

נושאים מתקדמים בתכנות מונחה עצמים

הרצאה 4

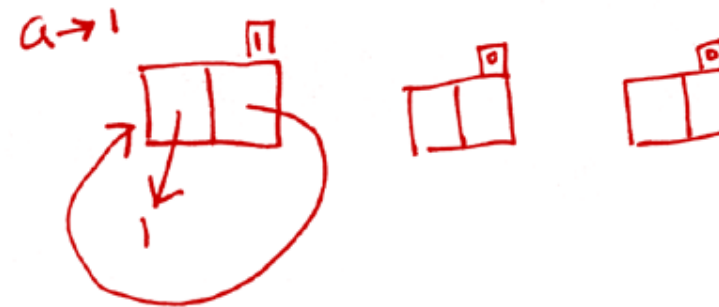
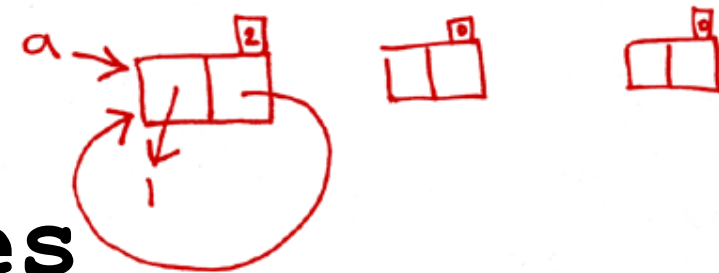
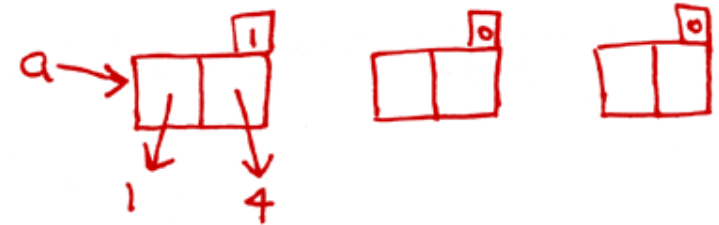
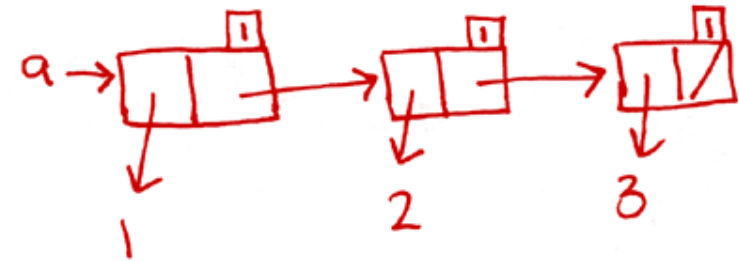
פרופ' עפר שיר
ofersh@telhai.ac.il

החוג למדעי המחשב



מבנה ההרצאה

- *Shared structures and reference counting*
- Garbage collection
- Strong and weak pointers
- C++0x Smart→Pointers

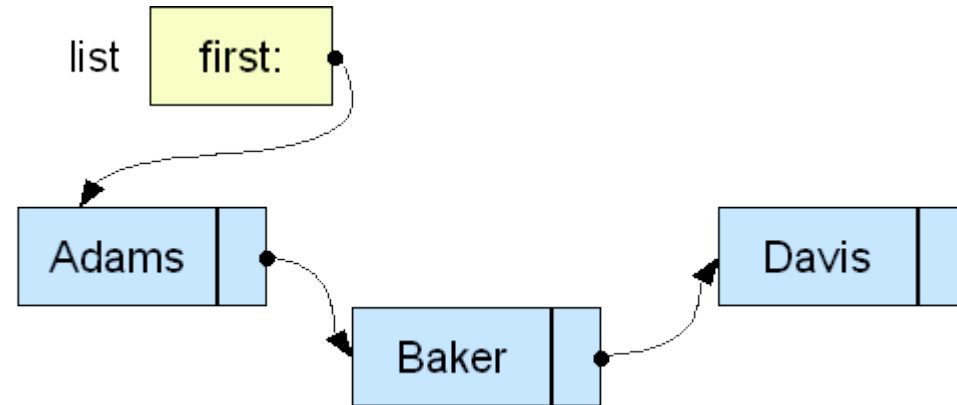


Shared Structures

Source: Steven Zeil, CS361 ODU.EDU

Singly Linked Lists (SLL)

```
struct SLNode {  
    string data;  
    SLNode* next;  
    :  
    ~SLNode () {delete next;}  
};  
class List {  
    SLNode* first;  
public:  
    :  
    ~List() {delete first;}  
};
```



בעיה: מחסנית רקורסיה בגודל $O(n)$

- בהינתן רשימה מקושרת באורך n , המחסנית תידרש לאחסן $O(n)$ קריאות בלתי-שלמות לשם ביצוע הפירוק הרקורסיבי.
- אם גודל המחסנית אינו מהווה בעיה, זהו פתרון קביל.
- אם גודל המחסנית מהווה שיקול, נבחר בגישה אגרסיבית יותר בה נמחק את הרשימה עצמה ולא את הצמתים שלה (בשקף הבא).

ריסון מחסנית רקורסיה גדולה

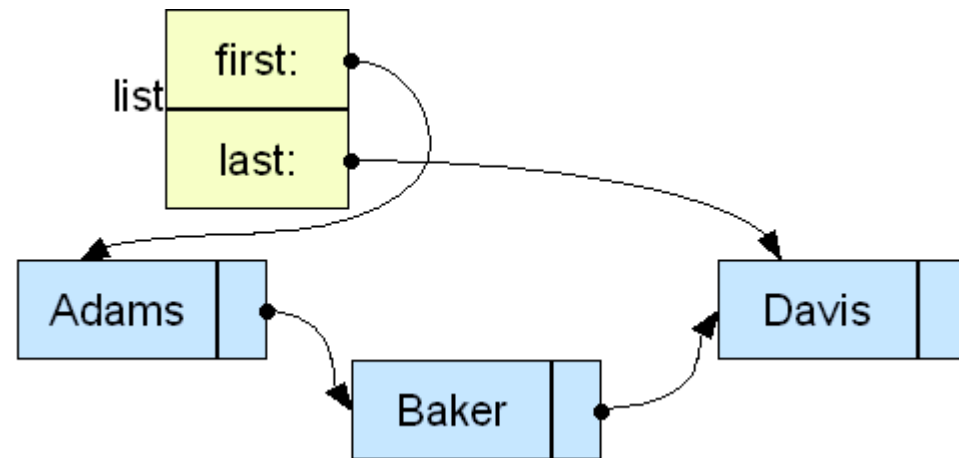
```
struct SLNode {
    string data;
    SLNode* next;
    :
    ~SLNode () { /* do nothing */}
};

class List {
    SLNode* first;
public:
    :
    ~List() {
        while (first != 0) {
            SLNode* next = first->next;
            delete first;
            first = next;
        }
    }
};
```

First-Last Headers

```
struct SLNode {
    string data;
    SLNode* next;
    :
    ~SLNode () {delete next;}
};

class List {
    SLNode* first;
    SLNode* last;
public:
    :
    ~List() {
        delete first;
        delete last;
    }
};
```

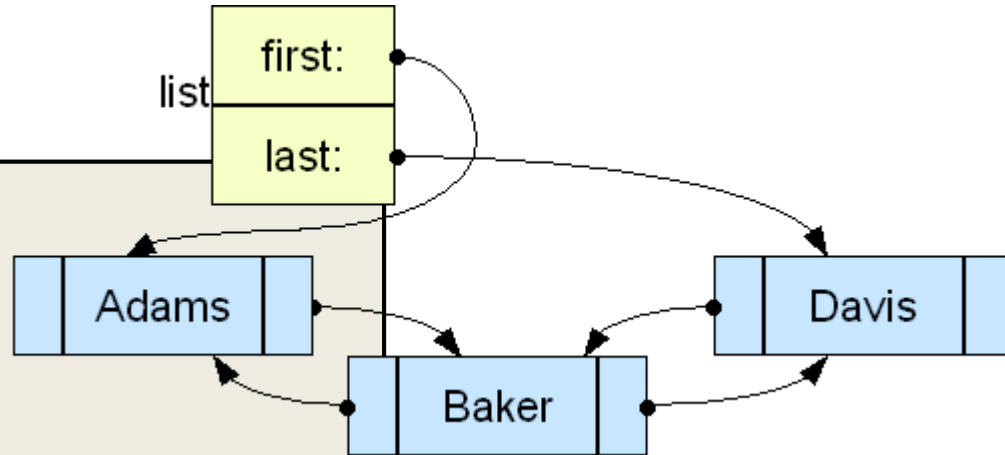


בעיה: מחיקה כפולה

- בעת פירוק הרשימה, הצומת האחרונה תימחק פעמיים.
- הדבר עשוי לגרום להשחתת זיכרון (*corrupt the heap*) ואף להביא לקריסת התכנית.

Doubly Linked Lists

```
struct DLNode {  
    string data;  
    DLNode* prev;  
    DLNode* next;  
    :  
    ~DLNode () {delete prev; delete next;}  
};  
  
class List {  
    DLNode* first;  
    DLNode* last;  
public:  
    :  
    ~List() {delete first; delete last;}  
};
```



בעיה: רקורסיה אינסופית

- גישת המחיקה האגרסיבית תגרור מחיקות כפולות, אך גרוע מכך, תמשיך להפעיל מפרקים באופן אינסופי.
- הדבר יימשך עד להשחתה מוחלטת של מרחב הזיכרון הדינאמי או לשימוש מקסימלי במחסנית הזיכרון הזמינה לקריאות הרקורסיה.
- מה מקור הבעייתיות?
 - שיתוף באמצעות מצביעים
 - קיומם של מעגלים

אובייקטי גרף

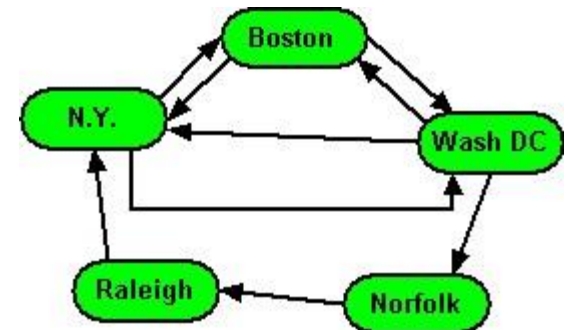
- הבעיה המוזכרת לא תתרחש רק במימוש מבני נתונים ב-low-level, אלא אף ברמת האפליקציה.
- המקרה הנפוץ יהיה עולם מונחה עצמים בעל קשרים פנימיים המתוארים באמצעות גרף מתמטי.
- הדוגמא הבאה תעסוק באובייקט "שדה תעופה".

Airline Connections

```
class Airport
{
    :

private:
    vector<Airport*> hasFlightsTo;
};

Airport::~~Airport()
{
    for (int i=0; i < hasFlightsTo.size(); ++i)
        delete hasFlightsTo[i];
}
```

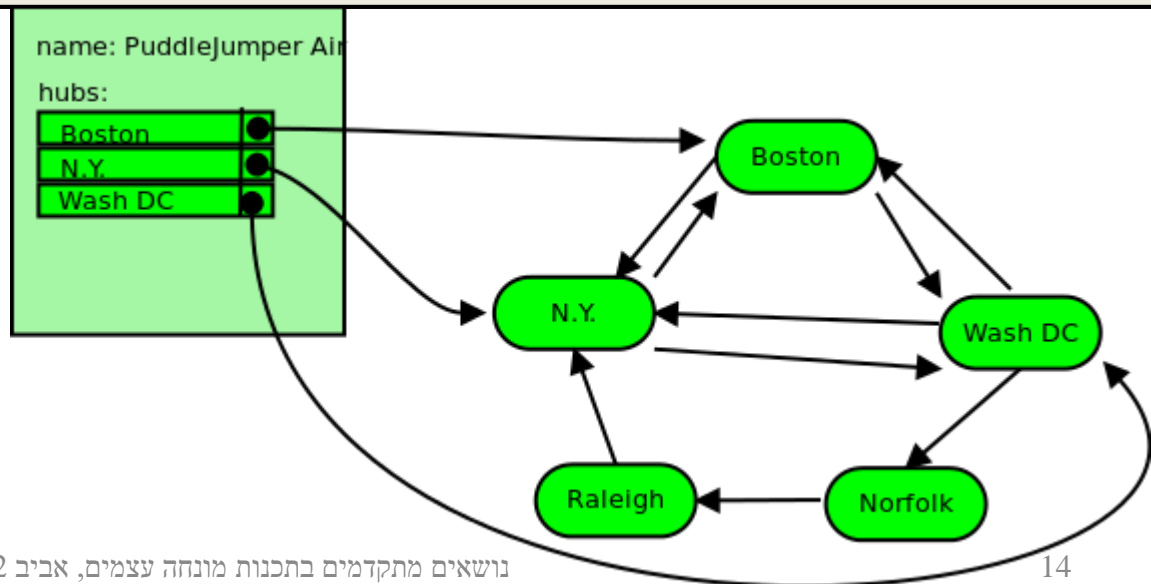


מחיקת שדה-תעופה השייך למעגל

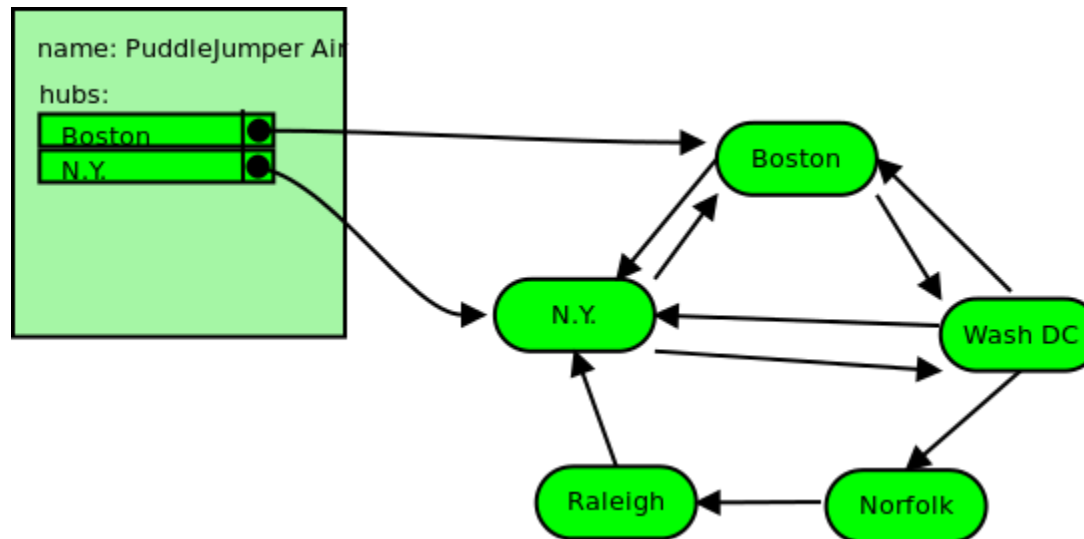
- ניסיון למחוק את שדה התעופה Boston תביא לקטסטרופה, בדומה לדוגמאות הרשימות המקושרות:
 - מחיקות כפולות
 - רקורסיה אינסופית בין המפרקים של שדות התעופה השונים
- מהו ההקשר הכולל של המחיקה? כיצד ייתכן ששדה התעופה Boston יימחק?
 - ההקשר הרחב בשקף הבא

The Airline Structure

```
class AirLine {  
    :  
    string name;  
    map<string, Airport*> hubs;  
};  
AirLine::~~Airline() {  
    for (map<string,Airport*>::iterator i=hubs.begin();i != hubs.end();  
    ++i) delete i->second;  
}
```

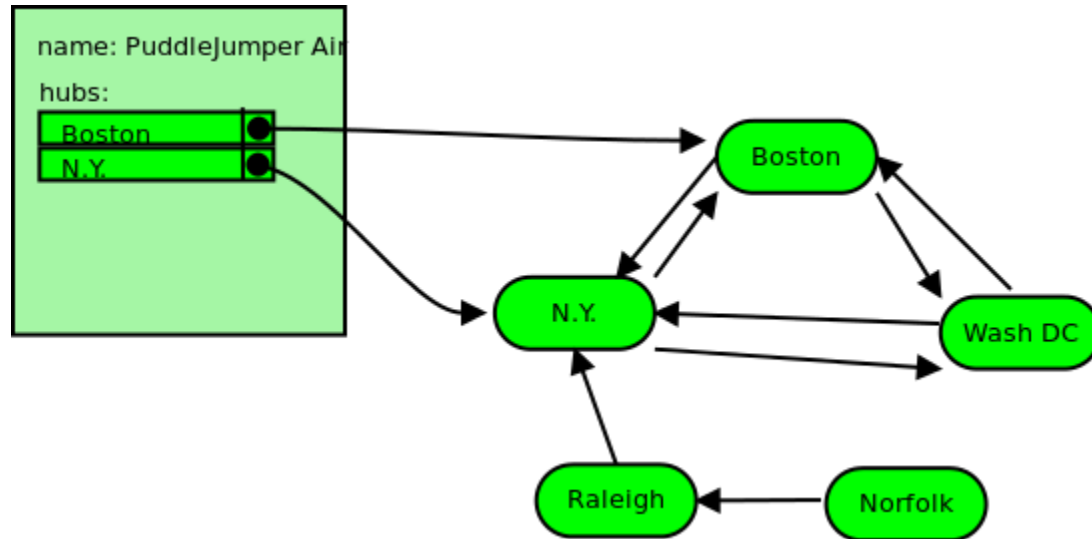


DC loses status as a hub



*Even though the pointer to it were removed from the hubs table, the Wash DC airport needs to remain in the map.

DC drops service to Norfolk



*Norfolk and Raleigh should be deleted, as there would be no way to reach them.

האתגר הקיים

- במקרה הראשון: יש להוציא את שדה התעופה DC **מרשימת ה-hubs בלבד**, מבלי למחוק אותו.
- במקרה השני: שדות התעופה Norfolk, Raleigh צריכים שניהם להימחק, מפני שאבדה הגישה אל שניהם.
- האתגר: כתיבת קוד (מפרקים וכו') המסוגל להפריד בין שני המקרים הללו ולבצע את התרחישים המתוארים.

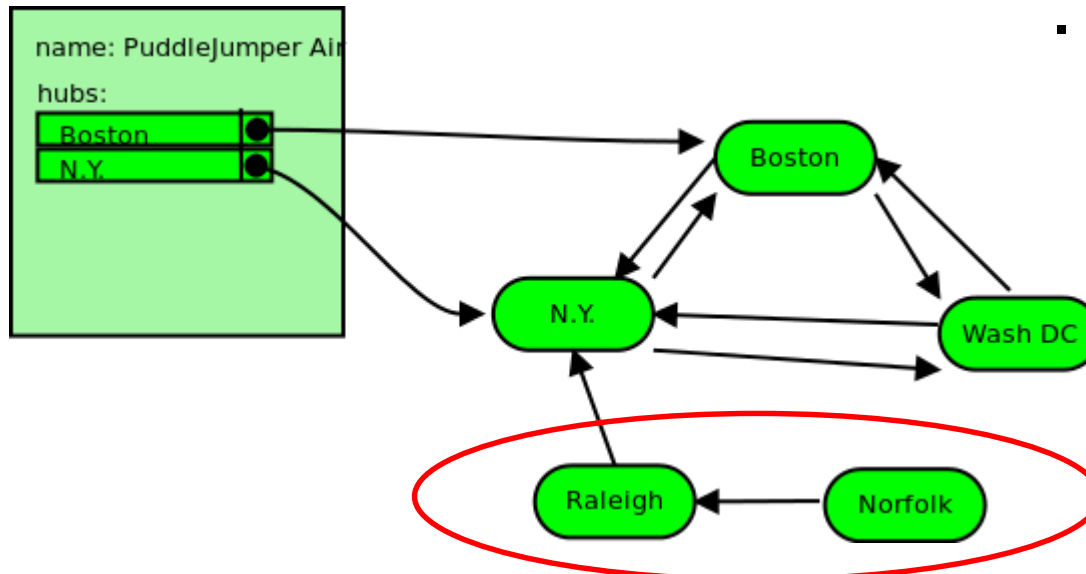


Garbage Collection

Source: Steven Zeil, CS361 ODU.EDU

Garbage

- אובייקטים במרחב הזיכרון הדינאמי (*heap*) אליהם לא קיימת גישה (במספר צעדים כלשהו) – ממצביעים במחסנית ההפעלה (*activation stack*), או – ממצביעים במרחב הזיכרון הסטטי (משתנים סטטיים) מוגדרים כ"זבל".



Garbage vs. “Nothing points to it”

- בדוגמא הנתונה: Norfolk, Raleigh הם "זבל", אלא אם כן הושמט פוינטר המצביע אליהם ומאפשר אליהם גישה.
- סטטוס "זבל" שונה מסטטוס "שום דבר אינו מצביע אליו"
- Raleigh בסטטוס "זבל" למרות שקיימת אליו הצבעה

Garbage Collection (GC)

- קביעה האם אלמנט במרחב הזיכרון הדינאמי עובר לסטטוס "זבל" היא קשה דייה, באופן ששפות תכנות רבות לוקחות משימה זו מן המתכנת
- תומכי בקרת הריצה עבור שפות אלו מספקים *automatic garbage collection*
- מתכנתי JAVA עושים שימוש בפוינטרים רבים (יותר ממתכנתי ++C...) אבל מנגנון ה-GC דואג לנקות אחריהם.

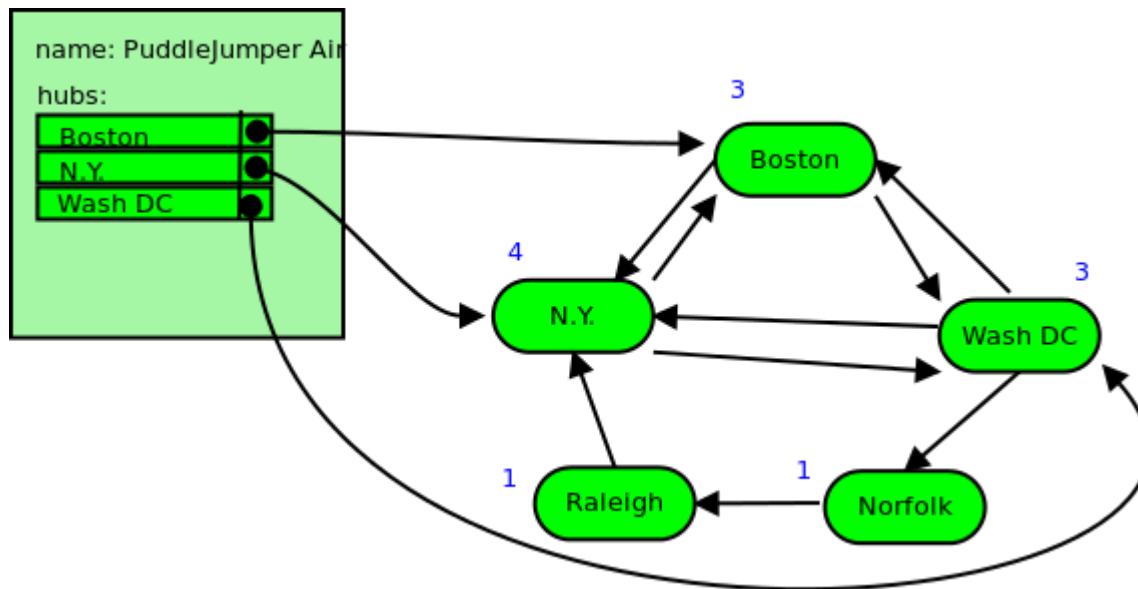
Reference Counting (RC)

RC היא אחת מהטכניקות הבסיסיות למימוש GC:

- עבור כל אובייקט ב-*heap*, נמנה את מספר המצביעים אליו
- בעת השמה של מצביע אליו לכתובת אחרת, המונה יקטן באחד
- בעת השמה של מצביע אל כתובתו, המונה יגדל באחד
- כאשר המונה מקבל ערך 0, האובייקט מפורק

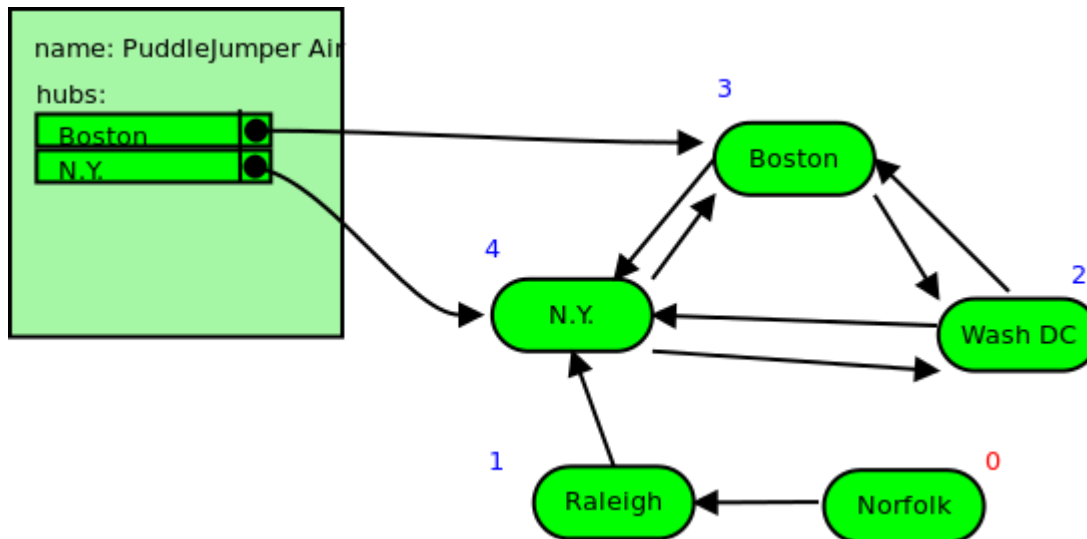
RC Example (1)

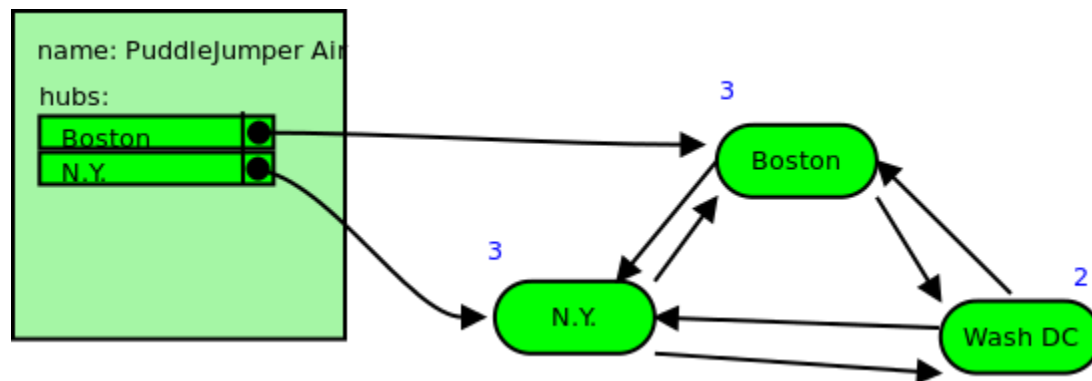
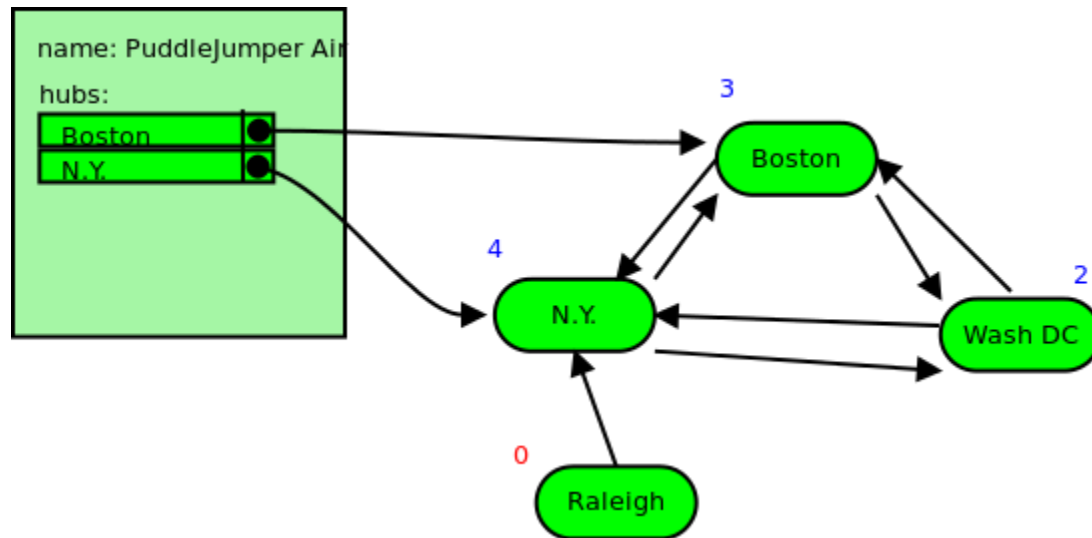
- הסרת DC מרשימת ה-hubs לא תגרוור את מחיקת האובייקט לחלוטין:



RC Example (2)

- ביטול הטיסות DC→Norfolk תגורר את הסרת
:Norfolk,Raleigh





```

template <class T>
class RefCountPointer {
    T* p;
    unsigned* count;
    void checkIfScavengable() {
        if (*count == 0) {
            delete count;
            delete p;
        }
    }
public:
    RefCountPointer (T* s) : p(s), count(new unsigned) {*count = 1;}
    RefCountPointer (const RefCountPointer& rcp) :
        p(rcp.p), count(rcp.count)
    { ++(*count); }
    ~RefCountPointer() { --(*count); checkIfScavengable(); }
    T& operator*() const {return *p;}
    T* operator->() const {return p;}

```

```

RefCountPointer& operator= (const RefCountPointer& rcp) {
    ++(*rcp.count);
    --(*count);
    checkIfScavengable();
    p = rcp.p;
    count = rcp.count;
    return *this;
}

bool operator== (const RefCountPointer<T>& ptr) const
{return ptr.p == p;}

bool operator!= (const RefCountPointer<T>& ptr) const
{return ptr.p != p;}

};

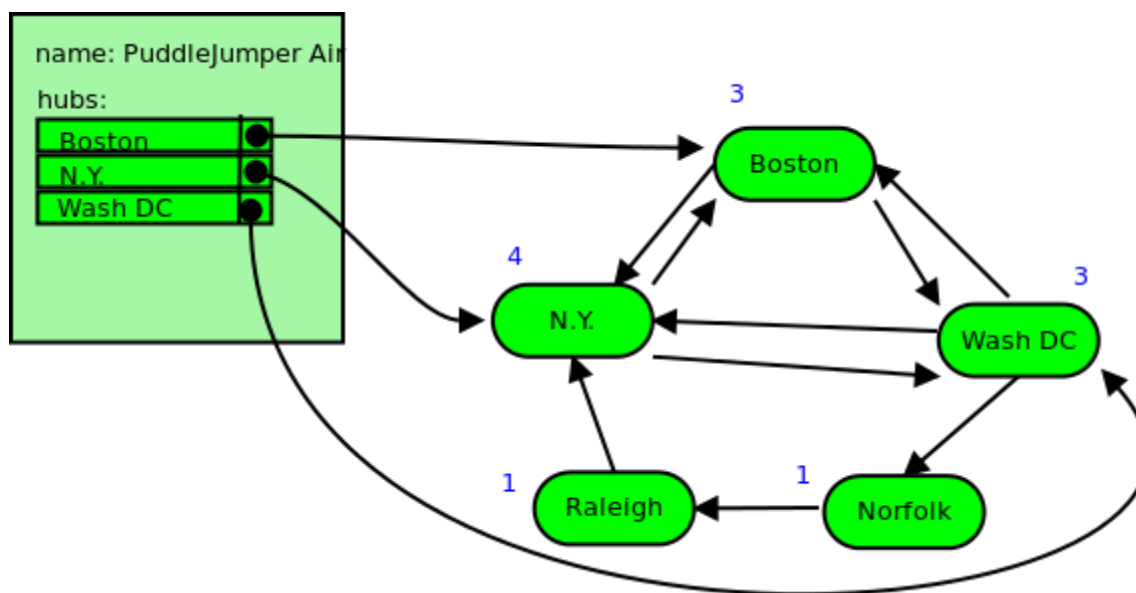
```

Remaining **TODOs**:

1. Null pointers
2. Const pointers

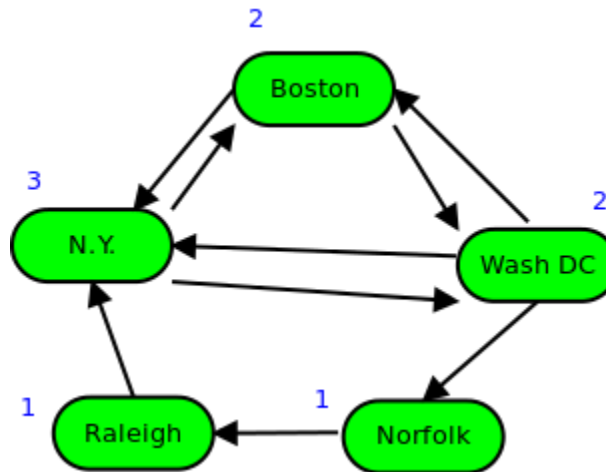
מחיקת Airline במודל RC הנוכחי

- נניח כי מופע Airline הוא משתנה מקומי בפונקציה, והתכנית עומדת לחזור מפונקציה זו



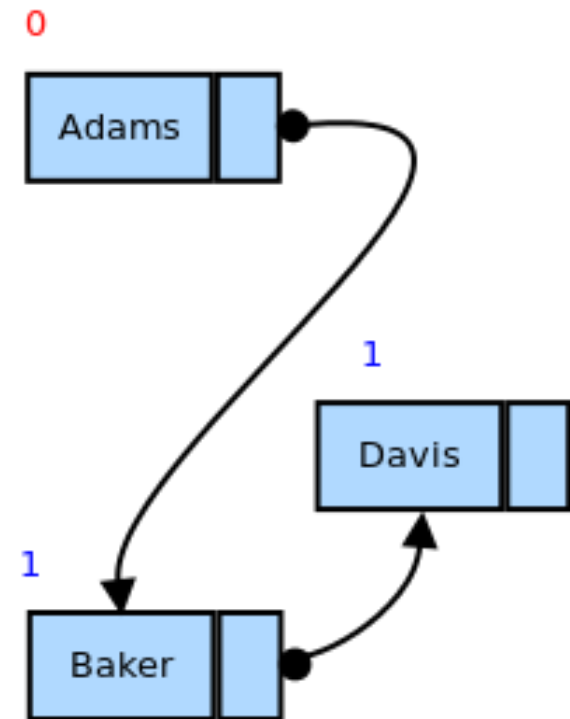
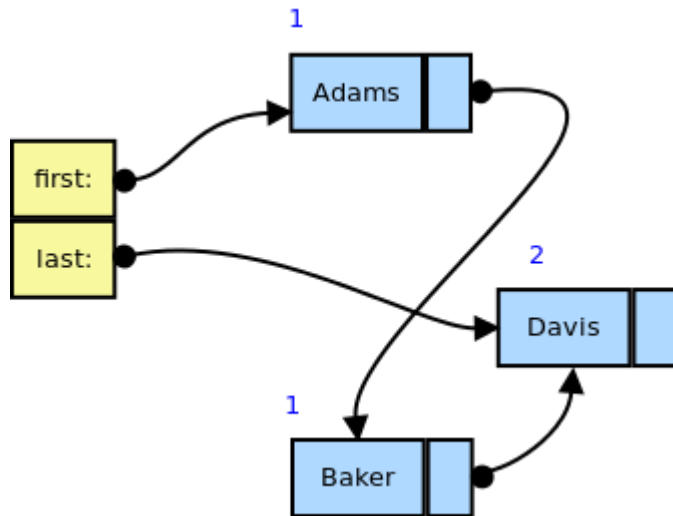
- מופע זה יימחק ואיתו המצביעים לשלושת ה-hubs

התוצאה: דליפת אובייקטי Airport

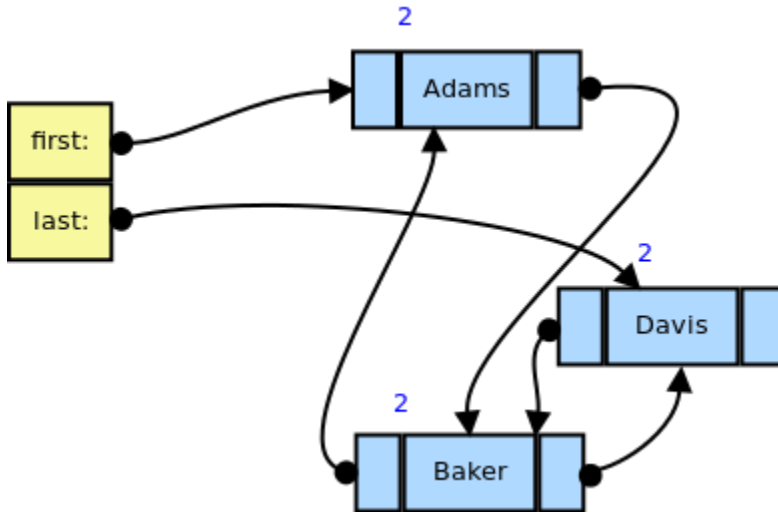


• מה השתבש כאן?

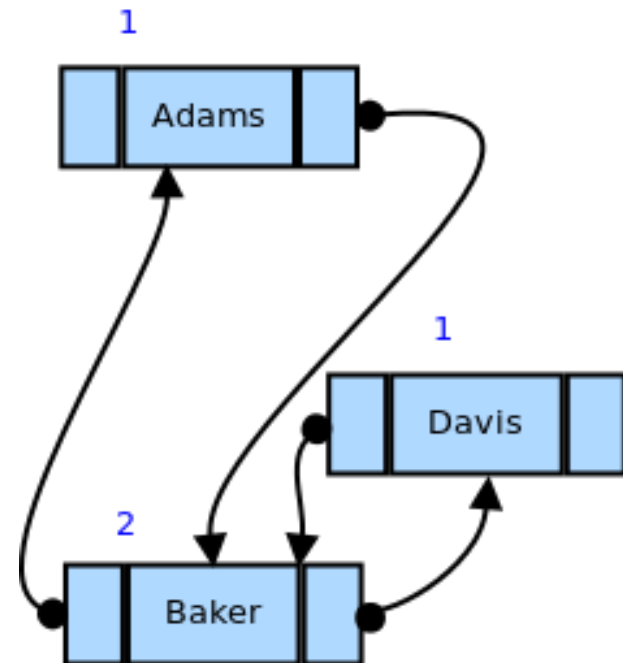
Ref Counted SLL



Ref Counted DLL



None of the reference counters have gone to zero, so nothing will be scavenged, even though all three nodes are garbage.



עקב אכילס של RC

- ש: מהו הגורם המשותף לכשלונות RC הללו?
- ת: מעגלים בגרף ! (*cyclic garbage*)
- אם הפוינטרים יוצרים מעגל הצבעה, האובייקטים במעגל זה לא יוכלו לאפס את ה-RC והגישה תיכשל
- נדרשת גישה כללית יותר!

Mark and Sweep

- אחד מאלגוריתמי ה-GC הראשונים והידועים
 - GC שעובד היטב עם מעגלים, אבל
 - דורש תמיכה משמעותית מהקומפיילר ומבקרת הריצה
- הנחות:
 - לכל אובייקט על ה-*heap* יש ביט סמן סמוי
 - כל המצביעים אל ה-*heap* זמינים
 - עבור כל אובייקט על ה-*heap* ניתן לאתר את כל המצביעים
 - ניתן לעבור בצורה שיטתית על כל האובייקטים על ה-*heap*

```

void markAndSweep() {
    // mark
    for (all pointers P on the run-time stack or in the static data area ) {
        mark *P;
    }
    //sweep
    for (all objects *P on the heap) {
        if *P is not marked then
            delete P
        else
            unmark *P
    }
}

template <class T>
void mark(T* p) {
    if *p is not already marked {
        mark *p;
        for (all pointers q inside *p) {
            mark *q;
        }
    }
}

```

ניתוח האלגוריתם

האלגוריתם פועל בשני שלבים:

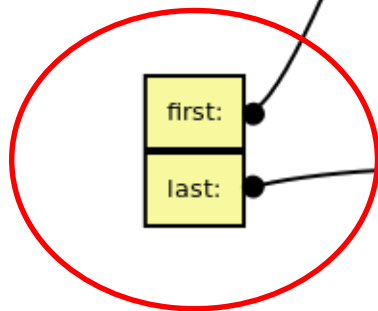
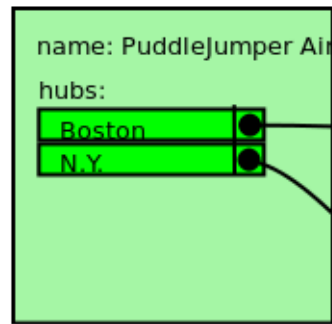
- בשלב הראשון, עבור כל פוינטר מחוץ ל-*heap*, נסמן באופן רקורסיבי כל אובייקט הניתן להגעה באמצעות מצביע זה.

– במונחים של גרפים: *depth-first traversal*

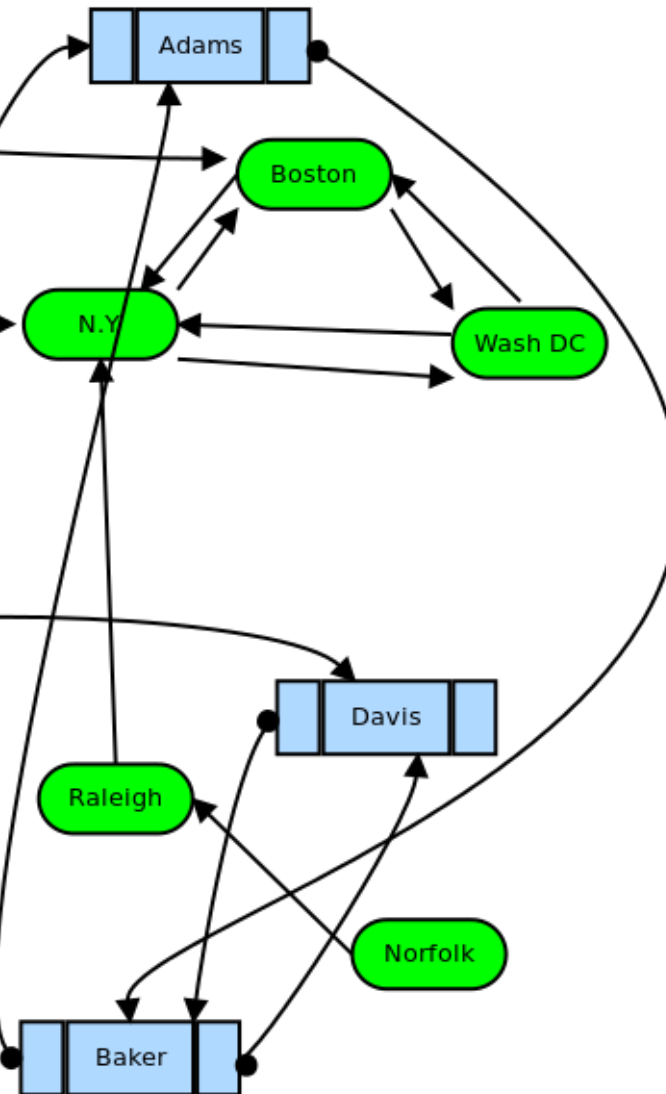
- בשלב השני, נבחן את כל האובייקטים על ה-*heap*:
 - אם הוא מסומן, ניתן להגיע אליו ממצביע מחוץ ל-*heap*, לכן הוא איננו "זבל" וניתן להניח לו
 - אם האובייקט על ה-*heap* איננו מסומן, מדובר ב"זבל" למחיקה

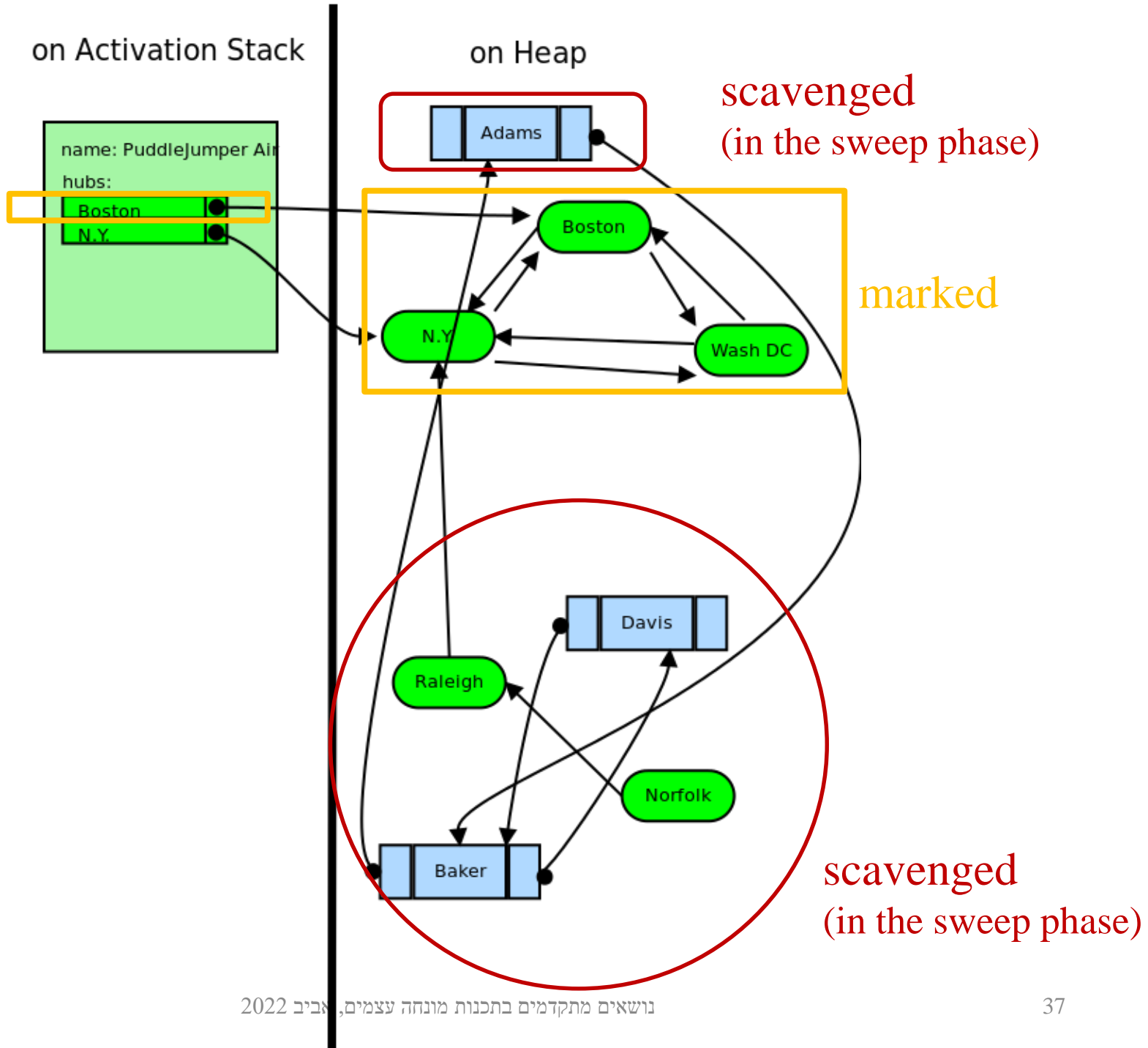
on Activation Stack

on Heap



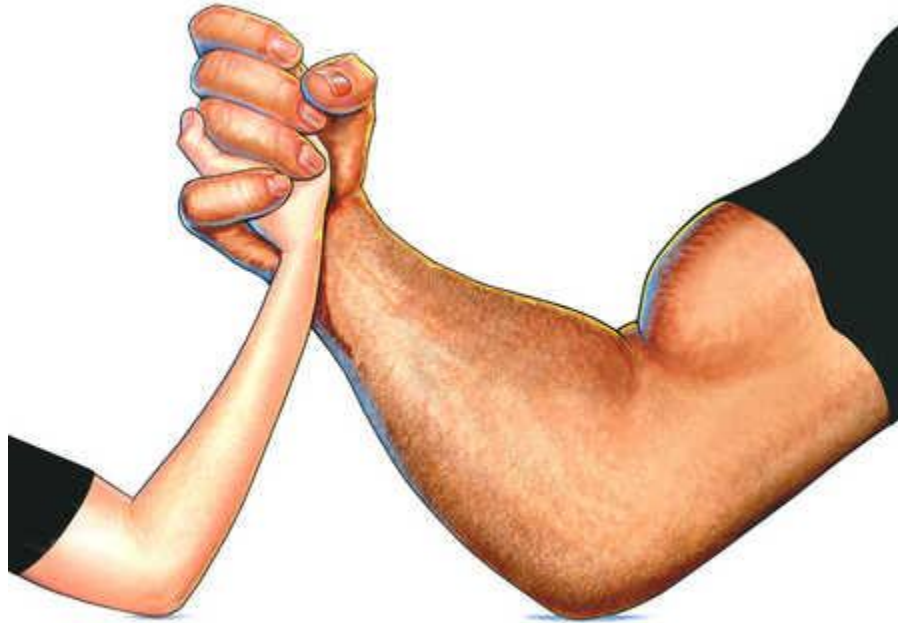
נניח כי ראש הרשימה הינו
משתנה לוקאלי שנמחק





ביקורת

- האופי הרקורסיבי עשוי לדרוש נפח מחסנית גדול – הקריאות ל- `mark()` עשויות לצלול לעומקי רקורסיה
- אפילו תחת שיפורים (גרסה איטרטיבית), מערכות המשתמשות באלגוריתם זה נחשבות לאיטיות – אחת הבעיות היא מעבר על כל אובייקט על ה-*heap*



Strong and Weak Pointers

Source: Steven Zeil, CS361 ODU.EDU

OK, garbage collection is great if you
can get it....

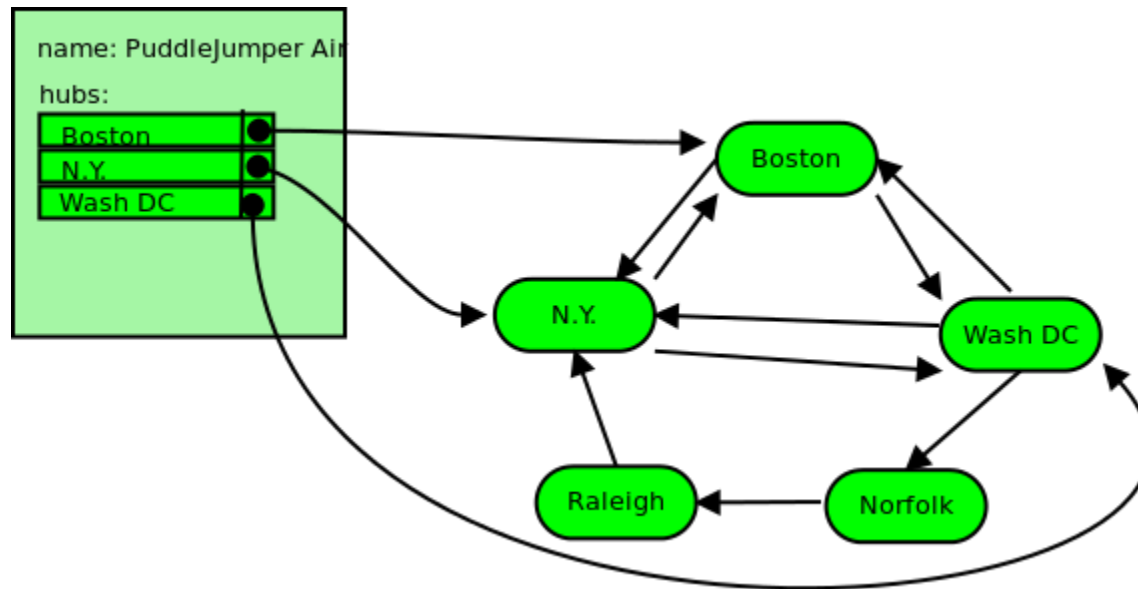
- אבל C++ אינה מספקת GC (וכנראה לעולם לא תלך לכיוון הזה...)
- קומפיילרים של C++ אפילו לא מספקים את התמיכה הנדרשת למימוש Mark&Sweep או למימוש אלגוריתמי GC מתקדמים יותר.
- אז מה יכולים לעשות מתכנתי C++ בהינתן מבני נתונים החולקים בלוקים של זיכרון במרחב הדינאמי (*heap*)?

Ownership

- גישה בסיסית תהיה לזהות אילו טיפוסים נתונים מופשטים (ADT) הם הבעלים של הזיכרון המשותף, ואילו ADT בקושי עושים בו שימוש.
- בעלים של זיכרון משותף נושאים באחריות ליצור אותו, לשתף מצביעים אחרים אליו, ולמחוק אותו.
- ADT אחרים, השותפים לזיכרון אך אינם מחזיקים בבעלות, אינם אמורים ליצור או למחוק מופעים של הזיכרון הזה.

Airport revisited

- בדוגמה של אובייקט חברת התעופה, ראינו שאם אובייקטי Airport בצד שמאל ובצד ימין ימחקו את הפוינטרים אליהם מצביעים, התכנית תקרוס:



Airline *owns* Airport descriptors

```
class Airport {
    :
private:
    vector<Airport*> hasFlightsTo;
};

Airport::~Airport() {
    // for (int i=0; i < hasFlightsTo.size();++i) delete hasFlightsTo[i];
}

class AirLine {
    :
    string name;
    map<string, Airport*> hubs;
};

AirLine::~Airline() {
    for (map<string,Airport*>::iterator i=hubs.begin();
        i != hubs.end(); ++i)

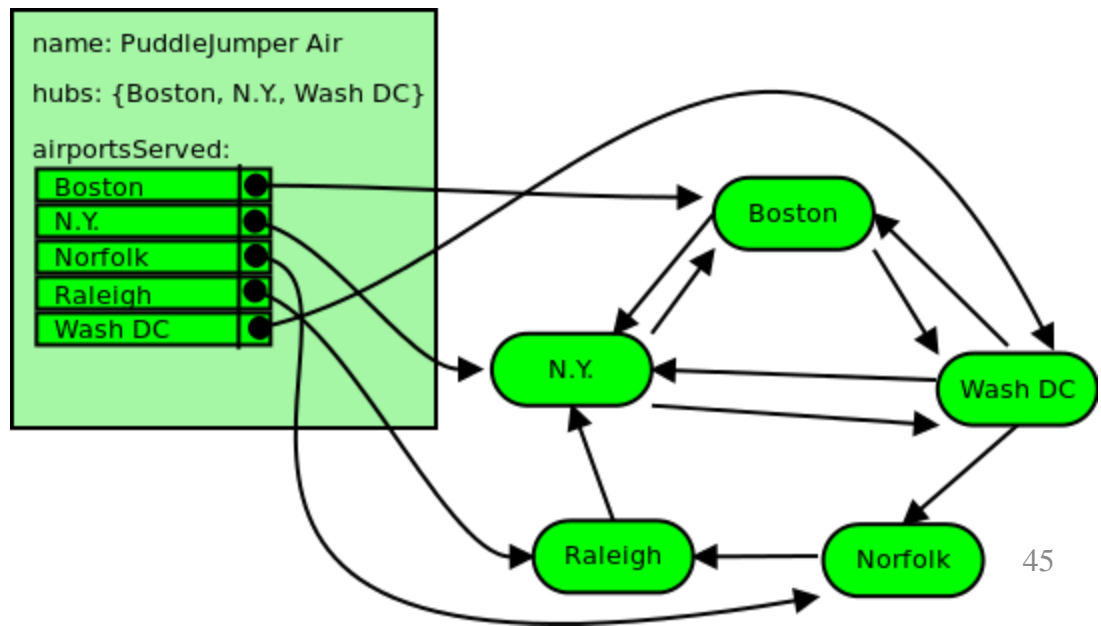
        delete i->second;
}
28/03/2022
```

This solution isn't perfect...

- שדות התעופה Norfolk, Raleigh צריכים שניהם להימחק ("זבל") אך זה לא יתרחש כאן (דלף!); זאת מפני שיש שדות תעופה עליהם אין בעלות...
- גישת פתרון אפשרית תהיה שינוי הדיזיין המקורי, כך שאובייקט חברת התעופה ישלוט טוב יותר על שדות התעופה באמצעות הפרדת מבני נתונים:
 - קבוצה של *hubs*
 - מפה של שדות תעופה בהם החברה נותנת שירות

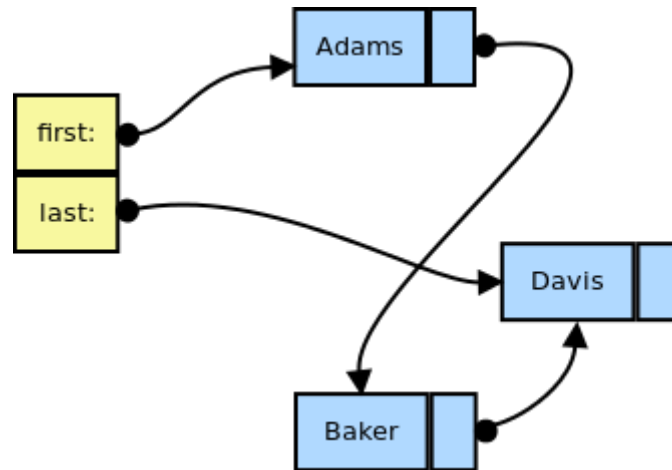
Airline redesigned

```
class AirLine {  
    :  
    string name;  
    set<string> hubs;  
    map<string, Airport*> airportsServed;  
};  
AirLine::~~Airline() {  
    for (map<string,Airport*>::iterator i= airportsServed.begin();  
i != airportsServed.end(); ++i) delete i->second;  
}
```



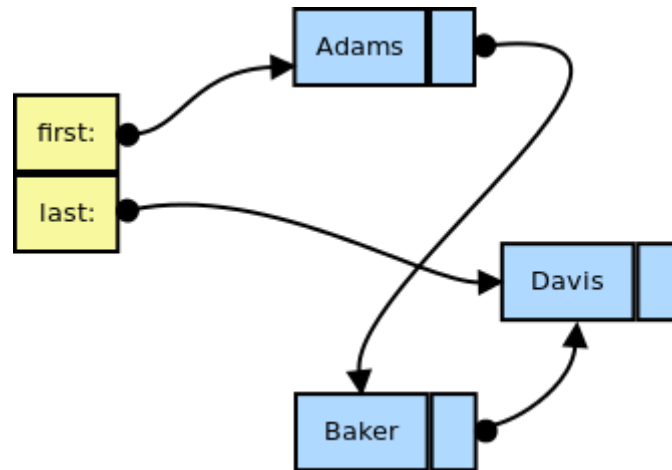
Ownership can be too strong (1)

בחזרה לרשימה המקושרת, אם נאמר שצומת ראש הרשימה מחזיק בבעלות הצמתים אליהם הוא מצביע, אז נמחוק את הצמתים הראשון והאחרון אך הצומת Baker לא יימחק, ויישאר דולף במרחב הדינאמי:



Ownership can be too strong (2)

ואם נאמר שכל צומת מחזיק בבעלות הצומת עליו
הוא מצביע, ובנוסף צומת ראש הרשימה מחזיק
בבעלות הצמתים עליהם הוא מצביע, אז נמחוק את
הצומת האחרונה פעמיים:



Strong and Weak Pointers

נכליל את מושג הבעלות ע"י הגדרת ההצבעה כחזקה או כחלשה:

- מצביע חזק הוא שדה מחלקה פוינטר המצביע על אובייקט אשר חייב להישאר בזיכרון
 - מצביע חלש הוא שדה מחלקה פוינטר אשר מותר לו להצביע על אובייקט שעשוי (היה) להימחק
- ← כאשר אובייקט המחזיק פוינטרים כשדות מחלקה יפורק – הוא ימחוק את המצביעים החזקים שלו ולא יפעל על המצביעים החלשים שלו.

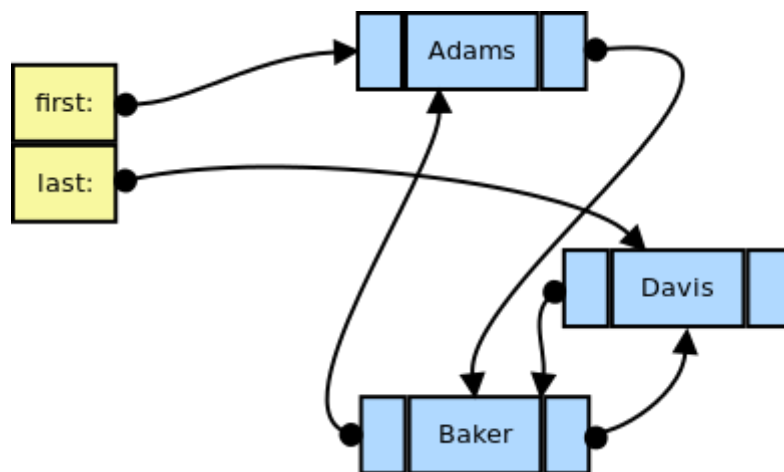
Strong and Weak SLL

```
struct SNode {
    string data;
    SNode* next; // strong
    :
    ~SNode () {delete next;}
};

class List {
    SNode* first; // strong
    SNode* last;  // weak
public:
    :
    ~List()
    {
        delete first; // OK, because this is strong
        /*delete last;*/ // Don't delete. last is weak.
    }
};
```

Picking the strong ones

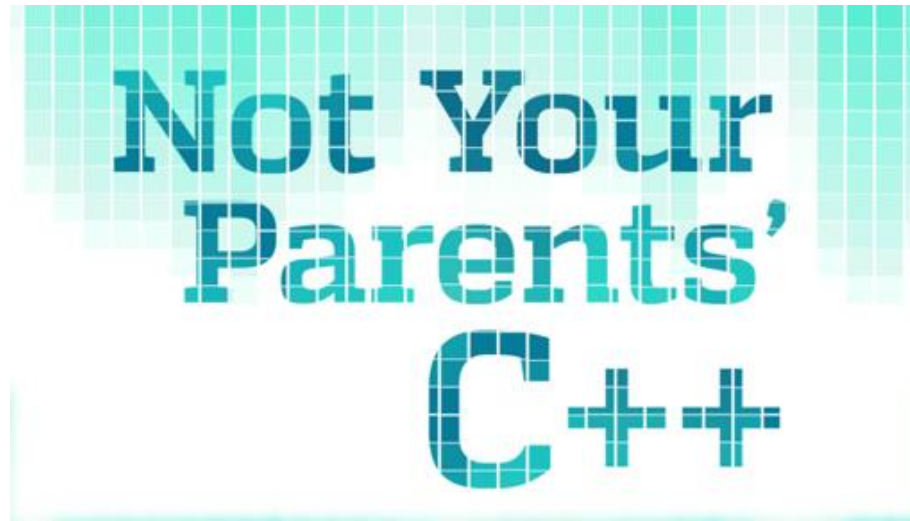
המפתח הוא לזהות את קבוצת הפוינטרים המינימלית שיש ביכולתה לחבר יחדיו את כל האובייקטים המוקצים – ובמקביל מספקים בדיוק מסלול אחד ויחיד לכל אובייקט כזה.



Strong and Weak DLL

```
struct DLNode {
    string data;
    DLNode* prev; // weak
    DLNode* next; // strong
    :
    ~DLNode () { delete next; }
};

class List {
    DLNode* first; // strong
    DLNode* last;  // weak
public:
    :
    ~List() {delete first;}
};
```



C++0x

Smart -> Pointers*

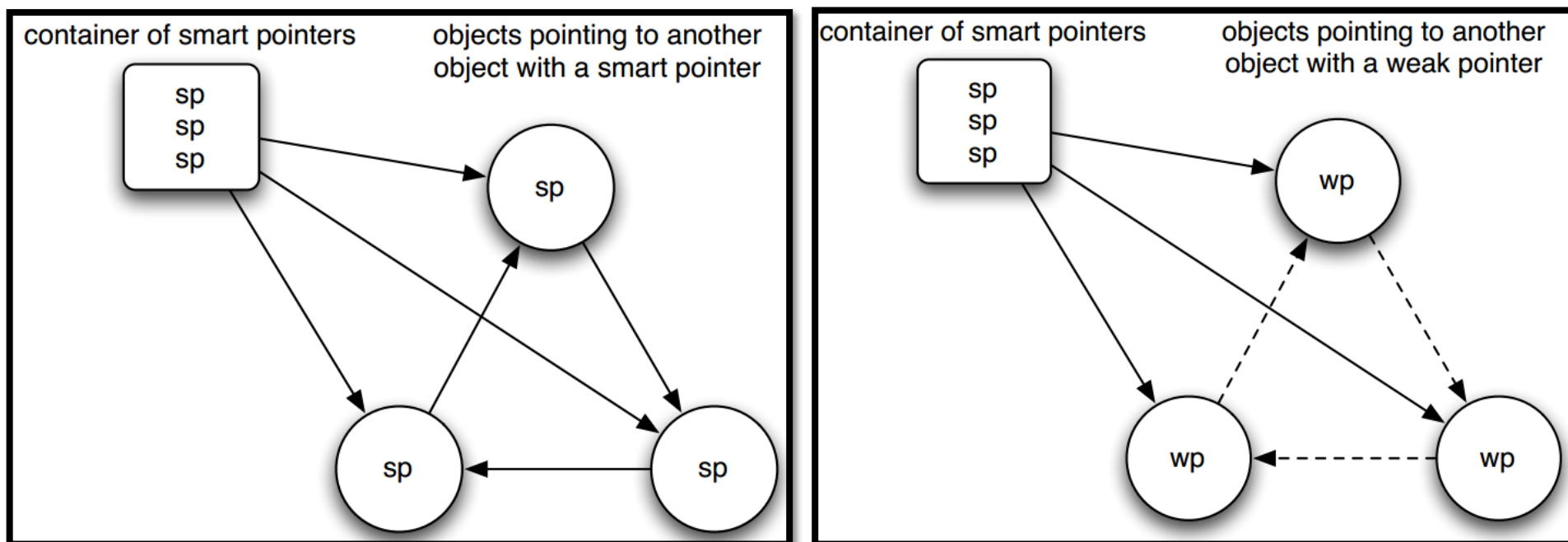
Source: David Kieras, EECS U-Michigan

C++0x Proudly Presents:

- `std::shared_ptr`
 - `std::weak_ptr`
 - `std::unique_ptr`
- * `#include<memory>`

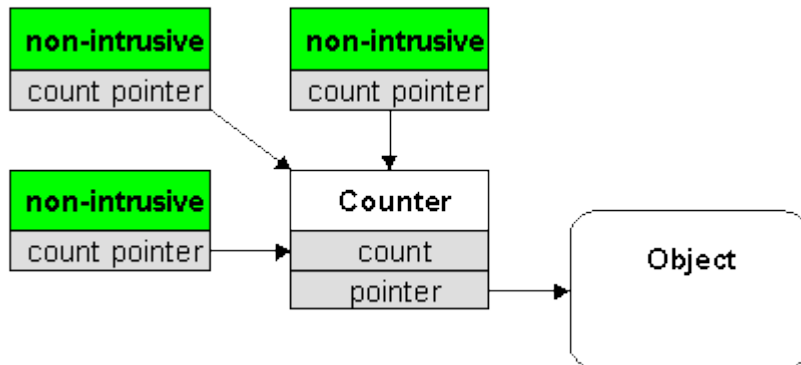
Background

תסריטים בעייתיים של מצביעים חכמים עם RC
בהצבעה מעגלית (תרשים שמאל) יכולים להיפתר כאשר
המצביעים החכמים מתנהגים היטב ע"פ שני טיפוסים
מוגדרים – חזק (sp) וחלש (wp) (תרשים ימין):

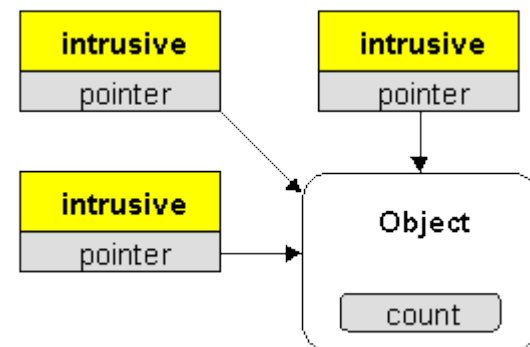


Intrusive vs. Non-intrusive RC

Non-Intrusive Pointers

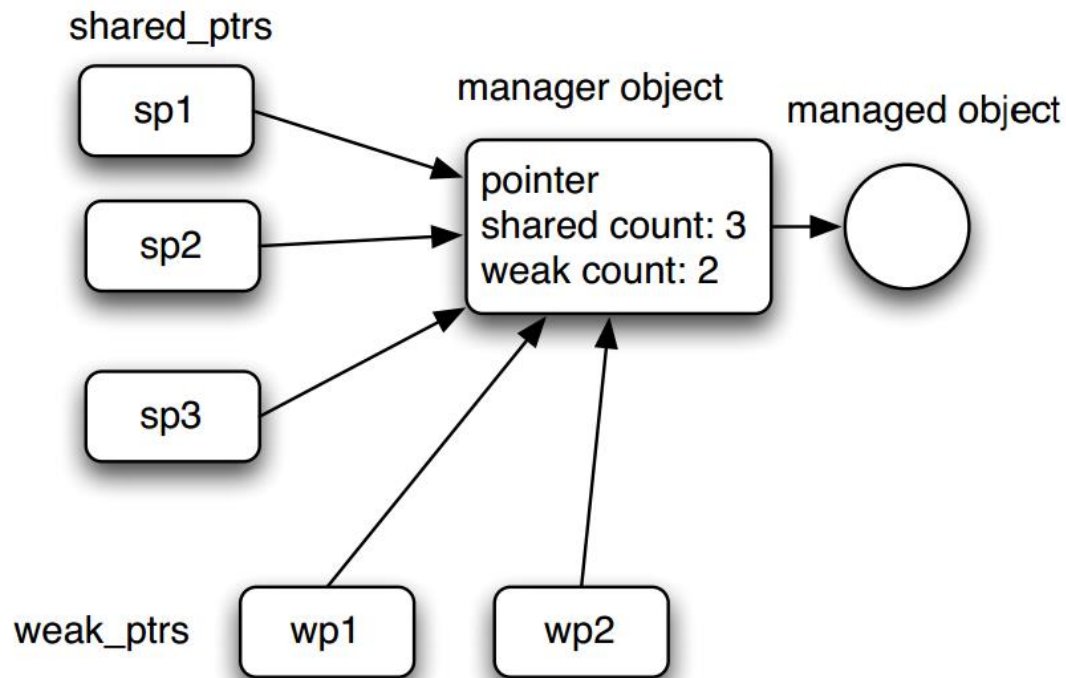


Intrusive Pointers



C++0x Solution

- קבוצת Boost (boost.org) השקיעה מאמצים גדולים לאורך זמן רב על מנת להבטיח מצביעים חכמים המתנהגים היטב בגישה *non-intrusive*:



shared and weak ptrs

- **`shared_ptr<T>` - מחלקת תבנית עבור מצביע חכם**
***reference counting* עם**
 - ניתנים להשמה חופשית בין מופעים של המחלקה
 - אם כל המצביעים מטיפוס זה יפורקו או יקבלו השמה להצבעה אחרת, ערך ה-RC יתאפס והאובייקט יימחק
- **`weak_ptr<T>` - מחלקת תבנית עבור מצביע חכם**
חלש עם *reference counting*, אבל
 - יצירה, העתקה, השמה או מחיקה/פירוק של טיפוס זה אינו משפיע על ערך ה-RC
- מצביעים חכמים חזקים וחלשים ניתנים להעתקה בינם לבין עצמם.

DLL revisited by `std C++0x`

```
struct DLNode {
    string data;
    weak_ptr<DLNode> prev;
    shared_ptr<DLNode> next;
    :
    ~DLNode ()
    {
        // do nothing - the reference counting will take care of it
    }
};

struct List {
    shared_ptr<DLNode> first;
    weak_ptr<DLNode> last;
    :
    ~List() { /* do nothing */ }
};
```

Client Code

```
void addToFront (List& list, string newData)
{
    shared_ptr<DLNode> newNode (new DLNode());
    newNode->data = newData;
    if (list.first == shared_ptr<DLNode>()) //equivalent to null
    {
        // list is empty
        list.first = list.last = newNode;
    }
    else
    {
        newNode->next = list.first;
        list.first->prev = newNode;
        list.first = newNode;
    }
}
```

אילוצים ומגבלות בשימוש (1)

- ניתן להשתמש במצביעים חכמים אלו רק עבור אובייקטים המוקצים דינאמית על ה-*heap* באמצעות **new** ואשר יכולים להימחק באמצעות **delete**.
- יש לוודא שיש רק *manager* יחיד לכל אובייקט. הדרך להשיג זאת היא יצירת **shared_ptr** מיד עם יצירת האובייקט:

new object as argument for a **shared_ptr** c'tor –
make_shared function template –

כאשר כל **shared_ptr** או **weak_ptr** הנדרשים להצביע אליו יועתקו או יקבלו השמה מהעותק המקורי.

אילוצים ומגבלות בשימוש (2)

- יש להימנע ככל האפשר משימוש מקביל בפוינטרים "טיפשים" להצבעה על אותם אובייקטים, מפני שצפויים תסריטים בעייתיים:

dangling pointers –

double deletions –

* קיימת פונקציית מחלקה `get()` המחזירה את הפוינטר הגולמי – השימוש בה לרוב אינו הכרחי!

`std::shared_ptr`

- **c'tors, d'tor, operator=**
 - Default c'tor generates an empty object, the equivalent to `nullptr`
- **reset()**
 1. `void reset() noexcept;`
 2. `template <class U> void reset (U* p);`

For (1) the object becomes *empty* (as if default-constructed). For (2), the `shared_ptr` acquires ownership of `p` with a `use_count` of 1

`std::shared_ptr`

- `long int use_count() const noexcept;`
Returns the number of `shared_ptr` objects that share ownership over the same pointer as this object (including it). If this is an *empty* `shared_ptr`, the function returns 0.

```

struct Thing {
    void dada();
};

ostream& operator<< (ostream&, const Thing&);
// a function can return a shared_ptr:
shared_ptr<Thing> locate();
// takes a shared_ptr parameter by value:
shared_ptr<Thing> boogie(shared_ptr<Thing> p);
//...

void foo() {
    shared_ptr<Thing> p1(new Thing);
    shared_ptr<Thing> p2 = p1; // p1&p2 now share ownership
    //...

    shared_ptr<Thing> p3(new Thing); // another Thing
    p1 = locate(); // p1 may no longer point to first
    boogie(p2);
    p3->dada(); // call a member function
    cout << *p2 << endl; // dereferencing
    p1.reset();
    p2 = nullptr;
}

```


Bad Idea

```
Thing* bad_idea() {
    shared_ptr<Thing> sp; // an empty pointer
    Thing* raw_ptr = new Thing;
    sp = raw_ptr; // Compilation ERROR
    //...
    return raw_ptr; // RISKY - caller could make a mess
}

shared_ptr<Thing> better_idea() {
    shared_ptr<Thing> sp(new Thing);
    //...
    return sp;
}
```

Another Bad Idea

```
Thing* another_bad_idea() {  
    shared_ptr<Thing> sp(new Thing);  
    Thing* raw_ptr = sp; // Compilation ERROR  
  
    Thing* raw_ptr = sp.get();  
    // you should have a good reason to do this!  
    //...  
    return raw_ptr;  
    // RISKY - caller could make a mess  
}
```

Inheritance and **shared_ptr**

The following code of built-in pointers,

```
class Base {};  
class Derived : public Base {};  
// ...  
Derived * dp1 = new Derived;  
Base * bp1 = dp1;  
Base * bp2(dp1);  
Base * bp3 = new Derived;
```

should well-behave similarly:

```
shared_ptr<Derived> dp1(new Derived);  
shared_ptr<Base> bp1 = dp1;  
shared_ptr<Base> bp2(dp1);  
shared_ptr<Base> bp3(new Derived);
```

Casting a `shared_ptr`

C++0x מספקת פונקציות תבנית המבצעות המרות כמו במצביעים הטיפשים (המימוש מסתמך על `get()`):

```
shared_ptr<Base> base_ptr (new Base);  
shared_ptr<Derived> derived_ptr;  
/* if static_cast<Derived *>(base_ptr.get())  
is valid, then the following is valid: */  
derived_ptr =  
static_pointer_cast<Derived>(base_ptr);
```

בנוסף –

- `dynamic_pointer_cast` •
- `const_pointer_cast` •

`std::weak_ptr`

- הטיפוס `weak_ptr` הוא בגדר "משקיף", ללא אחריות או מעורבות בגורלו של האובייקט עליו מצביעים – אין לו בעלות.
- ברמת הדיזיין – חשיבותו בשבירת המעגלים
- ברמת הפונקציונליות – הוא מוגבל ביותר:
 - הוא מיועד לבדוק באם האובייקט עודנו חי ולספק `shared_ptr` בכדי לעבוד עליו במקרה הצורך
 - הוא משולל יכולת ל-`dereference`; המתודות `operator*`, `operator->`, `get()` אינן קיימות

Initializing a `weak_ptr`

- בנאי ברירת-מחדל יוצר מופע ריק המצביע לכלום (הוא אפילו אינו מצביע ל-*manager object*)
- ניתן להצביע לאובייקט רק באמצעות בנאי העתקה או אופרטור השמה מעתיק ממופעים קיימים של `weak_ptr` או `shared_ptr`

```
shared_ptr<Thing> sp(new Thing);  
weak_ptr<Thing> wp1(sp); //copy c'tor from a shared_ptr  
weak_ptr<Thing> wp2; // an empty weak_ptr  
wp2 = sp; // wp2 now points to the new Thing  
weak_ptr<Thing> wp3 (wp2); // copy c'tor  
weak_ptr<Thing> wp4;  
wp4 = wp2; // wp4 now points to the new Thing
```

Using a **weak_ptr** to refer to an object

- כאמור, טיפוס זה משולל יכולת *dereferencing*; יש קודם כל לקבל ממנו טיפוס **shared_ptr** באמצעות הפונקציה **lock()**
- פונקציה זו בודקת את מצב ה-*manager object*:
 - אם אינו קיים, יוחזר **shared_ptr** ריק
 - אם קיים, יוחזר מופע **shared_ptr** המצביע אליו
- כשמה כן היא – נועלת את ה-*manager object*, באם קיים, כל עוד עובדים עליו.
- בהמשך לקוד של השקף הקודם:

```
shared_ptr<Thing> sp2 = wp2.lock();
```

- אך מה קורה באם ה-*manager object* אינו קיים?

Solutions #1,#2: Test/Ask

```
void do_it(weak_ptr<Thing> wp) {  
    shared_ptr<Thing> sp = wp.lock();  
    if(sp)  
        sp->dada(); // tell the Thing to do something  
    else  
        cout << "The Thing is gone!" << endl;  
}
```

```
bool is_it_there(weak_ptr<Thing> wp) {  
    if(wp.expired()) {  
        cout << "The Thing is gone!" << endl;  
        return false;  
    }  
    return true;  
}
```


Solution #3: Catch it

```
void do_it(weak_ptr<Thing> wp) {  
    shared_ptr<Thing> sp(wp);  
    // exception thrown if wp is expired, so if  
    here, sp is good to go  
    sp->dada(); // tell the Thing to do something  
}  
//...  
try {  
    do_it(wpx);  
}  
  
catch(bad_weak_ptr&) {  
    cout << "A Thing has disappeared!" << endl;  
}
```

Special Case: Getting a `shared_ptr` for `this`

```
struct Thing {
    void foo();
    void dada();
};
void persist(Thing*);
int main(void) {
    Thing* t1 = new Thing;
    t1->foo();
    //...
    delete t1; // done with the object
}
//...
void Thing::foo() {
    // we need to persist this object
    persist(this);
}
//...
void persist(Thing* ptr) {
    ptr->dada();
    /* et cetera */
}
```

Rewriting using `shared_ptr`

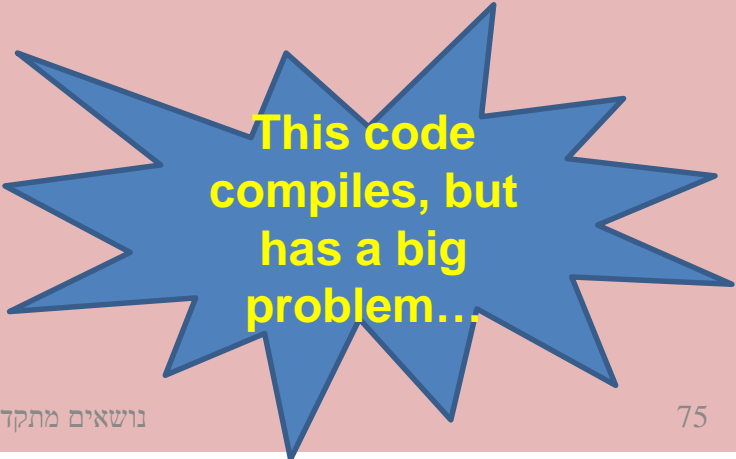
```
struct Thing {
    void foo();
    void dada();
};

void persist(shared_ptr<Thing>);

int main(void) {
    shared_ptr<Thing> t1(new Thing); // start a manager for the Thing
    t1->foo();
    //...
    // Thing is supposed to get deleted when t1 goes out of scope
}
//...

void Thing::foo() {
    // we need to persist this object
    shared_ptr<Thing> sp_for_this(this); // A second manager object!
    persist(sp_for_this);
}
//...

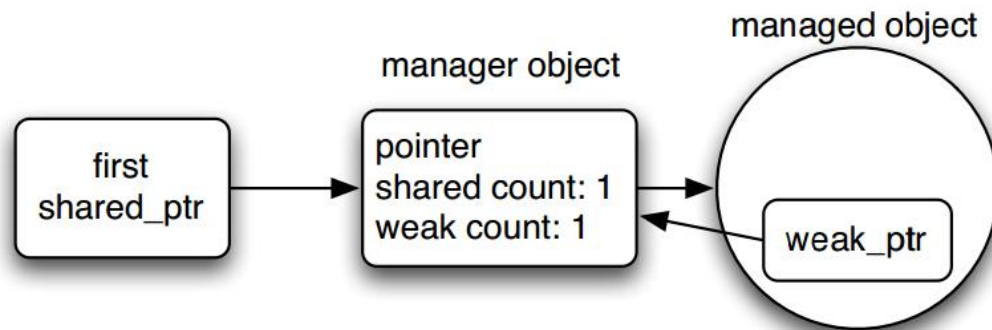
void persist(shared_ptr<Thing> ptr) {
    ptr->dada();
    /* etc. */
}
```



This code
compiles, but
has a big
problem...

הבעיה: כפילות manager object

- ה-*manager object* השני נוצר בפונקציה `foo`, ולכן צפויה מחיקה כפולה והשחתה של מרחב הזיכרון.
- השאיפה היא לפתרון כללי, בו האובייקט עצמו מכיל `weak_ptr` המצביע לאותו *manager object* – עליו מצביע ה-`shared_ptr` המקורי:



std::enable_shared_from_this

```
struct Thing : public enable_shared_from_this<Thing> {
    void foo();
    void dada();
};

int main(void) {
    // The following starts a manager object for the Thing and also
    // initializes the weak_ptr member that is now part of the Thing.
    shared_ptr<Thing> t1(new Thing);
    t1->foo();
    //...
}
//...

void Thing::foo() {
    // we need to persist this object
    // get a shared_ptr from the weak_ptr in this object
    shared_ptr<Thing> sp_this = shared_from_this();
    persist(sp_this);
}
//...

void persist(shared_ptr<Thing> ptr) {
    ptr->dada();
    /* et cetera */
}
```

`std::unique_ptr`

- טיפוס מצביע חכם נוסף בתקן החדש, המחזיק בבעלות בלעדית (*unique ownership*).
 - נבדל מ-`shared_ptr` בשני היבטים:
1. חסכון – אינו דורש *manager object* – משולל כל תקורה בעת יצירתו
 2. בלעדיות – אובייקט יכול להיות בבעלות של `unique_ptr` אחד ויחיד בכל נקודת זמן.

Unique ownership *feat.* deleted functions

- הבלעדיות מתקבלת מעצם ביטולם של בנאי ההעתקה ואופרטור ההשמה המעתיק (!)
- תקן C++0x מבטל פונקציות באמצעות התחביר המיוחד: `=delete;`

```
unique_ptr (const unique_ptr&) = delete;  
unique_ptr& operator=(const unique_ptr&) =  
delete;
```

```
unique_ptr<Thing> p1 (new Thing); // p1 owns the Thing  
unique_ptr<Thing> p2 (p1); // ERROR  
unique_ptr<Thing> p3; // an empty unique_ptr;  
p3 = p1; // ERROR
```

העברת בעלות?

- אסור ליצור עותקים נוספים, כלומר תמיד מתקיים עיקרון הבלעדיות, אך האם ניתן לבצע העברת בעלות?
- התשובה היא חיובית, אך לשם כך נזדקק להכיר את מושג ההעברה (חדש בתקן C++0x) והתחביר הנלווה לו (*move semantics*) – בהרצאה הבאה.