บทที่ 3: การโปรแกรมที่เปลี่ยนไปของ Objective-C จากอดีตถึงปัจจุบัน

เราเริ่มกันด้วยการเปลี่ยนแปลงในส่วนที่ลึกที่สุดกันเสียก่อน นั่นก็คือการเปลี่ยนแปลงที่ระดับของตัวภาษาเอง ซึ่งการเปลี่ยนแปลงในระดับของภาษาทุกครั้งนั้น จะส่งผลกับการเขียนโปรแกรมอย่างมาก ไม่ว่าจะเป็นการคิดถึงรูปแบบการเขียนต่างๆ หรือการจัดระบบระเบียบโครงสร้างของโปรแกรม

ภาษา Objective-C เป็นภาษาที่มีอายุมากกว่า 25 ปีแล้ว และได้มีการเปลี่ยนแปลงเพิ่มเติมความสามารถในระดับของ “ตัวภาษา” (ไม่ใช่ในระดับของ Standard Library หรือชุดคำสั่งมาตรฐาน) หลายครั้ง โดยเฉพาะอย่างยิ่งในช่วง 5-6 ปีล่าสุด ที่มีการเพิ่มเติมความสามารถใหม่ๆ ลงไปมากมาย

สิ่งที่เพิ่มเข้ามาแต่ละครั้งใน Objective-C นั้นอาจถึงขั้น “ล้างตำรา” กันหลายเล่มเลยทีเดียว เพราะว่าเป็นเรื่องที่เป็นหัวใจของการโปรแกรมเชิงวัตถุใน Objective-C มาโดยตลอด นั่นก็คือ “การทำงานกับวัตถุ” โดยเริ่มจากการมี Property ใน Objective-C 2.0 (ท่ามกลางฟีเจอร์อื่นๆ อีกหลายอย่าง) ซึ่งเปลี่ยนการเข้าถึง Data และการเขียน Data Accessor Methods และล่าสุดนี้ก็คือ “การจัดการวัตถุ” ผ่าน Reference Counting โดยในงาน WWDC 2011 ที่ผ่านมา Apple ได้ประกาศรูปแบบการทำงานใหม่ นั่นคือ **Automatic Reference Counting (ARC)** มาแทนที่การบริหารจัดการด้วยตัวเอง โดย ARC นี้เป็นฟีเจอร์มาตรฐานของ LLVM Compiler 3.0 ซึ่งเป็นคอมไพเลอร์ภาษา Objective-C ของ Xcode 4.2

สำหรับบทนี้ ผมจะขอเริ่มต้นจากการทบทวนสิ่งที่เราได้เคยเห็นกันไปแล้วจากหนังสือเล่มแรก นั่นก็คือ “การประกาศและการเขียน Data Accessor Methods” ซึ่งเราจะลุยกันแบบเจาะเวลาหาอดีต จาก Objective-C 1.0 จากนั้นต่อด้วย Objective-C 2.0 ซึ่งเราได้เห็นกันมามากมาย และปิดท้ายด้วยน้องใหม่ล่าสุด สดๆ ร้อนๆ ซึ่งก็คือ ARC ก่อนจะขยายผลจากจุดนี้ ไปสู่รายละเอียดและกฎเกณฑ์ต่างๆ ของ ARC ซึ่งจะมีผลกับการคิดและเขียนโปรแกรมของเรา

เจาะเวลาหาอดีต ผ่าน Data Accessor Methods

ในหนังสือเล่มแรกนั้น ผมได้เขียนเรื่องนี้ไปโดยอ้างอิงกับ Objective-C 2.0 เป็นหลัก นั่นก็คือเริ่มต้นด้วยการใช้ “Property” ซึ่งเป็นความสามารถในการประกาศและสร้าง Data Accessors ให้กับเราตามที่เรากำหนดอย่างอัตโนมัติ (ผ่าน **@property** และ **@synthesize**) และสำหรับตอนนี้เราจะย้อนอดีตกันเล็กน้อย ว่าในอดีตนั้นเราจะต้องทำอะไรเองบ้าง ตัวภาษาและคอมไพเลอร์ช่วยเราได้แค่ไหน

สมมติว่าเรามีโปรแกรม “โปรเจคจบนักศึกษา” ซึ่งมีรูปแบบคือ นักศึกษา (Student) แต่ละคนจะต้องทำโปรเจค (Project) หนึ่งโปรเจค เป็นแบบ 1-to-1 ซึ่งนักศึกษาและโปรแกรมก็จะมีข้อมูลเพิ่มเติมก็คือ “ชื่อ” เท่านั้น โดยทั้งสองวัตถุนี้จะอ้างอิงซึ่งกันและกัน เพื่อตอบคำถามที่ว่า “คนนี้ทำโปรเจคอะไร” และ “โปรเจคนี้ใครเป็นคนทำ”

ย้อนอดีต: Objective-C 1.0

ใน Objective-C 1.0 ก่อนที่จะมีเรื่องของ Property เข้ามาเกี่ยวข้อง เราจะต้องเขียน Student และ Project ซึ่งค่อนข้างตรงไปตรงมา ดังนี้

Student.h

```
#import <Foundation/Foundation.h>
```

```
@class Project;

@interface Student : NSObject {
    NSString *_name;
    Project  *_project;
}

- (NSString *)name;
- (void)setName:(NSString *)name;
- (Project *)project;
- (void)setProject:(Project *)project;

- (id)initWithName:(NSString *)name andProject:(Project *)project;
@end
```

Project.h

```
#import <Foundation/Foundation.h>

@class Student;

@interface Project : NSObject {
    NSString *_name;
    Student *_student;
}

- (NSString *)name;
- (void)setName:(NSString *)name;
- (Student *)student;
- (void)setStudent:(Student *)student;

- (id)initWithName:(NSString *)name;
@end
```

หลังจากประกาศส่วนของ Interface เรียบร้อยแล้ว ก็ถึงเวลาของ Implementation ทั้งหมด ซึ่งเราจะเริ่มพบกับความยุ่งยาก แม้แต่กับโปรแกรมง่ายๆ แค่นี้

โดยเราจะเริ่มจากส่วนของ Student ก่อน ซึ่งส่วนของ Getter นั้น ไม่มีอะไรยุ่งยาก เพียงแค่คืนการอ้างอิงไปยังวัตถุที่เก็บไว้

Student.m

```
- (NSString *)name {
    return _name;
}

- (Project *)project {
    return _project;
}
```

แต่สำหรับ Setter ล่ะ? เราทำแค่เพียงอ้างอิงไปหาเฉยๆ เหมือนโค้ดต่อไปนี้

```
- (void)setName:(NSString *)name {
```

```
    _name = name;
}

- (void)setProject:(Project *)project {
    _project = project;
}
```

ไม่ได้อย่างแน่นอน! เหตุผลก็คือเรื่องของการบริหารจัดการวัตถุ ตามที่เราได้เห็นกันอย่างละเอียดแล้วในหนังสือเล่มแรก (บทที่ 3) ว่าการที่อ้างอิงไปเฉยๆ นั้น มีโอกาสที่จะเกิดปัญหา “วัตถุไม่อยู่แล้ว” หรือที่เรียกเป็นศัพท์เทคนิคว่า Dangling Pointer ขึ้น

ดังนั้นเราก็ต้องมาจัดการมันทีละตัว เริ่มจาก name ก่อน โดยเราจะทำการ copy ชื่อที่รับเข้ามา แล้วให้ name อ้างอิงไปหา โดยที่ก่อนจะอ้างอิงไปหานั้น ก็ release ตัวที่กำลังอ้างอิงไปหาอยู่เสียก่อน โดยที่เราจะต้องระวังว่าชื่อที่ name อ้างอิงไปหาอยู่ กับชื่อที่จะรับเข้ามาเพื่อทำการ copy นั้นไม่ใช่ตัวเดียวกัน เพราะหากว่าเป็นตัวเดียวกันแล้ว การที่เราส่งข้อความ release ไป อาจทำให้ชื่อที่รับเข้ามาหายไปทันทีด้วย

ดังนั้น Setter สำหรับ name ของ Student จะมีหน้าตาดังนี้

Student.m

```
- (void)setName:(NSString *)name {
    if (_name != name) {
        [_name release];
        _name = [name copy];
    }
}
```

และสำหรับ Project นั้น เราไม่ต้องการ copy มาไว้อีกชุดหนึ่ง เราต้องการให้ Student ทำการอ้างอิงไปหา และมีความเป็นเจ้าของ Project นั้นๆ ดังนั้นเราจึงทำการอ้างอิงไปหา และทำการ retain เอาไว้ โดยที่เราต้อง release ตัวที่ project ปัจจุบันอ้างอิงไปหาอยู่ด้วย โดยที่เราทำการ retain วัตถุ Project ที่รับเข้ามาก่อนที่จะ release ตัวที่อ้างอิงไปหาอยู่ในปัจจุบัน เพื่อเป็นการป้องกันปัญหา Dangling Pointer เพราะหากวัตถุที่รับเข้ามาเป็น Project เดียวกันกับที่อ้างอิงอยู่ การ release ก่อน จะทำให้ตัวที่รับเข้ามาหายทันที ดังนั้นการเพิ่ม retain ไปก่อนจึงเป็นการป้องกันปัญหานี้ที่เรียบง่ายที่สุด

สุดท้าย เมื่อเราทำการอ้างอิง project ไปยัง Project ที่รับเข้ามาเรียบร้อยแล้ว ก็ส่งข้อความไปให้มันอ้างอิง Student ที่กำลังทำการอ้างอิงอยู่นี้กลับมาด้วย โดยผ่าน Setter ของ Project (จะเขียนต่อไป)

ดังนั้น Setter สำหรับ project จะมีหน้าตาดังนี้

Student.m

```
- (void)setProject:(Project *)project {
    [project retain];
    [_project release];
    _project = project;

    [_project setStudent:self];
```

```
}
```

เมื่อได้ Getter/Setter เรียบร้อยแล้ว เราก็เขียน Task Method สำหรับตั้งค่าตั้งต้น (Initializer) ให้กับ Student

Student.m

```
- (id)initWithName:(NSString *)name andProject:(Project *)project {
    self = [super init];
    if (self) {
        [self setName:name];
        [self setProject:project];
    }
    return self;
}
```

และลงท้าย Method สำหรับเก็บกวาดตัวเองไม่มีการอ้างอิงอีกต่อไปแล้ว

Student.m

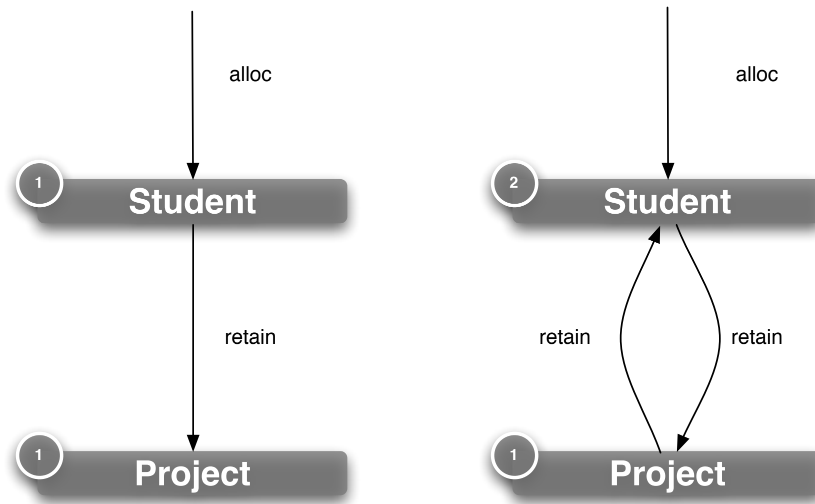
```
- (void)dealloc {
    [_name release];
    [_project release];
    [super dealloc];
}
```

เราอาจเขียนทับ description Method มาตรฐานของ NSObject สำหรับ Student

```
- (NSString *)description {
    return [NSString stringWithFormat:@"%@ (%@): %@",
                [self name], [super description], [self project]];
}
```

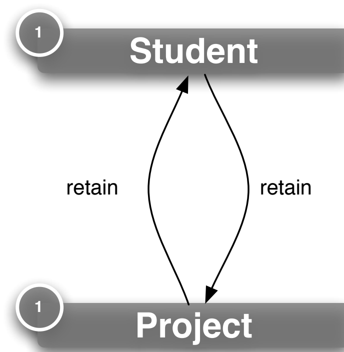
สำหรับ Project ก็คล้ายกันเกือบทั้งหมด เว้นไว้แต่ Setter สำหรับ Project ไปยัง Student เท่านั้น ที่จะไม่ retain วัตถุ Student เอาไว้

ทั้งนี้เนื่องจากหากทั้ง Student และ Project ต่าง retain ซึ่งกันและกันเอาไว้ จะเกิดปัญหาที่เรียกว่า **Retain Cycle** ขึ้น อธิบายดังนี้



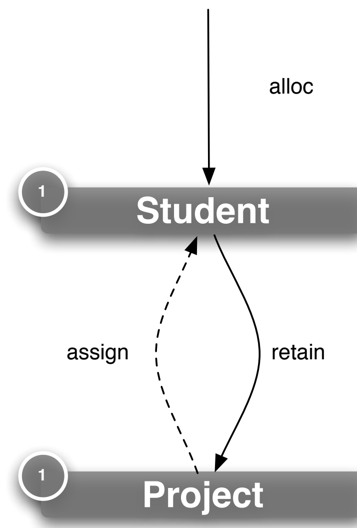
เมื่อเราทำการสร้าง Student ด้วย alloc และอ้างอิงไปหามันนั้น จะมี retain count เป็น 1 และเมื่อ Student อ้างอิงไปหา Project และทำการ retain เอาไว้ ตัว Project นั้นๆ ก็จะมี retain count เพิ่มขึ้นมาอีก 1 และหาก Project นั้นทำการ retain ตัว Student ไว้อีกครั้ง ก็จะทำให้ retain count ของตัว Student เพิ่มขึ้นเป็น 2 ซึ่งเมื่อมองผ่านๆ คร่าวๆ อาจจะมองดูเป็นเรื่องดี เพราะว่าตัว Student จะไม่มีทางหายไปไหนแน่นอน ทั้งจากการอ้างอิงของเรา และจากการอ้างอิงของ Project

แต่ปัญหาจะเกิดขึ้น เมื่อเราทำการส่ง release ให้กับ Student ที่เราทำการอ้างอิงถึง และเปลี่ยนไปอ้างอิงยังวัตถุอื่นเรียบร้อยแล้ว สิ่งที่เกิดขึ้นจะเป็นดังภาพต่อไปนี้



ซึ่งจะเห็นว่าตัว Student และ Project ต่างก็ยังคงมี retain count ซึ่งเกิดจากการ retain กันเองอยู่ แต่ในตัวโปรแกรมของเราไม่มีการอ้างอิงไปยังมันทั้งคู่แล้ว ซึ่งทำให้เกิดปัญหา Memory Leak ขึ้นกับวัตถุทั้งคู่ (เพราะว่าตราบใดก็ตามที่ retain count ของ Student ยังไม่เป็น 0 นั้น dealloc ก็จะไม่ถูกเรียก ทำให้ Project ไม่ได้รับการปล่อย)

ดังนั้น เพื่อไม่ให้เกิดปัญหานี้ขึ้น วิธีการง่ายที่ง่ายที่สุดคือ Project จะต้องไม่ retain Student เอาไว้ เพียงแต่อ้างอิงกลับไปหาเท่านั้น



เป็นหลักการเขียนง่ายๆ ว่า

“วัตถุที่เป็นแม่ (Parent) จะเป็นเจ้าของลูก (Child) แต่ลูกจะไม่เป็นเจ้าของแม่”

ดังนั้นกรณีนี้ Student จะเป็นเจ้าของ Project ผ่าน retain ownership แต่ Project จะไม่เป็นเจ้าของ Student ซึ่งทำให้เราได้ Setter สำหรับ Project ไปหา Student ดังนี้

Project.m

```
- (void)setStudent:(Student *)student {
    _student = student;
}
```

เพื่อให้เห็นภาพทั้งหมดอีกครั้ง ผมขอนำโค้ดในส่วน Implementation ของทั้ง Student และ Project มาแสดงตรงนี้อีกครั้งหนึ่ง

Student.m

```
#import "Student.h"
#import "Project.h"

@implementation Student
- (NSString *)name {
    return _name;
}

- (void)setName:(NSString *)name {
    if (_name != name) {
        [_name release];
        _name = [name copy];
    }
}

- (Project *)project {
```

```
    return _project;
}

- (void)setProject:(Project *)project {
    [project retain];
    [_project release];
    _project = project;

    [_project setStudent:self];
}

- (id)initWithName:(NSString *)name andProject:(Project *)project {
    self = [super init];
    if (self) {
        [self setName:name];
        [self setProject:project];
    }
    return self;
}

- (void)dealloc {
    [_name release];
    [_project release];
    [super dealloc];
}

- (NSString *)description {
    return [NSString stringWithFormat:@"%@ (%@): %@",
            [self name], [super description], [self project]];
}

@end
```

Project.m

```
#import "Project.h"
#import “Student.h”

@implementation Project
- (NSString *)name {
    return _name;
}

- (void)setName:(NSString *)name {
    if (_name != name) {
        [_name release];
        _name = [name copy];
    }
}

- (Student *)student {
    return _student;
}

- (void)setStudent:(Student *)student {
    _student = student;
}

- (id)initWithName:(NSString *)name {
    self = [super init];
    if (self) {
        [self setName:name];
    }
    return self;
```

```
}

- (void)dealloc {
    [_name release];
    [super dealloc];
}

- (NSString *)description {
    return [NSString  stringWithFormat:@"%@ (%@)",
            [self name], [super description]];
}

@end
```

ทดสอบทุกอย่างที่สร้างขึ้นมา โดยการสร้าง Project ขึ้นมา 2 ตัว และนักศึกษา 2 คน และตรวจสอบ retain count ในการอ้างอิงวัตถุ

main.m

```
#import <Foundation/Foundation.h>
#import "Project.h"
#import "Student.h"

int main (int argc, char const *argv[])
{
      NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
      Project *project = [[Project alloc] initWithName:@"Destroy Konoha"];
      Student *student = [[Student alloc] initWithName:@"Uchiha Sasuke" andProject:project];

      Project *project2 = [[Project alloc] initWithName:@"Save Rukia"];
      Student *student2 = [[Student alloc] initWithName:@"Kurasaki Ichigo" andProject:project2];

      [project release];
      [project2 release];

      NSLog(@"%@", student);
      NSLog(@"%@", student2);

      NSLog(@"%@", [[project student] name]);

      NSLog(@"retain counts: project1:%lu project2:%lu student1:%lu student2:%lu",
            [project retainCount], [project2 retainCount],
            [student retainCount], [student2 retainCount]);

      [student release];
      [student2 release];
      [pool drain];
      [pool release];
      return 0;
}
```

และจะได้ผลการรันโปรแกรมดังนี้

```
Uchiha Sasuke (<Student: 0x10e614b40>): Destroy Konoha (<Project: 0x10e614060>)
Kurasaki Ichigo (<Student: 0x10e614bb0>): Save Rukia (<Project: 0x10e614b90>)
Uchiha Sasuke
retain counts: project1:1 project2:1 student1:1 student2:1
```

เหนื่อยไหมครับ กับโปรแกรมง่ายๆ ไม่มีอะไรเลย ในอดีตอันไกลโพ้นเราต้องเขียนกันแบบนี้ตลอด และต่อไปนี้ผมขอพาไปพบกับ “อดีตเมื่อวันวาน” คือ Objective-C 2.0 ที่ยังไม่มี Automatic Referencing Counting (ARC) ครับ

วันวานยังหวานอยู่: Objective-C 2.0

ทวนความจำกันเล็กน้อย จากบทที่ 3 ในหนังสือเล่มแรกของ เราใช้ **@property** ทำการกำหนดคุณสมบัติของ Data Accessors และใช้ **@synthesize** ในการให้คอมไพเลอร์สร้างให้ตามที่เรากำหนด และท้ายบทที่ 3 นั้น ก็มีการพูดถึงว่าสำหรับ iOS และ Mac OS X 64bit นั้น จะมี Objective-C runtime ตัวใหม่ ที่ทำให้ไม่ต้องประกาศวัตถุมารองรับแต่อย่างใด ดังนั้นโค้ดทั้งหมดของเราจะเหลือแค่นี้

Student.h

```
#import <Foundation/Foundation.h>

@class Project;

@interface Student : NSObject
@property (nonatomic, copy) NSString *name;
@property (nonatomic, retain) Project *project;

- (id)initWithName:(NSString *)name andProject:(Project *)project;
@end
```

Project.h

```
#import <Foundation/Foundation.h>

@class Student;

@interface Project : NSObject
@property (nonatomic, copy) NSString *name;
@property (nonatomic, assign) Student *student;

- (id)initWithName:(NSString *)name;
@end
```

และทำการเขียนส่วนของ Implementation ด้วยการให้คอมไพเลอร์ช่วยสร้างให้ ซึ่งจะมีข้อยกเว้นสำหรับกรณี Setter ของ Student ไปยัง Project เนื่องจากจะไม่ใช่แค่การอ้างอิงไปหา และ retain ไว้เท่านั้น แต่จะต้องให้ Project ทำการอ้างอิงกลับมาด้วย ซึ่งกรณีนี้จะ synthesize ตรงๆ ไม่ได้ เนื่องจากการ synthesize นั้นจะเขียนให้เราได้แต่กรณีอ้างอิงไปหาอย่างเดียวเท่านั้น จะไม่จัดการกรณีอ้างอิงกลับให้ ซึ่งเราจะต้องเขียนเอง

ทำให้เราได้โค้ดส่วนของ Implementation ของทั้งสองตัว ซึ่งเป็นโค้ดแบบที่เราจะคุ้นเคยกันมากกว่า ดังนี้

Student.m

```
#import "Student.h"
#import "Project.h"

@implementation Student
@synthesize name = _name;
@synthesize project = _project;

-(void)setProject:(Project *)project {
```

```
    [project retain];
    [_project release];
    _project = project;

    self.project.student = self;
}

- (id)initWithName:(NSString *)name andProject:(Project *)project {
    self = [super init];
    if (self) {
        self.name = name;
        self.project = project;
    }
    return self;
}

- (void)dealloc {
    [_name release];
    [_project release];
    [super dealloc];
}

- (NSString *)description {
    return [NSString stringWithFormat:@"%@ (%@): %@",
            self.name, [super description], self.project];
}
@end
```

Project.m

```
#import "Project.h"
#import "Student.h"

@implementation Project
@synthesize name = _name;
@synthesize student = _student;

- (void)dealloc {
    [_name release];
    [super dealloc];
}

- (id)initWithName:(NSString *)name {
    self = [super init];
    if (self) {
        self.name = name;
    }
    return self;
}

- (NSString *)description {
    return [NSString  stringWithFormat:@"%@ (%@)",
            self.name, [super description]];
}
@end
```

จะเห็นว่า Objective-C 2.0 ได้ย้ายงานหนักที่ต้องอาศัยความละเอียด ความระมัดระวัง แต่มีความซ้ำซ้อนสูง และมีลักษณะการเขียนที่เป็นรูปแบบ ไปเป็นหน้าที่ของคอมไพเลอร์ ที่จะช่วยเราเขียน โดยที่เราต้องเป็นผู้กำหนด ซึ่งเราต้องรู้และเข้าใจว่าจะกำหนดอะไร กำหนดอย่างไร และกำหนดเพื่ออะไร

วันนี้ที่รอคอย: Objective-C 2.0 ARC

การพัฒนาภาษา Objective-C ในลักษณะที่ย้ายงานหนักไปเป็นภาระหน้าที่ของคอมไพเลอร์ และโปรแกรมเมอร์กลายเป็นผู้กำหนดคุณสมบัติ บอกคอมไพเลอร์ให้เขียนโปรแกรมส่วนที่เป็นรูปแบบซ้ำซ้อนแทนนั้น ไม่หยุดอยู่แค่นี้ เพราะว่ายังมีอีกอย่างหนึ่งที่โปรแกรมเมอร์ภาษา Objective-C ปฏิบัติกันมากว่า 25 ปี ตั้งแต่ก่อน NeXTSTEP มาเป็น GNUstep และ Mac OS X จนมาแพร่หลายใน iOS ถึงทุกวันนี้ นั่นก็คือการบริหารจัดการวัตถุด้วยวิธีการแบบ Referencing Counting

ภาษาโปรแกรมหลายภาษามีนำระบบ Garbage Collection (GC) ซึ่งจะคอยจัดการเก็บกวาดวัตถุที่ไม่มีการอ้างอิงแล้ว โดยที่โปรแกรมเมอร์ไม่ต้องทำอะไรทั้งสิ้น เข้ามาใช้งาน ซึ่ง Objective-C 2.0 บน Mac OS X ก็ไม่ใช่ข้อยกเว้น แต่ว่าทาง Apple ได้แสดงจุดยืนชัดเจนว่าจะไม่มีการนำระบบ GC เข้ามาใช้ในระบบ iOS เด็ดขาด ด้วยเหตุผลด้านประสิทธิภาพการทำงาน เพราะการทำงานของระบบ GC นั้นโดยปกติแล้วจะไม่แน่นอน ว่าจะทำงานเวลาไหน ทำงานนานเท่าไหร่ หากทำงานน้อยเกินไป ก็จะเหลือหน่วยความจำน้อยลงๆ เรื่อยๆ แต่หากโผล่ขึ้นมาทำงานผิดจังหวะ ก็จะทำให้ประสิทธิภาพเสียอย่างไม่ควรเกิดขึ้นในจังหวะนั้น เช่น กำลังควบคุมตัวละครในเกมจังหวะหัวเลี้ยวหัวต่อ เป็นต้น

อันที่จริงแม้ว่าการจัดการวัตถุแบบ Reference Counting นั้น แม้จะไม่ยากเท่าไหร่ เมื่อเทียบกับการจัดการหน่วยความจำแบบไม่ผ่านตัวนับ เช่นแบบภาษา C หรือ C++ และไม่มีข้อจำกัดของ GC ดังที่กล่าวมา แต่ก็มีข้อผิดพลาดเล็กๆ น้อยๆ เกิดขึ้นได้หลายจุด อันเกิดจากการ alloc/copy/retain หรือ release ไม่สมดุลกัน

และสำหรับ Objective-C 2.0 ARC นั้น การเขียน retain, release นั้น ก็กลายเป็น “หน้าที่ของคอมไพเลอร์” อีกอย่างแล้ว ทำให้เราไม่ต้องจัดการ Reference Counting ด้วยตัวเอง แต่ทั้งนี้ต้องทำความเข้าใจเบื้องต้นเสียก่อนว่า **ARC นั้นไม่ใช่ Garbage Collection** แต่อย่างใด การจัดการหน่วยความจำด้วย Reference Counting ก็ยังมีเหมือนเดิม เพียงแต่คนที่เขียนเปลี่ยนจากเราเป็นคอมไพเลอร์เท่านั้น โดยคอมไพเลอร์จะทำการเขียน retain/release ให้เราตอนที่ทำการคอมไพล์ (compile-time) ในขณะที่ GC นั้นจะทำงานในขณะที่โปรแกรมกำลังทำงานอยู่ (runtime) ดังนั้นการเขียนโปรแกรมด้วย ARC นั้น ก็ยังสามารถเกิด Memory Leak และ Dangling Pointer ได้เช่นเดิมหากไม่ระมัดระวังทำตามกฎเกณฑ์ของ ARC

รายละเอียดของการใช้งาน ARC นั้น ผมจะเขียนถึงอย่างละเอียดต่อไปในบทนี้ แต่ตอนนี้ลองมาดูก่อนว่า โปรแกรมของเราจะเปลี่ยนแปลงไปเป็นแบบไหนบ้าง

เริ่มจากส่วนของ Interface ทั้ง Student และ Project ก่อน

Student.h

```
#import <Foundation/Foundation.h>

@class Project;

@interface Student : NSObject
@property (nonatomic, copy) NSString *name;
@property (nonatomic, strong) Project *project;

- (id)initWithName:(NSString *)name andProject:(Project *)project;
@end
```

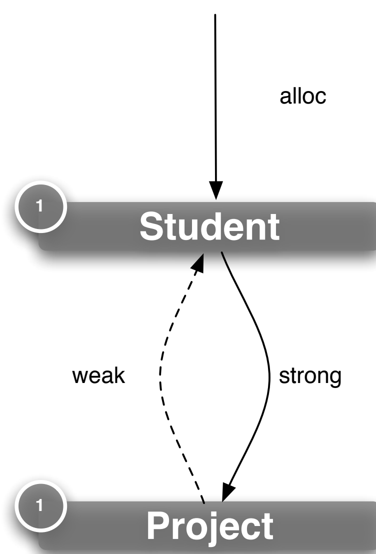
Project.h

```
#import <Foundation/Foundation.h>

@class Student;
@interface Project : NSObject
@property (nonatomic, copy) NSString *name;
@property (nonatomic, weak) Student *student;

- (id)initWithName:(NSString *)name;
@end
```

จะเห็นว่า ไม่มีการเปลี่ยนแปลงอะไรมากมาย นอกจากตัว Property นั้นเป็น **strong** แทนที่ retain และ **weak** แทน assign ซึ่งเป็นการบ่งบอก “ลักษณะความสัมพันธ์ของวัตถุ” ว่าสัมพันธ์กันแบบไหน มากกว่าจะเป็นการบอกลักษณะการจัดการวัตถุว่าจะอ้างอิงอย่างไร ซึ่งในขณะที่ strong นั้นจะไม่แตกต่างจาก retain มากนัก แต่ weak นั้นจะแตกต่างจาก assign พอสมควร เนื่องจากจะทำการอ้างอิงไปยัง nil ในกรณีที่วัตถุที่ถูกอ้างอิงถึงนั้น ถูกปล่อยไปแล้วด้วย ดังนั้นจะเป็นการแก้ปัญหา Dangling Pointer อันเกิดจากการพยายามใช้งานวัตถุที่ไม่มีตัวตนอีกด้วย



สำหรับ Setter ของ Student นั้น ก็เรียบง่ายและตรงไปตรงมาขึ้นมาก ดังนี้

Student.m

```
-(void)setProject:(Project *)project {
    _project = project;

    self.project.student = self;
}
```

จะเห็นว่า การ retain/release นั้น หายไปจากระบบของคิดของเราเลย เราแค่ให้ _project อ้างอิงไปหา project ซึ่งหากเราสังเกตดีๆ ก็จะเหมือนกับโค้ดที่ “อยากจะเขียนนะ แต่เขียนแบบนั้นไม่ได้” ใน Objective-C 1.0 เปี๊ยบเลย และเมื่ออ้างอิงไปถึงแล้วก็ทำการอ้างอิงกลับ เท่านั้นก็เป็นที่เรียบร้อย

และนี่คือโค้ดทั้งหมดของ Student.m และ Project.m

Student.m

```
#import "Student.h"
#import "Project.h"

@implementation Student
@synthesize name = _name;
@synthesize project = _project;

-(void)setProject:(Project *)project {
    _project = project;

    self.project.student = self;
}

- (id)initWithName:(NSString *)name andProject:(Project *)project {
    self = [super init];
    if (self) {
        self.name = name;
        self.project = project;
    }
    return self;
}


- (NSString *)description {
    return [NSString stringWithFormat:@"%@ (%@): %@",
            self.name, [super description], self.project];
}
@end
```

Project.m

```
#import "Project.h"
#import "Student.h"

@implementation Project
@synthesize name = _name;
@synthesize student = _student;


- (id)initWithName:(NSString *)name {
    self = [super init];
    if (self) {
        self.name = name;
    }
    return self;
}

- (NSString *)description {
    return [NSString  stringWithFormat:@"%@ (%@)",
            self.name, [super description]];
}
@end
```

จะเห็นว่าเราไม่จำเป็นต้องเขียน dealloc ด้วย เนื่องจากคอมไพเลอร์จะจัดการเขียนให้เราเอง

สำหรับ Driver-Program เพื่อทำการทดสอบนั้น ก็จะเห็นว่ามีความเปลี่ยนแปลงไปอีก แทนที่ต้องทำการสร้าง Autorelease Pool ขึ้นมาทีละตัว ก็ใช้ **@autoreleasepool** กับโค้ดทั้งส่วน เพื่อให้โค้ดทั้งส่วนถูกจัดการด้วย Autorelease Pool แทน

main.m

```
#import <Foundation/Foundation.h>
#import "Project.h"
#import "Student.h"

int main (int argc, const char * argv[])
{

    @autoreleasepool {
        Project *project = [[Project alloc] initWithName:@"Destroy Konoha"];
        Student *student = [[Student alloc] initWithName:@"Uchiha Sasuke" andProject:project];

        Project *project2 = [[Project alloc] initWithName:@"Save Lukia"];
        Student *student2 = [[Student alloc] initWithName:@"Kurasaki Ichigo" andProject:project2];


        NSLog(@"%@", student);
        NSLog(@"%@", student2);

        NSLog(@"%@", [[project student] name]);

    }
    return 0;
}
```

อย่างไรก็ตาม เราไม่สามารถตรวจสอบจำนวนการอ้างอิงของวัตถุเมื่อทำงานกับ ARC ได้ เนื่องจากทุกอย่างที่เกี่ยวข้องกับ Reference Counting นั้นจะถูกปิดตายจากเข้าถึงของโปรแกรมเมอร์ รวมถึง retainCount ด้วย

จะเห็นว่า ARC ทำให้การทำงานหลายอย่าง “เหมือนจะง่ายขึ้น” และมีโค้ดที่ต้องเขียนน้อยลง แต่ก็แลกมากับการที่ต้องมีสิ่งที่เราต้องเข้าใจมากขึ้น ต้องเรียนรู้ในเชิงลึกมากขึ้น เพื่อกำหนดแนวทางให้คอมไพเลอร์ทำงานอย่างถูกต้อง

ข้อสรุปจากอดีตถึงปัจจุบัน

จะเห็นว่าตั้งแต่ Objective-C 1.0 มาถึง Objective-C 2.0 ARC นั้น นอกจากเราจะค่อยๆ เขียนโค้ดน้อยลงๆ แล้ว เรายังสามารถเขียนโค้ดได้ “อย่างที่เราคิด” มากยิ่งขึ้น และยิ่งไปกว่านั้นคือ ในขณะที่โค้ดเราสั้นลง มันกลับมี “รายละเอียด” ที่สำคัญมากขึ้น โดยไม่จำเป็นต้องเขียนคอมเมนท์บรรยายโค้ดซ้ำซ้อนไปอีก เช่น

- ใน Objective-C 2.0 เราเห็นได้ทันทีว่าวัตถุนี้มี Getter/Setter อะไรบ้าง และทำงานอย่างไร จากการอ่านลักษณะ Property
- ใน Objective-C 2.0 ARC เราเห็นความสัมพันธ์ของวัตถุทันที ว่าตัวไหนมีความสัมพันธ์ในลักษณะ Parent-Child อย่างไร จาก strong/weak
- ใน Objective-C 2.0 ARC เราเห็นทันทีว่าโค้ดส่วนไหนอยู่ใน Autorelease Pool

ไม่เพียงเท่านั้น โค้ดที่ถูกเขียนขึ้นสุดท้ายโดย “โปรแกรมเมอร์+คอมไพเลอร์” นั้น จะมีความถูกต้องมากขึ้น ขาดตกบกพร่องในรายละเอียดเล็กๆ น้อยๆ ต่างๆ น้อยลง และมีประสิทธิภาพในการทำงานสูงขึ้น (เนื่องจากโค้ดจากการเขียนของคอมไพเลอร์นั้น จะเป็นโค้ดที่ถูกรีดประสิทธิภาพโดยผู้เชี่ยวชาญ มากกว่าที่เราเขียนเอง)

ARC “เท่าที่ต้องรู้”

จากโค้ด Student Project เมื่อสักครู่นี้ เราคงจะสังเกตได้ว่า ARC ไม่ได้ทำหน้าที่แค่ใน Setter เท่านั้น แต่ทำหน้าที่ทั้งโปรเจค สังเกตจากการที่เราไม่ต้อง release อะไรเลย และเมื่อเรา alloc วัตถุ เราก็ใช้มันโดยไม่ต้องมีห่วงอะไรทั้งสิ้น รวมถึงการตัด dealloc ออกจากสิ่งที่เราต้องเขียน

ใช่แล้วครับ เมื่อใดก็ตามที่เราใช้ ARC ในโปรเจค ARC ก็จะทำหน้าที่จัดการกับ retain/release แทนเราทั้งโปรเจค (นอกเสียจากเราจะกำหนดให้ยกเว้นการทำงานสำหรับบางไฟล์) ไม่ว่าจะเป็นการประกาศวัตถุอะไรก็ตาม ทุกเรื่องที่เกี่ยวข้องกับการจัดการหน่วยความจำ “ด้วยวิธีการ Reference Counting” นั้น จะถูกจัดการด้วย ARC ทั้งหมด นั่นคือ เมื่อเรามีโค้ด

```
ObjectType *someObject = .......;
```

ไม่ว่าจะเป็นการสร้างใหม่จาก alloc ขึ้นมาใหม่ retain วัตถุที่มีอยู่แล้วเอาไว้ การสร้างใหม่โดยการ copy สิ่งที่มีอยู่แล้ว หรือการใช้ convenient object (ซึ่งจากความรู้เดิมจากเล่มที่แล้วนั้นจะเป็น autorelease object) ก็จะถูกจัดการโดย ARC ทั้งสิ้น

ซึ่งในการที่จะทำงานกับ ARC อย่างราบรื่นไม่มีปัญหานั้น เราจะต้องทำความรู้จักกับมัน ว่า ARC คาดหวังอะไรจากเรา และ ARC จัดการอะไรให้เราบ้าง รวมถึงลักษณะความสัมพันธ์ระหว่างวัตถุ และการจัดการในแต่ละกรณีว่า ARC ทำอะไรบ้าง

Qualifier สำหรับระบุคุณลักษณะความสัมพันธ์ (ownership) ให้กับวัตถุ

ใน ARC จะมีการกำหนดคุณลักษณะวัตถุทั้งหมด 4 แบบ ซึ่งเราจะใช้วางไว้หน้าวัตถุต่างๆ เพื่อให้ ARC จัดการวัตถุให้กับเราได้อย่างถูกต้อง

1. **Strong** ซึ่งเราได้เห็นกันไปแล้ว มี qualifier คือ **__strong** โดยลักษณะนี้หมายความว่า “เป็นเจ้าของ” ตราบใดก็ตามที่ยังมีการอ้างอิงไปหาวัตถุ วัตถุนั้นจะถูก retain เอาไว้เสมอ ไม่มีการหายไปไหน แต่หากไม่มีการอ้างอิงไปยังวัตถุนั้นๆ แล้ว ก็จะถูกปล่อยออกไป กรณีนี้จะเป็นค่าโดยปริยาย (default) ของวัตถุทุกชนิด เมื่อเราสร้างวัตถุขึ้นมาใช้งาน หรือได้วัตถุมาใช้งาน วัตถุนี้จะคงอยู่ตราบที่เราต้องใช้มันเสมอ
2. **Weak** ซึ่งเราได้เห็นไปแล้วเช่นกัน มี qualifier คือ **__weak** โดยลักษณะนี้หมายถึงว่า “รู้จัก แต่ไม่ได้เป็นเจ้าของ” วัตถุที่อ้างอิงนี้จะไม่ถูก retain ไว้ตลอด แต่จะถูก retain ไว้ตลอดการทำงาน 1 expression และถูก release ทันทีหลังจากนั้น ดังนั้นจึงจะหายไปเมื่อไหร่ก็ได้ เมื่อวัตถุหายไป ก็จะอ้างอิงไปยัง nil เพื่อไม่ให้เกิดปัญหา Dangling Pointer ขึ้น ซึ่งโดยปกติจะใช้สำหรับวัตถุที่จะไม่หายไปไหน ตราบที่ยังต้องการใช้งาน เช่น Parent-Child ซึ่ง Parent จะไม่หายไปก่อนที่ Child จะหายไปแน่นอน
3. **Unsafe & Unretained** มี qualifier คือ **__unsafe_unretained** กรณีนี้ ARC จะไม่ทำอะไรกับวัตถุเราเลย นอกจากอ้างอิงไปหาเฉยๆ ซึ่งอาจทำให้เกิด Dangling Pointer
4. **Auto-releasing** มี qualifier คือ **__autoreleasing** เป็นคนละตัวกับ @autoreleasepool โดยตัวนี้จะใช้กับวัตถุที่ใช้ผ่านเข้าไปยัง Method หรือฟังค์ชั่นต่างๆ ในแบบ pass-by-reference (กรณีที่เจอบ่อยคือวัตถุ NSError)

Qualifier สำหรับกำหนด Property

การกำหนดคุณลักษณะให้กับ Property นั้น วัตถุที่ Property เหล่านั้นทำงานด้วย จะต้องมี ownership ที่สอดคล้องกับ Property ดังนี้

Property	Object Ownership
assign, unsafe_unretained	__unsafe_unretained
copy, retain, strong	__strong
weak	__weak

การคิดเกี่ยวกับ "วัตถุ" ในระบบ ARC

ระบบ ARC ทำให้เราต้องเปลี่ยนแนวคิดกับการโปรแกรมวัตถุ โดยเฉพาะอย่างยิ่งเรื่องการบริหารจัดการอายุวัตถุ เพราะการที่เราไม่ต้อง retain/release เองนั้น ทำให้เราสามารถคิดในกรอบความคิดระดับสูงขึ้น นั่นคือ "ระดับความสัมพันธ์ของวัตถุ" (Object Graph) และคิดในระดับของคอนเซปท์ (Conceptual Level) มากกว่ารายละเอียดของการเขียน (Implementation Level) เช่นเดียวกับที่ Property ทำให้เราคิด Data Accessors ในระดับ Conceptual ว่าจะมีอะไรบ้าง ทำงานอย่างไร มากกว่ารายละเอียดของการเขียน Accessors เหล่านั้น

การคิดเกี่ยวกับวัตถุใน ARC นั้น ผมขอสรุปเป็นข้อๆ ดังนี้

1. การอ้างอิงไปหา หมายถึง ownership ซึ่งจะเป็นไปตาม qualifier ที่เรากำหนด
2. การอ้างอิงไปหา เป็นการรักษาชีวิตให้วัตถุ ตามลักษณะของ ownership ซึ่งวัตถุจะยังคงมีชีวิตตราบเท่าที่มีการอ้างอิงไปหามัน เมื่อไม่มีการอ้างอิงไปหา วัตถุก็จะหายไป
3. คิดโครงสร้างของระบบวัตถุ ในระดับความสัมพันธ์ของวัตถุ (Object-Graph) ไม่ใช่ระดับการอ้างอิงวัตถุ
4. เลิกห่วงว่าจะ retain/release วัตถุไหนเมื่อไหร่
5. เชื่อว่า ARC จะจัดการ retain/release ให้เราอย่างถูกต้องและมีประสิทธิภาพ

ขอยกตัวอย่างเล็กน้อยให้พอเห็นภาพว่า ARC ทำอะไรกับโค้ดเราเพื่อความสบายใจ

โค้ดที่เราเขียน	โค้ดสุดท้าย (ตัวเอียงหนาคือสิ่งที่ ARC เพิ่มให้)
NSString *name;	__strong NSString *name = nil;

if (someCondition) {
    NSString *name = ...;
}
	if (someCondition) {
	    NSString *name = ...;
	    [name release];
	}

name = newName;	[newName retain]; NSString *oldName = name; name = newName; [oldName release];

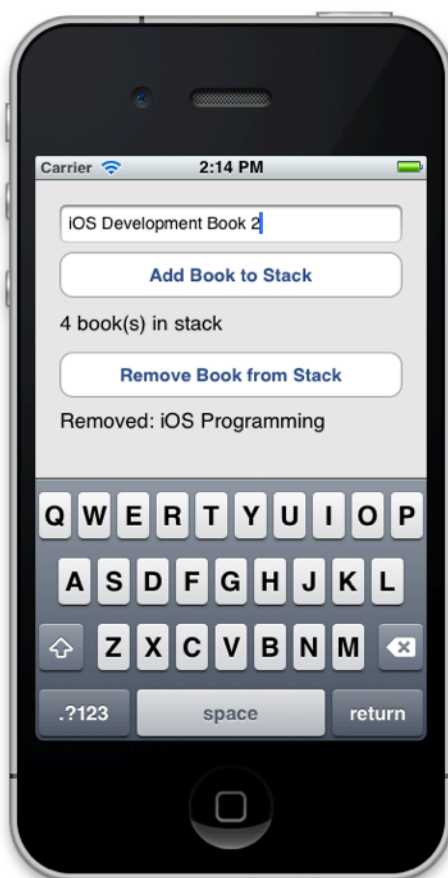
โค้ดที่เราเขียน	โค้ดสุดท้าย (ตัวเอียงหนาคือสิ่งที่ ARC เพิ่มให้)

`-(void)saveWithError:(NSError **)err {`
`    if(!validCondition)`
`        *err = ....;`
`    }`
`}`
	`-(void)saveWithError:(`***`__autoreleasing`***` NSError **)err {`
	`    if(!validCondition)`
	`        *err = `***`[[....`***` `***`retain]`***` `***`autorelease]`***`;`
	`    }`
	`}`

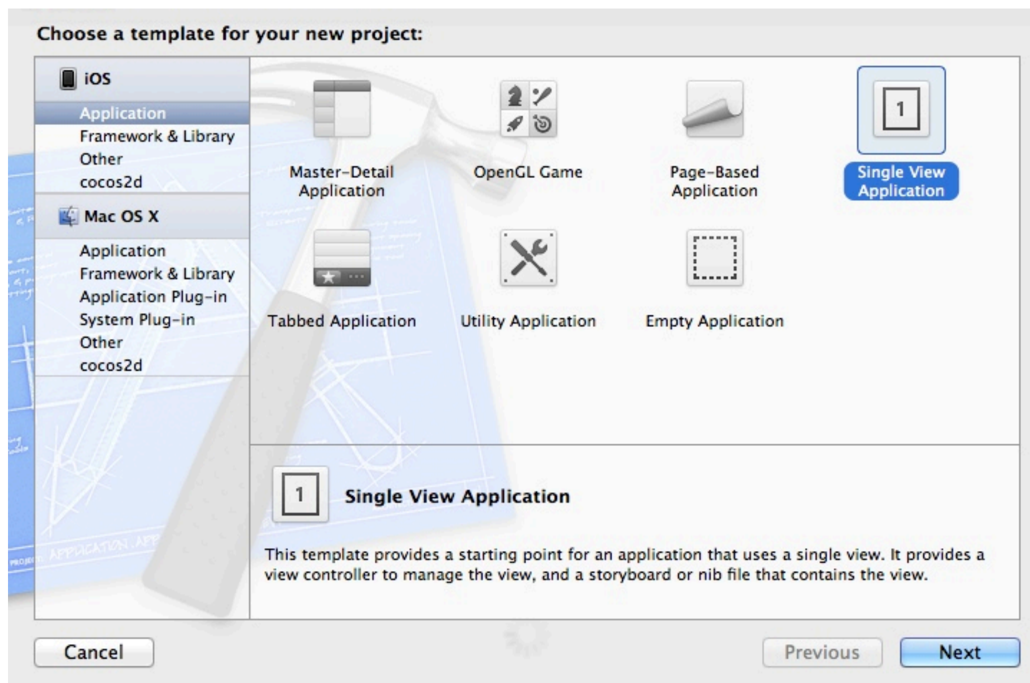
จะเห็นว่า ARC ทำงานหนักๆ แทนเราค่อนข้างเยอะ ซึ่งเดี๋ยวเราจะได้เห็นอย่างชัดเจนในโปรเจคต่อไป

App: BookStack

ลักษณะ:	1. เป็น iOS app ที่ให้ผู้ใช้กรอกชื่อของหนังสือ 2. ผู้ใช้สามารถวางหนังสือลงบนตั้งหนังสือ และหยิบเล่มบนสุดจากตั้งหนังสือได้
เป้าหมายการเรียนรู้:	1. การทำงานกับ ARC ใน iOS app 2. การเปรียบเทียบระหว่าง ARC และ non-ARC ในโปรเจคจริง 3. การสร้างโปรเจคใน Xcode 4.2

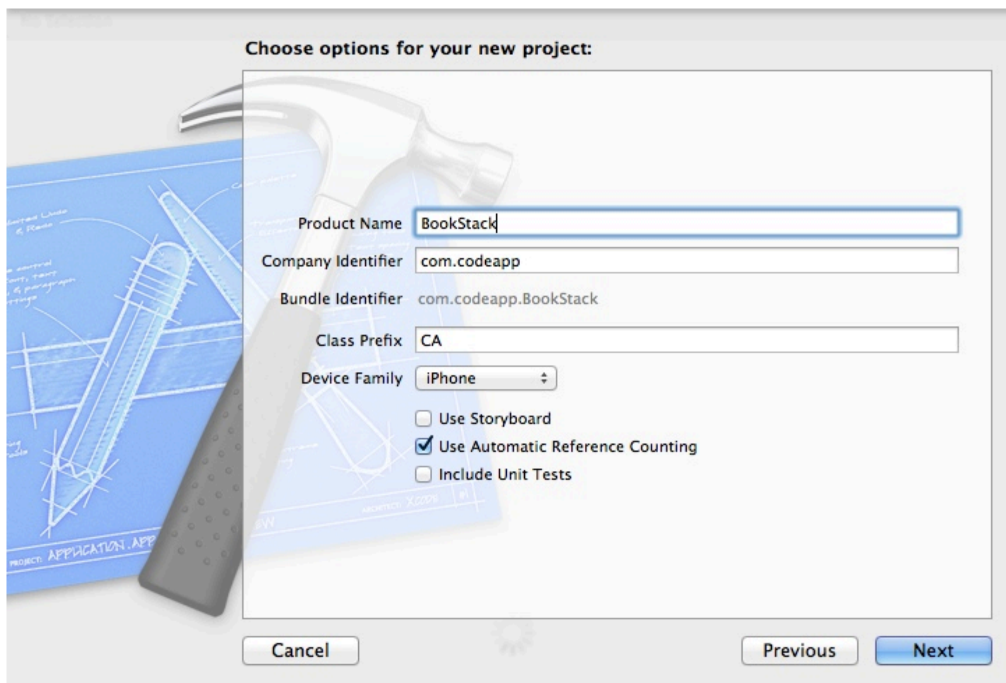


เรียก Xcode 4.2 ขึ้นมาทำงาน และเนื่องจากนี่จะเป็นโปรเจคแรกของเรากับ Xcode 4.2 เราจะค่อยๆ ไปทีละขั้น โดยเริ่มจากการเลือกสร้างโปรเจคใหม่ และเลือก iOS > Application > Single View Application



จากนั้นตั้งค่าต่างๆ ตามนี้

- ตั้งชื่อโปรเจคว่า BookStack
- ทำการกำหนด Company Identifier ตามความเหมาะสม ซึ่ง Xcode จะนำไปสร้าง Bundle Identifier ให้ตามรูปแบบ Company Identifier ตามด้วยชื่อโปรเจค
- กำหนด Class Prefix ซึ่งจะเป็นตัวอักษรแรกของชื่อคลาสที่ Xcode จะสร้างให้ (ตาม Coding Convention ที่พบบ่อย เช่น NS, UI, MK, CL และอื่นๆ โดยปกติจะเป็นตัวย่อของชื่อ Framework หรือชื่อผู้สร้าง) ซึ่งนอกจากจะมีผลกับชื่อคลาสแล้วยังมีผลกับชื่อไฟล์อีกด้วย ดังนั้นเพื่อความสอดคล้องกัน ตอนนี้ขอให้ทุกคนตั้งว่า IDB (iOS Development Book)
- เลือก Device Family เป็น iPhone
- สุดท้ายเลือกที่จะใช้ ARC (อย่าเพิ่งเลือกใช้ Storyboard ในตอนนี้ เราจะใช้ Storyboard ในบทถัดไป)

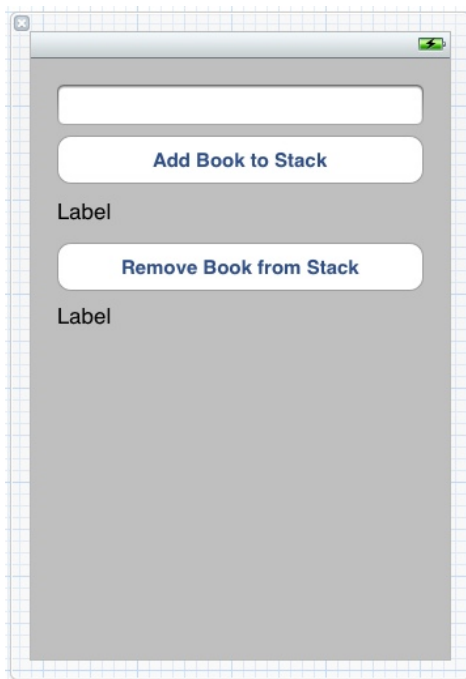


เมื่อเราสร้างโปรเจคเรียบร้อยแล้ว ก็สำรวจไฟล์ต่างๆ ที่ Xcode สร้างให้ จะพบว่าแตกต่างไปจาก Xcode เวอร์ชั่นก่อนหน้านี้พอสมควร เช่น

- ชื่อไฟล์ต่างๆ จะมี Class Prefix ตามที่เรากำหนดนำหน้า
- ชื่อของ Application Delegate จะเป็น <Class Prefix>AppDelegate แทนที่จะเป็น <ชื่อโปรเจค>AppDelegate
- Application Delegate เป็น Subclass ของ UIResponder แทน NSObject
- Property ใน Application Delegate เป็น strong แทนที่จะเป็น retain
- โค้ดใน View Controller (IDBViewController) เรียบง่ายขึ้น

ออกแบบส่วนติดต่อกับผู้ใช้

งานแรกของเราก็คือ ออกแบบส่วนติดต่อกับผู้ใช้ โดยเลือกไฟล์ IDBViewController.xib ขึ้นมาทำงาน และทำการสร้างส่วนติดต่อกับผู้ใช้ โดยมี Text Field สำหรับใส่ชื่อหนังสือ ปุ่มเพิ่มหนังสือลงบนตั้งหนังสือ ข้อความแสดงว่ามีหนังสือในตั้งหนังสือกี่เล่ม ปุ่มหยิบหนังสือออกจากตั้ง และข้อความแสดงหนังสือเล่มที่หยิบออกมา



จากนั้นทำการเชื่อม Outlet เหล่านี้ไปยัง View Controller ดังนี้

IDBViewController.h

```
@interface IDBViewController : UIViewController
@property (weak, nonatomic) IBOutlet UITextField *bookNameField;
@property (weak, nonatomic) IBOutlet UILabel *bookCountLabel;
@property (weak, nonatomic) IBOutlet UILabel *removedBookLabel;

- (IBAction)addBookButtonTapped:(id)sender;
- (IBAction)removeBookButtonTapped:(id)sender;
@end
```

สังเกตว่าเมื่อเราพยายามเชื่อม Outlet ด้วยวิธีการ “ลากจาก View ไปยัง Controller” นั้น ค่าโดยปริยายจะเป็นความสัมพันธ์แบบ weak ซึ่งเป็นไปตามคำอธิบายใน Developer Documentation ของทาง Apple ซึ่งกล่าวไว้ว่า

> โดยทั่วไป Outlet จะเป็น weak นอกจากตัวที่เป็น Outlet ที่อยู่บนสุดของ File’s Owner แต่ละตัว ที่จะเป็น strong ดังนั้น Outlet ที่เราสร้างโดยปกติจะเป็น weak โดยปริยาย ทั้งนี้เพราะ
>
> - ความสัมพันธ์ระหว่าง Controller และ View โดย Controller ควรรู้จักและอ้างอิงถึงวัตถุใน View ได้ แต่ไม่ได้เป็นเจ้าของ
> - Outlet ที่จำเป็นต้องเป็น strong จะถูกกำหนดจากคลาสจาก Framework ต่างๆ อยู่แล้ว เช่น view Outlet ของ UIViewController เป็นต้น

ทั้งนี้ weak จะต้องใช้กับ iOS 5 ขึ้นไป (Mac OS X 10.7 ขึ้นไป สำหรับการเขียนโปรแกรมบน OS X) ถ้าต้องการให้โปรแกรมทำงานได้บน iOS 4 ด้วย ต้องสร้าง Outlet เป็น unsafe_unretained

ปิดท้ายด้วยการใส่ค่าตั้งต้นให้กับส่วนต่างๆ เช่น ลบคำว่า Label ออกจาก Label ทั้งสองตัว ใส่ Place Holder Text ให้กับ Text Field เป็นต้น

สร้าง Model สำหรับ Stack

งานต่อไป ก็คือการสร้าง Model ของ Stack ซึ่งจะถูกนำมาใช้เป็น “ตั้งหนังสือ” โดย Stack เป็นโครงสร้างข้อมูลที่มีลักษณะ Last-In, First-Out ซึ่งมีการนำไปประยุกต์ใช้งานในหลายรูปแบบ

สร้างไฟล์ใหม่ โดยเลือก iOS > Cocoa Touch > Objective-C class จากนั้นตั้งชื่อว่า IDBStack และให้เป็น Subclass ของ NSObject จากนั้นสร้าง Interface ของ IDBStack ดังนี้

IDBStack.h

```
#import <Foundation/Foundation.h>

@interface IDBStack : NSObject
-(void)push:(id)object;
-(id)pop;
-(NSUInteger)count;
@end
```

จะเห็นว่า Stack ตัวนี้จะรับวัตถุอะไรก็ได้ และเมื่อเราหยิบของออกมา ก็จะได้วัตถุคืนมาจาก Stack

ถึงตรงนี้บางคนอาจเริ่มสังเกตเห็นอะไรบางอย่างด้วยความสงสัย แล้ว Stack ตัวนี้ไม่มีวัตถุอะไรที่ใช้เก็บข้อมูลเลยหรือ? โดยปกติน่าจะต้องมี NSMutableArray ในส่วนการประกาศ Data หรือในส่วนของ Property ไม่ใช่หรือ?

นอกจาก ARC แล้ว Objective-C ในปัจจุบัน ยังเพิ่มความสามารถเล็กๆ น้อยๆ อีกหลายอย่าง เช่น การประกาศ Data ในส่วนของ Implementation แทนที่จะต้องประกาศทั้งหมดในส่วนของ Interface ซึ่งเป็นส่วน “สาธารณะ” ของวัตถุ ดังนั้นเราจะประกาศ NSMutableArray สำหรับการเก็บข้อมูลในส่วนของ Implementation

ส่วนของ Implementation จึงมีหน้าตาดังนี้

IDBStack.m

```
#import "IDBStack.h"

@implementation IDBStack
{
    NSMutableArray *_array;
}

-(id)init {
    self = [super init];
    if (self) {
        _array = [NSMutableArray array];
    }
    return self;
}

-(void)push:(id)object {
    [_array addObject:object];
}
```

```
-(id)pop {
    id object = [_array lastObject];
    [_array removeLastObject];
    return object;
}

-(NSUInteger)count {
    return _array.count;
}
@end
```

ค่อนข้างเรียบง่ายและตรงไปตรงมาทีเดียว เมื่อ ARC ทำงานหนักๆ ให้เรา (ท้ายโปรเจคนี้ ผมจะให้ดูเวอร์ชั่น non-ARC นะครับ ว่าเป็นอย่างไร)

เขียน Controller

งานสุดท้าย ก็คือการเขียน Controller เพื่อเชื่อมระหว่าง Model กับ View ซึ่งเริ่มด้วยการประกาศ IDBStack ในส่วน Data Implementation

IDBViewController.m

```
@implementation IDBViewController
{
    IDBStack *stack;
}
```

เตรียม Method สำหรับอัพเดทจำนวนหนังสือใน Stack

IDBViewController.m

```
- (void)updateBookCountLabel {
    self.bookCountLabel.text = [NSString stringWithFormat:@"%lu book(s) in stack", [stack count]];
}
```

จากนั้นก็สร้าง Stack ตัวนี้เมื่อ View ถูกโหลดขึ้นมาเรียบร้อย และบอกให้ Text Field เป็น First Responder พร้อมรับ Input ทันที (ซึ่งจะทำให้คีย์บอร์ดโผล่ขึ้นมาเลย ไม่ต้องไปแตะตรง Text Field) และเรียก Method ที่เตรียมไว้

IDBViewController.m

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    stack = [[IDBStack alloc] init];
    [self.bookNameField becomeFirstResponder];
    [self updateBookCountLabel];
}
```

เมื่อผู้ใช้กดปุ่มเพิ่มหนังสือ ก็เอาชื่อหนังสือจาก Text Field ไปใส่ใน Stack และสำหรับกรณีที่ผู้ใช้ไม่ใส่ชื่ออะไรมา ก็ใส่ Unnamed Book เข้าไป จากนั้นอัพเดทจำนวนหนังสือในตั้งหนังสือ และล้างค่าของ Text ใน Text Field

IDBViewController.m

```
- (IBAction)addBookButtonTapped:(id)sender {
    NSString *bookName = self.bookNameField.text;
    if (bookName == nil || bookName.length == 0) {
        bookName = @"Unnamed Book";
    }
    [stack push:bookName];
    [self updateBookCountLabel];
    self.bookNameField.text = nil;
}
```

สุดท้าย เมื่อเอาหนังสือออกจากนั้งหนังสือ ก็เอาชื่อหนังสือไปแสดงในช่องแสดงชื่อหนังสือ จากนั้นก็อัพเดทจำนวนหนังสือใน Stack

IDBViewController.m

```
- (IBAction)removeBookButtonTapped:(id)sender {
    NSString *bookName = [stack pop];
    self.removedBookLabel.text = [NSString stringWithFormat:@"Removed: %@", bookName];
    [self updateBookCountLabel];
}
```

เป็นอันเรียบร้อย

สำหรับผู้สนใจ: Stack แบบ non-ARC

เพื่อให้เห็นภาพอีกครั้ง ว่า ARC ทำอะไรให้เราบ้าง และทำให้การเขียนโปรแกรมง่ายขึ้นแค่ไหน ผมขอหยิบ Stack ที่เราเพิ่งสร้างมาเป็นกรณีศึกษา โดยขอให้ดูที่ส่วน Implementation ของ Stack กันทีละ Method เลย

เริ่มจาก init กันก่อน

```
-(id)init {
    self = [super init];
    if (self) {
        _array = [NSMutableArray array];
    }
    return self;
}
```

จากความรู้เดิม เราจะเห็นได้เลยว่ามีปัญหาเกิดขึ้นแน่นอน เพราะว่า _array เป็นวัตถุแบบ Convenient Object จาก Convenient Constructor ที่จะถูกปล่อยเมื่อไหร่ก็ไม่รู้ ดังนั้นเราก็ต้อง retain มันเอาไว้ หรือเปลี่ยนวิธีสร้างเป็น alloc-init แทน ดังนี้

```
-(id)init {
    self = [super init];
    if (self) {
        _array = [[NSMutableArray array] retain]; // OR _array = [[NSMutableArray alloc] init];
    }
    return self;
}
```

สิ่งที่ตามมาก็คือ Memory Leak เพราะว่าเราได้ retain ตัว _array เอาไว้ ทำให้ต้องเพิ่ม dealloc เข้าไปในโค้ด

```
-(void)dealloc {
    [_array release];
    [super dealloc];
}
```

ส่วนของการ pop วัตถุออกจาก Stack ก็จะมีปัญหาด้วย

```
-(id)pop {
    id object = [_array lastObject];
    [_array removeLastObject];
    return object;
}
```

เพราะว่าเราให้ object อ้างอิงไปยังวัตถุตัวสุดท้ายของ _array ก่อนจะทำการเอาวัตถุนั้นออกจาก _array ซึ่ง ณ จุดนี้ อาจทำให้วัตถุตัวนั้นหายไปเลย (หากไม่มีตัวไหนอ้างไปหาวัตถุตัวนั้นจากที่อื่น -- ซึ่งดูจากการใช้งานใน BookStack ของเราแล้ว หายแน่นอน เพราะไม่มีวัตถุไหนอ้างอิงไปหาของแต่ละสิ่งที่ Stack เลย) ดังนั้นเราจะต้องทำการ retain วัตถุตัวสุดท้ายเอาไว้ก่อน แต่เมื่อเรา retain เอาไว้ และไม่ release ก็อาจทำให้เกิด Memory Leak ได้ แต่เราต้องการ return ค่าของมันกลับไปไม่ใช่เหรอ แล้วจะ release เมื่อไหร่ คำตอบก็คือ บอกให้มัน autorelease เมื่อทำการ return ซึ่งจะพอดีกับจังหวะที่จะมีตัวอ้างอิงไปหามันพอดี

```
-(id)pop {
    id object = [[_array lastObject] retain];
    [_array removeLastObject];
    return [object autorelease];
}
```

ดังนั้นเราจะได้โค้ดของ Stack ง่ายๆ ระหว่าง ARC และ non-ARC เวอร์ชั่นดังนี้

<table>
<tr><th>ARC</th><th>non-ARC</th></tr>
<tr><td><pre>
@implementation IDBStack

-(id)init {
    self = [super init];
    if (self) {
        _array = [NSMutableArray array];
    }
    return self;
}

-(void)push:(id)object {
    [_array addObject:object];
}

-(id)pop {
    id object = [_array lastObject];
    [_array removeLastObject];
    return object;
}

-(NSUInteger)count {
    return _array.count;
}
@end
</pre></td><td><pre>
@implementation IDBStack

-(id)init {
    self = [super init];
    if (self) {
        _array = [[NSMutableArray alloc] init];
    }
    return self;
}

-(void)push:(id)object {
    [_array addObject:object];
}

-(id)pop {
    id object = [[_array lastObject] retain];
    [_array removeLastObject];
    return [object autorelease];
}

-(NSUInteger)count {
    return _array.count;
}

-(void)dealloc {
    [_array release];
    [super dealloc];
}
@end
</pre></td></tr>
</table>

จะเห็นว่าโค้ดน้อยลงเล็กน้อย แต่สะอาดขึ้นและเรียบง่ายขึ้นมาก โดยเฉพาะอย่างยิ่งเป็นการลดข้อผิดพลาดบางอย่างที่อาจเกิดขึ้นได้จากการบริหารจัดการ Reference Counting เอง (นึกถึงกรณีของ pop Method ซึ่งหลายคนจะลืมนึกถึงการ retain ไว้ และ autorelease เมื่อ return) รวมถึงการลืมปล่อยวัตถุอย่างครบถ้วนทุกตัวใน dealloc ด้วย

แปลงโปรเจคเก่าให้เป็น ARC

ถ้าเรามีโปรเจคเก่าที่ทำมาตั้งแต่ก่อน Xcode 4.2 และต้องการย้ายมาเป็น ARC เราจะต้องทำอย่างไรบ้าง? จะต้องไล่เก็บ retain, release, autorelease เยอะแค่ไหน จะต้องปรับแก้การตั้งค่าอะไรยุ่งยากซับซ้อนหรือเปล่า?

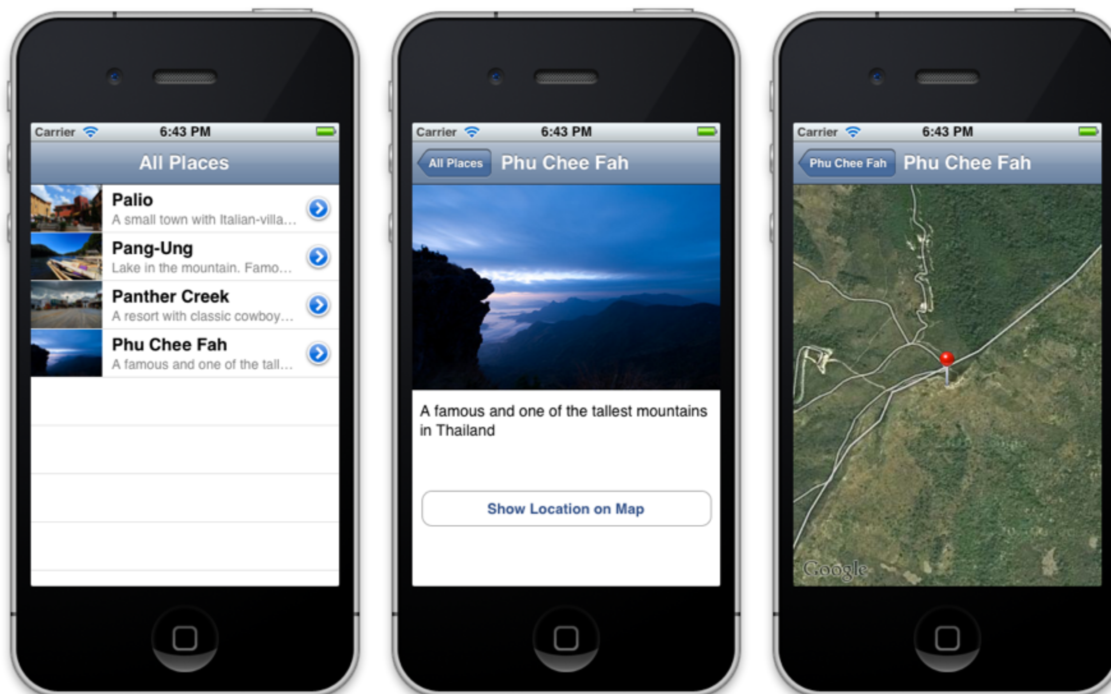
คำตอบก็คือ Xcode 4.2 ได้เตรียมเครื่องมือสำหรับแปลงโปรเจคเก่า ให้กลายเป็น ARC ซึ่งการทำงานจะประกอบด้วย 4 ขั้นตอนดังนี้

1. ให้ Xcode ทำการตรวจสอบโค้ดที่มีอยู่
2. หาก Xcode พบปัญหาที่ไม่สามารถแปลงได้อย่างอัตโนมัติ ก็จะแจ้งให้เรารู้ถึงปัญหาต่างๆ ให้เราทำการแก้ไขโดยการใส่ qualifier หรือย้ายบางส่วนของโค้ด ซึ่งบางอย่าง Xcode จะเสนอแนะว่าควรแก้อย่างไร
3. ทำซ้ำขั้นตอนที่ 1-2 จนกระทั่งไม่เจอปัญหา
4. ให้ Xcode ทำการแปลงโค้ดทั้งหมดเป็น ARC

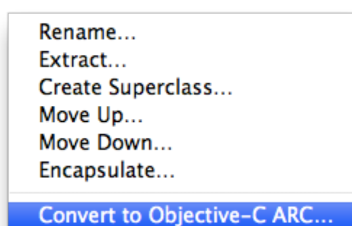
ตอนนี้เรามาลองแปลงโปรเจคเก่าเป็น ARC กันครับ

App: PlacesGuide

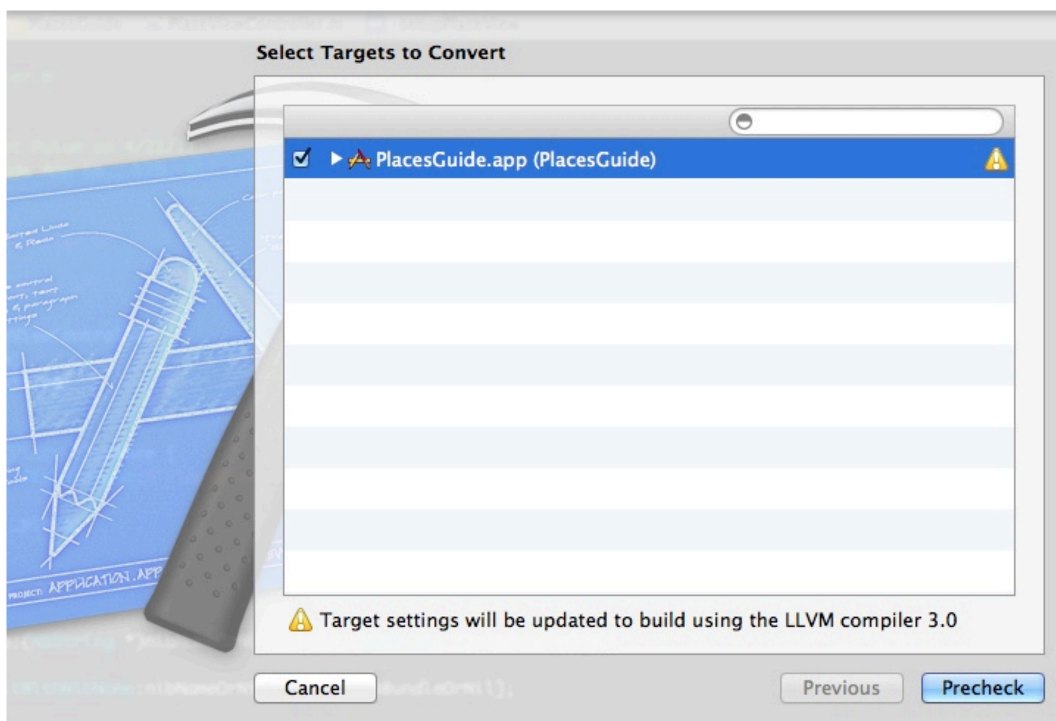
ลักษณะ:	1. เหมือนกับ PlacesGuide ในบทที่ 9 ของหนังสือเล่มแรก แต่ใช้ ARC
เป้าหมายการเรียนรู้:	1. การแปลงโปรเจคเก่าที่มีอยู่แล้วเป็น ARC



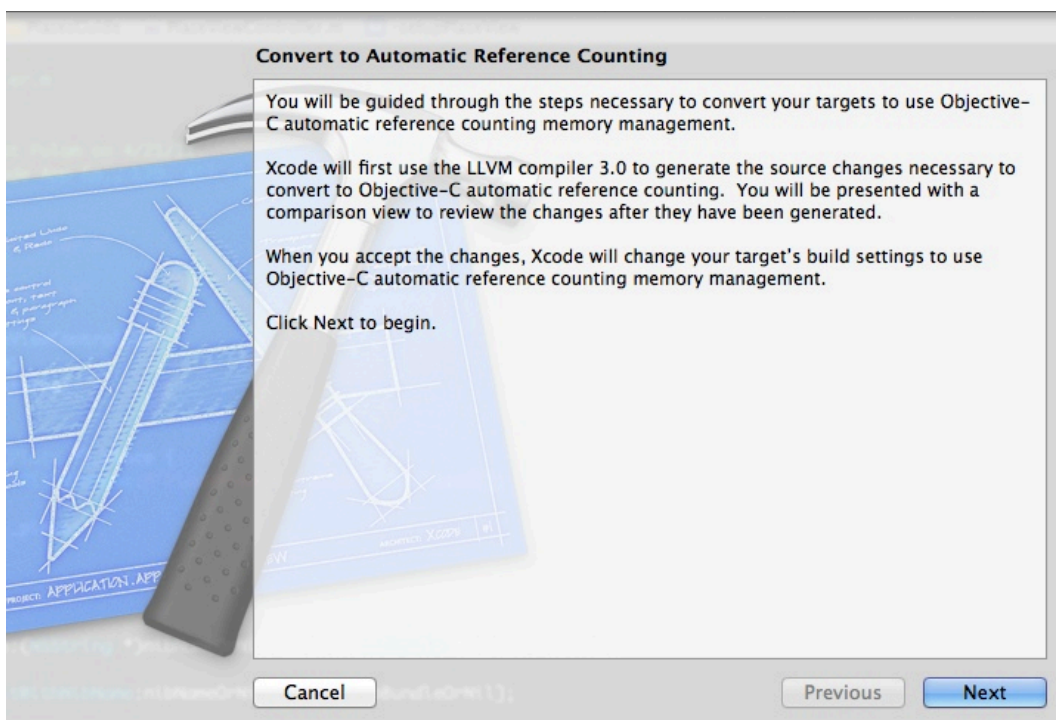
เปิดโปรเจค PlacesGuide ที่มีอยู่จากหนังสือเล่มแรก (อาจทำสำเนาไว้อีกที่หนึ่งก่อน) ใน Xcode จากนั้นเลือกเมนู Edit > Refactor > Convert to Objective-C ARC



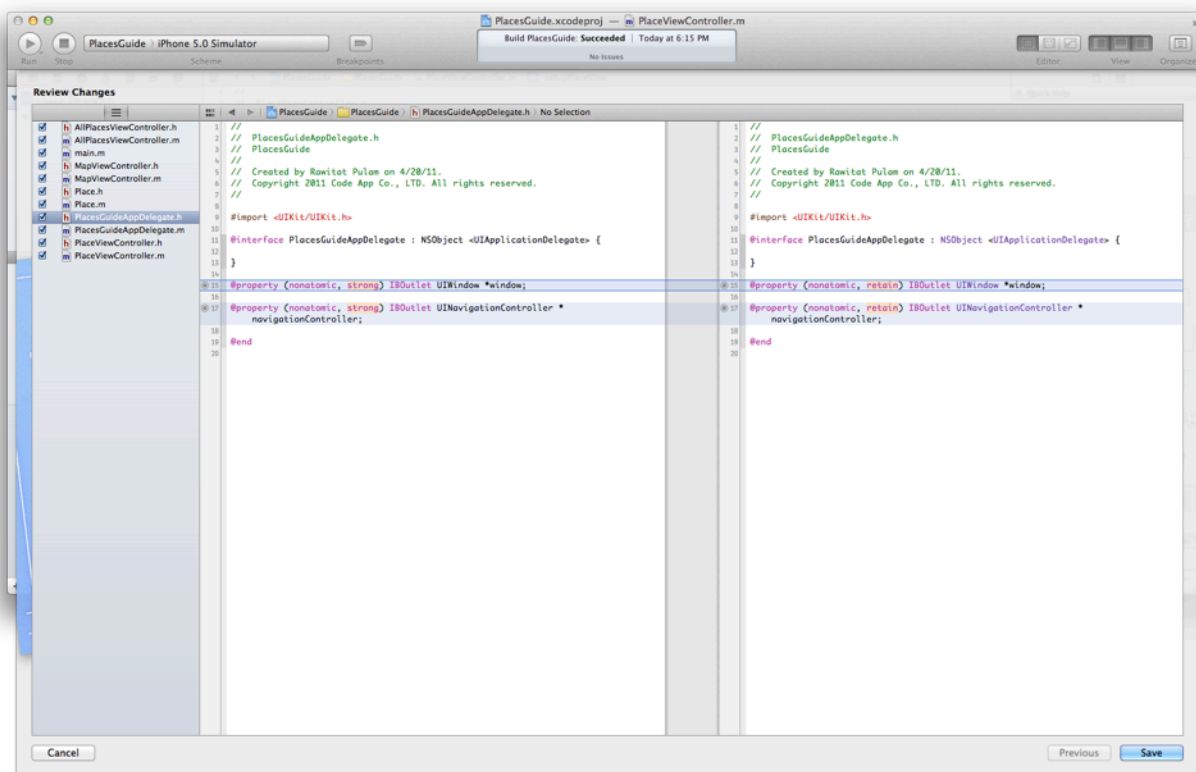
จากนั้นเลือก Target ที่จะทำการแปลง ซึ่งกรณีนี้คือ PlacesGuide และกดเลือก Precheck ซึ่งจะเข้ากระบวนที่ 1 ตามขั้นตอนที่ได้ระบุไว้ด้านบน



สำหรับกรณีนี้ Xcode ไม่พบปัญหาในการแปลง ดังนั้นจึงข้ามขั้นตอนที่ 2-3 เข้าสู่กระบวนการที่ 4 คือการให้ Xcode จัดการแปลงโค้ดต่อไปได้เลย



เมื่อ Xcode แก้ไขโค้ดเรียบร้อยแล้ว ก็แสดงส่วนต่างให้เราเห็น เพื่อให้เราตรวจสอบอีกครั้งหนึ่ง



ซึ่งเราจะเห็นไฟล์ทั้งหมด และโค้ดทั้งหมดที่ถูกแก้ไข ซึ่ง Xcode แสดงส่วนที่ถูกแก้ไขเทียบก่อนและหลังให้เห็นชัดเจน ซึ่งจุดนี้เราสามารถตรวจสอบและแก้ไขเพิ่มเติมได้หากต้องการ ซึ่งในกรณีนี้เราต้องการไขเพิ่มเติมเล็กน้อย ดังนี้

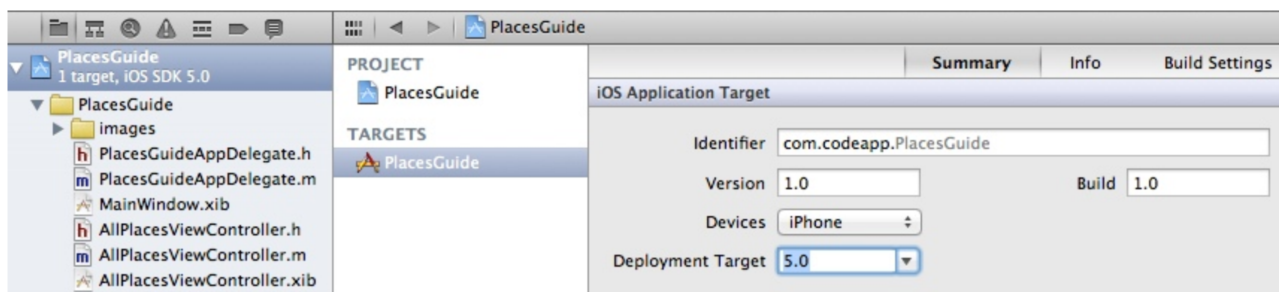
1. เลือกไฟล์ MapViewController.h และทำการแก้ไข Outlet ทั้งสองตัว จาก strong เป็น weak ตามข้อแนะนำของ Apple
2. เลือกไฟล์ PlaceViewController และทำการแก้ไข Outlet ทั้งสองตัว จาก strong เป็น weak เช่นเดียวกัน
3. เลือกไฟล์ PlaceViewController และแก้ไขการอ้างอิง Place จาก strong เป็น weak เนื่องจาก PlaceViewController จะไม่ได้เป็นเจ้าของ Place แต่จะอ้างอิงไปยัง Place ที่อยู่ใน NSArray ใน AllPLacesViewController ซึ่งเป็น Parent View Controller
4. เลือกไฟล์ MapViewController และแก้ไขการอ้างอิง Place จาก strong เป็น weak ด้วยเหตุผลเดียวกัน

กดบันทึกการแก้ไขทั้งหมด จากนั้นลอง Build & Run โปรเจคดู จะพบว่าไม่สามารถ Build ได้ เนื่องจาก

1. เป้าหมายการใช้งานโปรเจคนี้ ถูกตั้งไว้ที่ iOS 4.x ซึ่งไม่รองรับการอ้างอิงแบบ weak
2. วัตถุที่ทำงานกับ weak Property ไม่ได้ถูกกำหนดไว้เป็น weak

ดังนั้นเราจึงแก้ไขปัญหาดังนี้

1. เลือกที่ตัวโปรเจคจาก Navigation Panel และเลือก Deployment Target เป็น 5.0 หรือหากเราต้องการให้ใช้งานได้บน iOS 4.x ด้วยจริงๆ ก็เปลี่ยนจาก weak เป็น unsafe_unretained หรือ strong ก็ได้ (แต่ unsafe_unretained อาจเกิดปัญหา Dangling Pointer และ strong นั้นไม่ก่อให้เกิดปัญหาอะไรในกรณีนี้ เนื่องจากไม่เกิด Retain Cycle ขึ้น)



2. กำหนด __weak เพิ่มเข้าไปหน้าวัตถุที่ทำงานกับ weak Property ในส่วนการประกาศ Data ของวัตถุ เช่น

PlaceViewController.h

```
@interface PlaceViewController : UIViewController {
    __weak UIImageView *imageView;
    __weak UITextView *detailTextView;
    __weak Place *place;
}
```

หรือจะเอาส่วนประกาศ Data เหล่านี้ออกไปเลยก็ได้ และใช้ความสามารถของคอมไพเลอร์ในการสร้างวัตถุเหล่านี้ให้เอง

เมื่อเรียบร้อยแล้วก็ลอง Build & Run อีกครั้งหนึ่ง จะพบว่าโปรแกรมสามารถทำงานได้เป็นปกติ

สรุปท้ายบท

สิ่งที่เราได้เรียนรู้ในบทนี้นั้น สรุปได้ดังนี้

1. การพัฒนาการของภาษา Objective-C ซึ่งทำให้เราเขียนโปรแกรมง่ายขึ้น มีโค้ดที่สั้นคง แต่มีความหมายมากขึ้น ตั้งแต่ Objective-C 1.0 มาเป็น 2.0 ซึ่งเพิ่มความสามารถเรื่อง Property จนมาถึง Objective-C 2.0 ARC ในปัจจุบัน
2. อีกหนึ่งปัญหาคลาสสิคในเรื่องการจัดการวัตถุ: Retain Cycle
3. การทำงานของ ARC โดยละเอียด
4. การเขียนโปรแกรมโดยใช้ ARC
5. การแปลงโปรเจคเก่าจาก non-ARC มาเป็น ARC

ARC ช่วยให้เราเขียนโปรแกรมได้ง่ายขึ้นมากก็จริงอยู่ แต่พึงระลึกไว้เสมอนะครับว่า ARC นั้นยังคงเป็นระบบ Reference Counting เช่นเดิม ไม่ใช่ Garbage Collection ทำให้ไม่มีผลประทบต่อประสิทธิภาพของโปรแกรมขณะทำงาน แต่ก็ยังคงสามารถเกิดปัญหา Memory Leak และ Dangling Pointer ได้เหมือนเดิมหากเราเขียนอย่างไม่ระมัดระวัง (แม้ว่าจะเกิดยากขึ้นมากก็ตาม)