

# C++

Namespace e oggetti

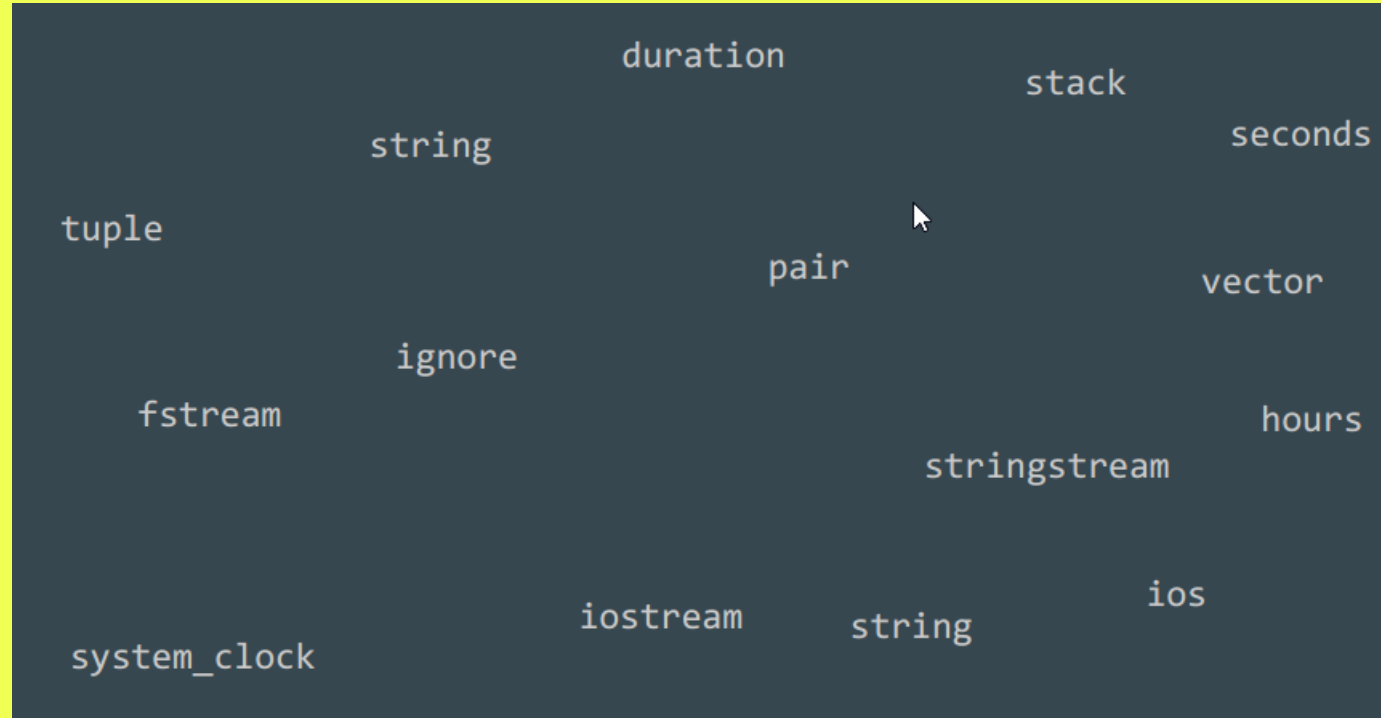
NAMESPACE

# NAMESPACE

- ▶ Negli esempi ho scritto linee di codice del tipo `std::cout` o `std::vector<int>`
- ▶ `std` è un esempio di namespace
- ▶ Usati per raggruppare in scope separati variabili, oggetti, funzioni in modo da prevenire conflitti di denominazione all'interno del progetto

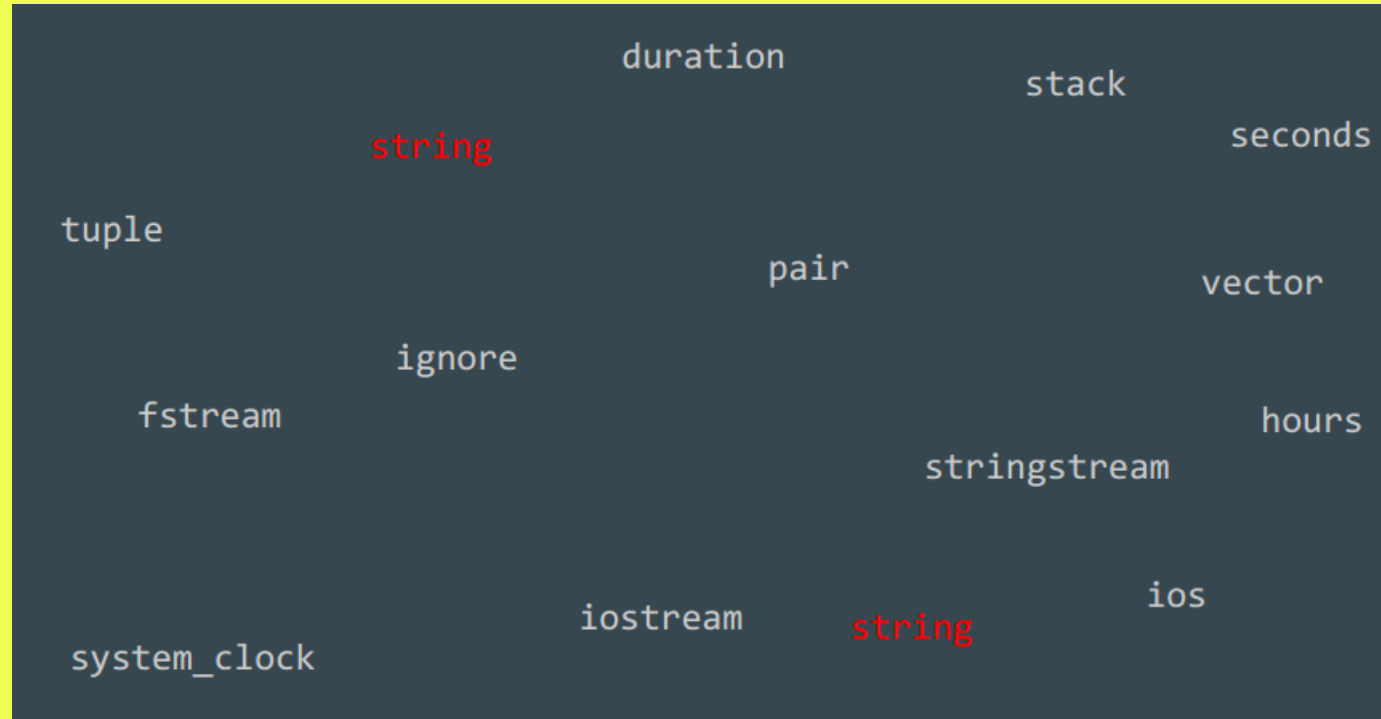
# NAMESPACE

► Es:




# NAMESPACE

► Es:



duration stack seconds  
string pair vector  
tuple ignore fstream stringstream hours  
system\_clock istream string ios

# NAMESPACE

- ▶ E' un problema se definisco delle entità che hanno lo stesso nome di quelle presenti, per es., nella standard library? E' un problema se le entità che definisco nella mia libreria hanno lo stesso nome di quelle presenti in altre librerie che ho importato nel progetto?
  - ▶ In generale, sì.
  - ▶ Molti linguaggi di programmazione risolvono questo problema con i namespace.
- 

# NAMESPACE

- ▶ `namespace identifier {  
 entities  
}`

- ▶ Questa è la definizione di un nuovo namespace, dove `identifier` è il nome che voglio dare al namespace, e `entities` l'insieme di classi, oggetti, funzioni e variabili che voglio siano rinchiusi nel namespace.

- ▶ Ad es.

```
namespace myNamespace  
{  
    int a, b;  
}
```

- ▶ Per accedere alle entità contenute all'interno di un particolare namespace si usa l'operatore di risoluzione dello scope `::`

- ▶ `myNamespace::a`  
`myNamespace::b`

# NAMESPACE

- ▶ Particolarmente utili nel caso in cui ci sia la possibilità che due entità globali abbiano lo stesso nome, generando un conflitto:

```
// namespaces
#include <iostream>
using namespace std;

namespace first
{
    int var = 5;
}

namespace second
{
    double var = 3.1416;
}

int main () {
    cout << first::var << endl;
    cout << second::var << endl;
    return 0;
}
```

```
5
3.1416
```

- ▶ Nota: std è il namespace della standard library, ma non è il "punto più basso" (radice) dei namespace!



# NAMESPACE

- ▶ "Ma è necessario scrivere l'intera catena delle inclusioni?" E' scomodo, specie se ho tanti namespace e classi innestati.
- ▶ La keyword `using` rende visibile un particolare membro del namespace (o l'intero namespace stesso) nella scope corrente (current declarative region).
- ▶ `using namespace std;`  
rende visibile tutti i membri di `std` all'interno dello scope corrente;
- ▶ `using std::string`  
rende visibile solo l'entità `string`
- ▶ "Ma io ho visto usare `::` anche per accedere ai membri di una classe!"  
Questa è un'altra storia...

# NAMESPACE

- ▶ Sfortunatamente...

<http://stackoverflow.com/questions/2712076/how-to-use-an-iterator/2712125>

▶  
I'm trying to calculate the distance between two points. The two points I stored in a vector in C++:  
(0,0) and (1,1).

I'm supposed to get results as

```
0  
1.4  
1.4  
0
```

But the actual result that I got is

```
0  
1  
-1  
0
```

I think there's something wrong with the way I use iterator in vector. How can I fix this problem?

I posted the code below.

# NAMES

## ► Sfortunata

<http://stackoverflow.com/questions/2712125>



```
typedef struct point {
    float x;
    float y;
} point;

float distance(point *p1, point *p2)
{
    return sqrt((p1->x - p2->x)*(p1->x - p2->x) +
                (p1->y - p2->y)*(p1->y - p2->y));
}

int main()
{
    vector<point> po;
    point p1; p1.x = 0; p1.y = 0;
    point p2; p2.x = 1; p2.y = 1;
    po.push_back(p1);
    po.push_back(p2);

    vector<point>::iterator ii;
    vector<point>::iterator jj;
    for (ii = po.begin(); ii != po.end(); ii++)
    {
        for (jj = po.begin(); jj != po.end(); jj++)
        {
            cout << distance(ii,jj) << " ";
        }
    }
    return 0;
}
```

# NAMESPACE

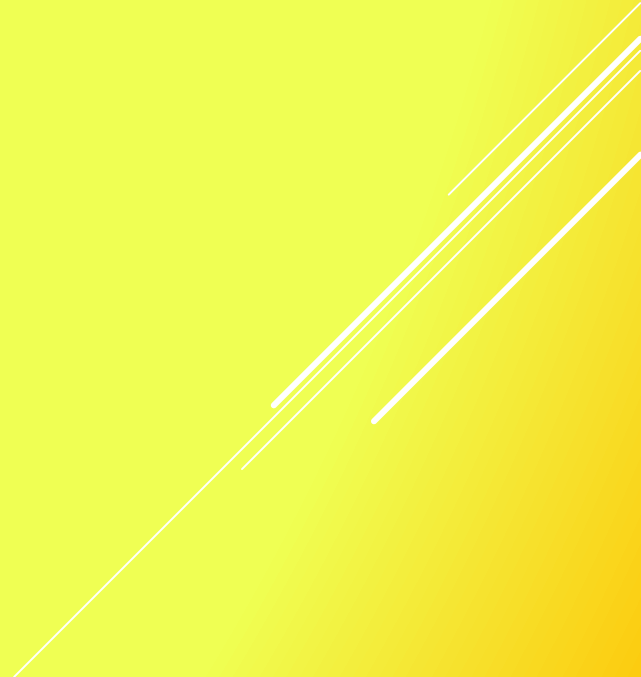
- ▶ Sfortunatamente...  
<http://stackoverflow.com/questions/2712076/how-to-use-an-iterator/2712125>
- ▶ Da qualche parte (penso all'inizio del codice) è stato dichiarato `using namespace std;`  
(altrimenti `vector` sarebbe stato indicato come `std::vector`)
- ▶ Sfortunatamente, il compilatore "preferisce" considerare `std::distance()`, invece che la funzione `distance()` definita poco sopra!
- ▶ `std::distance()` prende in ingresso due iteratori e ne calcola la distanza...
- ▶ <http://www.cplusplus.com/reference/iterator/distance/>

# NAMESPACE

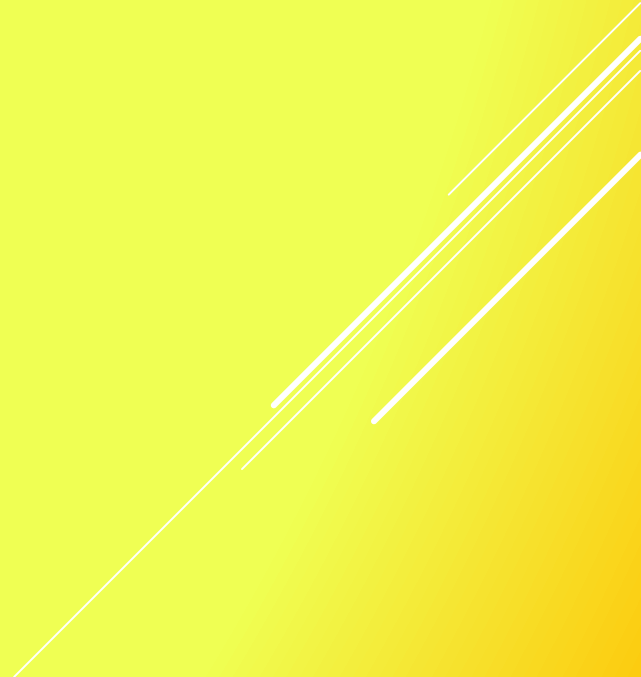
- ▶ Sfortunatamente...  
<http://stackoverflow.com/questions/2712125/using-namespace-std-with-vector>
- ▶ Da qualche parte (penso che sia in `std::vector`) è dichiarato  
`using namespace std;`  
(altrimenti `vector` sarebbe `std::vector`)
- ▶ Sfortunatamente, il compilatore cerca `std::distance()`, invece che la  
funzione `distance()` definita per `std::vector`
- ▶ `std::distance()` prende in ingresso due iteratori e calcola la distanza...
- ▶ <http://www.cplusplus.com/reference/iterator/distance/>



# CLASSI E OGGETTI



# SARO' BRE...

- ▶ Mi aspetto che conosciate già i principi di oop!
  - ▶ Rapida carrellata su come si definiscono classi ed oggetti
  - ▶ Approfondimenti su RAI, smart pointers e const-correctness
- 
- Several thin, parallel white lines are drawn diagonally across the bottom right corner of the slide, extending from the bottom edge towards the right edge.

# CLASSI

- ▶ Classe: rappresentazione di un concetto  
Es: vettore, matrice, input stream, finestra, braccio di un robot, anta di un frigorifero...
- ▶ *«Una classe è un tipo definibile che specifica come un oggetto di tale tipo possa essere creato ed utilizzato»* (B. Stroustrup)

▶                      classe <-> tipo    oggetto <-> variabile

```
▶ class X {      // nome della classe
public:          // membri pubblici - interfaccia per l'utente (accessibili da tutti!)
                // funzioni
                // tipi
                // variabili (dati e oggetti)
private:        // membri privati - dettagli dell'implementazione
                // (accessibili solo dai membri di questa classe)
                // funzioni
                // tipi
                // variabili (dati e oggetti)
};
```



# CLASSI

- ▶ Distinzione tra membri `public` e `private` utile, per es., per evitare che ad una particolare variabile di una classe venga assegnato un valore non valido (implementando una funzione `setValue( .. )`, evito questa situazione)
- ▶ Nelle classi, tutti i membri sono di default `private`. Devo specificare quali siano membri pubblici.
- ▶ Al contrario, nelle `struct` tutti i membri sono di default `public`.
- ▶ `struct` e `class` sono per lo più intercambiabili e al programmatore è lasciata la scelta di cosa usare (io uso `struct` per identificare strutture che assomigliano a contenitori per dati, mentre a classi per identificare funzionalità)

# CLASSI

## ► Esempio conciso (Starstroup)

```
class X {  
private:           // the representation (implementation) is private  
    int m;  
public:           // the user interface is public  
    X(int i = 0) : m{i} { } // a constructor (initialize the data member m)  
  
    int mf(int i) // a member function  
    {  
        int old = m;  
        m = i;    // set a new value  
        return old; // return the old value  
    }  
};  
  
X var {7}; // a variable of type X, initialized to 7  
  
int user(X var, X* ptr)  
{  
    int x = var.mf(7); // access using . (dot)  
    int y = ptr->mf(9); // access using -> (arrow)  
    int z = var.m;      // error: cannot access private member  
}
```

# CLASSI

- ▶ Definizione e dichiarazione di una classe in tempi (e luoghi) separati: non è necessario definire e dichiarare la classe tutta in una volta.
- ▶ Buona pratica: dichiarazione in file .h e implementazione in uno (o più) .cpp

- ▶ Crect.h

- ▶ 

```
#pragma once
#include <iostream>

class CRectangle {
    int width, height;
public:
    CRectangle(int,int);
    int area();
};
```

- ▶ Main.cpp

- ```
#include "Crect.h"
CRectangle::CRectangle (int a, int b) {
    width = a;
    height = b;
}

int Crectangle::area() {
    return width * height;
}

int main () {
    CRectangle rect (3,4);
    std::cout << "rect area: " <<
        rect.area() << endl;
    return 0;
}
```

# CLASSI

- ▶ Costruttore è la funzione invocata quando un oggetto viene creato
  - definizione di variabile -> allocazione sullo stack
  - new -> allocazione sullo heap
- ▶ Costruttore inizializza l'oggetto al suo stato iniziale

```
class Vector {  
    Vector() {  
        logicalSize = 0;  
        allocatedSize = 8;  
        elems = new int[allocatedSize];  
    }  
};  
  
// Both of these lines call the constructor  
Vector x;  
Vector *y = new Vector();
```

- ▶ Perché abbiamo i costruttori? Non basterebbe una funzione "init"?

# CLASSI

- Perché abbiamo i costruttori? Non basterebbe una funzione "init"?

```
struct foo {  
    int value;  
    void init(int v) {value = v;}  
};  
  
foo x;  
x.init(42);  
cout << x.value << endl;
```

```
struct ConstMember {  
    const int value;  
    void init(int v) {value = v;}  
};  
  
ConstMember x;  
x.init(42); // Error: assignment to const!
```

# CLASSI

- Perché abbiamo i costruttori? Non basterebbe una funzione "init"?

```
struct foo {  
    int value;  
    void init(int v) {value = v;}  
};
```

```
foo x;  
x.init(42);  
cout << x.value << endl;
```

```
struct ConstMember {  
    const int value;  
    void init(int v) {value = v;}  
};
```

```
ConstMember x;  
x.init(42); // Error: assignment to const!
```

- 1) se dimenticassimo di chiamare x.init() ?

# CLASSI

- ▶ Perché abbiamo i costruttori? Non basterebbe una funzione "init"?

```
struct foo {  
    int value;  
    void init(int v) {value = v;}  
};
```

```
foo x;  
x.init(42);  
cout << x.value << endl;
```

```
struct ConstMember {  
    const int value;  
    void init(int v) {value = v;}  
};
```

```
ConstMember x;  
x.init(42); // Error: assignment to const!
```

- ▶ 1) se dimenticassimo di chiamare x.init() ?
- ▶ 2) membri con qualificatore const!?

# CLASSI

- ▶ Inizializzazione ed assegnamento sono due concetti ben distinti in C++:
- ▶ **Inizializzazione** trasforma i dati iniziali (casuali) di un oggetto in dati validi
- ▶ **Assegnamento** sostituisce dati validi già esistenti con altri dati validi
- ▶ Inizializzazione fatta da costruttore
- ▶ Assegnamento fatta da operatore di assegnamento (ad es. operatore " = " )
- ▶ Verranno affrontati (rule of three):
  - default constructor
  - copy constructor (costruisce un'istanza come copia di un'altra)
  - copy assignment (non proprio un costruttore)  
(assegna l'istanza di un tipo come copia di un'altra)
- ▶ Incidentalmente, questi sono creati anche di default dal compilatore!



# CLASSI

- ▶ Es.
- ▶ Inizializzazione: default constructor  
`Crectangle x;`
- ▶ Inizializzazione: copy constructor  
`Crectangle y(x);`
- ▶ Inizializzazione: copy constructor  
`Crectangle z = x;`
- ▶ Assegnamento: copy assignment  
`z = y;`

# CLASSI

- ▶ Default constructor è molto simile a qualsiasi altra funzione membro contenuta all'interno di una classe, ma con qualche piccola eccezione
- ▶ - I costruttori **non** hanno un return value (nemmeno void)
- ▶ - I costruttori hanno lo stesso nome del tipo in questione
- ▶ - I costruttori hanno una initialization list che permette di inizializzare (non assegnare!) i membri quando viene inizializzato l'oggetto

```
// Assignment
struct Widget {
    const int value;
    Widget();
};
Widget::Widget() {
    value = 42; //ERROR
}

// Initialization
struct Widget {
    const int value;
    Widget();
};
Widget::Widget()
: value(42) {}
```

# CLASSI

- ▶ Altri esempi di initialization list:

```
struct Person {  
    int age;  
    string name;  
    Person();  
};  
  
Person::Person() : age(36), name("Kanye") {  
    //Empty constructor since nothing to assign  
}
```

- ▶ `std::vector<int> a;` // Inizializza un `vector<int>` vuoto  
`std::vector<int> b(42);` // `vector<int>` di 42 elementi  
`std::vector<int> c(42, 12);` // `vector<int>` di 42 elementi, inizializzati a 12

# CLASSI

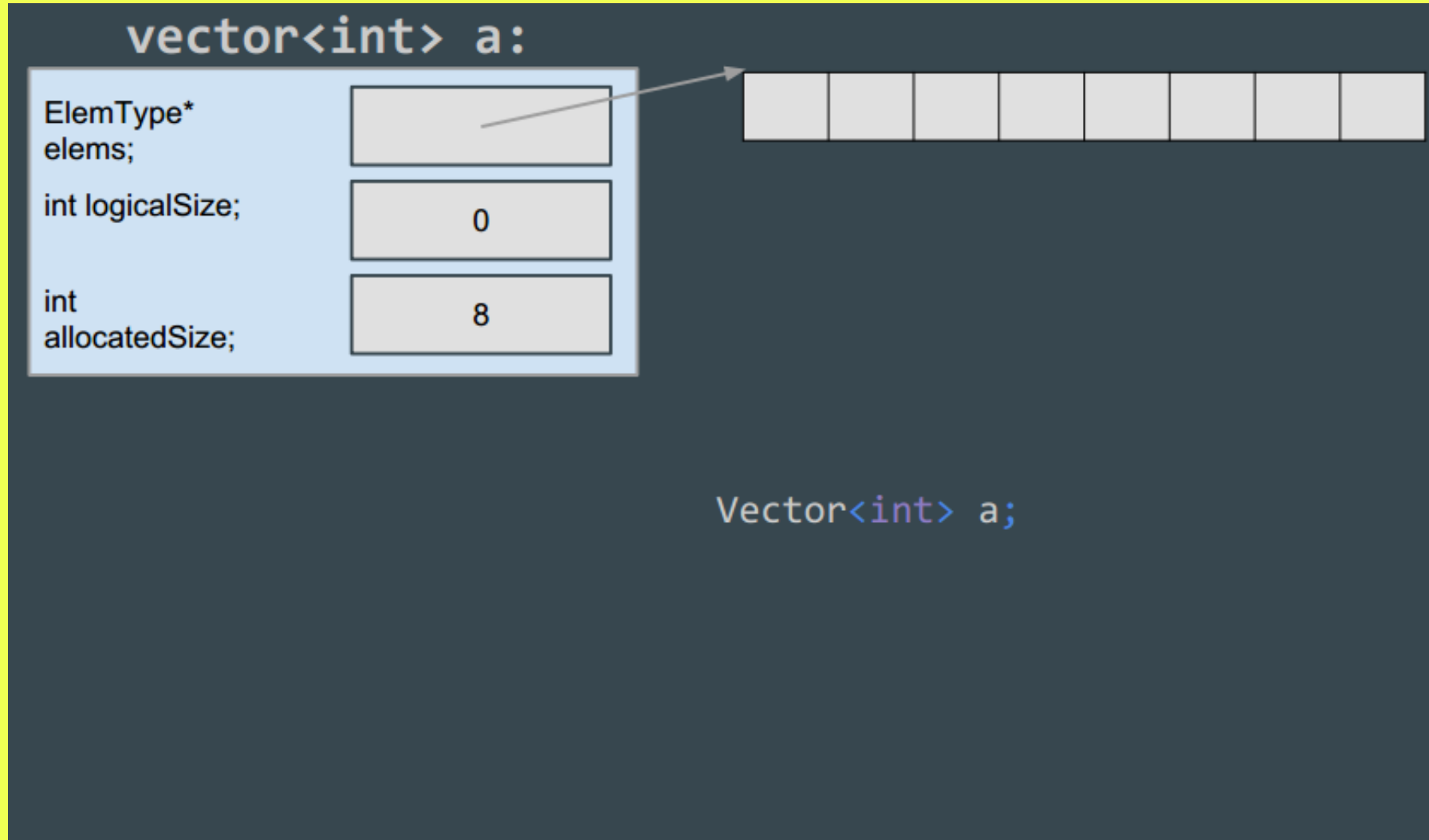
- ▶ Copy constructor viene chiamato per costruire una copia di una istanza di oggetto
- ▶ `std::vector<std::string> a(10, "no");`
- ▶ `std::vector<std::string> b(a);`      // chiamata diretta al copy constructor
- ▶ `std::vector<std::string> c = a;`      // chiamata implicita al copy constructor

## ▶ Sintassi

```
Crectangle::Crectangle( const Crectangle& cr ) {  
    ...  
}
```

- ▶ In linea di principio, copy constructor crea un nuovo oggetto e lo inizializza copiandone i valori con quello originale. Almeno... Questo è ciò che fa il default copy constructor che viene assegnato automaticamente dal compilatore.
- ▶ Ma cosa succede se un oggetto possiede un puntatore ad un altro oggetto?

# CLASSI



# CLASSI

▶ `vector<int> a:`

ElemType\*  
elems;

int logicalSize;

int  
allocatedSize;

1

8

8



```
Vector<int> a;  
a.push_back(8);
```

# CLASSI

▶ **vector<int> a:**

ElemType\*  
elems;

int logicalSize;

int  
allocatedSize;

2

8

8

6

```
Vector<int> a;  
a.push_back(8);  
a.push_back(6);
```

# CLASSI

▶ `vector<int> a:`

ElemType\*  
elems;

int logicalSize;

int  
allocatedSize;

3

8

8

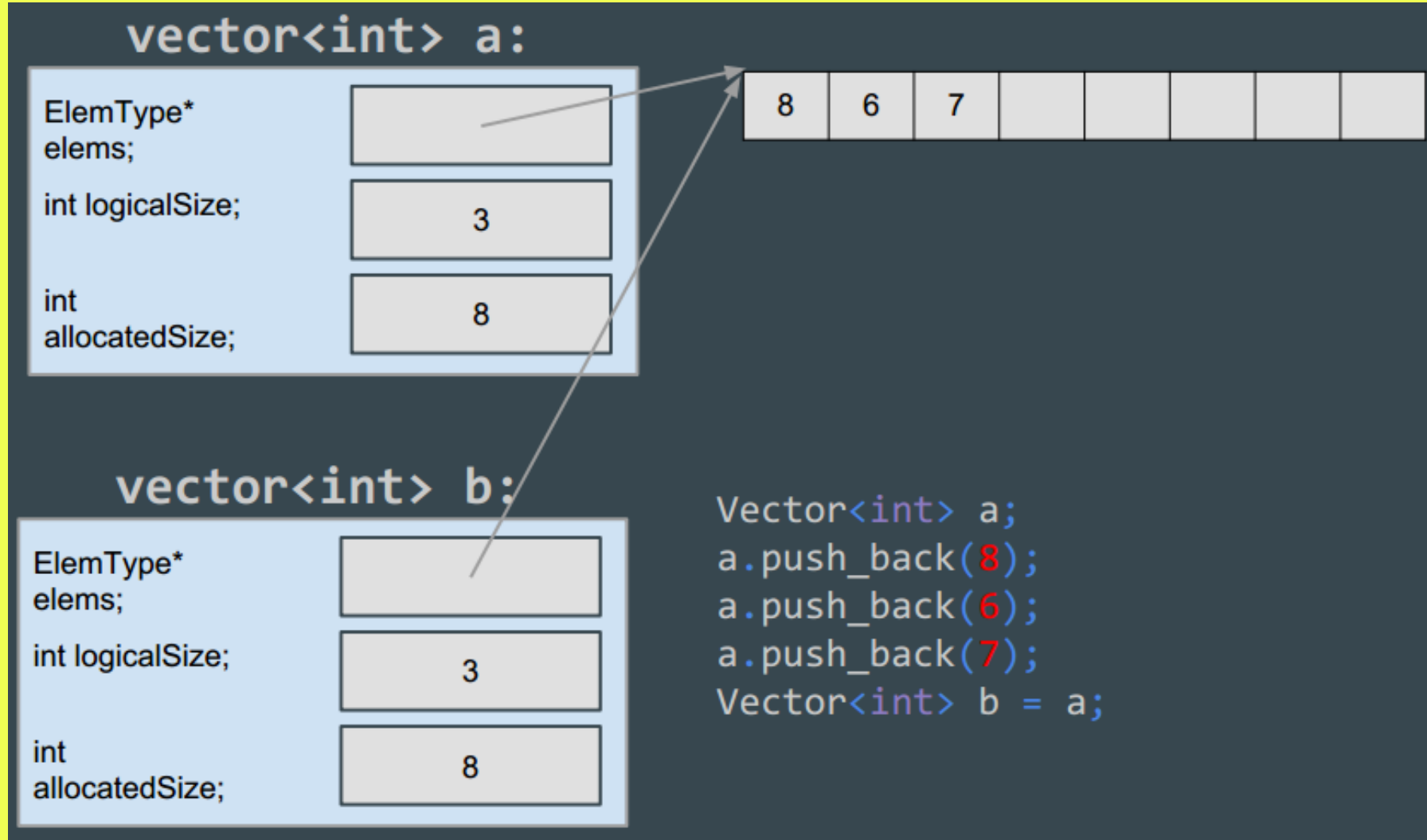
6

7

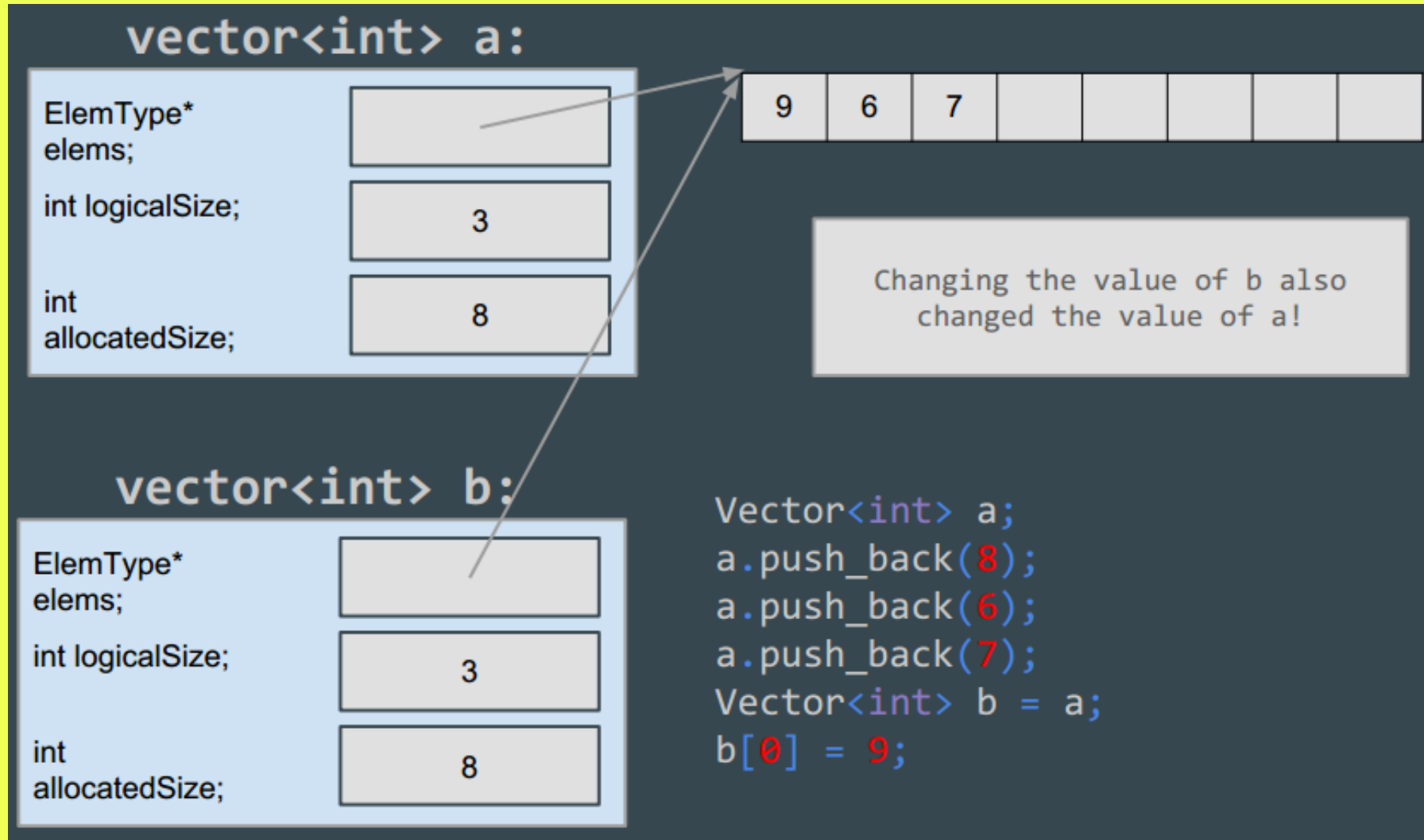
```
Vector<int> a;  
a.push_back(8);  
a.push_back(6);  
a.push_back(7);
```



# CLASSI



# CLASSI



# CLASSI

- ▶ Copy assignment constructor assegna dei nuovi valori ad un oggetto già inizializzato
- ▶ `int x = 4;      // Copy constructor (se gli int fossero oggetti)`  
`x = 2;            // Copy assignment (se gli int fossero oggetti)`
- ▶ Sintassi un po' più complicata, perché è definita tramite overload di operatore " = ":

```
class Widget {  
public:  
    Widget& operator=(const Widget& other);  
    //Other member vars and functions  
};  
  
Widget& Widget::operator=(const Widget& other) {  
    // Code to copy data from other  
}
```

# CLASSI

- ▶ Implementare un copy assignment constructor è incasinato!
  - memory leak
  - self assignment
  - return value
- ▶ Per non sudare troppo, usiamo paradigma del "copy and swap":
  - abbiamo un valore esistente da modificare, e un valore esistente da cui attingere
  - uso copy constructor per creare un valore temporaneo del valore da cui attingiamo
  - scambio il contenuto di questa copia con il valore da modificare

# CLASSI

```
class Widget {  
    int value;  
public:  
    void swap(Widget& other);  
    Widget& operator=(const Widget& other);  
};  
  
void Widget::swap(Widget& other) {  
    std::swap(value, other.value);  
}  
  
Widget& Widget::operator=(const Widget& other) {  
    Widget temp(other);  
    swap(temp);  
    return *this;  
}
```

# CLASSI

- ▶ Visto che siamo in tema di costruttori, cosa fa la seguente riga di codice?
- ▶ `vector<int> v( );`

# CLASSI

- ▶ Visto che siamo in tema di costruttori, cosa fa la seguente riga di codice?
- ▶ `Vector<int> v( );`
- ▶ `int x( );`

# CLASSI

- ▶ The **most vexing parse**:
- ▶ La linea:  
`TimeKeeper time_keeper(Timer());`
- ▶ - definizione di variabile `time_keeper` di tipo `TimeKeeper`, inizializzata con una istanza anonima della classe `Timer`
- ▶ - dichiarazione di funzione `time_keeper` che restituisce un oggetto di tipo `TimeKeeper` e ha un singolo parametro che è una funzione il cui return type è `Timer`
- ▶ In tanti ci aspettiamo il primo comportamento
- ▶ Lo standard prevede il secondo!!!

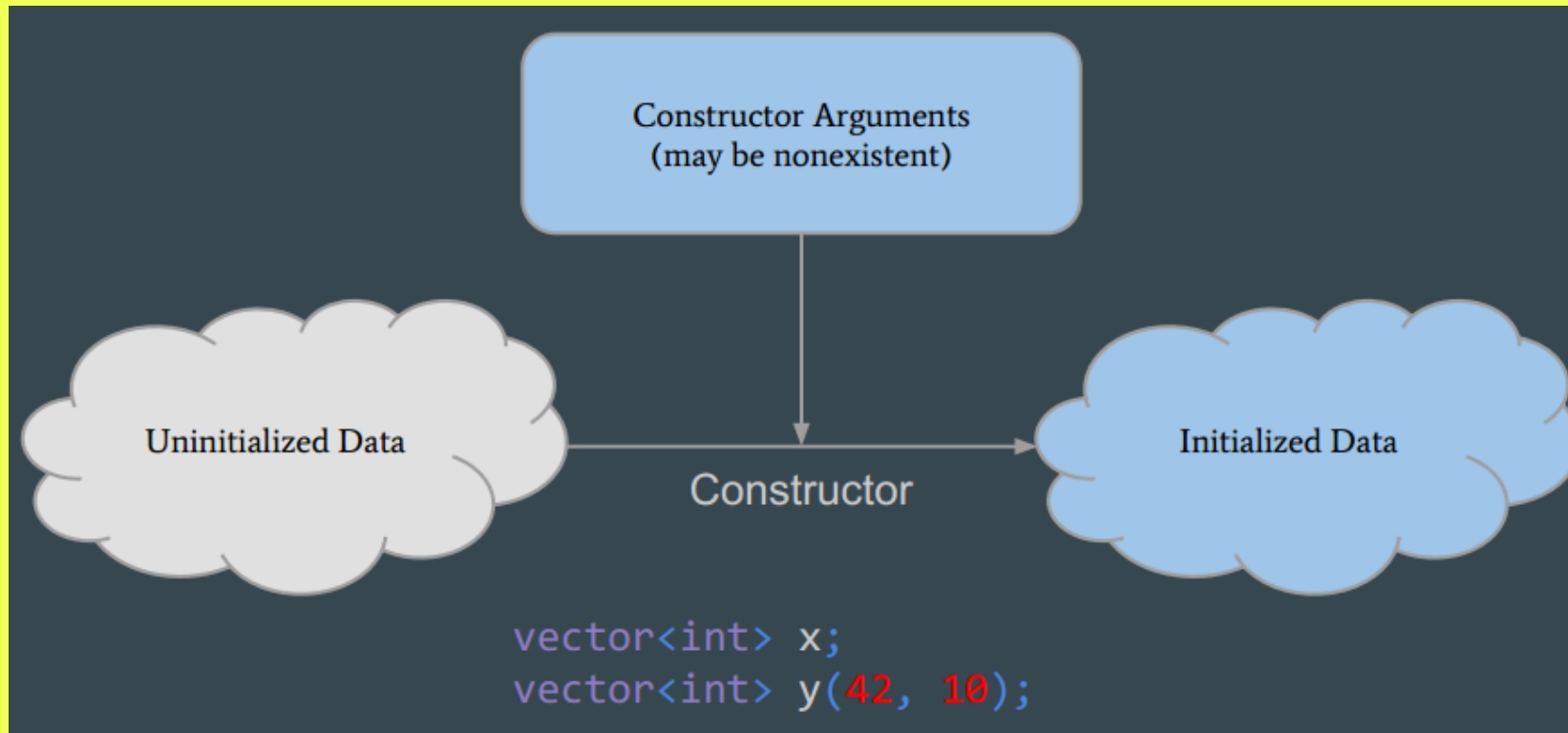
```
class Timer {  
    public:  
        Timer();  
};  
  
class TimeKeeper {  
    public:  
        TimeKeeper(const Timer& t);  
  
        int get_time();  
};  
  
int main() {  
    TimeKeeper time_keeper(Timer());  
    return time_keeper.get_time();  
}
```



RAII

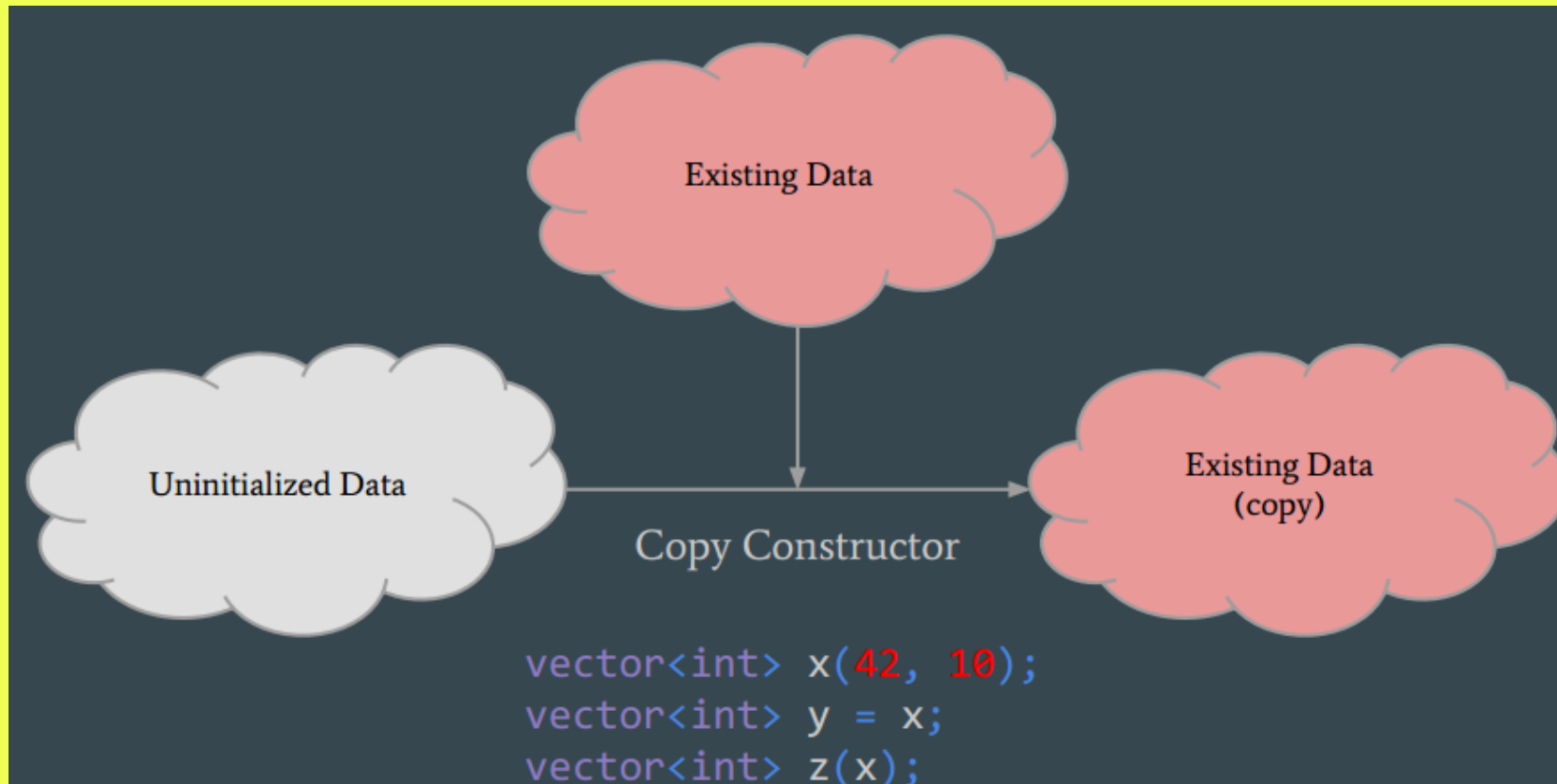
# RAII

- ▶ Abbiamo parlato di costruttori come strumenti per inizializzare strutture dati e assegnare ad esse dei valori di partenza validi



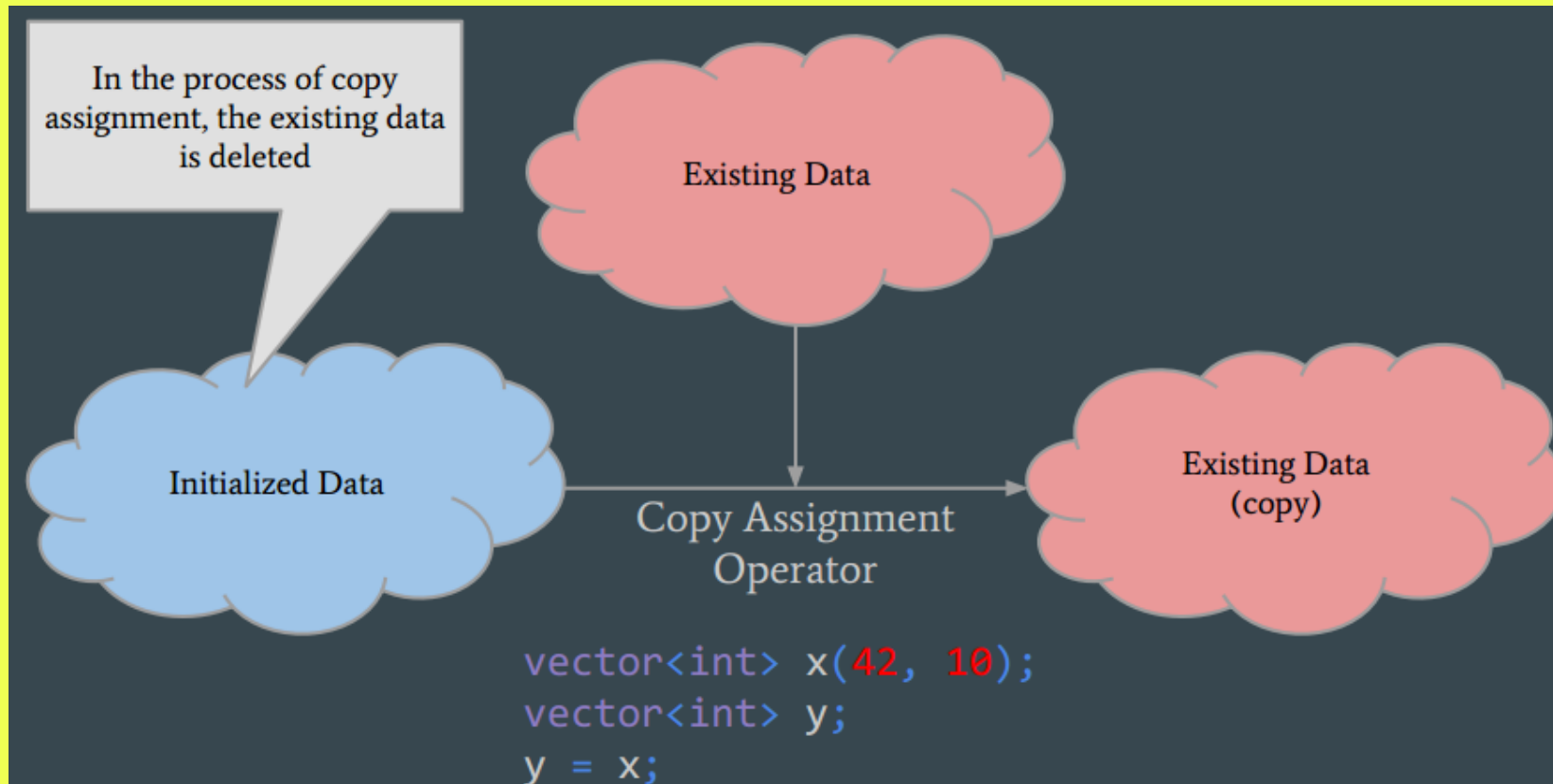
# RAII

- ▶ Abbiamo parlato di costruttori come strumenti per inizializzare strutture dati e assegnare ad esse dei valori di partenza validi, oppure – nel caso del copy constructor – inizializzare una copia



# RAII

- ▶ Abbiamo parlato di costruttori come strumenti per inizializzare strutture dati e assegnare ad esse dei valori di partenza validi, oppure – nel caso del copy constructor – inizializzare una copia, o rimpiazzare dati validi già esistenti con altri dati nel caso di assignment copy constructor



# RAII

- ▶ Prima di lanciarcì nella descrizione del paradigma RAII, ripassiamo come avviene la gestione delle risorse avviene in C, prendendo come esempio il file processing.
- ▶ Per leggere un valore da un file dobbiamo:
  - aprirlo con la funzione `fopen()`
  - leggere caratteri con `fgetc()` o `fgets()`
  - chiuderlo con `fclose()`
- ▶ Se ci si dimenticasse di chiuderlo, potrebbero succedere cose spiacevoli

# RAII

- ▶ Fino ad ora, abbiamo parlato di costruttori in termini di inizializzatori – trasforma dati random in dati che possiamo utilizzare.
- ▶ Ora vediamoli in termini di gestori di risorse
- ▶ Una risorsa è:
  - qualcosa che si deve acquisire e rilasciare
  - va acquisita prima di poterla utilizzare e rilasciata al termine
- ▶ Immaginiamo di essere dei fotografi subacquei con la passione per gli squali

# RAI

- ▶ Prima di gettarci in acqua, dobbiamo **acquisire** una gabbia a prova di squalo:



# RAI

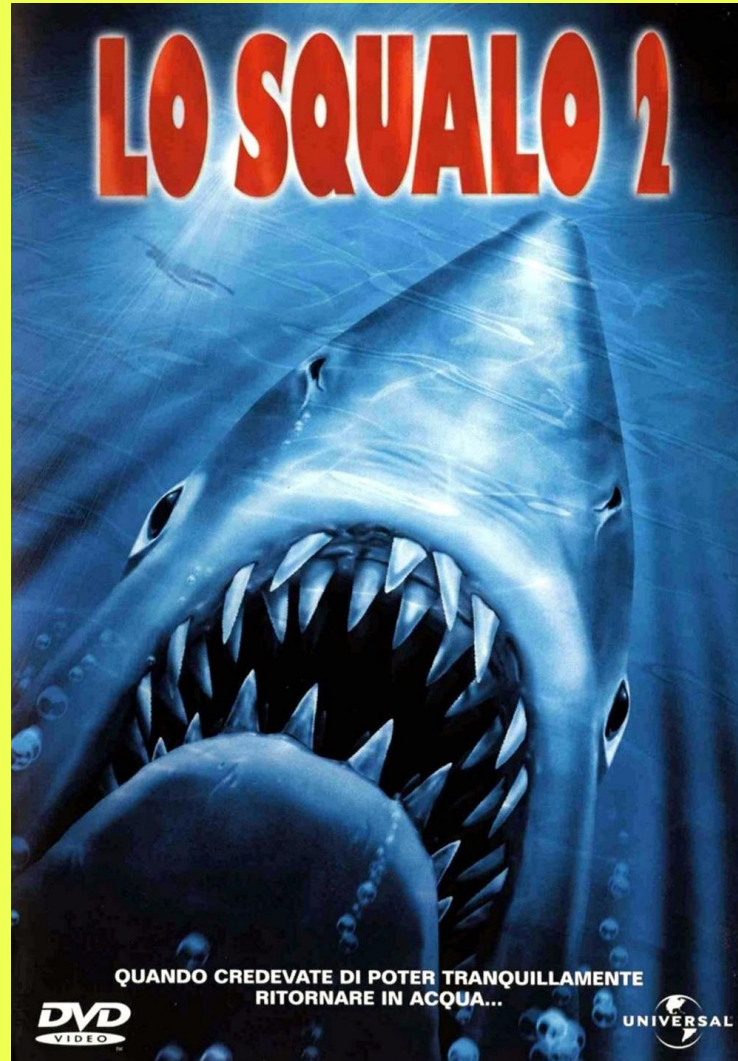
- Una volta acquisita, possiamo fotografare gli squali in tutta "tranquillità".





# RAI

- ▶ Se proviamo ad utilizzare una risorsa senza prima averla acquisita, possono accadere degli errori



# RAI

- ▶ Se ci dimentichiamo di **rilasciare** la risorsa, possono crearsi degli altri problemi



# RAII

- ▶ In C:
- ▶ 

```
void printFile( const char *nomeFile ) {  
    FILE *f = fopen( nomeFile, "r" );  
    fclose(f);  
}
```

# RAII

- ▶ In C:

- ▶ 

```
void printFile( const char *nomeFile ) {  
    FILE *f = fopen( nomeFile, "r" );  
    fclose(f);  
}
```

- ▶ 

```
void printFile( const char *nomeFile ) {  
    FILE *f;           //Uh oh...  
    fclose(f);  
}
```

- ▶ 

```
void printFile( const char *nomeFile ) {  
    FILE *f = fopen( nomeFile, "r" );  
}
```

- ▶ Ok, ma cosa hanno in comune le risorse con le strutture o classi?

# RAII

- ▶ RAII: Resource Acquisition Is Initialization
- ▶ (ok, non è un gran nome, e il sig. Strastrup ne è ancora pentito)
- ▶ 1) La creazione di un oggetto chiama il costruttore che acquisisce le risorse
  - questo avviene quando dichiaro una variabile o la creo con `new`
- ▶ 2) Quando il distruttore di un oggetto viene chiamato, la risorsa viene rilasciata
  - questo avviene quando un oggetto esce dalla validità dello scope in cui era definito, o quando viene esplicitamente invocato `delete`

# RAII

- Vediamo come può tornare utile nell'esempio del file:

```
// Remember this code?  
void printFile(const char *name) {  
    // Acquire the resource  
    FILE *f = fopen(name, "r");  
  
    // Print the contents of 'f'  
  
    // Release the resource  
    fclose(f);  
}
```

# RAII

- Vediamo come può tornare utile nell'esem

```
// Remember this code?  
void printFile(const char *name) {  
    // Acquire the resource  
    FILE *f = fopen(name, "r");  
  
    // Print the contents of 'f'  
  
    // Release the resource  
    fclose(f);  
}
```

```
struct FileObj {  
    FILE *ptr;  
    // Acquire the file resource  
    FileObj(char *name)  
        : ptr(fopen(name, "r")) {}  
  
    // Release the file resource  
    ~FileObj() {  
        fclose(ptr);  
    }  
};  
  
void printFile(const char *name) {  
    // Initialize the object  
    // Implicitly acquire the resource  
    FileObj o(name);  
  
    // Print the contents of the file  
  
    // Destructor the object  
    // Implicitly release the resource  
}
```

# RAII

- ▶ Una delle applicazioni più utili del paradigma RAII è nella gestione dei puntatori:

```
▶ void f() {  
    std::string *x = new std::string("Mamma!");  
  
    std::cout << *x << std::endl;  
    std::cout << x->size() << std::endl;  
  
    delete x;  
}
```

- ▶ Dimentichiamoci il delete => succede un casino (memory leaking)



# RAII

- ▶ Mi piacerebbe automatizzare allocazione e deallocazione di oggetti creati sullo heap.
- ▶ Potrei creare un oggetto per questo compito, qualcosa che encapsuli un puntatore e che ne gestisca il comportamento.
- ▶ Idealmente, questo oggetto dovrebbe:
  - gestire inizializzazione del puntatore e allocazione memoria nel costruttore
  - rilasciare la memoria quando l'oggetto viene distrutto
  - avere accesso al puntatore incapsulato tramite operatori \* e ->
  - quando copio questa struttura, copio il valore del puntatore

# RAII

```
▶ void f() {  
    std::string *x = new string( "Mommy!" );  
  
    std::cout << *x << std::endl;  
    std::cout << x->size() << std::endl;  
  
    delete x;  
}
```



# RAII

```
▶ void f() {  
    SmartPtr<std::string> x(new string( "Mommy!" ));  
  
    std::cout << *x << std::endl;  
    std::cout << x->size() << std::endl;  
  
    //delete x;  
    //la stringa è eliminata implicitamente quando x esce dallo scope  
}
```

▶ Ok, ma per la copia?

# RAII

- Ok, ma per la copia?

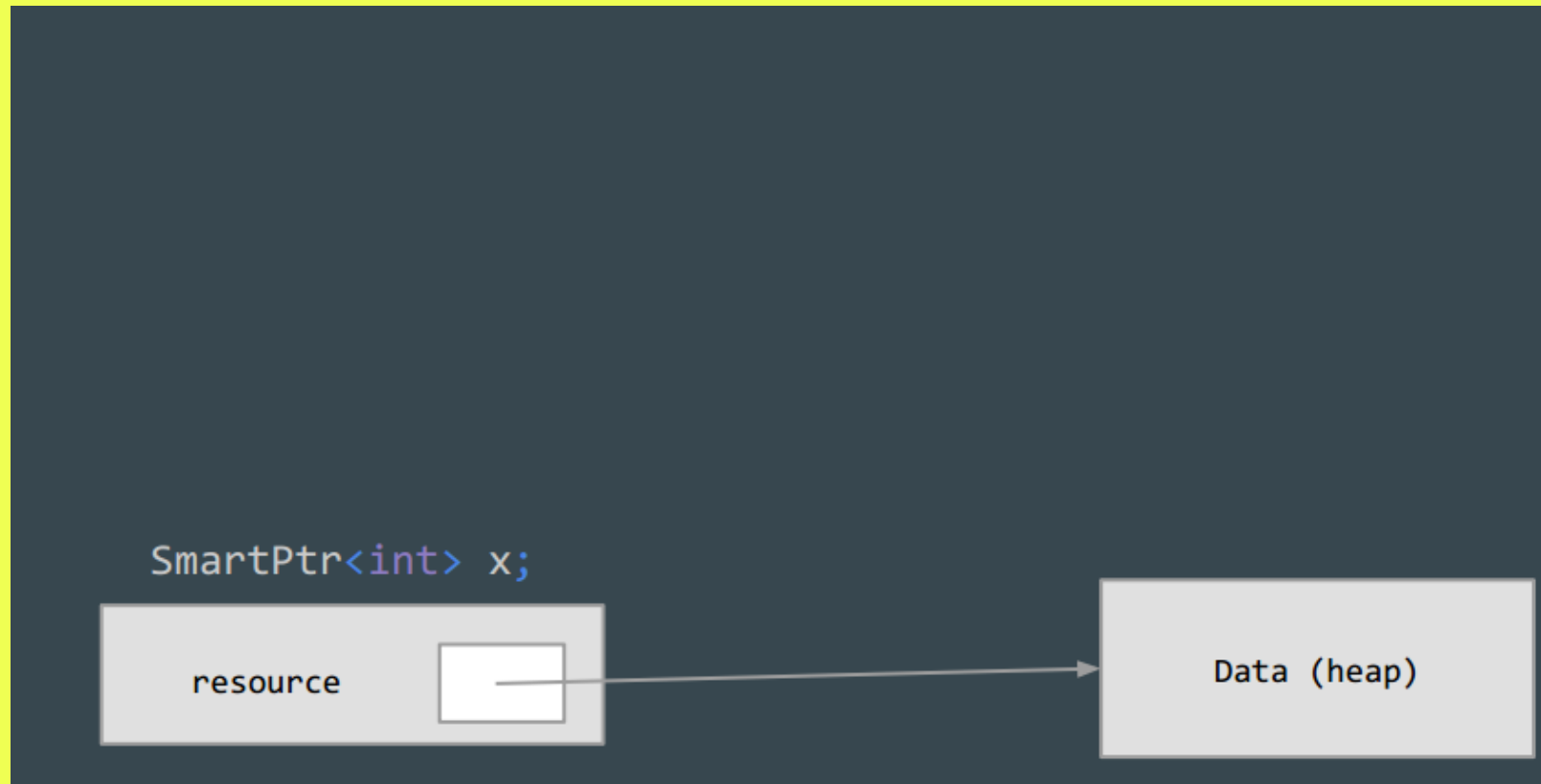
```
// Regular pointers implementation
void f() {
    int *x = new int(4);
    cout << *x << endl;
    int *y = x;
    *y = 8;
    cout << *x << endl;
    delete x;
}
```

```
// Will this work given my design?
void f() {
    SmartPtr<int> x(new int(4));
    cout << *x << endl;
    SmartPtr<int> y(x);
    *y = 8;
    cout << *x << endl;
}
```

```
// Will this work given my design?
void f() {
    SmartPtr<int> x(new int(4));
    cout << *x << endl;
    if (/* condition */) {
        SmartPtr<int> y(x);
        *y = 8;
    }
    cout << *x << endl;
}
```

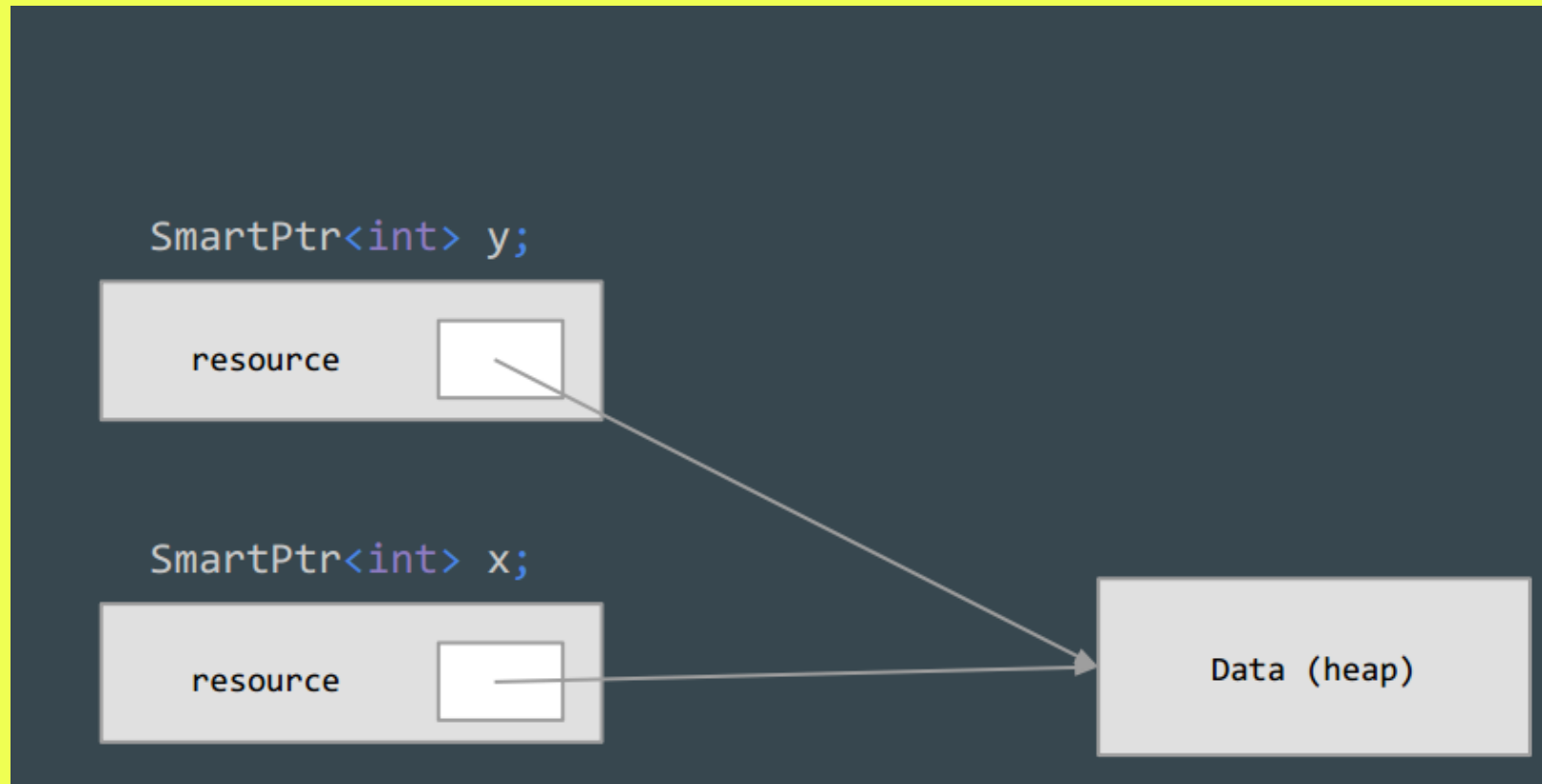
# RAII

- ▶ In realtà è un gran casino, perché...
- ▶ Per prima cosa, creiamo uno smart pointer che alloca sullo heap la struttura che vogliamo



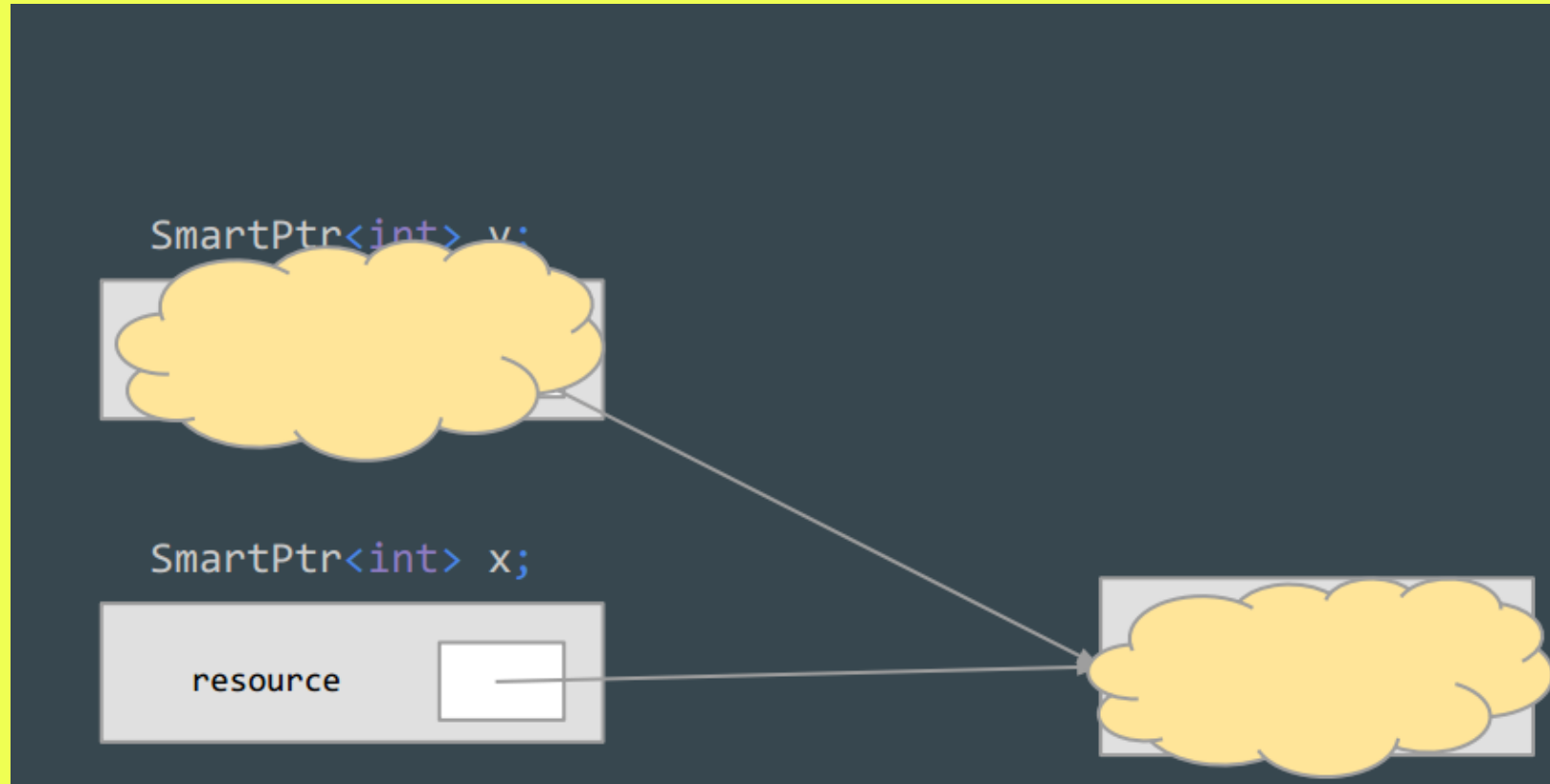
# RAII

- ▶ In realtà è un gran casino, perché...
- ▶ Poi facciamo una copia dello smart pointer



# RAII

- ▶ In realtà è un gran casino, perché...
- ▶ Quando y va fuori dallo scope, viene chiamato il distruttore, che elimina anche i dati dallo heap



# RAII

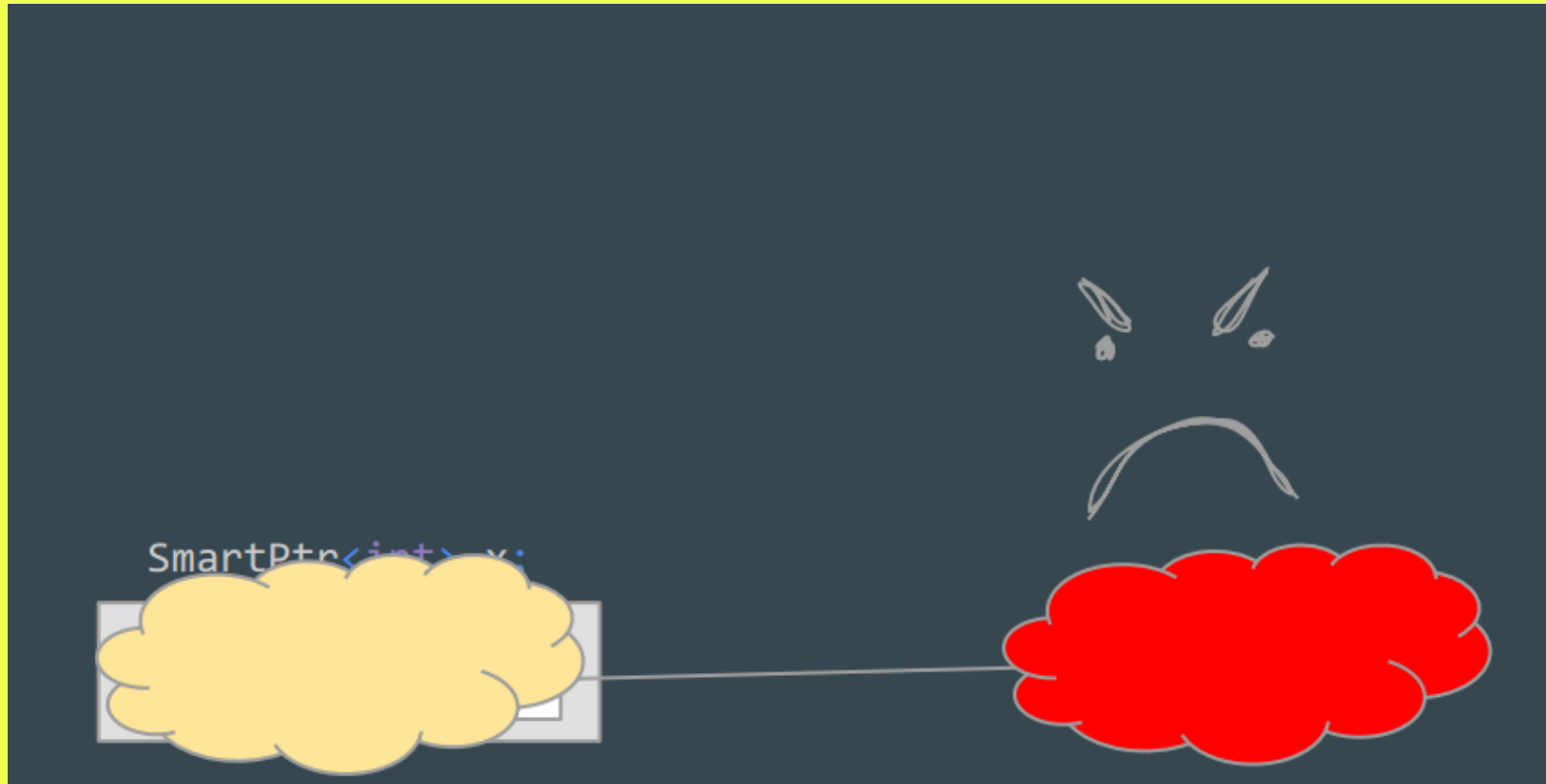
- ▶ In realtà è un gran casino, perché...
- ▶ Ora, però, x punta a dei dati che sono già stati deallocati





# RAII

- ▶ In realtà è un gran casino, perché...
- ▶ Quando anche x andrà fuori dallo scope, viene chiamato il suo distruttore che cercherà di dellocare di nuovo gli stessi dati!



# RAII

- ▶ Lavorare con le risorse RAII è un casino perché spesso può capitare di avere due oggetti che pensano di essere gli unici detentori di una risorsa.
- ▶ Per fortuna:
  - non dobbiamo scrivere la classi che implementano gli smart pointer
  - gli smart pointer della STL si prendono cura della gestione di casi come quello appena descritto

# RAII

- ▶ Lavorare con le risorse RAII è un casino perché spesso può capitare di avere due oggetti che pensano di essere gli unici detentori di una risorsa.
- ▶ Per fortuna:
  - non dobbiamo scrivere la classi che implementano gli smart pointer
  - gli smart pointer della STL si prendono cura della gestione di casi come quello appena descritto
- ▶ Infatti STL ha:
  - `unique_ptr`
  - `shared_ptr`
  - `weak_ptr`
  - `auto_ptr`

# RAII

- ▶ `unique_ptr`:
- ▶ Simile a `SmartPtr` che abbiamo appena descritto
- ▶ Si incarica di allocare la risorsa e distruggerla quando `unique_ptr` esce dallo scope o è distrutto
- ▶ Non può essere copiato (ma è possibile trasferire la ownership!)
- ▶ E' possibile allocare array
- ▶ 

```
{  
    std::unique_ptr<int> p = new int;  
    std::unique_ptr<int[50]> arr = new( int[50] );  
}
```

  
// qui p e arr non esistono più!

# RAII

- ▶ `shared_ptr`:
- ▶ Simile a `unique_ptr` nella creazione e distruzione, ma tiene traccia di quanti oggetti ne detengono il controllo (ownership) eseguendone un conteggio
- ▶ Si occupa di distruggere la risorsa solo quando non ci sono più oggetti che ne detengono il controllo
- ▶ Questo meccanismo funziona solo tramite copia
- ▶ 

```
{  
    std::shared_ptr<int> p1 = new int;  
    {  
        std::shared_ptr<int> p2 = p1;  
        // qui posso usare p1 e p2  
    }  
    // p2 non esiste più, ma p1 punta ancora alla risorsa  
}  
// qui p1 e p2 non esistono più, la risorsa viene rilasciata
```
- ▶ Occhio: in caso di referenze circolari, la risorsa non verrà mai rilasciata!!!

# RAII

- ▶ `weak_ptr`:
- ▶ Servono per risolvere il problema delle referenze circolari: possono essere creati a partire da uno `shared_ptr` esistente, ma non ne incrementano il contatore delle referenze
- ▶ Nota (mia): `unique_ptr` e `shared_ptr` dovrebbero essere sufficienti per ogni applicazioni!
- ▶ `auto_ptr`:
- ▶ Questo è facile: NON USATELO! E' stato deprecato.

# RAII

- ▶ Macro per gli smart pointer:
- ▶ Invece di usare `new`, STL mette a disposizione macro per creazione di smart ptr:
  - ▶ `std::unique_ptr<int> p1 = std::make_unique<int>();`
  - ▶ `std::shared_ptr<std::string> p2 = std::make_shared<std::string>( "Solita stringa" );`
- ▶ Consigli:
  - usare sempre gli `unique_ptr` (e la macro `make_unique`)
  - ne caso in cui sia necessario passarlo da uno scope ad un altro, usare la `move` o passare il raw pointer restituito dalla funzione membro `.get()`
  - poiché gli `unique_ptr` hanno un solo possessore, non si verificheranno mai dei memory leak

# RAII

- ▶ Macro per gli smart pointer:
- ▶ Invece di usare `new`, STL mette a disposizione macro per creazione di smart ptr:
  - ▶ `std::unique_ptr<int> p1 = std::make_unique<int>();`
  - ▶ `std::shared_ptr<std::string> p2 = std::make_shared<std::string>( "Mollatemi" );`
- ▶ Consigli:
  - usare sempre gli `unique_ptr` (e la macro `make_unique`)
  - ne caso in cui sia necessario passarlo da uno scope ad un altro, usare la `move` o passare il raw pointer restituito dalla funzione membro `.get()`
  - poiché gli `unique_ptr` hanno un solo possessore, non si verificheranno mai dei memory leak
- ▶ Svantaggio: se si distrugge il `unique_ptr` e da qualche parte esiste ancora un raw pointer che punta ai suoi dati...