ЛЕКЦИЯ 01
RAII

АЛГОРИТМИЗАЦИЯ И
ПРОГРАММИРОВАНИЕ

ЛЕКТОР ФУРМАВНИН С.А.

ПОНЯТИЕ RAII

RAII - Resource Acquisition Is Initialization

ПОНЯТИЕ RAII

RAII - Resource Acquisition Is Initialization

Что думаете об этом? Как бы вы определили что это такое?

ПОНЯТИЕ RAII

RAII - Resource Acquisition Is Initialization

Владение ресурсом – это его выделение и освобождение.

ПОНЯТИЕ RAII

RAII - Resource Acquisition Is Initialization

Владение ресурсом – это его выделение и освобождение.

Использование ≠ Владение

ПРОБЛЕМАТИКА

```
template <typename T>
T foo(T* p);

int main() {
    int *n = new int(5);
    foo<int>(n);
    delete n;
    return 0;
}
```

Кто владеет 5?

ПРОБЛЕМАТИКА

```
template <typename T>
T foo(T* p); //delete p;

int main() {
    int *n = new int(5);
    foo<int>(n);
    delete n;
    return 0;
}
```

Кто владеет 5?

РЕШАЕМ ПРОБЛЕМУ

Какие предложения у вас?

РЕШАЕМ ПРОБЛЕМУ

```
template <typename T>
class PRD_pointer final {
private:
    T* p;
public:
    PRD_pointer(T* p) : p(p) {}
    ~PRD_pointer() {delete p;};
};
```

РЕШАЕМ ПРОБЛЕМУ

```
template <typename T>
class PRD_pointer final {
private:
    T* p;
public:
    PRD_pointer(T* p) : p(p) {}
    ~PRD_pointer() {delete p;};
};
```

Всё ли тут хорошо?

РЕШАЕМ ПРОБЛЕМУ

```
template <typename T>
class PRD_pointer final {
private:
    T* p;
public:
    ??? operator*(???) ??? {return ???;}
    PRD_pointer(T* p) : p(p) {}
    ~PRD_pointer() {delete p;};
};
```

РЕШАЕМ ПРОБЛЕМУ

```
template <typename T>
class PRD_pointer final {
private:
    T* p;
public:
    T& operator*() ??? {return *p;}
    PRD_pointer(T* p) : p(p) {}
    ~PRD_pointer() {delete p;};
};
```

РЕШАЕМ ПРОБЛЕМУ

```
template <typename T>
class PRD_pointer final {
private:
    T* p;
public:
    T& operator*() const noexcept {return *p;}
    PRD_pointer(T* p) : p(p) {}
    ~PRD_pointer() {delete p;};
};
```

РЕШАЕМ ПРОБЛЕМУ

```
template <typename T>
class PRD_pointer final {
private:
    T* p;
public:
    T& operator*() const noexcept {return *p;}
    ??? operator->(???) ??? {return ???;}
    PRD_pointer(T* p) : p(p) {}
    ~PRD_pointer() {delete p;};
};
```

РЕШАЕМ ПРОБЛЕМУ

```
template <typename T>
class PRD_pointer final {
private:
    T* p;
public:
    T& operator*() const noexcept {return *p;}
    T* operator->() ??? {return p;}
    PRD_pointer(T* p) : p(p) {}
    ~PRD_pointer() {delete p;};
};
```

РЕШАЕМ ПРОБЛЕМУ

```
template <typename T>
class PRD_pointer final {
private:
    T* p;
public:
    T& operator*() const noexcept {return *p;}
    T* operator->() const noexcept {return p;}
    PRD_pointer(T* p) : p(p) {}
    ~PRD_pointer() {delete p;};
};
```

РЕШАЕМ ПРОБЛЕМУ : DRILL DOWN

```
template <typename T>
class PRD_pointer final {
…
    T* operator->() const noexcept {return p;}
…
};
```

- Вызов `p->x` эквивалентен `(p.operator->())->x` и так сколько угодно раз
- Стрелочка «зарывается» в глубину на столько уровней на сколько может

ОБСУЖДЕНИЕ

Хорош ли наш `PRD_pointer`? Подумайте о следующем:

```
S *a = new S{1}; S *b = new S{2};
std::swap(a, b); //что тут происходит?

PRD_pointer<S> x{new S{1}}, y{new S{2}};
std::swap(x, y); //А тут что происходит?
```

Hint: вспомните, что делает `std::swap()`

ПОСМОТРИМ НА ПАМЯТЬ

ПОСМОТРИМ НА ПАМЯТЬ

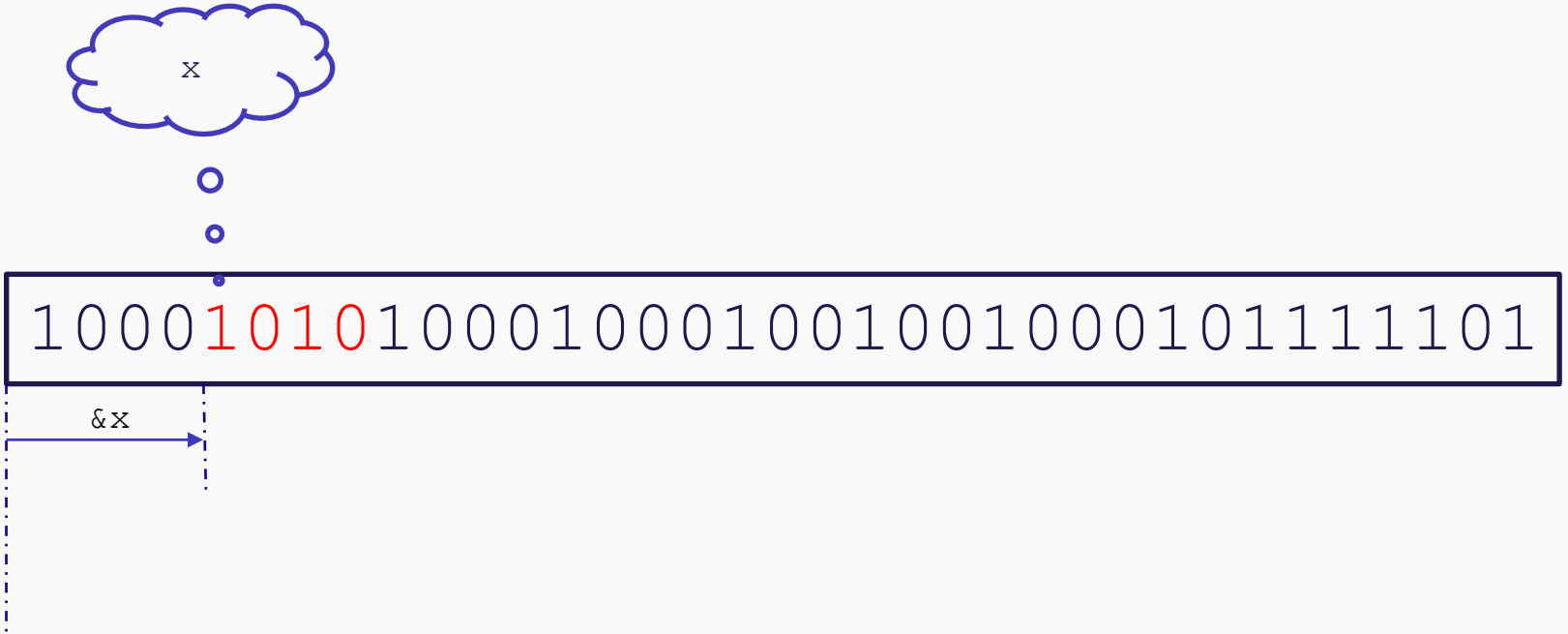
```
10001010100010001001001000101111101
```

ПОСМОТРИМ НА ПАМЯТЬ

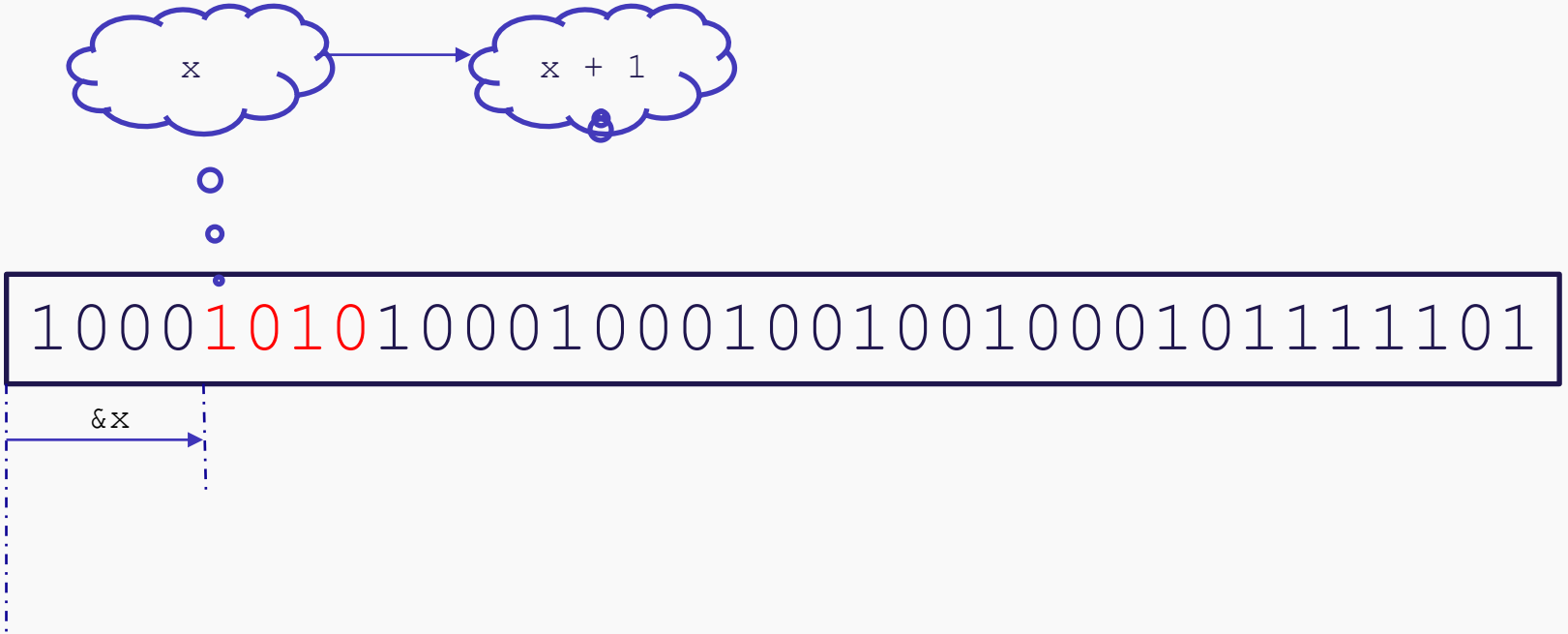
10001010100010001001001000101111101

ПОСМОТРИМ НА ПАМЯТЬ

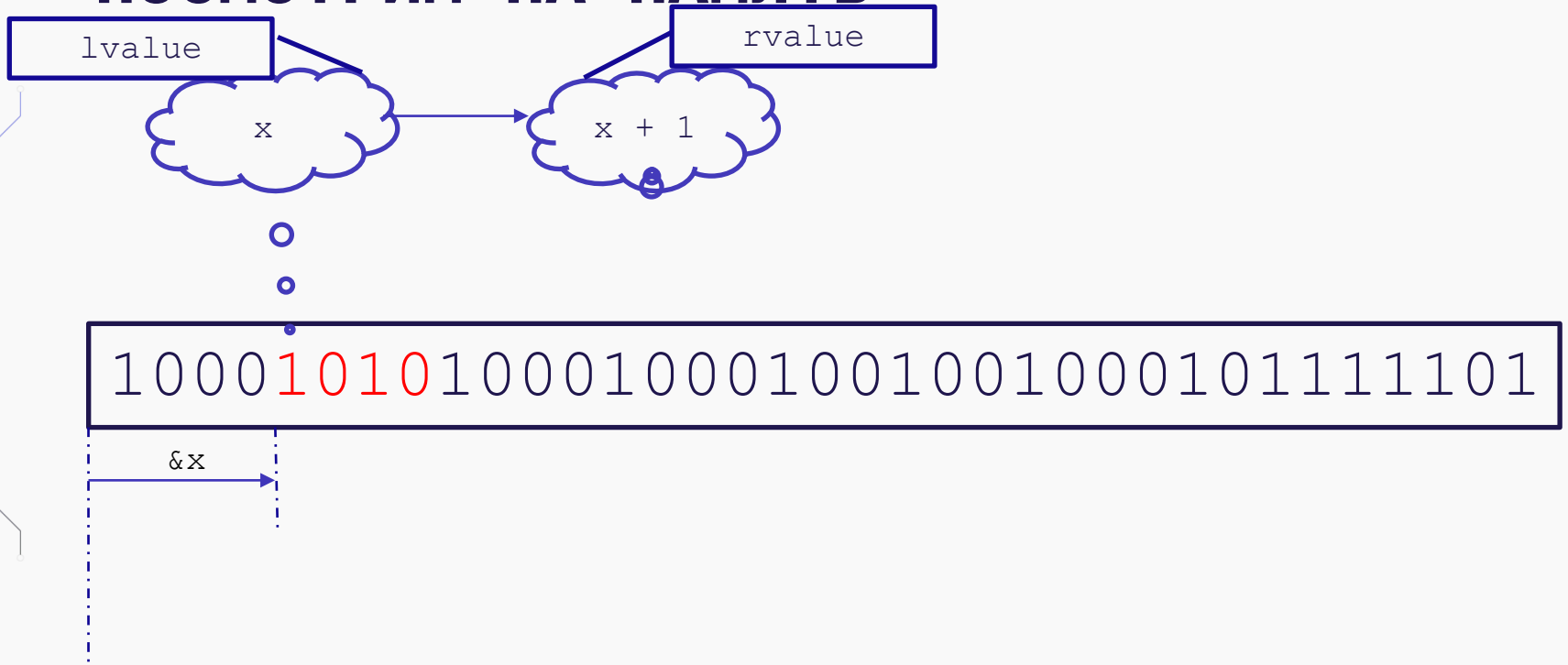
ПОСМОТРИМ НА ПАМЯТЬ



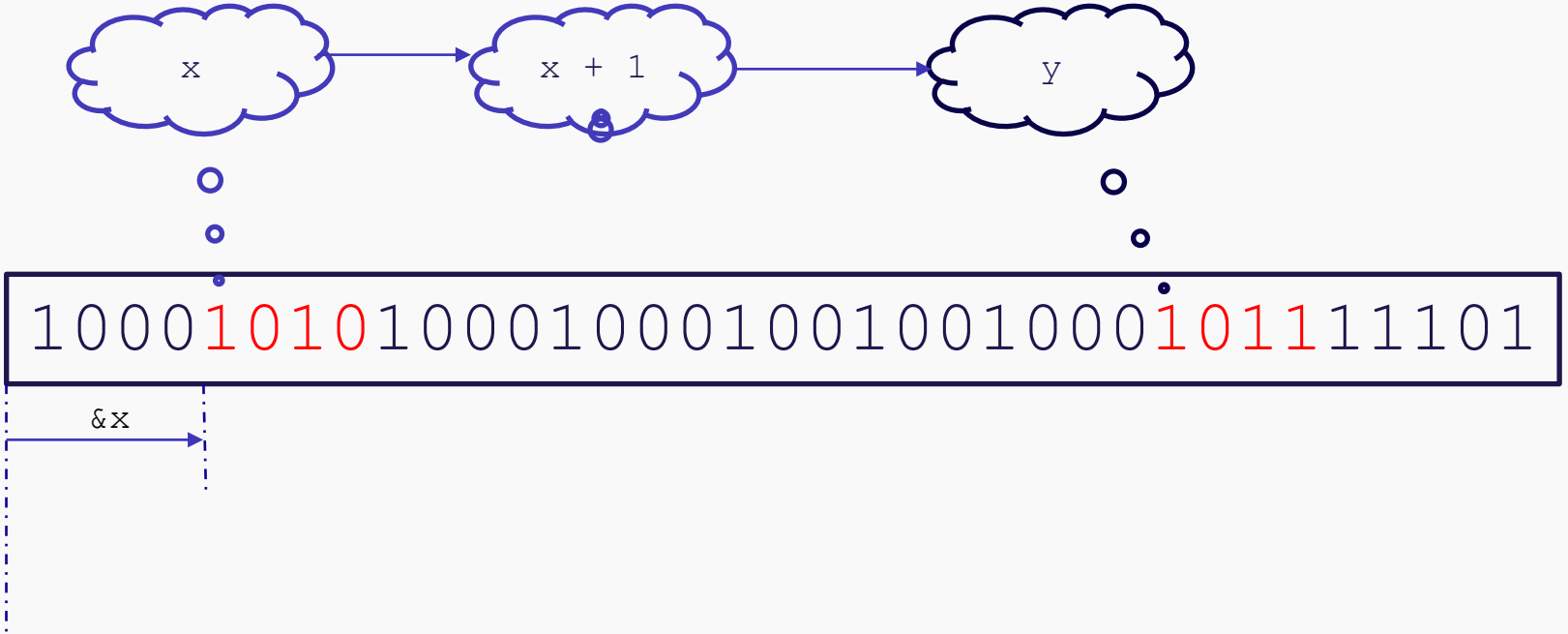
ПОСМОТРИМ НА ПАМЯТЬ



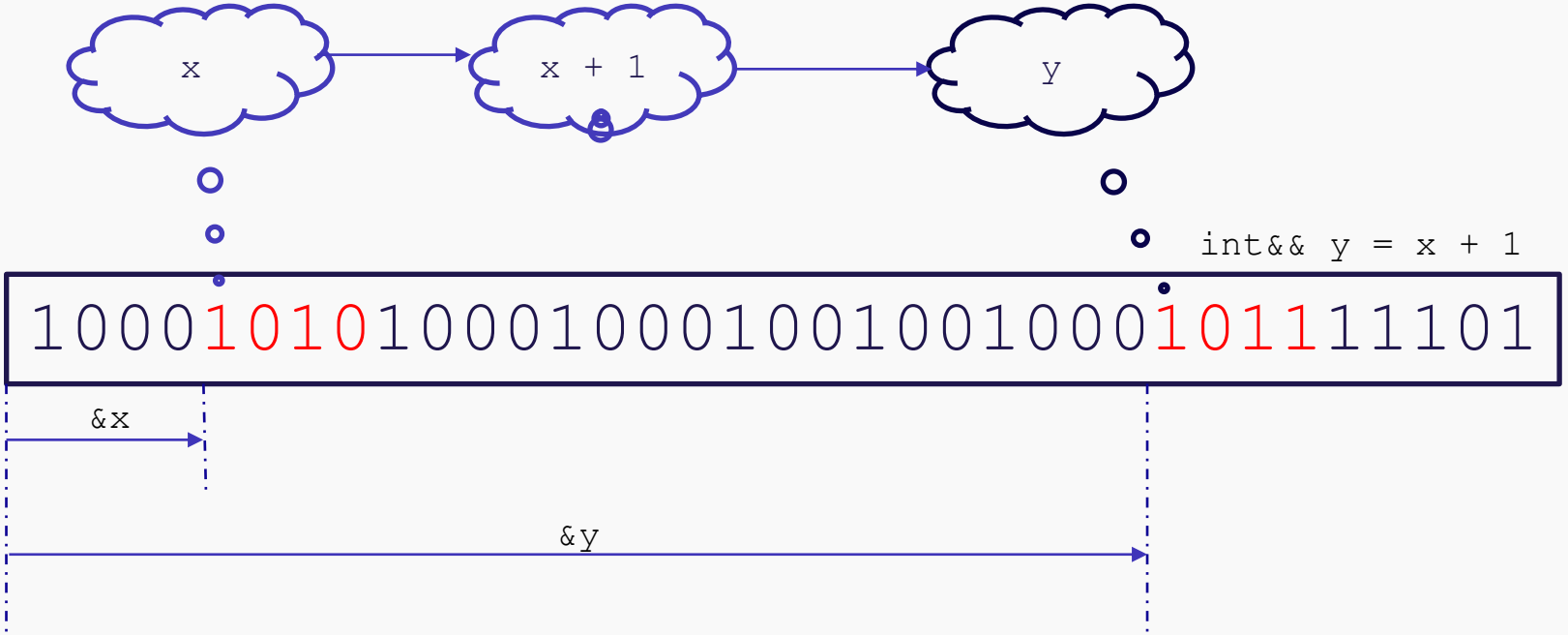
ПОСМОТРИМ НА ПАМЯТЬ



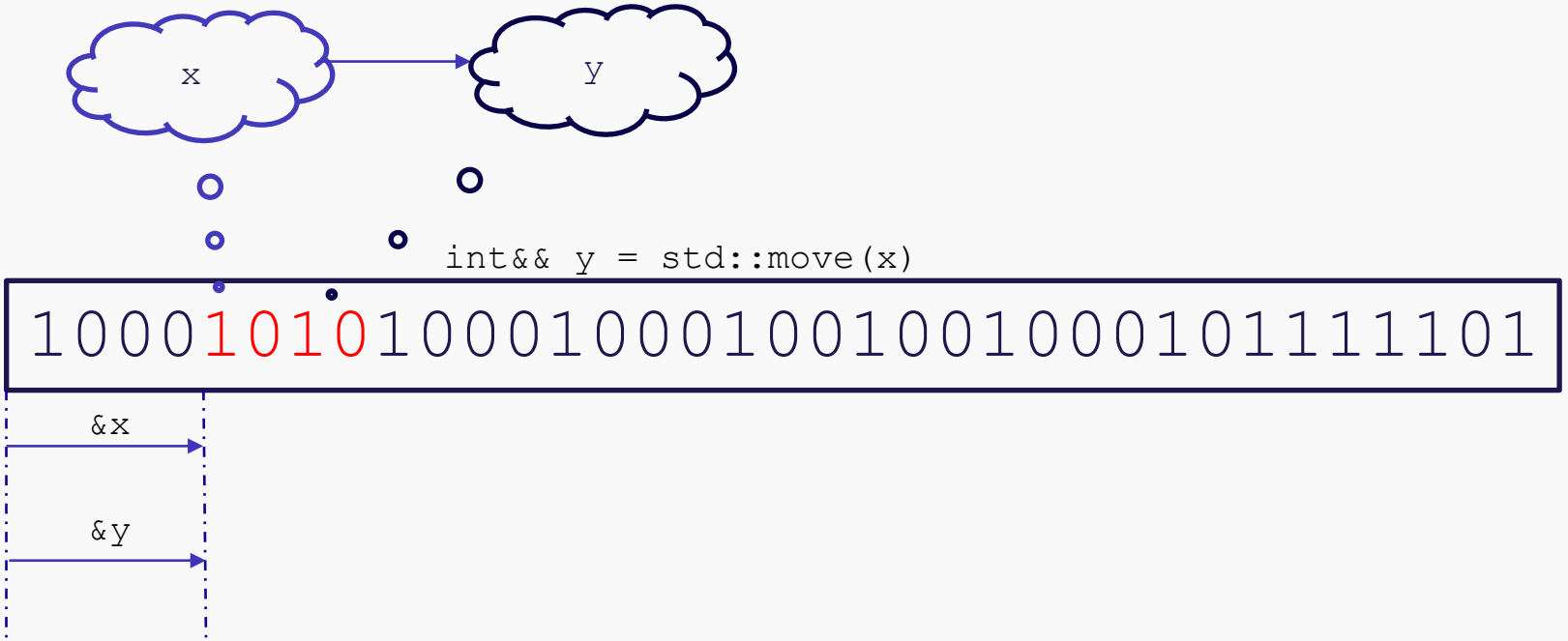
ПОСМОТРИМ НА ПАМЯТЬ



ПОСМОТРИМ НА ПАМЯТЬ



ПОСМОТРИМ НА ПАМЯТЬ



КРОСС-СВЯЗЫВАНИЕ

- `rvalue ref` не может быть связана с `lvalue`

```
int x = 1;
int &&y = x + 1;      // ok
int &&b = x;          // fail, not rvalue
```

КРОСС-СВЯЗЫВАНИЕ

- rvalue ref не может быть связана с lvalue

```
int x = 1;
int &&y = x + 1;     // ok
int &&b = x;         // fail, not rvalue
```

- non-const lvalue ref не может быть связана с rvalue

```
int &c = x + 1;      // fail, not lvalue
const int &d = x + 1; // ok, lifetime continue
```

КРОСС-СВЯЗЫВАНИЕ

- rvalue ref не может быть связана с lvalue

```
int x = 1;
int &&y = x + 1;     // ok
int &&b = x;         // fail, not rvalue
```

- non-const lvalue ref не может быть связана с rvalue

```
int &c = x + 1;      // fail, not lvalue
const int &d = x + 1; // ok, lifetime continue
```

- но при этом rvalue ref задает имя и адрес и является lvalue expression

```
int &&e = y;          // fail, not rvalue
int &f = y;           // ok
```

ОБСУЖДЕНИЕ: МЕТОДЫ ДЛЯ RVALUE

Помните, что метод может быть вызван для `rvalue exrp`

```
struct S {
    int x = 0;
    int& access() noexcept {return x;}
};

S x;
int &y = x.access(); // ok
int &z = S{}.access();  // что думаете?
```

ОБСУЖДЕНИЕ: МЕТОДЫ ДЛЯ RVALUE

Помните, что метод может быть вызван для `rvalue exrp`

```
struct S {
    int x = 0;
    int& access() noexcept {return x;}
};

S x;
int &y = x.access(); // ok
int &z = S{}.access();  // что думаете?
std::cout << z; // что мы увидим?
```

ОБСУЖДЕНИЕ: МЕТОДЫ ДЛЯ RVALUE

Помните, что метод может быть вызван для `rvalue exrp`

```
struct S {
    int x = 0;
    int& access() noexcept {return x;}
};

S x;
int &y = x.access(); // ok
int &z = S{}.access();  // ссылка на мертвый объект
std::cout << z; // что мы увидим?
```

ОБСУЖДЕНИЕ: МЕТОДЫ ДЛЯ RVALUE

Помните, что метод может быть вызван для `rvalue exrp`

```
struct S {
    int x = 0;
    int& access() noexcept {return x;}
};

S x;
int &y = x.access(); // ok
int &z = S{}.access();  // ссылка на мертвый объект
std::cout << z; // UB!!!
```

АННОТАЦИЯ МЕТОДОВ

Методы могут быть аннотированы и перегружены для `rvalue` и `lvalue expressions`

```
struct S {
    int foo() & {return 1;} // 1
    int foo() && {return 2;} // 2
};
extern S bar ();
S s {};
s.foo();   // 1
bar().foo(); // 2
```

АННОТАЦИЯ МЕТОДОВ

Теперь можно делать крутые штуки!

```
struct S {
    int x = 0;
    int& access() & noexcept {return x;}
};

S x;
int &y = x.access(); // ok
int &z = S{}.access();  // CE
```

ВОЗВРАТ ПРАВЫХ ССЫЛОК

- Возврат правых ссылок в большинстве случаев это плохо

```
int& foo(int &x) {return x;} // ok
const int& foo(const int &x) {return x;} // когда как
int&& buz(int&& x) {return std::move(x);} // DANGLE
```

- Вы не хотите возвращать `rvalue ref`, если у вас не `&&`-аннотированный метод

- При этом

```
int& bat(int &&x) {return x;} // когда как
```

- `rvalue ref` с точки зрения провисания гораздо опаснее `lvalue ref`

ДОРАБОТАЕМ НАШ УКАЗАТЕЛЬ

```
template <typename T>
class PRD_pointer final {
…
public:
      PRD_pointer(const PRD_pointer& rhs) :
            p(new T{*rhs.p}) {}
};
```

ДОРАБОТАЕМ НАШ УКАЗАТЕЛЬ

```
template <typename T>
class PRD_pointer final {
…
public:
      PRD_pointer(const PRD_pointer& rhs) :
            p(new T{*rhs.p}) {}
      PRD_pointer(PRD_pointer&& rhs) : p(rhs.p) {
            rhs.p = nullptr;}
};
```

ДОБАВИМ ПЕРЕМЕЩАЮЩЕЕ ПРИСВАИВАНИЕ

```
PRD_pointer& operator=(PRD_pointer&& rhs) {
      if (this == &rhs) return *this;
      // что напишете?
}
```

ДОБАВИМ ПЕРЕМЕЩАЮЩЕЕ ПРИСВАИВАНИЕ

```
PRD_pointer& operator=(PRD_pointer&& rhs) {
      if (this == &rhs) return *this;
// Вариант #1. Оставить пустое состояние
      delete p;
      p = rhs.p;
      rhs.p = nullptr;
      return *this;
}
```

Перемещающее присваивание ОБЯЗАНО оставить объект в консистентном состоянии

ДОБАВИМ ПЕРЕМЕЩАЮЩЕЕ ПРИСВАИВАНИЕ

```
PRD_pointer& operator=(PRD_pointer&& rhs) {
      if (this == &rhs) return *this;
// Вариант #1. Оставить пустое состояние
      delete p;
      p = rhs.p;
      rhs.p = nullptr;
      return *this;
}
```

Предложите вариант получше =)

ДОБАВИМ ПЕРЕМЕЩАЮЩЕЕ ПРИСВАИВАНИЕ

```
PRD_pointer& operator=(PRD_pointer&& rhs) {
      if (this == &rhs) return *this;
// Вариант #2. поменяем указатель, а деструктор удалит
      std::swap(p, rhs.p);
      return *this;
}
```

Консистентное состояние вообще-то не обязано быть предсказуемым

ПРОВЕРИМ ПОНИМАНИЕ

```
int x = 1;
int a = std::move(x);
assert (x == a); // ???
```

ПРОВЕРИМ ПОНИМАНИЕ

```
int x = 1;
int a = std::move(x);
assert (x == a); // ???

PRD_pointer y{new int(42)};
PRD_pointer b = std::move(y);
assert (y == b); // ???
```

ПРОВЕРИМ ПОНИМАНИЕ

```
int x = 1;
int a = std::move(x);
assert (x == a); // всегда верно

PRD_pointer y{new int(42)};
PRD_pointer b = std::move(y);
assert (y == b); // неизвестно
```

ДОСТАТОЧНО ЛИ ХОРОШ НАШ УКАЗАТЕЛЬ

```
PRD_pointer y{new int(42)};
while(…){
      PRD_pointer b = y;
      // do something
}
*y  = 40; // ???
```

ДОСТАТОЧНО ЛИ ХОРОШ НАШ УКАЗАТЕЛЬ

```
PRD_pointer y{new int(42)};
while(…){
      PRD_pointer b = y;
      // do something
}
*y  = 40; // ok, but…
```

УМНЫЕ УКАЗАТЕЛИ

Мы только что изобрели концепцию умных указателей

УМНЫЕ УКАЗАТЕЛИ

Доработаем наш указатель и сделаем его таким, чтобы только он мог владеть объектом и обеспечивал его уникальность. Что для этого нужно?

УМНЫЕ УКАЗАТЕЛИ

Доработаем наш указатель и сделаем его таким, чтобы только он мог владеть объектом и обеспечивал его уникальность. Что для этого нужно?

1. Запретить конструктор копирования

УМНЫЕ УКАЗАТЕЛИ

Доработаем наш указатель и сделаем его таким, чтобы только он мог владеть объектом и обеспечивал его уникальность. Что для этого нужно?

1. Запретить конструктор копирования
2. Запретить копирующее присваивание

УМНЫЕ УКАЗАТЕЛИ

Доработаем наш указатель и сделаем его таким, чтобы только он мог владеть объектом и обеспечивал его уникальность. Что для этого нужно?

1. Запретить конструктор копирования
2. Запретить копирующее присваивание

Отлично! Теперь наш указатель обеспечивает уникальность объекта и единоличное владение им

ДОМАШНЕЕ ЗАДАНИЕ

Написать свой unique_pointer, шаблонизированный двумя параметрами и помимо оговоренного на лекции определить следующие методы:

1. `operator bool` – приводит пустой указатель к `false`, а не пустой – к `true`
2. `get` – возвращает указатель на управляемый объект
3. `swap` – обмен управляемыми объектами между указателями

Это минимум на оценку УДОВЛЕТВОРИТЕЛЬНО

ДОМАШНЕЕ ЗАДАНИЕ

Написать свой unique_pointer, шаблонизированный двумя параметрами и помимо оговоренного на лекции определить следующие методы:

4. `reset` – заменяет объект, которым владеет
5. `release` – возвращает указатель на управляемый объект и освобождает его от своего владения
6. операторы сравнения
7. `get_delete` – возвращает `deleter`

Это минимум на оценку ХОРОШО

ДОМАШНЕЕ ЗАДАНИЕ

Написать свой unique_pointer, шаблонизированный двумя параметрами и помимо оговоренного на лекции определить следующие методы:

8. Написать частичную специализацию для `T[]`
9. `operator[]` для частичной специализации – обращение к элементу массива по индексу

Это минимум на оценку ОТЛИЧНО

ВНЕЗАПНАЯ ПРОБЛЕМА

```
int foo(int *a, int *b);
…
foo(new int(10), new int(20));
// что может пойти не так?
```

ВНЕЗАПНАЯ ПРОБЛЕМА

```
int foo(PRD_pointer<int> a, PRD_pointer<int> b);
…
foo(PRD_pointer<int>{new int(10)},
    PRD_pointer<int>{new int(20)});
// стало лучше?
```

ВНЕЗАПНАЯ ПРОБЛЕМА

```
int foo(PRD_pointer<int> a, PRD_pointer<int> b);
…
foo(PRD_pointer<int>{new int(10)},
    PRD_pointer<int>{new int(20)});
// стало лучше?
```

Как исправить?

ВНЕЗАПНАЯ ПРОБЛЕМА

```
int foo(std::unique_ptr<int> a,
        std::unique_ptr<int> b);
…
foo(std::unique_ptr<int>{new int(10)},
    std::unique_ptr<int>{new int(20)});
// А так стало лучше?
```

РЕШЕНИЕ ПРОБЛЕМЫ

Завернем создание умного указателя в отдельную функцию

РЕШЕНИЕ ПРОБЛЕМЫ

Завернем создание умного указателя в отдельную функцию

```
template <typename T>
PRD_pointer<T> make_unique(???) {
      // что тут написать?
}
```

РЕШЕНИЕ ПРОБЛЕМЫ

Завернем создание умного указателя в отдельную функцию

```
template <typename T, typename… Args>
PRD_pointer<T> make_unique(Args&&… args) {
      // что тут написать?
}
```

РЕШЕНИЕ ПРОБЛЕМЫ

Завернем создание умного указателя в отдельную функцию

```
template <typename T, typename… Args>
PRD_pointer<T> make_unique(Args&&… args) {
      return PRD_pointer<T>
      (new T(std::forward<Args>(args)...));
}
```

ВРЕМЯ ДЛЯ ВАШИХ ВОПРОСОВ

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Бьерн Страуструп, Язык программирования C++/ ред. А. Боборыкин. – 4-е изд. - Москва: Издательство БИНОМ, 2023. – 1213 с.
2. Скотт Мейерс, Эффективное использование C++. 55 верных способов улучшить структуру и код ваших программ / ред. Д.А. Мовчан – 3-е изд. – Москва: ДМК Пресс, 2017. – 300 с.
3. Скотт Мейерс, Наиболее эффективное использование C++. 35 новых рекомендаций по улучшению ваших программ и проектов / ред. Д.А. Мовчан – 3-е изд. – Москва: ДМК Пресс, 2016. – 298 с.