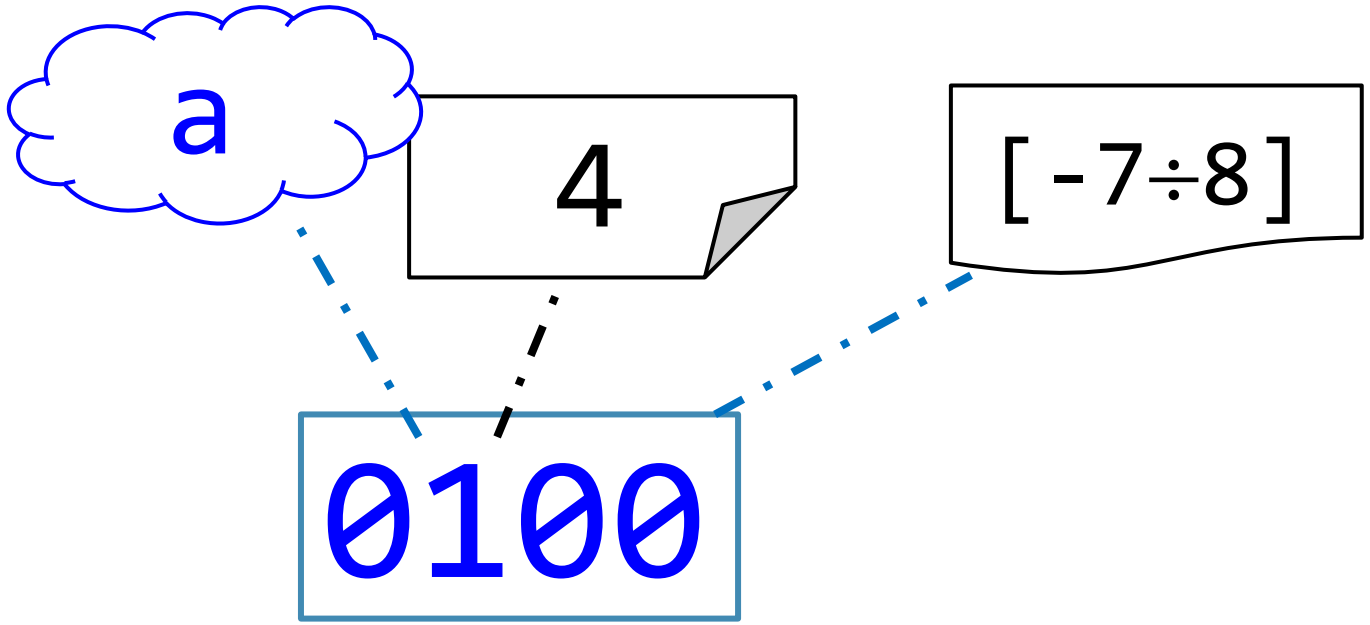
УКАЗАТЕЛИ И ССЫЛКИ

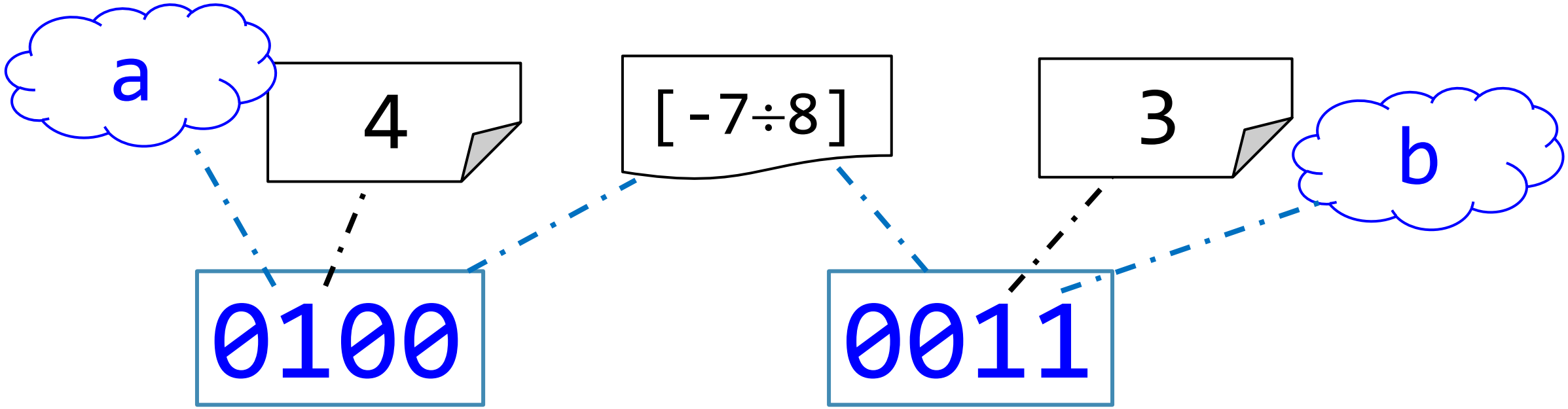
Genesis снова. Немного вычислительной геометрии. Инкапсуляция

К. Владимиров, Syntacore, 2022
mail-to: konstantin.vladimirov@gmail.com

- Genesis: имена и объекты
- Вычислительная геометрия
- Инкапсуляция
- Консистентность

0100


a
4
[-7÷8]
0100

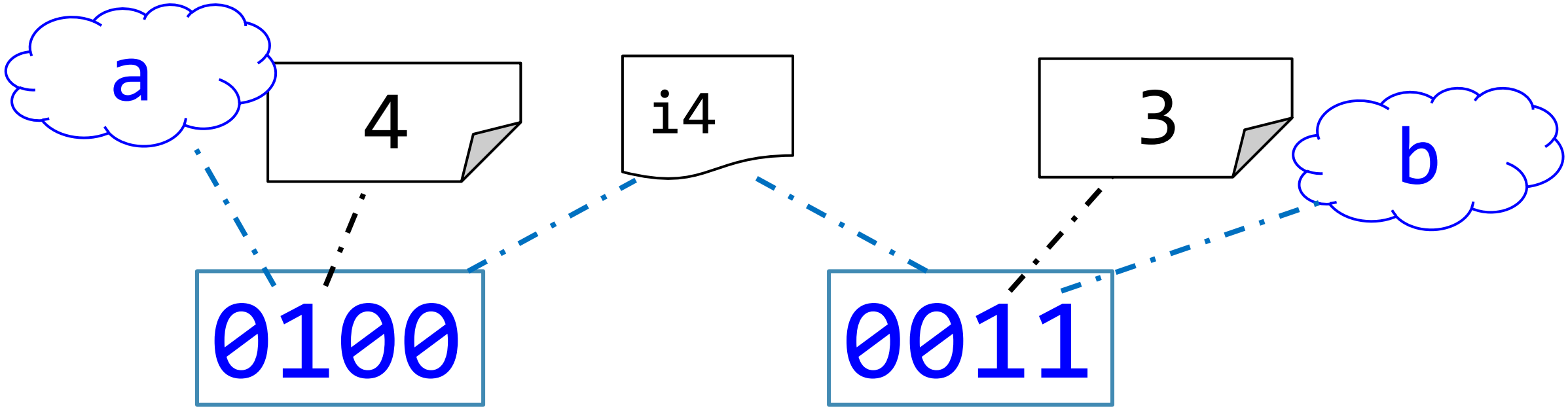

a
4
[-7÷8]
3
b
0100
0011

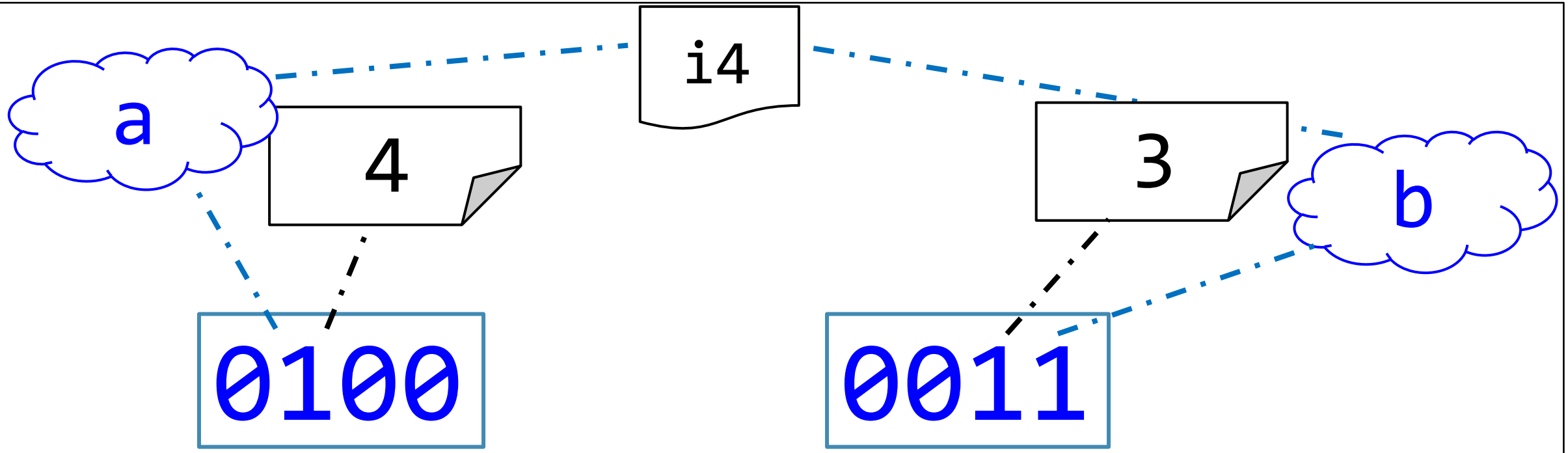
Обсуждение

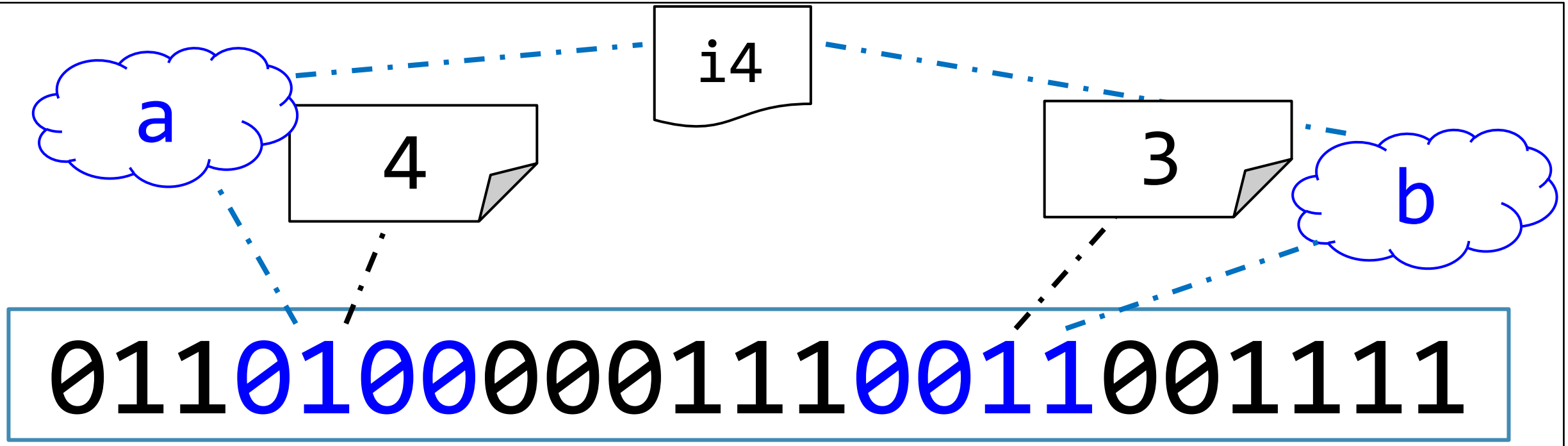
- Что такое тип?
- Достаточно ли для типа задать диапазон значений, например [-7÷8]?

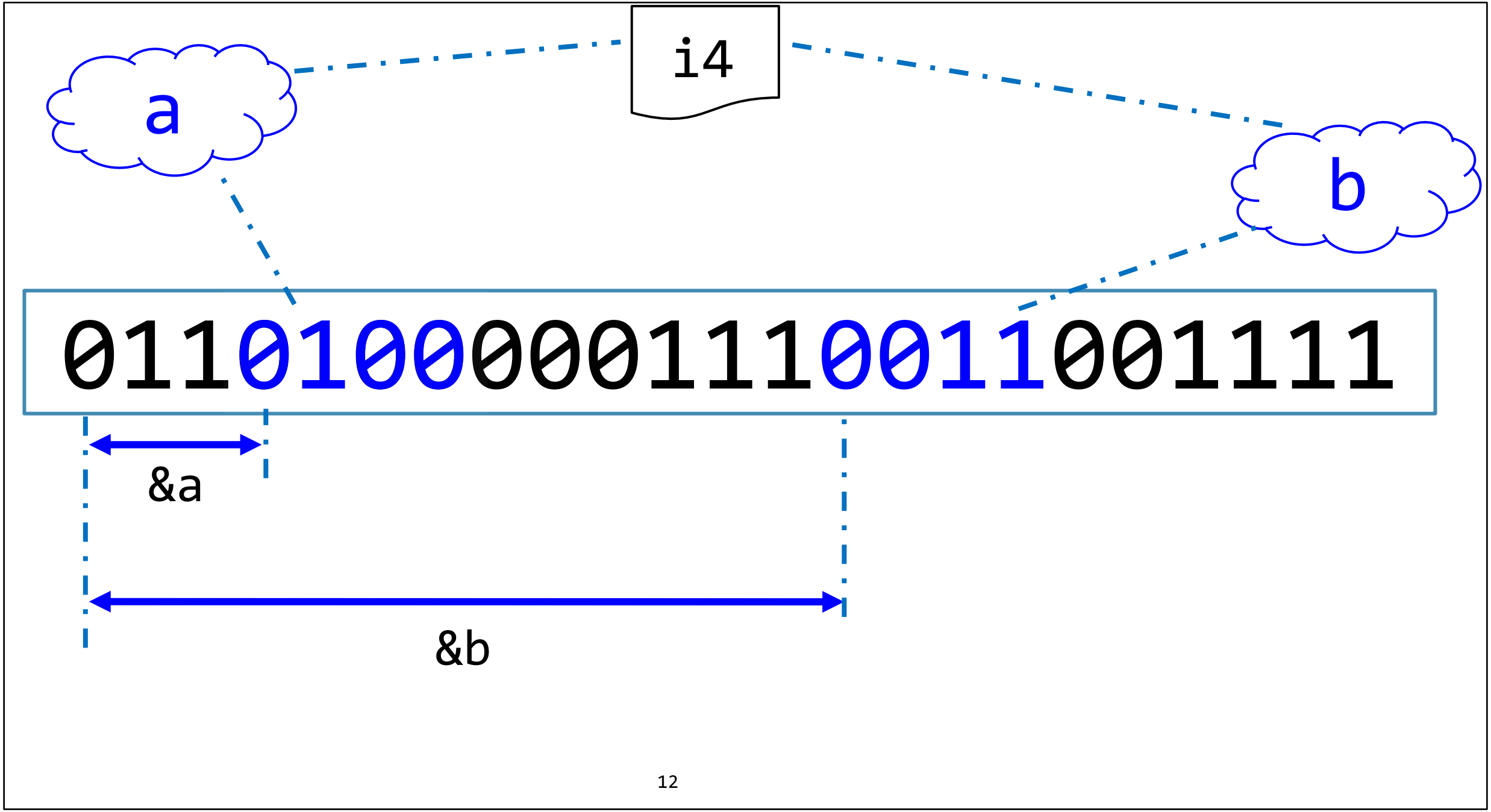
Типы: value types & object types

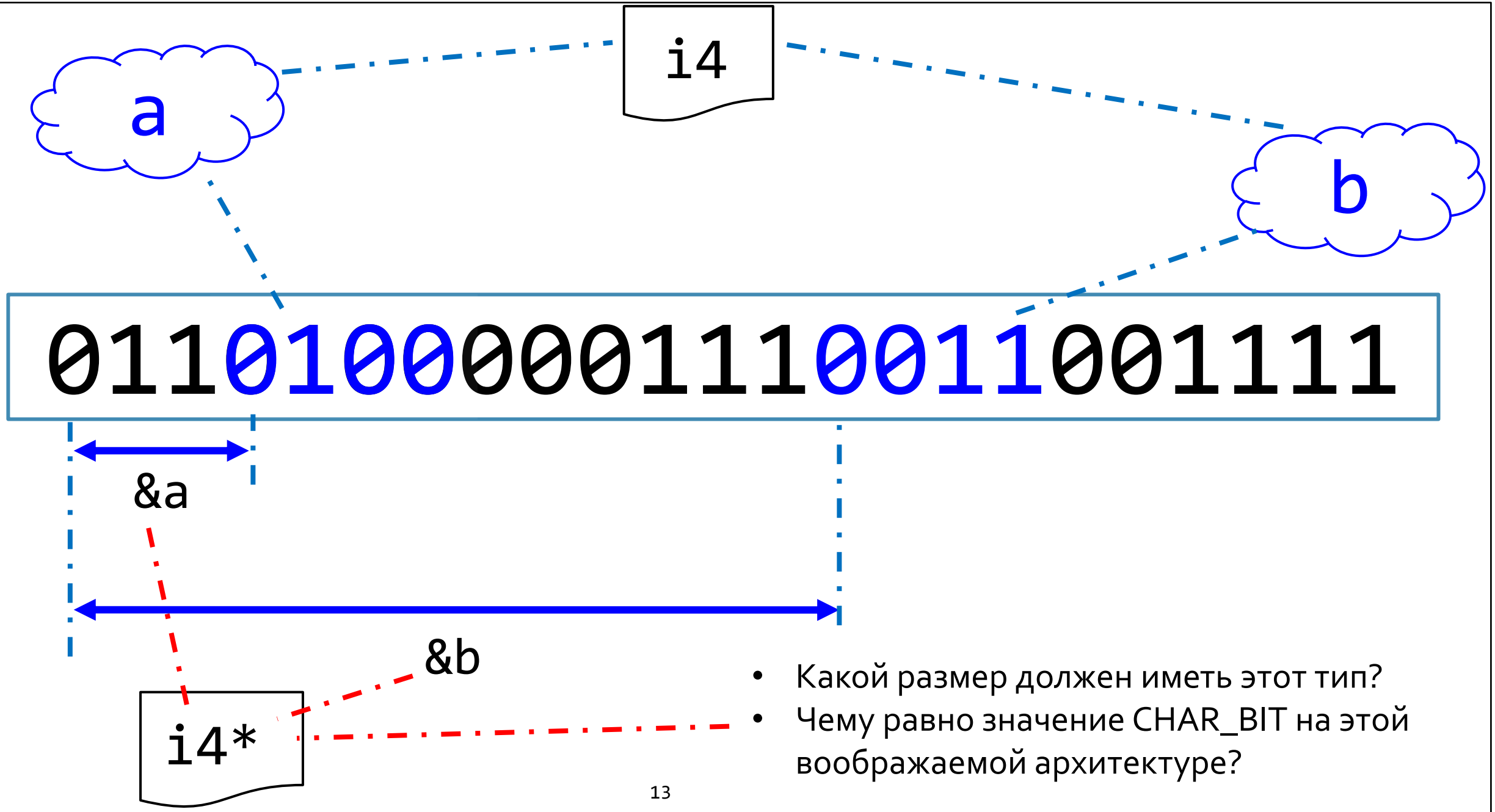
- Что такое тип?
- **value type**: диапазон значений объекта.
- **object type**: совокупность операций над объектом.
  - Например $5 \ / \ 2$ даст 2 для типа int но 2.5 для типа double
  - $0 \ - \ 1$ даст -1 для char, но 255 для unsigned char
- Назовём целочисленный четырёхбитный арифметический тип `i4`.


a
4
i4
3
b
0100
0011

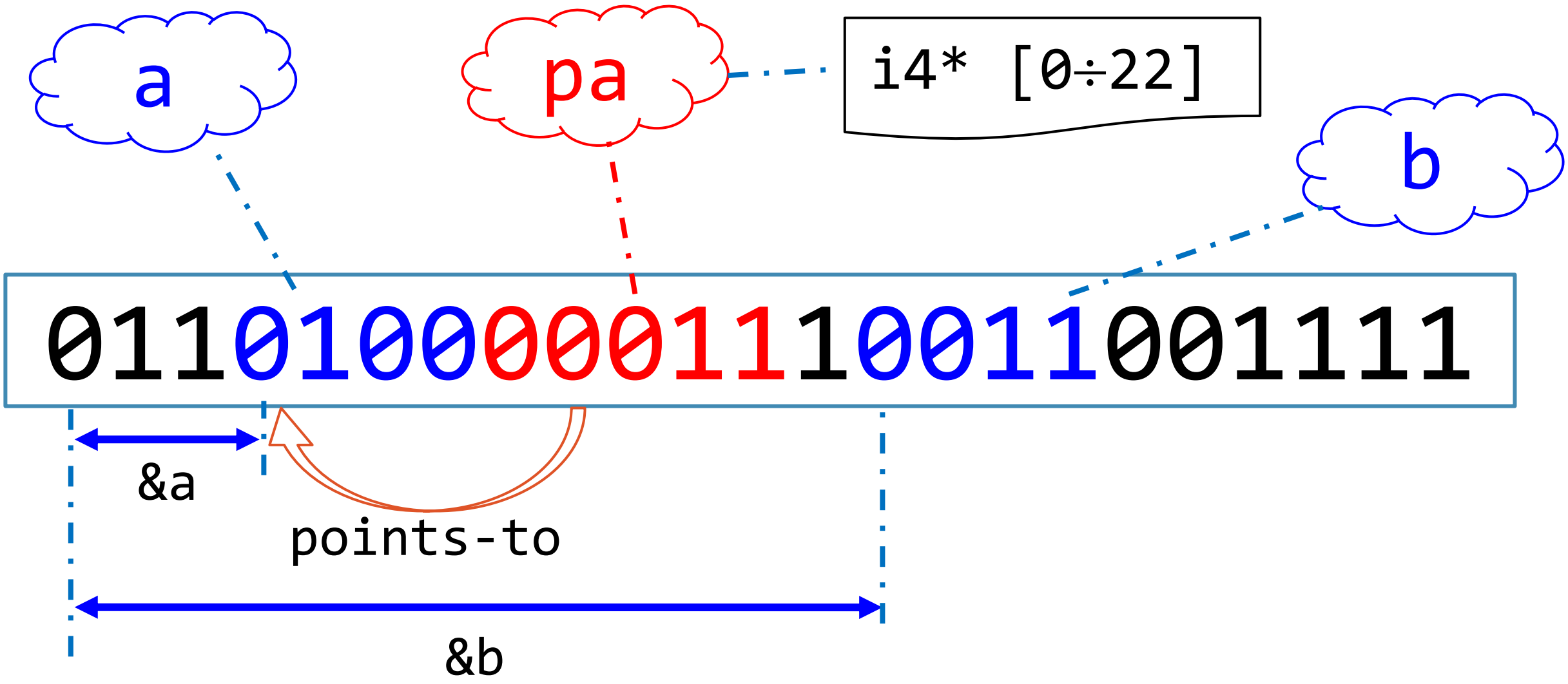

i4
a
4
3
b
0100
0011


i4
a
4
3
b
01101000000111001101001111


i4
a
b
01101000000111001101111
&a
&b



- Какой размер должен иметь этот тип?
- Чему равно значение CHAR_BIT на этой воображаемой архитектуре?


a
pa
i4* [0÷22]
b
01101000000111001101111
&a
points-to
&b

Нулевые указатели

- Если указатель это просто расстояние, может быть и нулевое расстояние?
- Нулевой указатель это специальный "маркер ничего". По нему ничего не лежит.
- Не надо путать `0`, `NULL` и `nullptr`.

```
if (!p) { smth(); } // сработает во всех трёх случаях
```

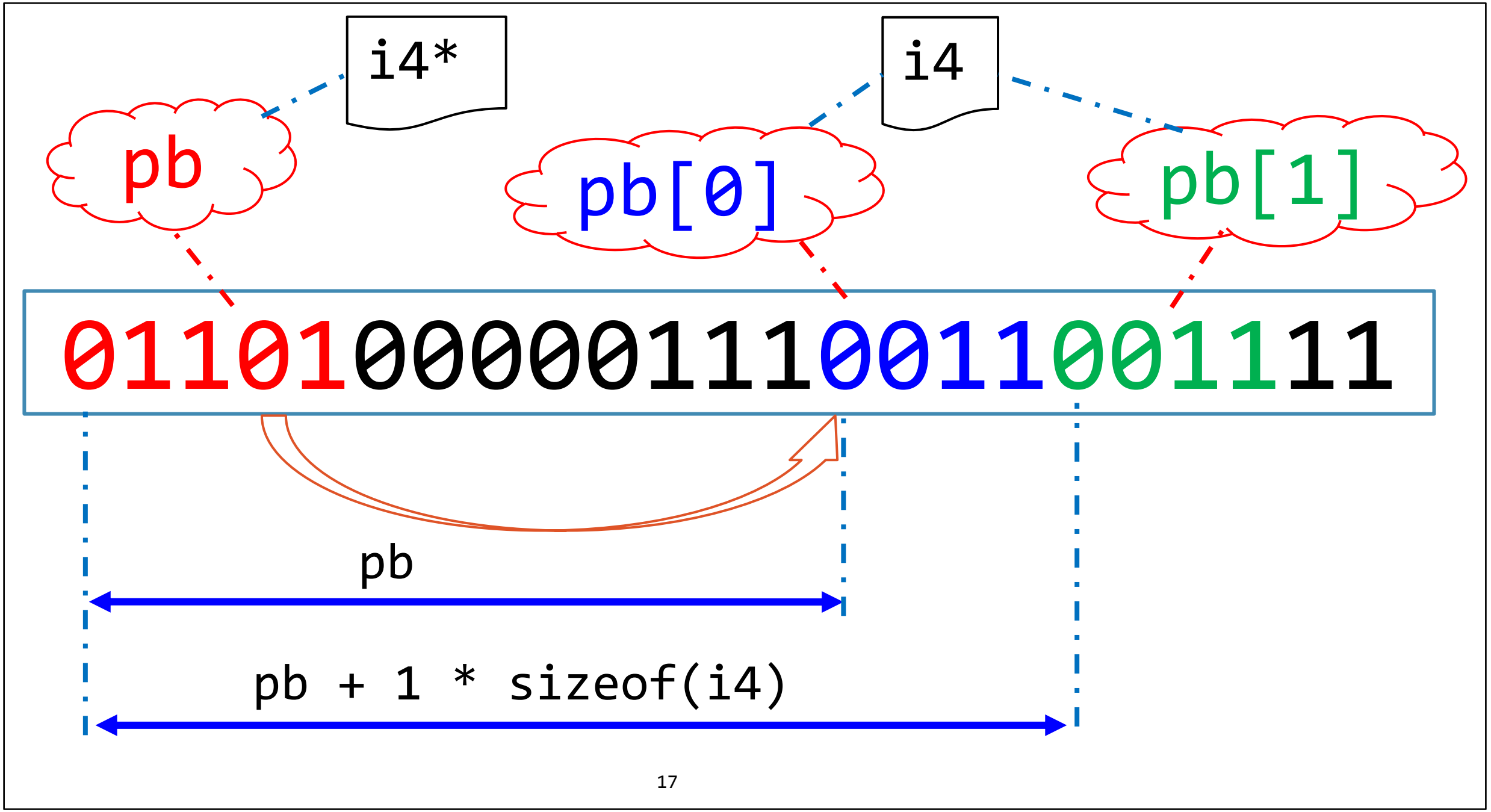
- В языке C++ наш выбор `nullptr` и мы поймём почему это так когда дойдём до перегрузки функций.

Индексация указателей

- Изначально указатели всегда были указателями внутрь массивов.

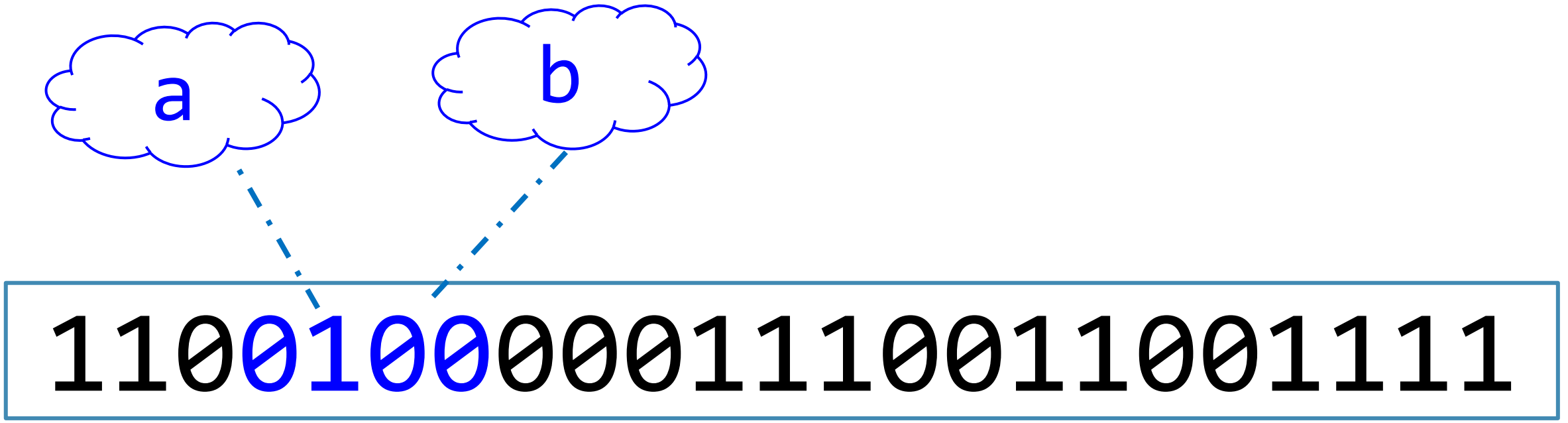
`p[2] == *(p + 2)`

- Поскольку сложение коммутативно, `2[p]` тоже сработает.
- Обсуждение: есть ли случаи когда такой синтаксис обоснован?
- Все ли понимают сколько байт будет добавлено к p при сложении с целым?


i4*
i4
pb
pb[0]
pb[1]
01101000000111001100111
pb
pb + 1 * sizeof(i4)

Ссылки (lvalue references)

- И если бы речь шла о языке C, то это было бы всё. Но в C++ есть уникальная возможность: два имени у одного объекта.



Синтаксис ссылок

- Базовый синтаксис lvalue ссылок это одинарный амперсанд.

```
int x;
int &y = x; // теперь y это просто ещё одно имя для x
```

- Не путайте его со взятием адреса!

```
int x[2] = {10, 20};
int &xref = x[0];
int *xptr = &x[0];
xref += 1;
xptr += 1;

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

Синтаксис ссылок

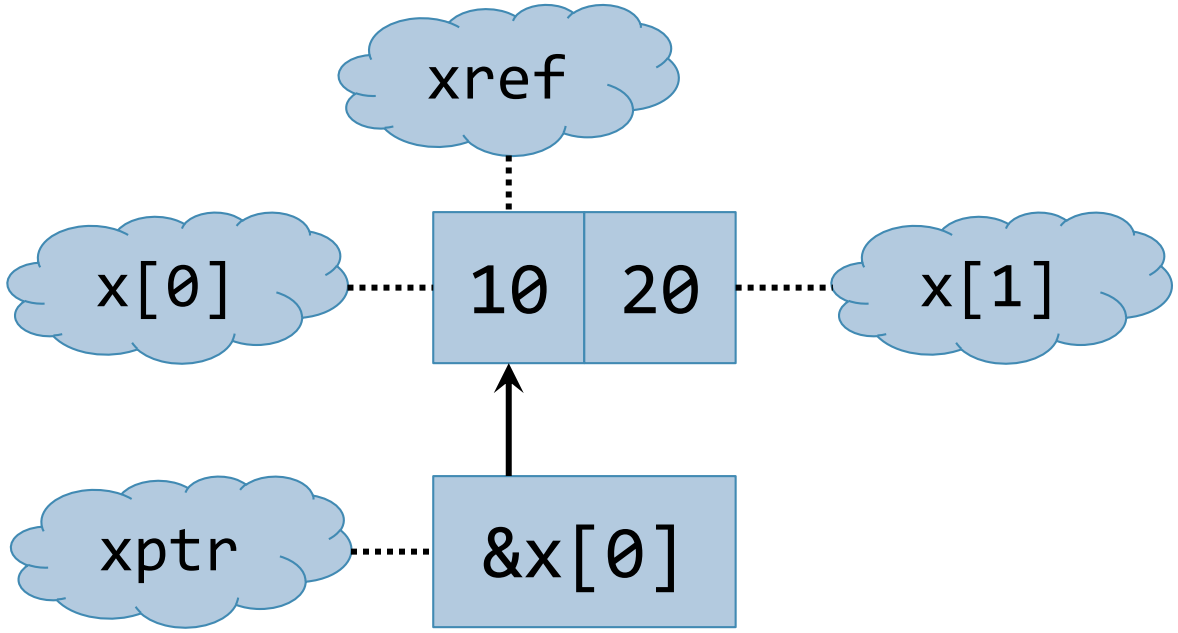
- Базовый синтаксис lvalue ссылок это одинарный амперсанд.

```
int x;
int &y = x; // теперь y это просто ещё одно имя для x
```

- Не путайте его с разыменованием!

```
int x[2] = {10, 20};
int &xref = x[0];
int *xptr = &x[0];
xref += 1;
xptr += 1;

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



Синтаксис ссылок

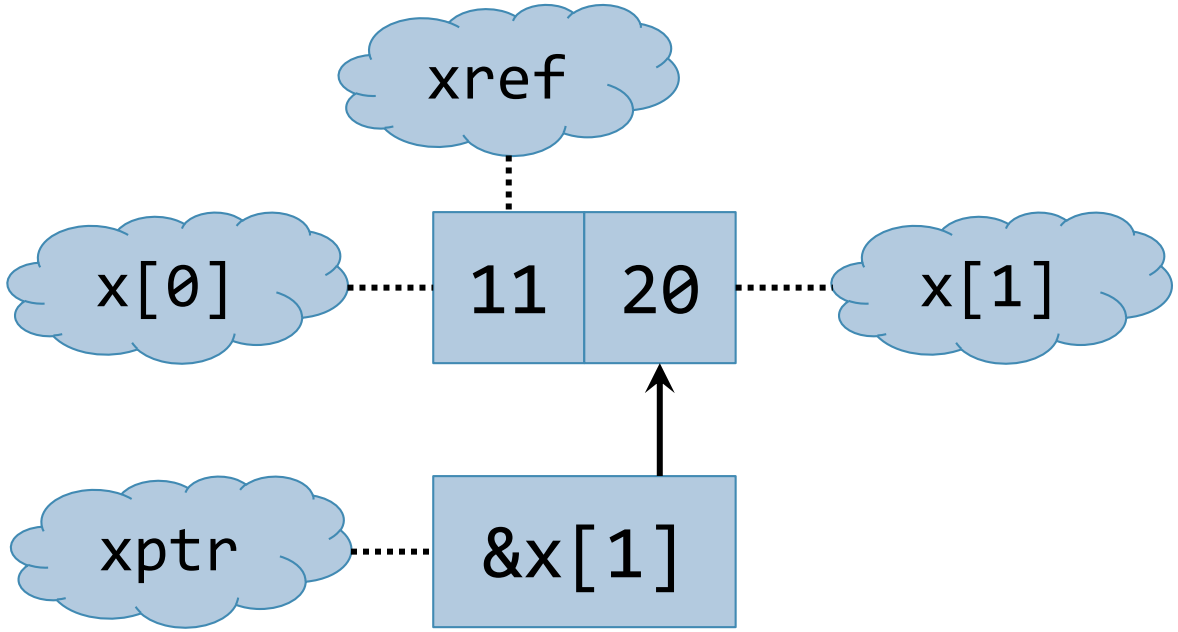
- Базовый синтаксис lvalue ссылок это одинарный амперсанд.

```
int x;
int &y = x; // теперь y это просто ещё одно имя для x
```

- Не путайте его с разыменованием!

```
int x[2] = {10, 20};
int &xref = x[0];
int *xptr = &x[0];
xref += 1;
xptr += 1;

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



Правила для ссылок

- Единожды связанную ссылку нельзя перевязать.

```
int x, y;

int &xref = x; // теперь нет возможности связать имя xref с переменной y
xref = y; // то же, что x = y
```

- Ссылки прозрачны для операций, включая взятие адреса.

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

- Сами ссылки не имеют адреса. Нельзя сделать указатель на ссылку.

```
int &*xrefptr = &xref; // ошибка

int *& xptrref = xptr; // ok, ссылка на указатель
```

Константность для ссылок

- Все ли помнят правила константности для указателей?

```
const char *s1; //?
char const *s2; //?
char * const s3; //?
char const * const s4; //?
```

Константность для ссылок

- Все ли помнят правила константности для указателей?

```
const char *s1; // указатель на константные данные (west-const)
char const *s2; // указатель на константные данные (east-const)
char * const s3; // константный указатель на (изменяемые) данные
char const * const s4; // константный указатель на константные данные
```

- Правила для ссылок гораздо проще.

```
char &r1 = r; // неконстантная ссылка (на изменяемые данные)
const char &r2 = r1; // константная ссылка (на константные данные)
```

Использование ссылок

- Представим некую функцию, которой нужно читать два тяжёлых объекта.
- Эта сигнатура плоха (все ли понимают чем?)

```
int foo(Heavy fst, Heavier snd) {  // fst.x
```

- Эта сигнатура куда лучше но придётся разыменовывать указатели.

```
int foo(const Heavy *fst, const Heavier *snd) { // fst->x
```

- Эта сигнатура использует указатели неявно.

```
int foo(const Heavy &fst, const Heavier &snd) { // fst.x
```

Использование ссылок

- Синонимы внутри больших объектов.

```
void mytype::change_internal(some_big_obj &obj) {
  int &internal = obj.somewhere[5].guts.internal;
  // код, активно изменяющий internal
}
```

- Здесь разница заметнее. Указатель был бы ячейкой памяти. Ссылка это просто имя.
- Кроме того, указатель всегда может быть изменён.

```
int *internal = &obj.somewhere[5].guts.internal;
internal += 5; // не имеет смысла тут, но всегда возможно!
```

Немного священных войн

- Многие считают, что ссылка это плохой out-параметр так как она не очевидна при вызове.
- Другие считают, что указатель плох как out-параметр, т.к. он двусмысленен.

```
void foo(int &);
void bar(int *); // не очевидно, что это не массив

int x;

foo(x); // не очевидно, что x это out-param
bar(&x);
```

- Что вы думаете?

Немного священных войн

- Дополнительный аргумент это состояние внутри функции.

```
void foo(int &x) {
  // очевидно, что x содержит int
}

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

- Более ограниченный интерфейс ссылок часто позволяет сократить рантайм-проверки.

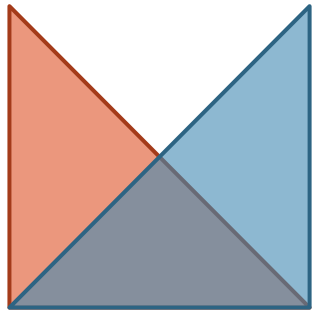
Обсуждение

- Как вы думаете, почему this это указатель а не ссылка?

- Имена и объекты
- Вычислительная геометрия
- Инкапсуляция
- Консистентность

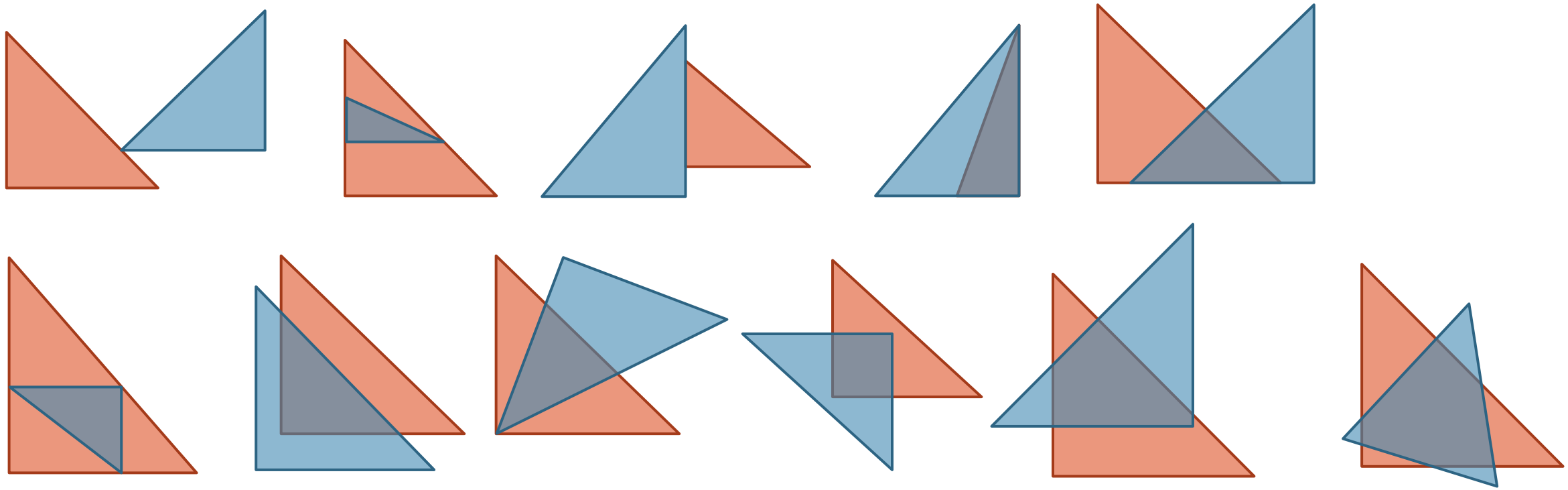
Я просто хочу пересечь треугольники

- Со стандартного ввода приходят два набора точек представляющих плоские треугольники. Задача: на стандартный вывод вывести площадь пересечения.
- Ввод: `0 0 0 1 1 0 0 0 1 1 1 0`
- Вывод: `0.25`
- Это сложная задача или простая?
- Как бы вы её решали?



На что похоже решение?

- Первый вопрос в таких обманчиво-простых задачах это: а как вообще может выглядеть решение?



Выделим предметную область

- Нам понадобятся:
  - Структуры для двумерной точки, отрезка, треугольника, полигона.
  - Важный инсайт: координаты лучше сразу закладывать FP.
  - Это будут типы в нашей программе.
- Выделим операции над объектами этих типов.
  - Пересечение отрезков, взаимоположение точки и отрезка, построение полигона как выпуклой оболочки множества точек, вычисление площади полигона, вероятно что-то ещё….
  - Это будут методы классов.
- На этапе проектирования алгоритмы менее важны. Хорошо спроектированная программа легко переживает смену алгоритмов.

lingeo.hpp

Скетч: структура для точки

```
struct point_t {
  float x = NAN, y = NAN;

  void print() const;
  bool valid() const;
  bool equal(const point_t &rhs) const;
};
```

- Такая точка может быть сконструирована по умолчанию (в невалидном состоянии).
- Метод `equal` должен проверять `std::abs(x - rhs.x) < flt_tolerance`.

Скетч: структура для линии

```
// line_t -- line in form of ax + by + c = 0
struct line_t {
  float a = -1.0f, b = 1.0f, c = 0.0f;

  void print() const;
  bool valid() const;
  line_t(const point_t &p1, const point_t &p2);
```

- Такая линия может быть и сконструирована по умолчанию и собрана из двух точек.
- Конструктор из двух точек уже не слишком тривиален. Как бы вы его написали?

Скетч: конструктор для линии

```
line_t(const point_t &p1, const point_t &p2) {
  float angle = std::atan((p2.y - p1.y) / (p2.x - p1.x));
  float sin_angle = std::sin(angle);
  float cos_angle = std::sqrt(1.0 - sin_angle * sin_angle);
  point_t normal_vect{-sin_angle, cos_angle};
  a = normal_vect.x;
  b = normal_vect.y;
  c = -(p1.x * normal_vect.x + p1.y * normal_vect.y);
}
```

- Один из вариантов реализации конструктора. Может быть можно лучше?
- Как бы вы его протестировали?

Идея unit-тестов

- Тесты на конкретные интерфейсы классов в терминах этих интерфейсов.

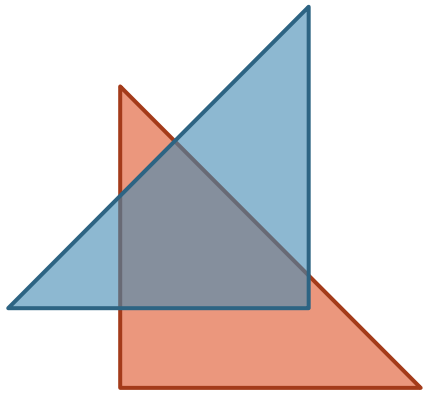
```
point_t p1{0, 0}, p2{1, 1};
line_t l1{p1, p2};
check(std::abs(l1.a - l1.b) < flt_tolerance);
check(std::abs(l1.c) < flt_tolerance);
```

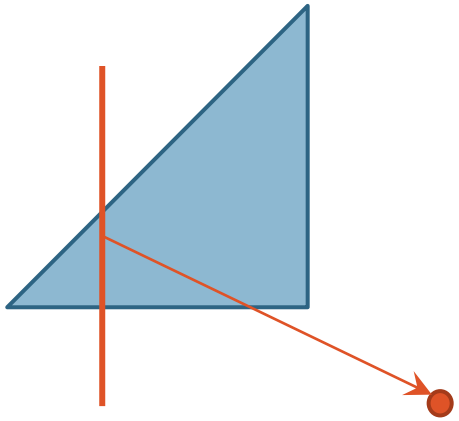
- Существует масса систем, облегчающих юнит-тестирование
  - Catch
  - Boost.Test
  - Google Test
- Попробуйте сами протестировать с их помощью выложенный код.

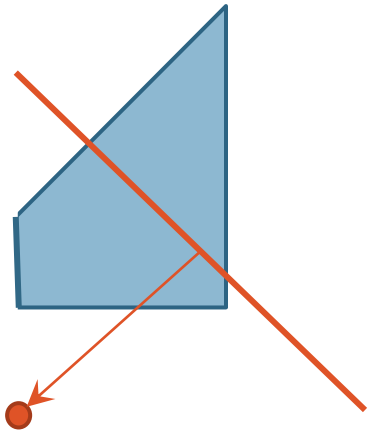
lingeo.hpp, square.cpp

Идея общего алгоритма

- Предположим что все вершины составляющие полигон отсортированы по кругу относительно его центра, **n_{th} side constructed by n_{th} and $(n + 1)_{th}$ vertices**.
- Тогда достаточно пересечь полигон каждой стороной другого полигона, каждый раз **for n_{th} side leave half-space with $(n + 2)_{th}$ point (wrap around)**.







Домашняя работа HW3D

- Со стандартного ввода приходит число `0` < N < `1000000`, а потом N наборов точек, представляющих трёхмерные треугольники. Задача: вывести номера всех треугольников, которые пересекаются с каким-либо другим
- Можно воспользоваться $[GCT]$ если вам не хватает базы в таких вопросах
- Как вы будете тестировать ваш алгоритм?

Обсуждение

- В принципе в опубликованном классе полигона есть существенная проблема.
- Ничто не мешает пользователю создать полигон не удовлетворяющий условию отсортированности вершин.
- Попытка пересечь его с другим полигоном (даже с корректным) вероятно даст самые причудливые результаты.
- Что делать? Можно ввести принудительную сортировку вершин по кругу перед каждым персечением, но это довольно дорогой шаг.

- Имена и объекты
- Вычислительная геометрия
- Инкапсуляция
- Консистентность

Case study: список

- Когда мы проектировали хэши на C, побочным эффектом был свой собственный список.
- Давайте ещё разок напишем его на C++.

```
template <typename T> struct list_t {
  struct node_t {
    node_t *next_, *prev_;
    T data_;
  };

  node_t *top_, *back_;
};
```

- Можем ли мы написать для него метод length?

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) { // а с чего мы взяли, что нет петли?
    len += 1;
    cur = cur->next_;
  }

  return len;
};
```

- Он может иметь недетерминированное время работы.

Case study: список

```
list_t<int> l;
// тут как-то заполняем
l.top_->next_ = l.top_; // oops
size_t len = l.length();
```

- Можем ли мы проверить, что в списке нет петли?
- Алгоритм Флойда вычисляет количество элементорв даже если петля есть.
- Но что если мы хотим теперь написать метод reverse?
- Надо ли в начале reverse **опять** вызывать алгоритм Флойда, проверяя нет ли петли и удваивая общее время работы?

Обсуждение

- У нас уже две очень похожие ситуации.
  1. Методы для полигона закладываются на то, что вершины отсортированы по кругу.
  2. Методы для списка закладываются на то, что он не содержит петли.
- Интуитивно "то на что рассчитывают методы конкретного типа" это нечто довольно важное.

Инварианты

- Предусловиями эффективного метода reverse 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 языке 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 и может работать только с указателем на объект только методами этого типа.
- Обсуждение: есть ли проблемы с этим подходом?

Инкапсуляция в языке C++

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

```
template <typename T> struct list_t {
private:
  struct node_t;
  node_t *top_, *back_;

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

- В структуре по умолчанию все поля public.

Инкапсуляция в языке C++

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

```
template <typename T> class list_t {
// private:
  struct node_t;
  node_t *top_, *back_;

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

- Новое ключевое слово class определяет по умолчанию закрытые поля.

Обсуждение: инварианты и линейность

- У нас есть линейная модель памяти.
- Разве это не значит, что просто приведя указатель на объект к `char*` мы можем нарушить все инварианты?

- Имена и объекты
- Вычислительная геометрия
- Инкапсуляция
- Консистентность

Неконсистентное состояние

- У нас есть линейная модель памяти
- Разве это не значит, что просто приведя указатель объект к char* мы можем нарушить все инварианты?
- Да можем (по крайней мере для standard-layout и для trivially copyable). Идея в том, что мы не хотим этого делать.
- Объект у которого нарушены инварианты это объект в неконсистентном состоянии операции над ним опасны и непредсказуемы.
- Никакой программист, будучи в своём уме, не приведёт свой или чужой объект в неконсистентное состояние по доброй воле.

Обсуждение: ссылки

- Ссылки тоже сохраняют инварианты.

```
int foo(const int *p) { int t = *p; delete p; return t; }

int bar(const int &p) { return p; }

foo(nullptr); // это невозможно проделать с bar

double d = 1.0;
int *q = *reinterpret_cast<int **>(&d);

foo(q); // это невозможно проделать с bar
```

- Инвариант const int reference: правильное и не вам принадлежащее целое число под ней. Именно поэтому побитовое представление ссылки скрыто.

Важное замечание

- Инкапсуляция это свойство типа, а не его объектов.

```
template <typename T> class list_t {
  node_t<T> *top_, *back_;

public:
  void concat_with(list_t<T> other) {
    for (auto cur = other.top_; // всё нормально, мы можем
         cur != other.back_;    // работать не только с this
         cur = cur->next_)      // а с любым объектом list_t<T>
      push(cur->data_);
  }
};
```

Важное замечание

- При этом разные шаблонные параметры инстанцируют разные типы.

```
template <typename T> class list_t {
  node_t<T> *top_, *back_;

public:
  template <typename U>
  void concat_with(list_t<U> other) {
    for (node_t<U> *cur = other.top_; // нарушение инкапсуляции
         cur != other.back_;
         cur = cur->next_)
      push(cur->data_);
  }
};
```

Конструкторы и деструкторы

- Инкапсуляция делает критически важными конструкторы.
- Теперь состояние объектов просто нельзя установить извне.

```
template <typename T> class list_t {
  node_t<T> *top_ = nullptr, *back_ = nullptr;
public:
  list_t(size_t initial_len); // ctor
  ~list_t(); // dtor
```

- Но в случае со списком, его нельзя и очистить извне. Поэтому важными становятся также деструкторы. Синтаксис показан на слайде.

Обсуждение

- Увы, старые добрые malloc и free ничего не знают о конструкторах и деструкторах.
- Созданный с их помощью в динамической памяти объект не будет корректно инициализирован и будет создан в невалидном состоянии.
- Что делать?

Аллокация динамической памяти

- В языке C++ аллокация делается через new и delete. Они вызывают конструкторы и деструкторы создаваемых объектов.
- Важно запомнить парность операторов.

```
int *t = new Widget;     // выделяем скалярный объект
delete t;                // освобождаем скалярный объект

int *p = new Widget[5]; // выделяем массив
delete [] p;             // освобождаем массив
```

- Вы не должны пытаться освободить через delete выделенное через new[] и наоборот.
- Вы не должны смешивать new/delete с механизмом malloc/free.

Семантика new и delete

- Парность вызовов крайне важна.

```
using mvi = MyVector<int>;
mvi *pv = new mvi{};                                  // ctor
mvi *pvs = new MyVector<int>[5];                      // 5 ctors
mvi *vpv = static_cast<mvi *> malloc(sizeof(mvi));    // no ctor

delete pv;                                            // dtor
delete [] pvs;                                        // 5 dtors
free(vpv);                                            // no dtor
```

- По типу pv и pvs очень похожи. Как в точке удаления по pvs понять что нужно пять деструкторов?

newdelete.cc

Обсуждение

- Что вы думаете о ссылке на выделенную память?

```
int *p = new int[5];
int &x = p[3];
```

Обсуждение

- Что вы думаете о ссылке на выделенную память?

```
int *p = new int[5];
int &x = p[3];
```

- Вроде всё хорошо если бы не червячок сомнения. А что будет после delete?

Литература

- [CC11] ISO/IEC 14882 – "Information technology – Programming languages – C++", 2011
- [BS] Bjarne Stroustrup – The C++ Programming Language (4th Edition) , 2013
- [GB] Grady Booch – Object-Oriented Analysis and Design with Applications, 2007
- [GCT] Eberly, Schneider – Geometric Tools for Computer Graphics, 2002
- [GS] Gilbert Strang – Introduction to Linear Algebra, Fifth Edition, 2016
- [BB] Ben Saks – Back to Basics: Pointers and Memory, CppCon, 2020

СЕКРЕТНЫЙ УРОВЕНЬ

Качество инкапсуляции

Качество инкапсуляции

- Обычно рекомендуется по умолчанию делать все поля закрытыми и писать для них геттеры и (при необходимости) сеттеры

```
class AlmostOpen {
  int x;
public:
  int get_x() const { return x; }
  void set_x(int xval) { x = xval; }
};
```

- Иногда это вызывает вопросы не много ли лишнего мы тут печатаем

Обсуждение

- Лучше ли инкапсулирован x в первом фрагменте кода, чем во втором?

```
class AlmostOpen { // 1
  int x;
public:
  int get_x() const { return x; }
  void set_x(int xval) { x = xval; }
};

struct Open { // 2
  int x;
};
```

Обсуждение

- Лучше ли инкапсулирован x в первом фрагменте кода, чем во втором?
- Как ни странно ответ да. Он не может утечь по косвенности

```
class AlmostOpen { // AlmostOpen a; int *px = &a.x; // ERROR
  int x;
public:
  int get_x() const { return x; }
  void set_x(int xval) { x = xval; }
};

struct Open { // Open o; int *px = &o.x; // OK
  int x;
};
```

Обсуждение

- Лучше ли инкапсулирован x в первом фрагменте кода, чем во втором?

```
class AlsoOpen { // 1
  int x;
public:
  int& access_x() { return x; }
};

struct Open { // 2
  int x;
};
```

Обсуждение

- Лучше ли инкапсулирован x в первом фрагменте кода, чем во втором?
- Не лучше, но мы тем не менее закладываем на будущее возможности

```
class AlsoOpen { // 1
  int x;
public:
  int& access_x() { return x; }
};

struct Open { // 2
  int x;
};
```

СЕКРЕТНЫЙ УРОВЕНЬ

Двухфазная инициализация глобальных переменных

Глобальные переменные

- Переменная в global scope имеет static storage duration и две инициализации

```
int a; // statically initialized to 0

int b = 1; // statically initialized to 1

int c = a + b; // statically initialized to 0
               // then dynamically initialized to a + b
```

- Порядок динамической инициализации
  - Внутри модуля – строго сверху вниз.
  - Между модулями не определён (unspecified).

initorder