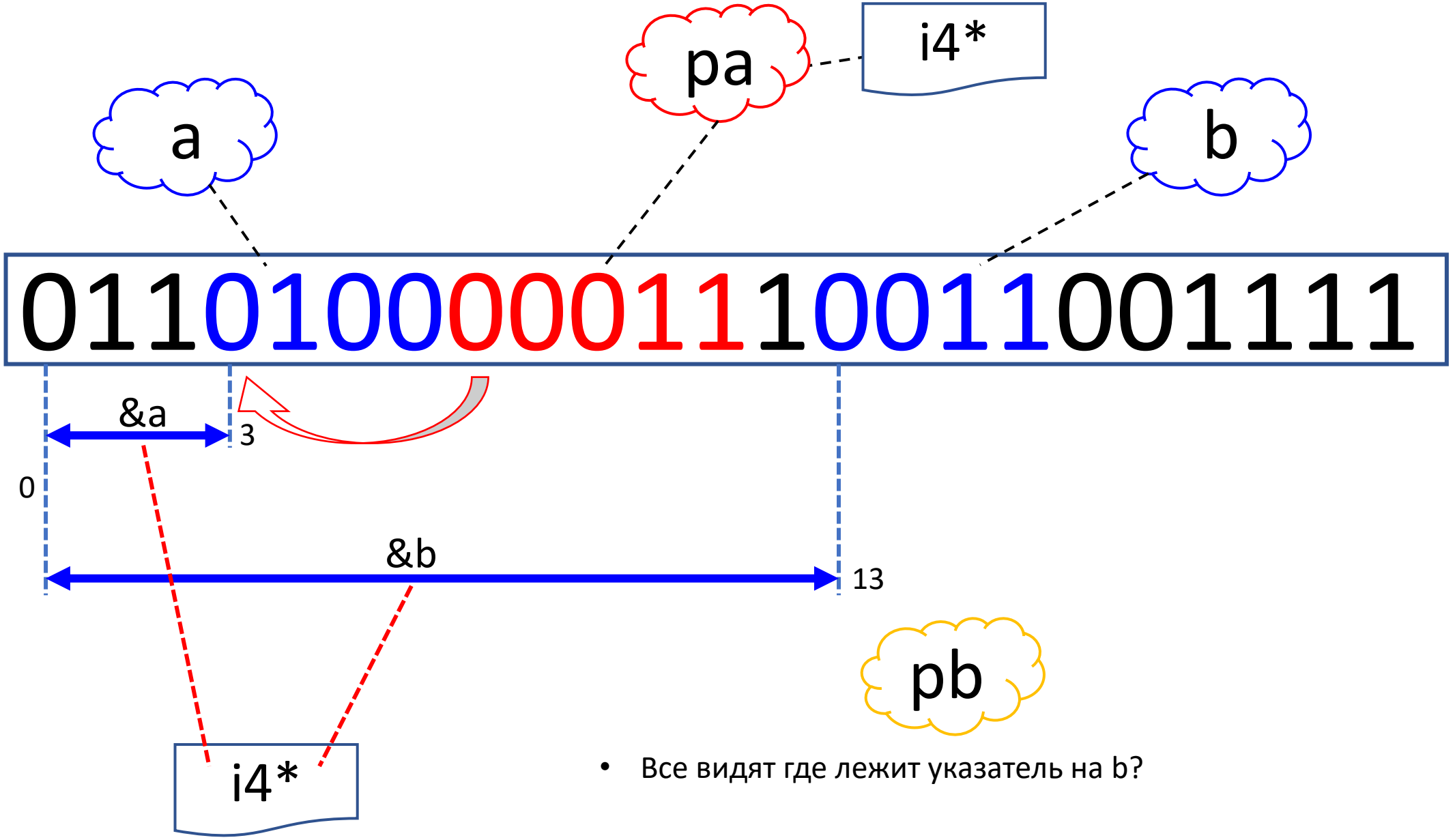

a
4
i4
3
b
01101000001110011001111


a
i4
b
01101000000111001111001
&a
&b

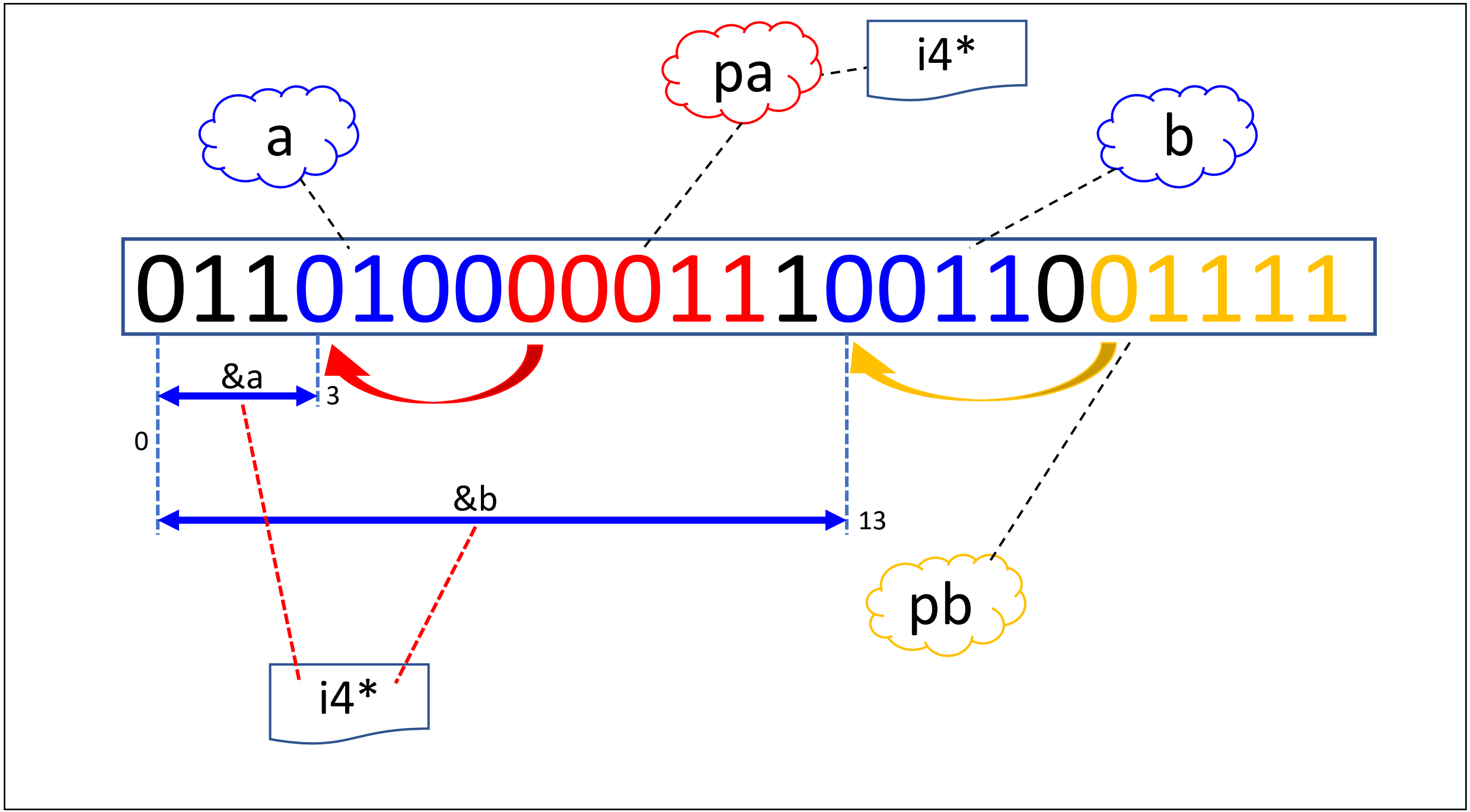


- Какой размер у i4* на этой выдуманной архитектуре?
- Чему равна константа CHAR_BIT для этой архитектуры?


pa
i4*
a
b
01101000000111001001111
&a
3
0
&b
13
i4*



- Все видят где лежит указатель на b?


pa
i4*
a
b
01101000000111001101001111
&a
3
0
&b
13
pb
i4*

Нулевые указатели

- Если указатель это просто расстояние, то может ли быть и нулевое расстояние?
- Нулевой указатель это специальный маркер «ничего». Там ничего не лежит.

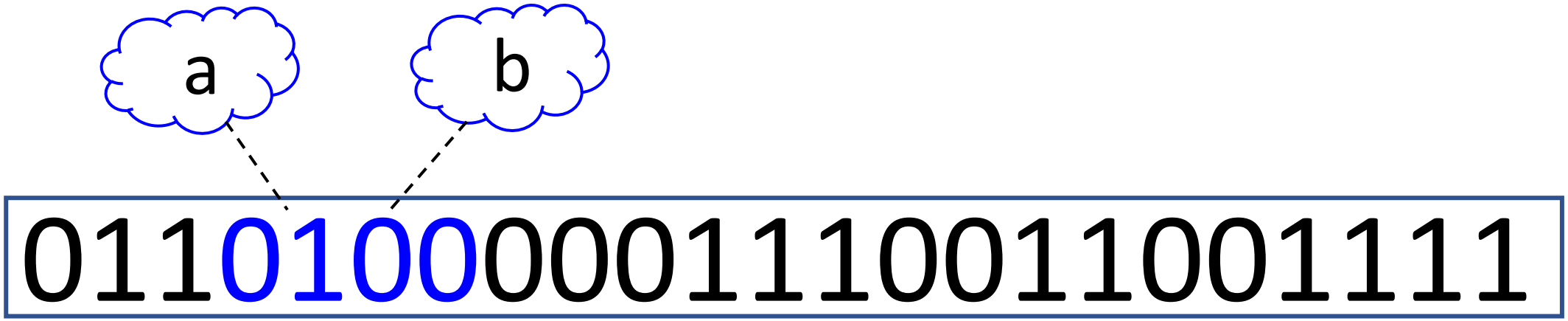
- Не путайте 0, `NULL` и `nullptr`.

```
if(!p) do_something(); // работает для всех 3х типов
```

- В чем отличие NULL от 0?
- В C++ всегда выбирайте `nullptr`.

Ссылки (lvalue references)

- Если бы мы говорили о языке C, то на этом можно было бы закончить.
- Однако в C++ есть возможность дать одному объекту два имени.



Синтаксис ссылок

- Синтаксис lvalue ссылок – одинарный амперсанд (можно добавлять cv-квалификаторы)

```
int x;
int& y = x; // теперь y еще одно имя для x
```

- Не путайте синтаксис ссылок со взятием адреса!

```
int x[2] = {10,20};
int& xref = x[0];
int* xptr = &x[0];
xptr++;
xref++;
```

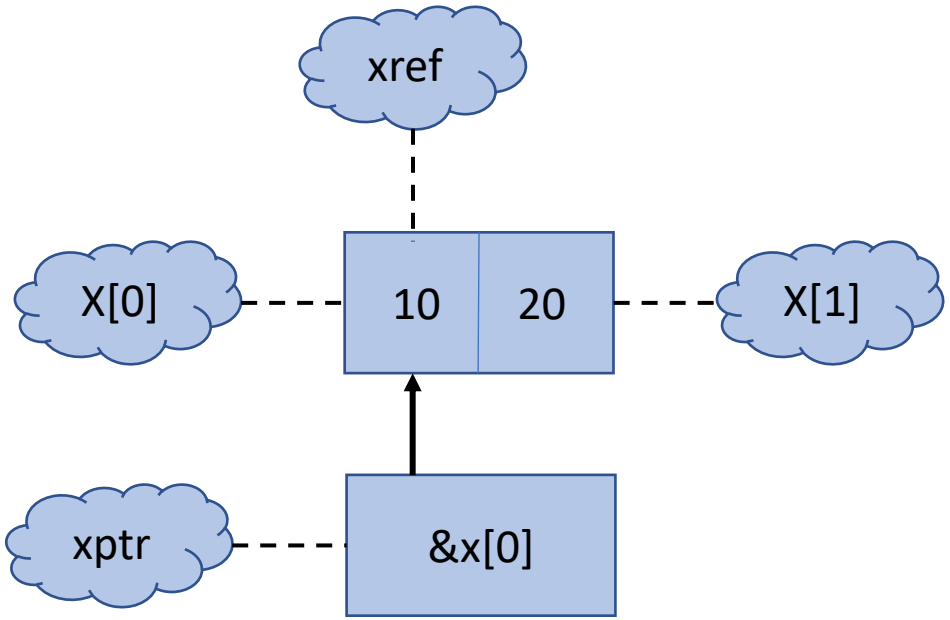
- Что в массиве?

Синтаксис ссылок

- Не путайте синтаксис ссылок со взятием адреса!

```
int x[2] = {10,20};
int& xref = x[0];      // еще одно имя для x[0]
int* xptr = &x[0];     // взятие адреса
xptr++;
xref++;

assert(xref == 11);
assert(*xptr == 20);
```

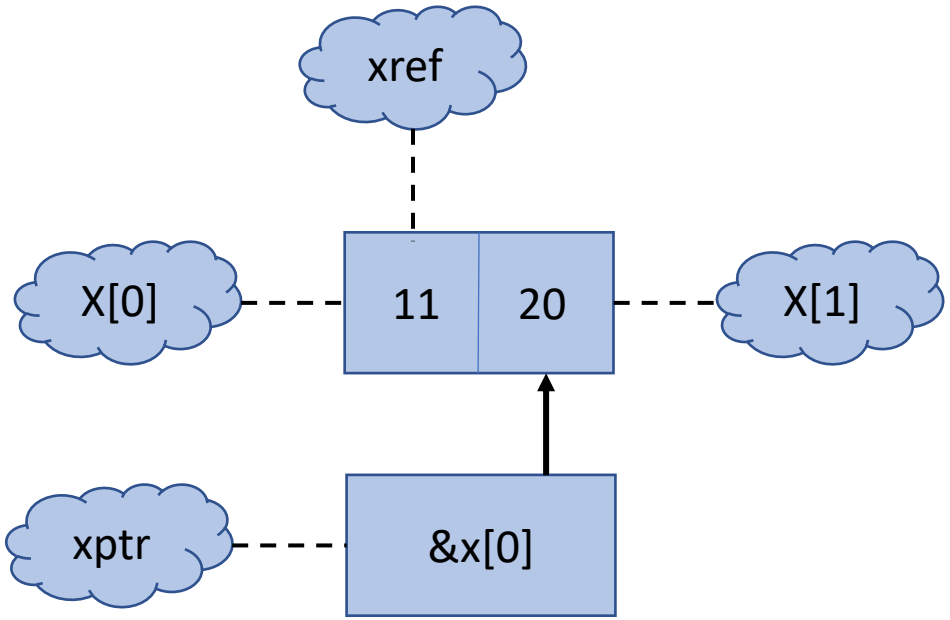


Синтаксис ссылок

- Не путайте синтаксис ссылок со взятием адреса!

```
int x[2] = {10,20};
int& xref = x[0];      // еще одно имя для x[0]
int* xptr = &x[0];     // взятие адреса
xptr++;
xref++;

assert(xref == 11);
assert(*xptr == 20);
```



Правила для ссылок

- Связанную ссылку нельзя перевязать на другое имя.

```
int x, y;
int& xref = x;  // больше нет возможности связать xref с y
xref = y;       // то же, что и x=y
```

- Ссылки прозрачны для операций, включая взятие адреса.

```
int* xptr = &xref;   // то же, что и int* xptr = &x
```

- Сами ссылки не имеют адреса, нельзя взять указатель на ссылку.

```
int &* xrefptr = &xref;   // ошибка, указатель на ссылку!
int *& xptrref = xref;    // ok, ссылка на указатель
```

Использование ссылок

- Представим некую функцию, которой нужно читать два тяжёлых объекта.

- Эта сигнатура плоха (все ли понимают чем?)
```
int foo(Heavy obj) { // obj.x }
```

- Эта сигнатура куда лучше но придётся разыменовывать указатели.
```
int foo(const Heavy *obj) { //obj->x }
```

- Эта сигнатура использует указатели неявно.
```
int foo(const Heavy &obj) { //obj.x }
```

Использование ссылок

- Синонимы внутри больших объектов

```
/* еще одно имя для доступа внутрь большого объекта */
int &inner = obj.big_data[5].matches.inner;
```

- Указатель всегда можно заменить, в отличие от ссылки

```
int *inner = &obj.big_data[5].matches.inner;
inner += 1;
```

Понятно, что вы хотели инкрементировать счетчик `inner`,
но забыли разыменовать указатель …только кого это волнует…

- Эта сигнатура куда лучше но придётся разыменовывать указатели.

```
int foo(const Heavy *obj) { //obj->x }
```

- Эта сигнатура использует указатели неявно.

```
int foo(const Heavy &obj) { //obj.x }
```

Священная война

- На другой стороне считают, что указатель является плохим out-параметром, поскольку он двусмыслен, например функция может рассчитывать, что out-параметр – массив.

```
void foo(int &);
void bar(int *); // это точно не массив?

int x;

foo(x);    // не очевидно, что x out-параметр?
bar(&x);

void foo(int &x) { // очевидно, что x содержит валидный int    }
void bar(int *x) { // не очевидно, что x не nullptr            }
```

А как вы думаете, почему `this` это указатель, ведь по всем соображениям он должен быть ссылкой?

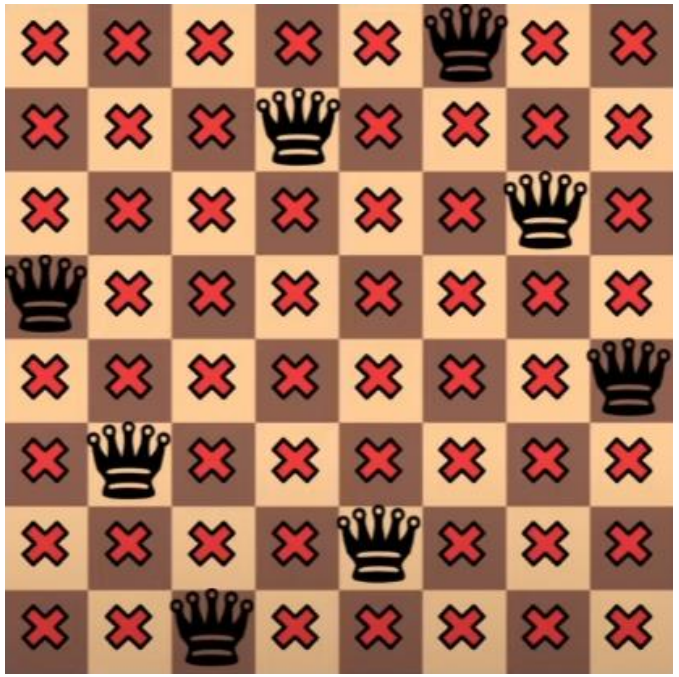
- ❑ Genesis: имена и объекты
  - ➢ Queens Problem
- ❑ Инкапсуляция
- ❑ Консистентность

8 Queens

- Эту задачу придумал в 1848 году немецкий шахматист Макс Фридрих Уильям Безель.

- Есть 8 ферзей и классическая шахматная доска (8x8 клеток).
- Расставить 8 ферзей на шахматной доске таким образом, что бы ни один не атаковал другого.

Возможное решение



- Эдсгер Дейкстра предложил разделить эту задачу и решить сначала ее уменьшенную часть в уме.

Возможное решение

- Возьмем шахматную доску 4x4 и попробуем расположить в ней ферзей так, чтобы они не атаковали друг друга.

- Расположите в их в уме и обратите внимание на ход своих мыслей.

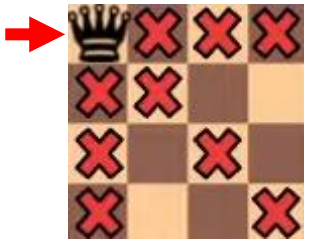
Возможное решение

- Возьмем шахматную доску 4x4 и попробуем расположить в ней ферзей так, чтобы они не атаковали друг друга.

- Расположите в их в уме и обратите внимание на ход своих мыслей.

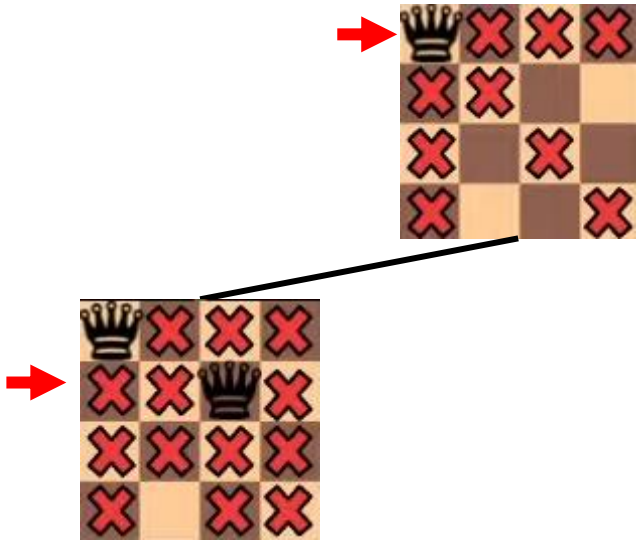
Возможное решение

- У себя в голове вы делали это как-то так:



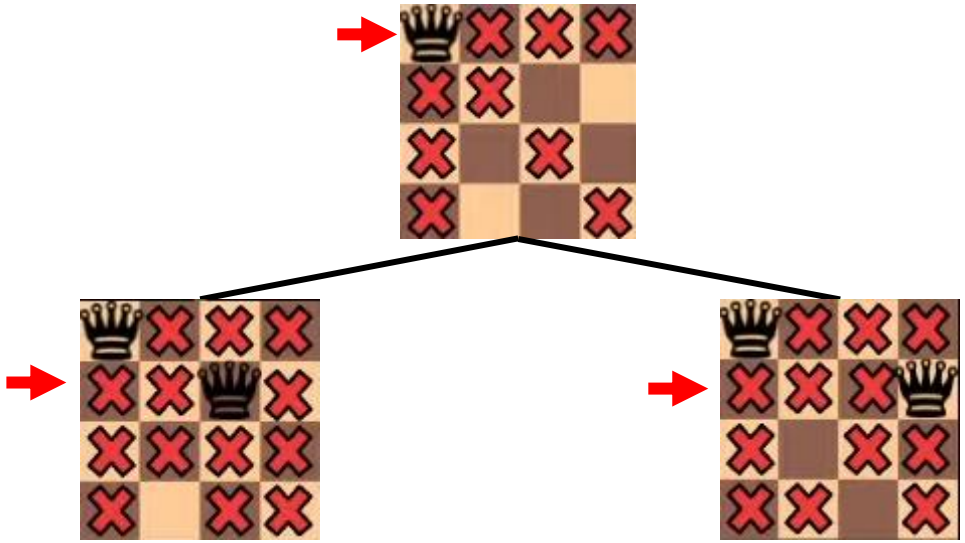
Возможное решение

- У себя в голове вы делали это как-то так:



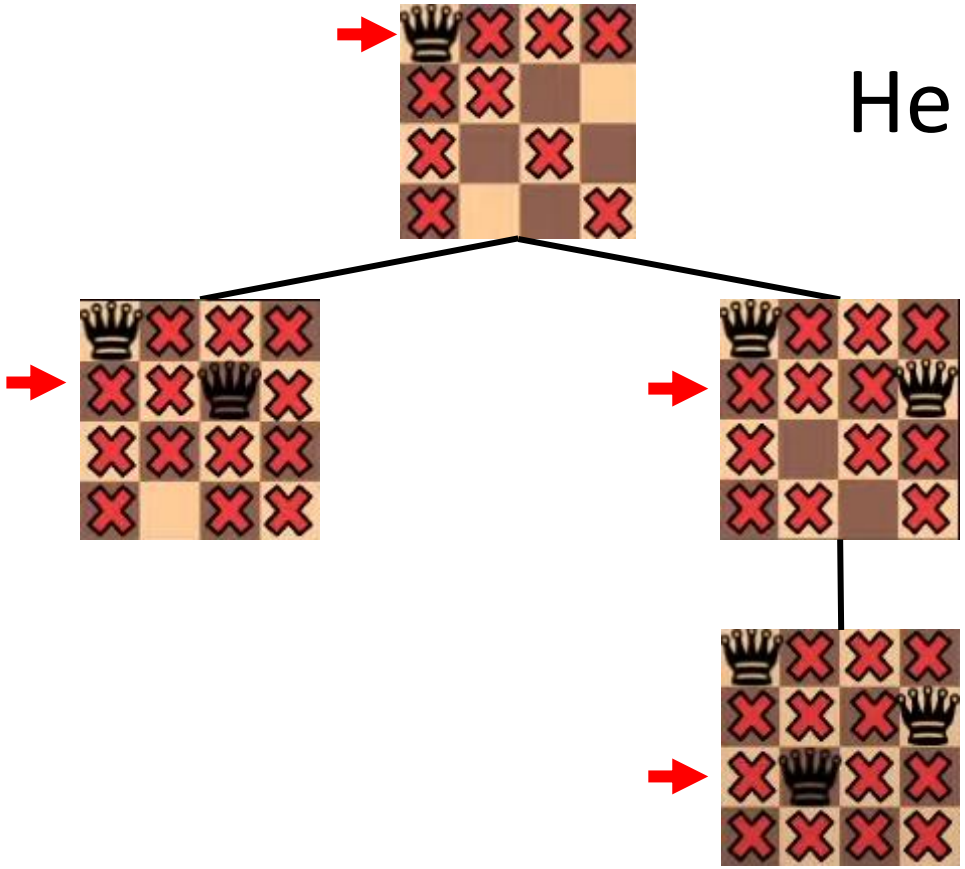
Возможное решение

- У себя в голове вы делали это как-то так:



Возможное решение

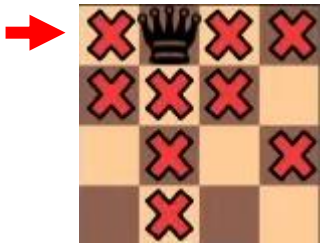
- У себя в голове вы делали это как-то так:



Не получилось, попробую сначала

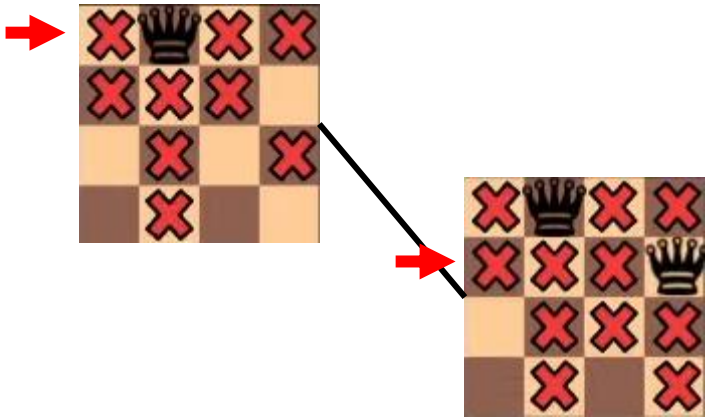
Возможное решение

- У себя в голове вы делали это как-то так:



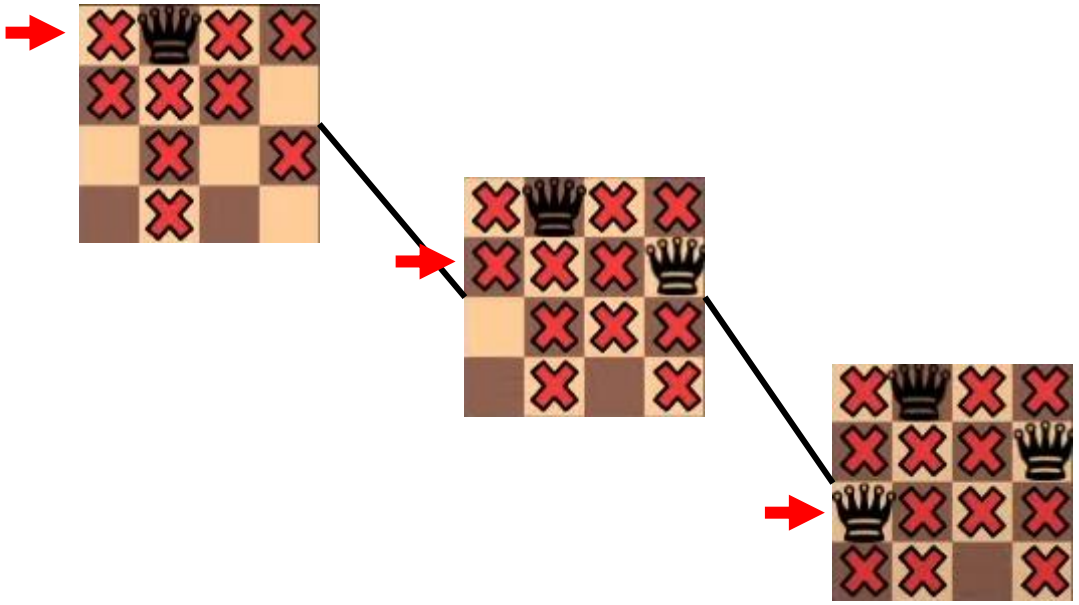
Возможное решение

- У себя в голове вы делали это как-то так:



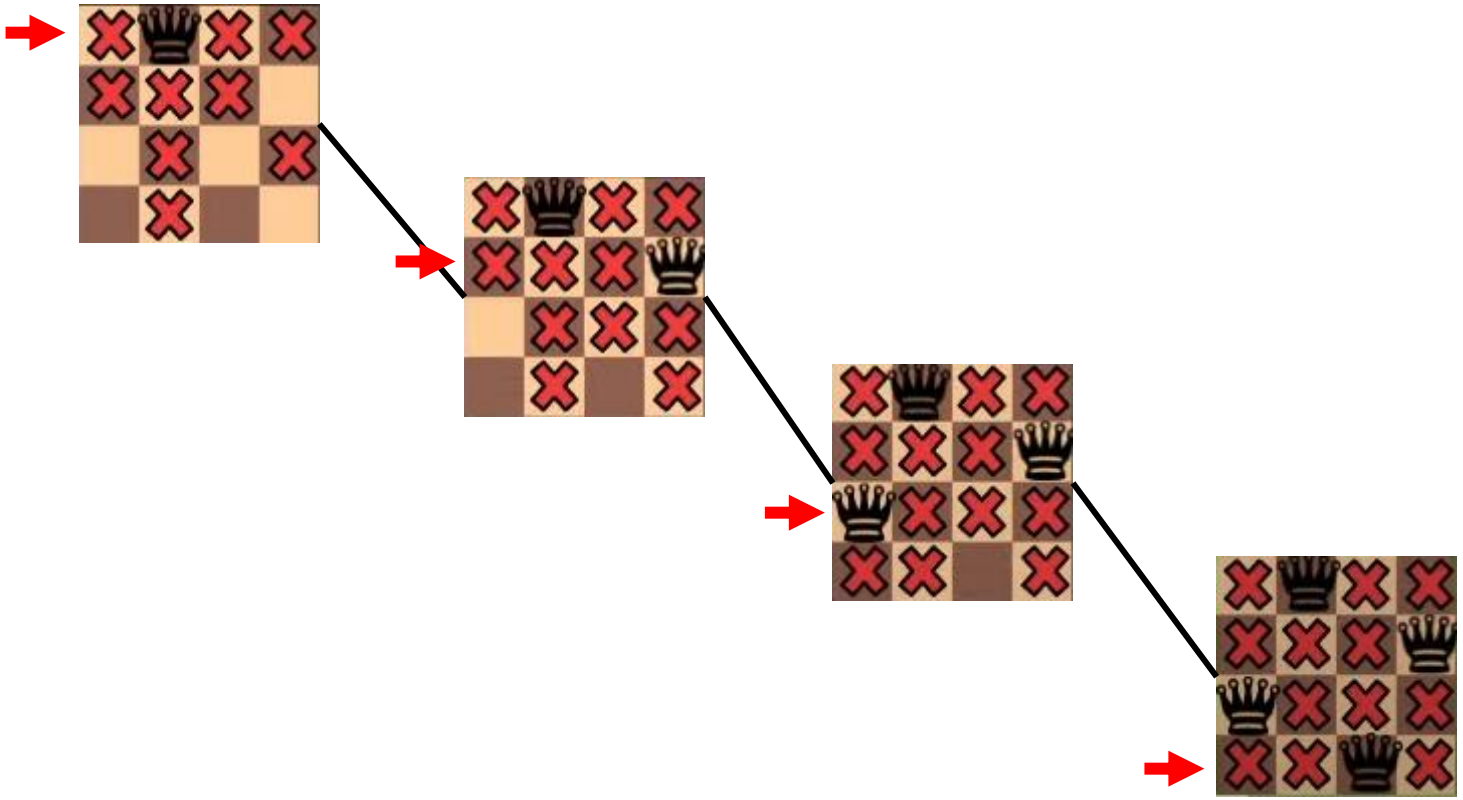
Возможное решение

- У себя в голове вы делали это как-то так:



Возможное решение

- У себя в голове вы делали это как-то так:



Выделяем предметную область

- Нам понадобятся:
- Структуры данных для ферзей и для доски
- Доску можно представить алиасом, например using Board = list<Queen>

- Выделим операции над объектами ферзей, такие как проверка на атакуемость другого ферзя.
- Это будут методы классов.

- На этапе проектирования алгоритмы менее важны. Хорошо спроектированная программа легко переживает смену алгоритмов.

Структура для королевы

```
struct Queen {
    int r, c;

    bool isAttack(const Queen &o) const {
        return r == o.r || c == o.c || abs(r - o.r) == abs(c - o.c);
    }
};
```

- В структуре Queen есть существенный недостаток – любой пользователь Queen может изменить координаты фигуры на доске.
- Ничего не мешает пользователю создать Queen с невалидными координатами вне доски.
- Попытка расчета возможности атаки на другие фигуры может привести к разными результатам.
- Можно принудительно в каждом методе проверять валидность координат, но выглядит это слишком печально.

- ❑ Genesis: имена и объекты
- ❑ Queens Problem
  - ➢ Инкапсуляция
- ❑ Консистентность

Case study: список

```
template <typename T>
size_t list_t::length() const {
    size_t len = 0;
    node_t* cur = top_;
    while (cur != nullptr) {
        len += 1;
        cur = cur->next_;
    }
    return len;
};
```

Что не так с этим методом?

Case study: список

```
template <typename T>
size_t list_t::length() const {
    size_t len = 0;
    node_t* cur = top_;
    while (cur != nullptr) {  // тут может быть петля
        len += 1;
        cur = cur->next_;
    }
    return len;
};
```

- Он может иметь недетерминированное время работы

```
list_t l;
l.top_->next_ = l.top_;  // так появилась петля
size_t len = l.length(); // а так мы застряли в вечном цикле
```

- Можем ли мы проверить, что в списке нет петли?

Обсуждение

- Есть две похожие ситуации:
  1. Методы Queen закладываются на то, что координаты находятся в валидной плоскости.
  2. Методы списка закладываются, что в списке нет петли.

- Интуитивно что-то подсказывает, что «то, на что рассчитывают методы конкретного типа» довольно важное знание

Инварианты

- Предусловиями эффективного метода length является тот факт, что список является корректным двусвязным списком, начинается нулём, завершается нулём, не сломан нигде внутри.

- В общем случае, мы не хотели бы всегда проверять контракт то есть предусловия и постусловия.

- Утверждение, которое должно быть верно всё время жизни объекта некоего типа называется инвариантом этого типа.

- Все методы списка существенно упростятся, если он сможет сохранять свои инварианты.

- Что для этого нужно?

Инварианты

- Все методы списка существенно упростятся, если он сможет сохранять свои инварианты.
- Что для этого нужно?
- Есть методы типа, которые пишем мы как разработчики типа. Сохранять инварианты в методах – обязанность разработчика и он обычно с ней справляется.
- Но есть внешние функции, работающие с объектами этого типа. И вот они как раз являются источником проблем.
- Есть ли у нас языковые средства, чтобы запретить всем, кроме методов класса, работать с его состоянием?

Инкапсуляция в C

- Мы можем использовать механизмы области видимости. Например сделать тип непрозрачным (opaque).

```
struct list_t;
struct list_t *list_create();
int list_length(struct list_t *list);
```

- Теперь пользователь не имеет доступа к состоянию `list_t` и может работать только с указателем на объект только методами этого типа.

- Обсуждение: есть ли проблемы с этим подходом?

Инкапсуляция в C++

- В языке C++ для инкапсуляции используется специальный механизм `private`, позволяющий ограничить видимость полей и методов.

```
template struct list_t {
private:
    struct node_t;
    node_t *top_, *back_;

public:
    int length() const;
};
```

- В структуре по умолчанию все поля `public`.

Структура для королевы

```
struct Cell {
    int row, col;
};

struct Queen {
private:
    Cell cell;
public:
    Queen(int row, int column) : cell{ row, column } { }

    Cell coordinates() const { return cell; }

    bool isAttack(const Queen& o) const {
        return cell.row == o.cell.row ||
               cell.col == o.cell.col ||
               abs(cell.row - o.cell.row) == abs(cell.col - o.cell.col);
    }
};
```

godbolt.org

Обсуждение

- У нас есть линейная модель памяти.

- Разве это не значит, что просто приведя указатель на объект к `char*` мы можем нарушить все инварианты?

- ❑ Genesis: имена и объекты
- ❑ Queens Problem
- ❑ Инкапсуляция
- ➢ Консистентность