ЛЕКЦИЯ 14
RAII И ПЕРЕМЕЩЕНИЕ

АЛГОРИТМИЗАЦИЯ И ПРОГРАММИРОВАНИЕ

ЛЕКТОР ФУРМАВНИН С.А.

МНОГОМОДУЛЬНЫЕ ПРОГРАММЫ

Предположим, что вы написали очень интересную программу

```
// pow возводит n в степень x
unsigned long long pow(unsigned n, unsigned x) {
    // тут очень умная реализация
}

int main() {
    // тут основная программа, использующая pow
}
```

Она работает, но вы обнаружили, что функция `pow` может быть вам нужна и в других программах, т.е. может быть **переиспользована**

ВЫНОСИМ ФУНКЦИЮ В МОДУЛЬ

Вы можете сделать отдельный модуль с функцией pow
В модуле myprog вам нужно только определение


```
// myprog.cpp
unsigned long long pow(unsigned n, unsigned x);
int main() {
    // тут основная программа, использующая pow
}
```

И это работает. Все ли видять проблему такого подхода?

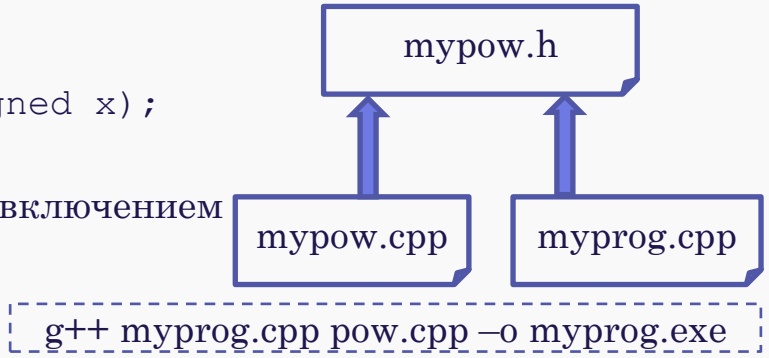
ЗАГОЛОВОЧНЫЕ ФАЙЛЫ

Выход – это составить заголовочный файл с определением и включить его и в файл с реализацией и в файл с использованием

```
// --- файл mypow.h ---
unsigned long long pow(unsigned n, unsigned x);
```

Внутри myprog.cpp мы воспользуемся текстовым включением

```
#include <iostream>
#include "mypow.h"
```



Вид скобок определяет способ поиска файлов: треугольные скобки – файл ищется по системным путям, кавычки – файл ищется от локальной директории

СТРАЖИ ВКЛЮЧЕНИЯ

Один заголовочный файл может быть включен в тысячи файлов в одном проекте
Чтобы избежать лишних включений, можно использовать прагму

```
// --- файл mypow.h ---
# pragma once
unsigned long long pow(unsigned n, unsigned x);
```

Позже, если успеем, поговорим о стражах включения подробнее

ВЛАДЕНИЕ РЕСУРСОМ

Памятью владеет тот, кто её выделяет и освобождает

```
S *p = new S;
foo(p); // foo(S*)
delete p;
```

Что может пойти не так в этом коде?

ВЛАДЕНИЕ РЕСУРСОМ

Памятью владеет тот, кто её выделяет и освобождает

```
S *p = new S;
foo(p); // foo(S* p) {delete p;}
delete p;
```

Что может пойти не так в этом коде?

В общем случае память это только один из возможных ресурсов

ПРИМЕР

```
int foo(int n) {
    S *p = new S{n};
    // ....some code....
    if (condition) {
        delete p;
        return FAILURE;
    }
    // ....some code....
    delete p;
    return SUCCESS;
}
```

Хотелось бы иметь одну точку освобождения, чтобы избежать проблем

ИСПОЛЬЗОВАНИЕ GOTO

```
int foo(int n) {
    S *p = new S{n};
    int result = SUCCESS;
    // ....some code....
    if (condition) {
        result = FAILURE;
        goto cond;
    }
    // ....some code....
cond:
    delete p;
    return result;
}
```

СОЦИАЛЬНО-ПРИЕМЛЕМОЕ GOTO

```
int foo(int n) {
    S *p = new S{n}; int result = SUCCESS;
    do {
        // ....some code....
        if (condition) {
            result = FAILURE;
            break;
        }
        // ....some code....
    } while (0);
    delete p;
    return result;
}
```

GOTO CONSIDERED HARMFUL

Что вы думаете об этом коде?

```
struct X {
    int smth = 42;
};

int foo(int cond) {
    switch(cond) {
        case 0: X x;
        case 1: return x.smth; // 42?
    }
}
```

GOTO CONSIDERED HARMFUL

Что вы думаете об этом коде?

```
struct X {
    int smth = 42;
};

int foo(int cond) {
    switch(cond) {
        case 0: X x;
        case 1: return x.smth; // CE!!
    }
}
```

К счастью это ошибка компиляции

ОБСУЖДЕНИЕ

Какие мы знаем goto-маскирующие конструкции?
`switch-case`, `break`, `continue`, `return`, ещё?

Будьте с ними КРАЙНЕ осторожны при работе с конструкторами и деструкторами. Ваш выбор – явные блоки.

```
int foo(int cond) {
    switch(cond) {
        case 0: { X x; }
        case 1: return x.smth; // очевидная ошибка
    }
}
```

RAII: RESOURCE ACQUISITION IS INITIALIZATION

Чтобы не писать goto, можно спроектировать класс, в котором конструктор захватывает владение, а деструктор освобождает ресурс.

```
int foo(int n) {
    ScopedPointer p {new S(n)}; // ownership passed
    // ....some code....
    if (condition)
        return FAILURE; // dtor called: delete
    // .... some code....
    return SUCCESS; // dtor called: delete
}
```

RAII-ОБЁРТКА

Как бы мог выглядеть упомянутый `ScopedPointer`?

```
class ScopedPointer {
    S *ptr_;
public:
    ScopedPointer(S *ptr = nullptr) : ptr_(ptr) {}
    ~ScopedPointer() { delete ptr_; }
};
```

И у нас две проблемы.

- Как написать копирование/присваивание?
- Как сделать с ним что-то полезное, не дав утечь указателю?

ГЛУБОКОЕ КОПИРОВАНИЕ

Начнем с копирования?

```
class ScopedPointer {
    S *ptr_;
public:
    ScopedPointer(S *ptr = nullptr) : ptr_(ptr) {}
    ~ScopedPointer() { delete ptr_; }
    ScopedPointer(const ScopedPointer& rhs) :
        ptr_(new S{*rhs.ptr_}) {}
    ScopedPointer&   operator=(const   ScopedPointer&
rhs);
};
```

ДОСТУП К СОСТОЯНИЮ

Можно сделать функцию access()

```
class ScopedPointer {
    S *ptr_;
public:
    ScopedPointer(S *ptr = nullptr) : ptr_(ptr) {}
    ~ScopedPointer() { delete ptr_; }
    S& access() {return *ptr_; }
    const S& access() const {return *ptr_; }
};
```

Итог немного многословен

```
ScopedPointer p{new S(n)}; int x = p.access().x; //
(*p).x
```

ПЕРЕГРУЗКА РАЗЫМЕНОВЫВАНИЯ

Разыменовывание указателя – это оператор и он перегружается

```
class ScopedPointer {
    S *ptr_;
public:
    ScopedPointer(S *ptr = nullptr) : ptr_(ptr) {}
    ~ScopedPointer() { delete ptr_; }
    S& operator*() {return *ptr_; }
    const S& operator*() const {return *ptr_; }
};
```

Уже сейчас стало гораздо лучше, хоть и не идеально

```
ScopedPointer p{new S(n)}; int x =(*p).x; // p->x
```

ПРОБЛЕМА СО СТРЕЛОЧКОЙ

Разыменовывание указателя – это оператор и он перегружается

```
class ScopedPointer {
    S *ptr_;
public:
    S& operator*() {return *ptr_; }
    const S& operator*() const {return *ptr_; }

    ??? operator->() {return ???; }
};
```

А что возвращать-то?

РЕШЕНИЕ: DRILL DOWN

Разыменовывание указателя – это оператор и он перегружается

```
class ScopedPointer {
    S *ptr_;
public:
    S& operator*() {return *ptr_; }
    const S& operator*() const {return *ptr_; }
    S* operator->() {return ptr_; }
    const S* operator->() const {return ptr_; }
};
```

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

ХОРОШ ЛИ НАШ SCOPEDPOINTER?

Подумайте вот о чём.

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

ScopedPointer x{new S(1)}, y{new S(2)};
std::swap(x, y); // а что тут?
```

Для справки: `std::swap` в C++98 был определен так:

```
template <typename T> void swap(T& x, T& y) {
    T tmp = x; // copy ctor
    x = y;          // assign
    y = tmp;    // assign
}
```

ХОРОШ ЛИ НАШ SCOPEDPOINTER?

Подумайте вот о чём.

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

ScopedPointer x{new S(1)}, y{new S(2)};
std::swap(x, y); // а что тут?
```

Для справки: `std::swap` в C++98 был определен так:

```
template <typename T> void swap(T& x, T& y) {
    T tmp = x; // copy ctor
    x = y;          // assign
    y = tmp;    // assign
}
```

ПОСМОТРИМ НА ПАМЯТЬ

ПОСМОТРИМ НА ПАМЯТЬ

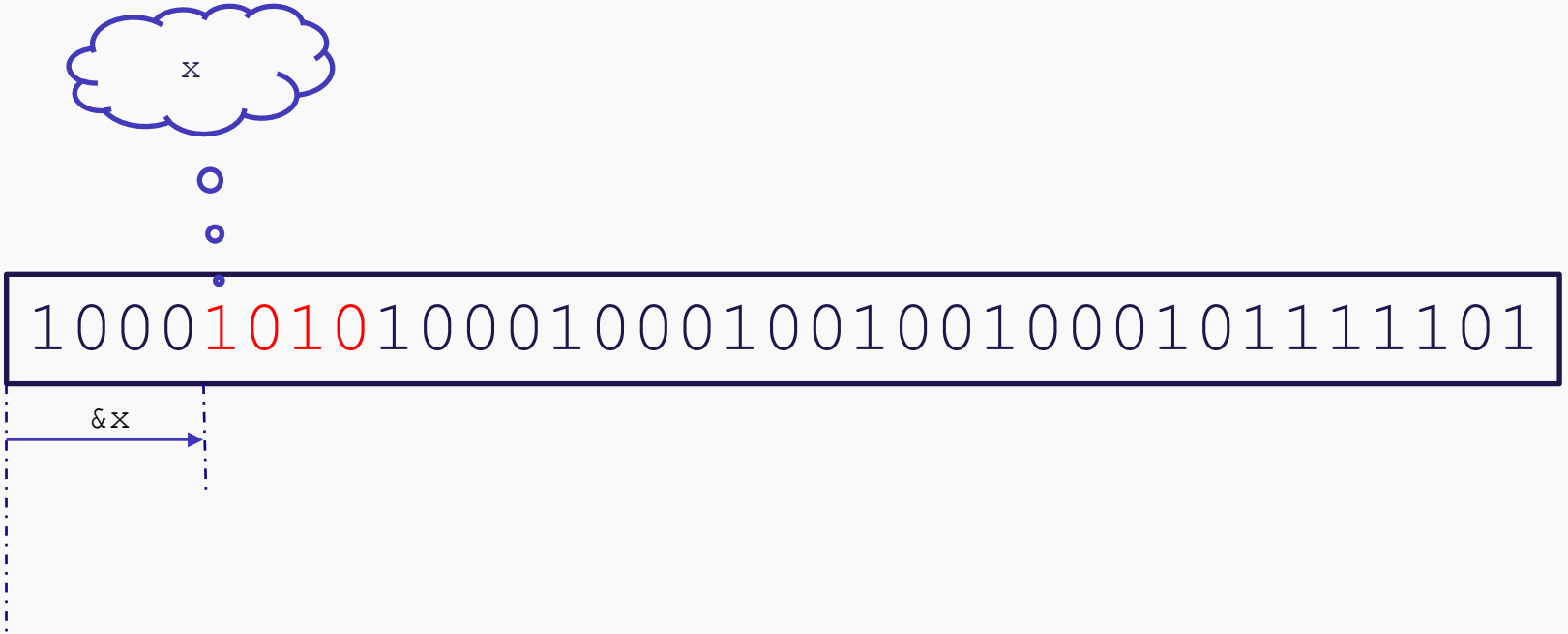
```
10001010100010001001001000101111101
```

ПОСМОТРИМ НА ПАМЯТЬ

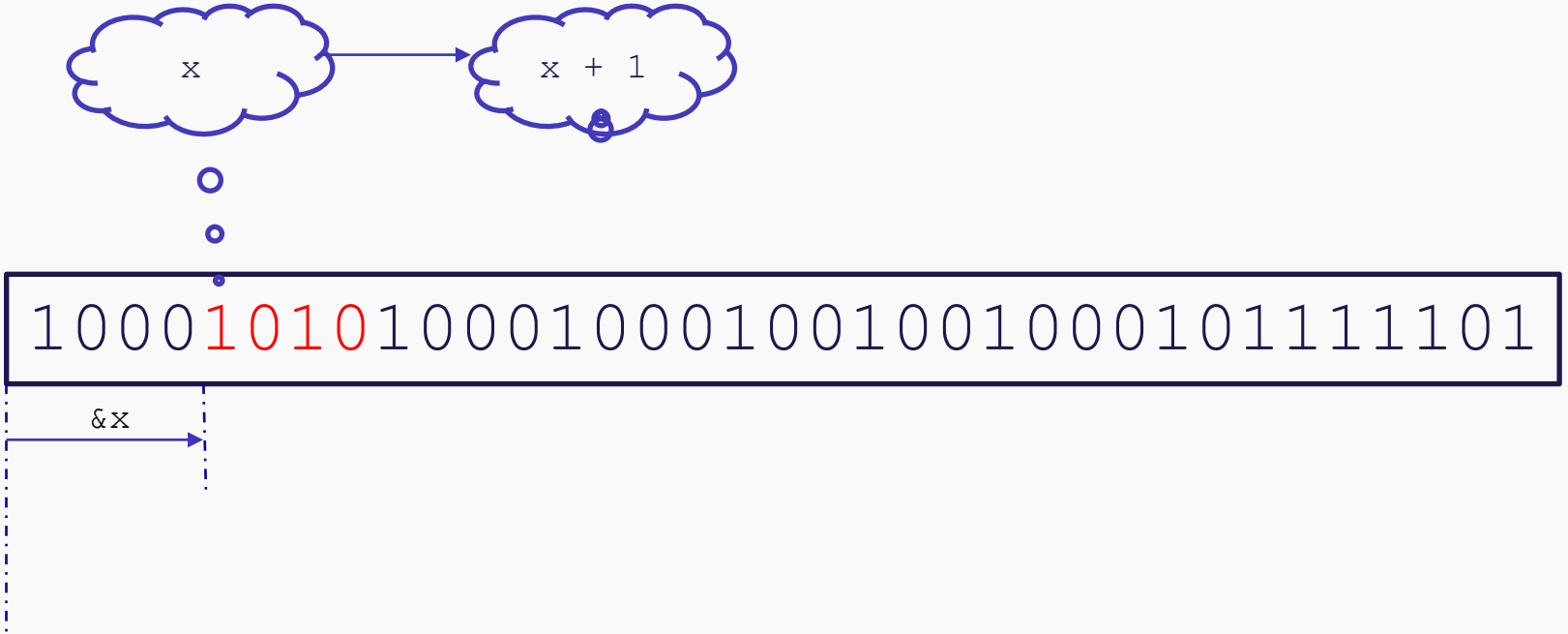
10001010100010001001001000101111101

ПОСМОТРИМ НА ПАМЯТЬ

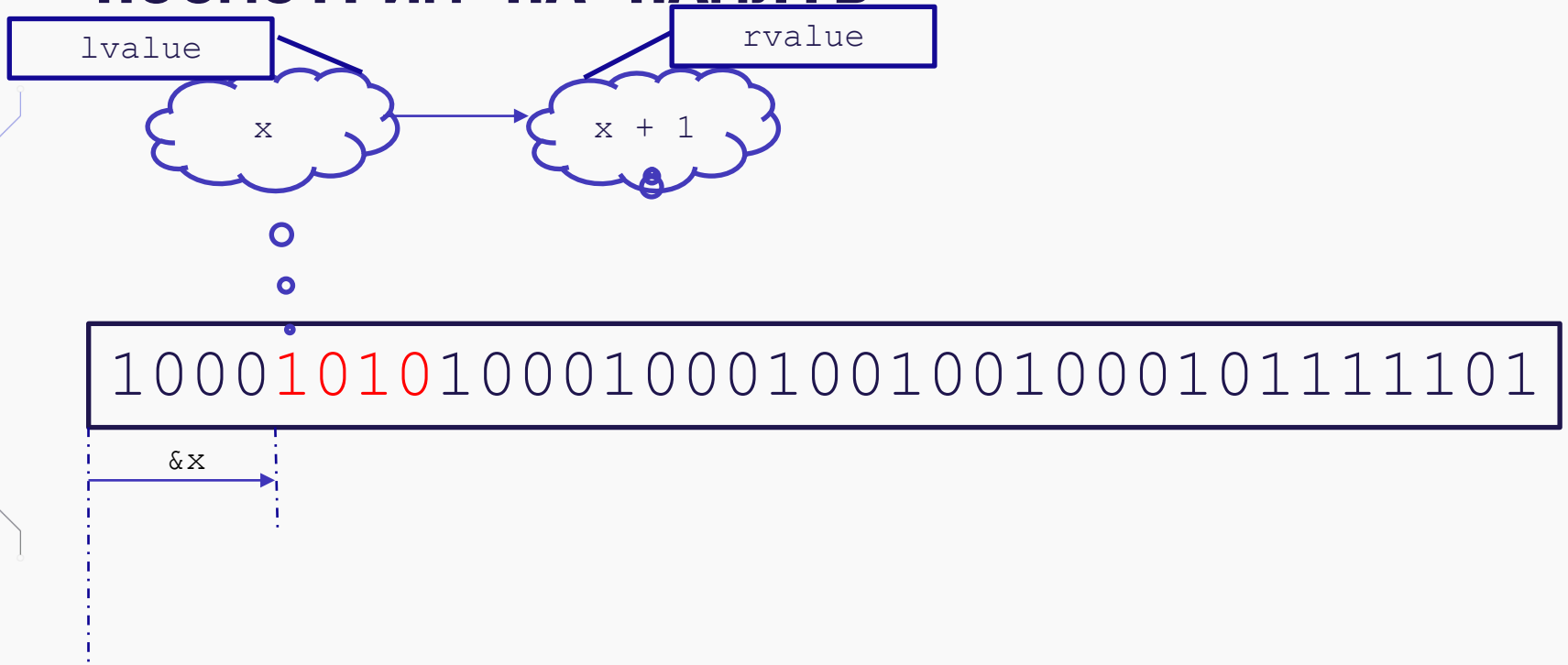
ПОСМОТРИМ НА ПАМЯТЬ



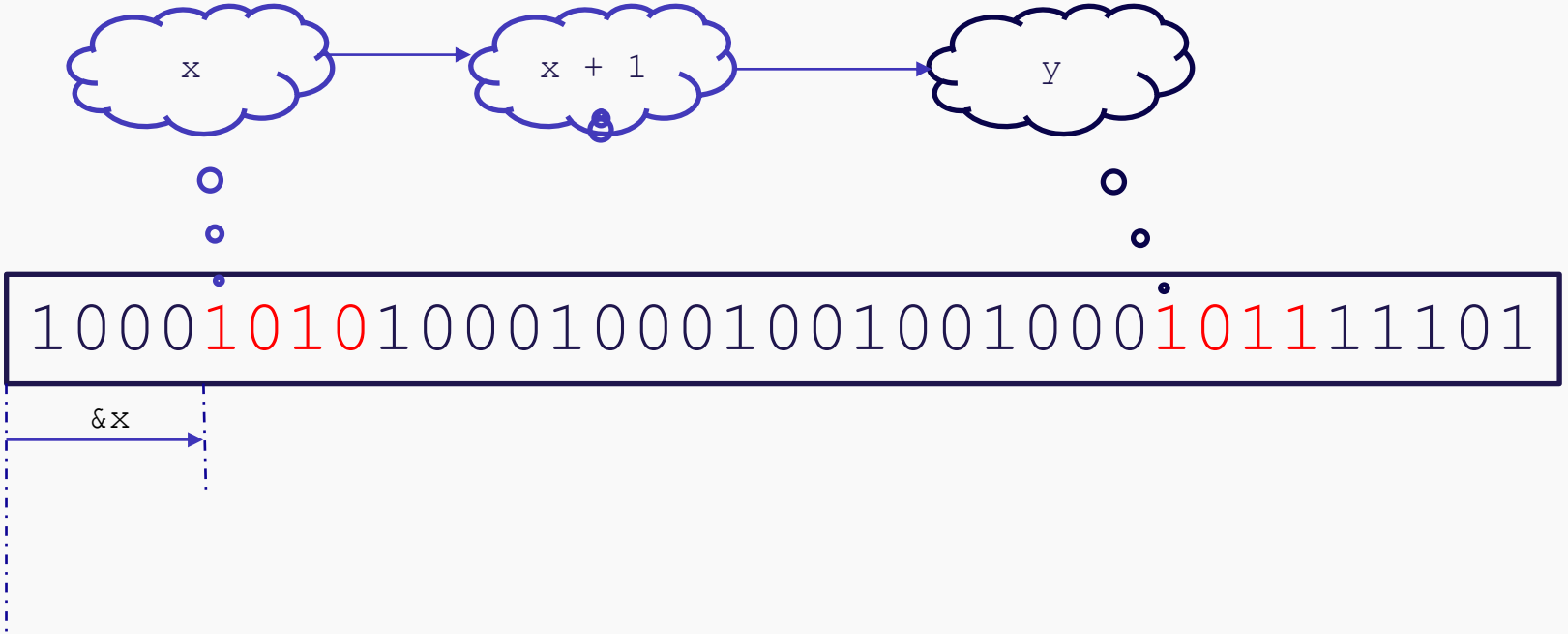
ПОСМОТРИМ НА ПАМЯТЬ



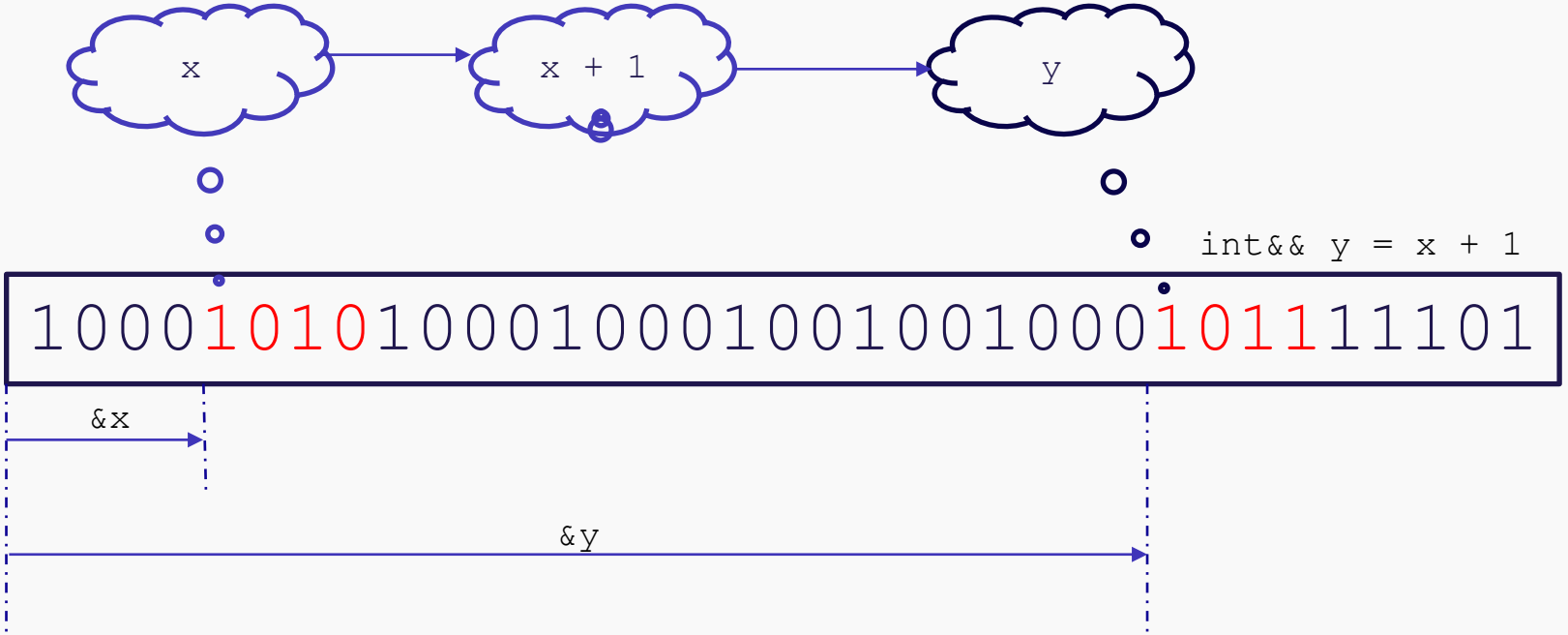
ПОСМОТРИМ НА ПАМЯТЬ



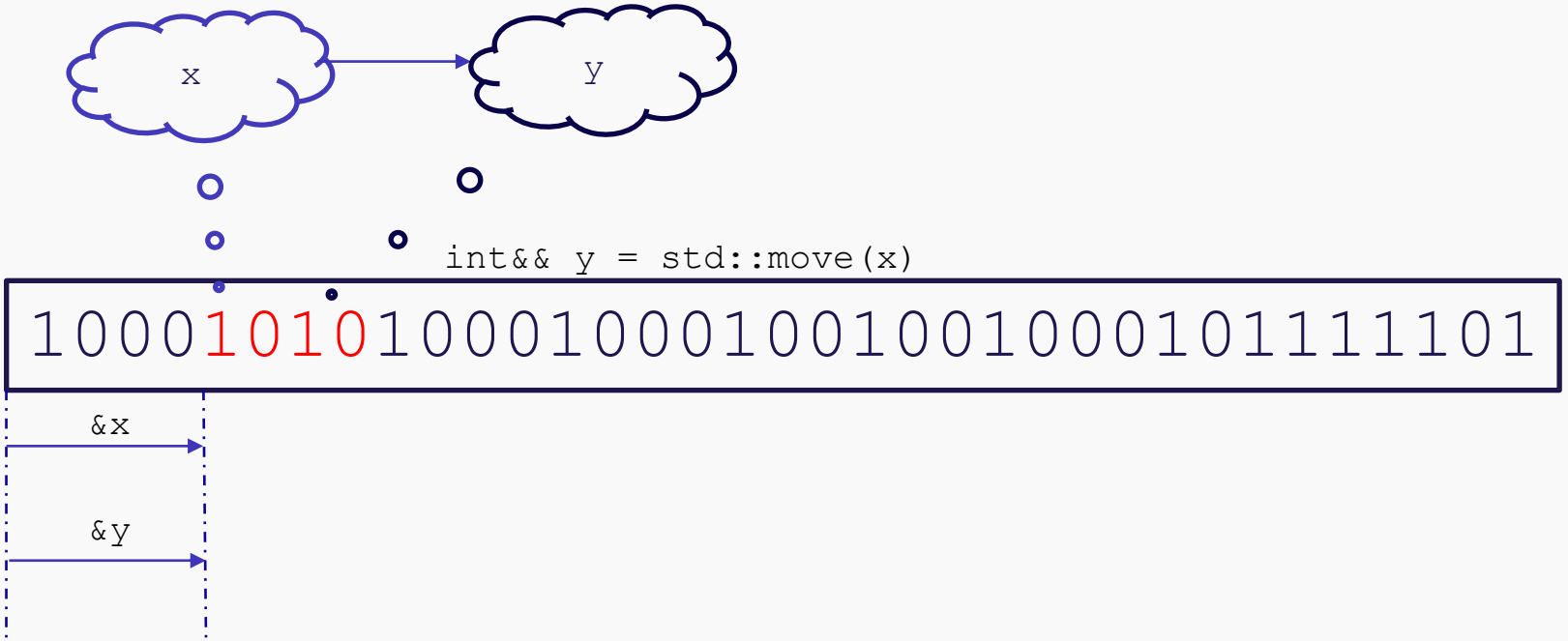
ПОСМОТРИМ НА ПАМЯТЬ



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() {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() {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() {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() {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() & {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`

ПЕРЕМЕЩАЮЩИЕ КОНСТРУКТОРЫ

- Конструктор, берущий rvalue ref не обязан сохранять значение (т.к. это rvalue)
- Это потрясающе выгодно там, где требуется глубокое копирование

```
class ScopedPointer {
    S *ptr_;
public:
    ScopedPointer(const ScopedPointer& rhs) :
        ptr_(new S{*rhs.ptr_}) {}
    ScopedPointer (ScopedPointer&& rhs) :
        ptr_(rhs.ptr_) {rhs.ptr_ = nullptr; }
};
```

ПЕРЕМЕЩАЮЩИЕ КОНСТРУКТОРЫ

- Конструктор, берущий rvalue ref не обязан сохранять значение (т.к. это rvalue)
- Это потрясающе выгодно там, где требуется глубокое копирование

```
class ScopedPointer {
    S *ptr_;
public:
    ScopedPointer(const ScopedPointer& rhs) :
        ptr_(new S{*rhs.ptr_}) {}
    ScopedPointer (ScopedPointer&& rhs) :
        ptr_(rhs.ptr_) {rhs.ptr_ = nullptr; }
};
```

ПЕРЕМЕЩАЮЩЕЕ ПРИСВАИВАНИЕ

- Для перемещающего присваивания есть варианты

```
ScopedPointer& operator=(ScopedPointer&& rhs) {
     if (this == &rhs) return *this;

     // вариант 1: оставим пустое состояние
     delete ptr_;
     ptr_ = rhs.ptr_;
     rhs.ptr_ = nullptr;
     return *this;
}
```

Оно обязано оставить объект в **консистентном** состоянии

КОРОТКО О КОНСИСТЕНТНОСТИ

- Объект находится в **консистентном** состоянии, если он может быть корректно удален и при этом:
  - Не будет утечек памяти
  - Не будет double delete
  - Ресурсы будут корректно освобождены

ПЕРЕМЕЩАЮЩЕЕ ПРИСВАИВАНИЕ

- Для перемещающего присваивания есть варианты

```
ScopedPointer& operator=(ScopedPointer&& rhs) {
     if (this == &rhs) return *this;

     // вариант 2: делаем обмен и пусть деструктор
удаляет
     std::swap(ptr_, rhs.ptr_);
     return *this;
}
```

Это состояние вообще **не обязано быть предсказуемым**.

АККУРАТНЕЕ С MOVE ON RESULT

Обычно в таком коде `std::move` просто не нужен

```
T foo(some args) {
    T x = some expression;
    // more code
    return std::move(x); // не ошибка, но зачем?
}
```

Функция, возвращающая by value это rvalue expression и таким образом всё равно делает move в точке вызова.

При этом использование `std::move` может сделать вещи несколько хуже, убив RVO

Ограничьте move on result случаями возврата ссылки

ЗАДАЧА: ОСОБЕННОСТИ MOVE

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

ScopedPointer y {new S(10)};
ScopedPointer b = std::move (y);
assert (y == b); // ???
```

Что можете сказать о приведенных assertions?

РЕШЕНИЕ: ОСОБЕННОСТИ MOVE

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

ScopedPointer y {new S(10)};
ScopedPointer b = std::move (y);
assert (y == b); // мы не знаем
```

- Использование move всего лишь получает &&, ничего не делая с переменной.
- Будет ли состояние потеряно, зависит от ого, есть ли у класса перемещающий конструктор и как он реализован
- У int его точно нет, на чем и построен первый ответ.

ПРОБЛЕМА IMPLICIT MOVE

Перемещение по умолчанию перемещает по умолчанию все поля

```
class SillyPointer {
    S *ptr_;
public:
    SillyPointer(S *ptr = nullptr) : ptr_(ptr) {}
    ~SillyPointer() { delete ptr_; }
};

template <typename T> void swap(T& lhs, T& rhs) {
    T tmp = std::move(lhs);
    lhs = std::move(rhs);
    rhs = std::move(tmp);
} // UB (segfault probably)
```

ПРАВИЛО ПЯТИ

Классическая идиома проектирования rule of five утверждает, что:

*Если ваш класс требует нетривиального определения **хотя бы одного из пяти методов**:*

- *копирующего конструктора;*
- *копирующего присваивания;*
- *перемещающего конструктора;*
- *перемещающего присваивания;*
- *деструктора,*

*то вам лучше бы нетривиально определить **все пять**.*

Очевидно SillyPointer его нарушает: он определяет нетривиальный деструктор и только его.

ПРАВИЛО НУЛЯ

Классическая идиома проектирования rule of zero утверждает, что:

Если ваш класс требует нетривиального определения ***хотя бы одного из пяти неявных методов****:*

- *копирующего конструктора;*
- *копирующего присваивания;*
- *перемещающего конструктора;*
- *перемещающего присваивания;*
- *деструктора,*

и, таким образом ***все пять,***

То в нем не должно быть никаких других методов.

Это правило много раз вас выручит и поэтому оно относится к моим любимым!

КРАЕВОЙ СЛУЧАЙ: MOVE FROM CONST

Хорошо организованный move ctor изменяет rhs. Но что если rhs нельзя изменять?

```
const Buffer y{new int(10)};
Buffer b = std::move(y); // копирование
```

- В этом случае move ctor просто не будет вызван, так как его сигнатура предполагает `Buffer&&`, а не `Buffer const &&`
- Вместо этого, `Buffer const &&` будет приведен к `Buffer const &` и вызовется копирующий конструктор, несмотря на явное указание move.

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Edsger W. Dijkstra – Go To Statement Considered Harmful, 1968
2. Edsger W. Dijkstra – The Humble Programmer, ACM Turing Lecture, 1972
3. Скотт Мейерс, Эффективный современный C++: 42 способа улучшить ваше использование C++11 и C++14
4. Klaus Iglberger – Back to Basics: Move Semantics, CppCon, 2019
5. RAII and Rule of Zero, Arthur O'Dwyer, CppCon, 2019
6. Nicolai Josuttis – The Nightmare of Move Semantics for Trivial Classes, 2017
7. Дональд Кнут, Искусство программирования. Том 1. Основные алгоритмы / Ю.В. Козаченко. - 3-е изд – Москва, Санкт-Петербург: ВИЛЬЯМС, 2018. – 721 с.
8. Дональд Кнут, Искусство программирования. Том 2. Получисленные алгоритмы / Ю.В. Козаченко. - 3-е изд – Москва, Санкт-Петербург: ВИЛЬЯМС, 2018. – 743 с.
9. Дональд Кнут, Искусство программирования. Том 3. Сортировка и поиск / Ю.В. Козаченко. - 3-е изд – Москва, Санкт-Петербург: ВИЛЬЯМС, 2018. – 767 с.