Dynamic Objects, Memory Management & Smart Pointers Objective-C: Introduction

David Marshall

School of Computer Science & Informatics
Cardiff University

CM2204

Dynamic Object Creation

- *Often we do not know the exact size of our array or objects beforehand*

The C Way: Dynamic memory allocation

Dynamic memory allocation functions such as `malloc( )` and `free( )` (and their variants) that allocate storage from the heap at runtime.

Dynamic Object Creation Cont.

The C++ Way

Dynamic memory allocation *à la* C **not a good idea** (even though it will compile!).
C++ essentially:

- Brings dynamic object creation into the core of the language.
  - In C, `malloc( )` and `free( )` are library functions, and thus **outside** the control of the compiler
- Likes to have control of **construction** and **destruction** of all data.

`new` and `delete`

C++ defines `new` and `delete` to elegantly and safely creating objects on the heap.

- Seen examples of `new` and `delete` already but look at this in more depth now.

new Operator Example Call

Example new Call

```
MyType *fp = new MyType(1,2);
```

When you create an object with new:

- It **allocates** enough storage on the heap to hold the object and calls the **constructor** for that storage.
  - The equivalent of `malloc(sizeof(MyType)` is called, and
  - The constructor for the Object Type (*e.g.* `MyType`) is called with the resulting address as the this pointer, (*e.g.* using (1,2) as the argument list).
  - By the time the **pointer** is **assigned** to `fp`, it's a **live**, **initialised** object
  - It's also automatically the proper MyType type so no cast is necessary.
- The default `new` checks to make sure the memory allocation was successful before passing the address to the constructor,
  - If there's **no memory left** an **exception** is thrown(**see later**).

The `delete` Operator

The complement to the `new` operator is `delete`:

- First calls the **destructor** and then **releases** the memory.
- **Requires** the **address** of an object.

```
delete fp;
```

  - This **destructs** and then **releases** the storage for the dynamically allocated `MyType` object above.
- `delete` can **only** be called for an object **created** by `new`.
- If the pointer you're deleting is **zero**, **nothing will happen**.
  - Recommend setting a pointer to zero immediately after you delete it, to prevent deleting it twice.
- Deleting an object **more than once** is **definitely** a **bad** thing to do, and **will** cause problems.

new & delete for arrays

C++ arrays of objects created on the stack or on the heap with equal ease,

- the constructor is called for each object in the array.

new arrays

Create arrays of objects on the heap using new, *e.g.*

```
MyType* fp = new MyType[100];
```

- This allocates enough storage on the heap for 100 MyType objects
  - Calls the constructor for each one.
- However, this simply gives a MyType*, which is exactly the same as from

  ```
  MyType* fp2 = new MyType;
  ```

new & delete for arrays cont.

Array pointers and delete

- We'd like fp to be starting address of an array (we know this is the case!),
  - so it makes sense to select array elements using an expression like fp[3].

  But what happens when you destroy the arrays (pointers)?

```
delete fp2; // OK
delete fp; // Not the desired effect
```

- The statements look exactly the same, and their ultimate effect will be the same:
  - The destructor will be called for the MyType object pointed to by the given **address**, and then the **storage** will be **released**. **However .....**

new & delete for arrays cont.

delete of arrays mechanism explained

- For fp2 this is fine — destruction is as to be expected.
- But for fp
  - the other **99 destructor** calls **will not be made**.
- **However**, the proper amount of storage will still be released,
  - it is allocated in one big chunk, and the size of the whole chunk is stashed somewhere by the allocation routine.

Array delete solution, Array delete syntax:

- The clean way to delete arrays Give the compiler the information that this is actually the **starting address** of an **array**

```
delete []fp;
```

  - The empty brackets tell the compiler to generate code that fetches the number of objects in the array, stored somewhere when the array is created, and calls the destructor for that many array objects.

Note: This is actually an improved syntax from the earlier form, which you may still occasionally see in old code:

```
delete [100]fp;
```

  - The programmer would could get the number of objects wrong!
    - The additional overhead of letting the compiler handle it was very low, and it was considered better to specify the number of objects in one place instead of two.

Making a pointer more like an array

- **As an aside**, the `fp` defined above can be changed to point to anything!
  - doesn't make sense for the starting address of an array.
- **However**, it makes more sense to define it as a `const`**ant**, so any attempt to modify the pointer will be flagged as an error, *i.e.*:

  ```
  int const* q = new int[10];
  ```

  or
  - const int* q = new int[10];
- **Both** are **wrong** however!
  - **Both** cases will bind the `const` to the `int`, what is being pointed **to**, **rather than** the quality of the pointer **itself**.

Making a pointer more like an array cont.

The correct way to make a pointer like an array in C++

- **Instead**, you **must** use:

```
int* const q = new int[10];
```

- **Now** the **array** elements in q **can** be **modified**, but any change to q (e.g. q++) is **illegal**, as it is with an **ordinary array identifier**.

Memory corruption

- It's easy to introduce bugs when using pointers, e.g. memory corruption

```
SomeClass* p = new SomeClass();
SomeClass* q = p;
p->doSomething();
delete p;
q->doSomething();
```

- See Corruption.cpp

Memory leaks

- A memory leak occurs when memory is not freed but never used again (e.g. where there is no pointer or reference available to access an object.)

```
void myFunction() {
  SomeClass* p = new SomeClass();
  p->doSomething();
}
```

- See MemoryLeak.cpp

Cleanup

- Destructors in C++ provide a means to automatically free the memory of objects, but it is still not always clear
- Which code is responsible for freeing the memory of the object returned by this function:

```
SomeClass* myFunction(SomeClass*);
```

- Which code is responsible for freeing the memory of these objects:

```
SomeClass** a = new SomeClass*[1000];
for (int i = 0; i < 1000; ++i) {
  a[i] = new SomeClass();
}
```

Smart pointers

- Smart pointers behave like pointers, but add extra functionality to improve memory management and/or efficiency
- There are many different forms – one of the simpliest is auto_ptr (which has since been deprecated)
- auto_ptr wraps a regular pointer (see http://ootips.org/yonat/4dev/smart-pointers.html):

```
template <class T> class auto_ptr
{
    T* ptr;
public:
    explicit auto_ptr(T* p = 0) : ptr(p) {}
    ~auto_ptr()                 {delete ptr;}
    T& operator*()              {return *ptr;}
    T* operator->()             {return ptr;}
    // ...
};
```

Reference counting

- Every object has a reference count
- This starts at 1 when the object is created
- When another reference takes ownership of the object, the count is incremented
- When you are finished with an object, you give up ownership and the count is decremented
- When the count reaches 0, the memory for the object can be freed
- This allows *copy-on-write*:
  - Copying an object simply adds one to the reference count
  - If a reference/smart pointer is used to modify an object, the data is copied to a new object and the reference count decremented

C++11 Smart Pointers

Smart pointer

- An abstract data type that simulates a pointer while providing additional features, such as automatic memory management or bounds checking.
  - intended to reduce bugs caused by the misuse of pointers while retaining efficiency.
  - keep track of the memory they point to.

To compile C++11 use something like (see example Makefile also):

```
c++ -std=c++11 -stdlib=libc++ ...
```

(**OS X**),

```
g++ -std=c++0x .....
```

(**Gnu Linux g++ 4.6.x**)

Google C++11 or for more details and examples see:

- 10 C++11 features,
- Smart Pointers (Wikipedia),
- Cplusplus.com Memory Reference

Three new pointer classes

- unique_ptr: should be used when ownership of a memory resource does not have to be shared (it doesn't have a copy constructor), but it can be transferred to another unique_ptr (move constructor exists).
- shared_ptr: should be used when ownership of a memory resource should be shared (hence the name).
- weak_ptr: holds a reference to an object managed by a shared_ptr, but does not contribute to the reference count; it is used to break dependency cycles (think of a tree where the parent holds an owning reference (shared_ptr) to its children, but the children also must hold a reference to the parent; if this second reference was also an owning one, a cycle would be created and no object would ever be released).

Smart pointers are **defined** in the std namespace (in the <memory> header file).

unique_ptr

- auto_ptr has strict ownership – only one auto_ptr can *own* an object
  - if we copy an auto_ptr the original is set to NULL
  - See Autocopy.cpp
  - Makes auto_ptr unsuitable for use in most collections (e.g. vector)
- C++11 introduced additional Smart Pointers (*e.g* unique_ptr along with *move semantics* to address this problem
- Created in the same way

```
unique_ptr<SomeClass> p(new SomeClass);
```

- Now we explicitly transfer ownership if needed

```
doSomething( move(p) ); // Pass to function
v.push_back( move(p) ); // Add to vector
```

Normal Pointers vs Smart Pointers

```
void UseRawPointer()
{
    // Using a raw pointer -- not recommended.
    Song* pSong = new Song("Starless", "King␣Crimson");

    // Use pSong...

    // Don't forget to delete!
    delete pSong;
}


void UseSmartPointer()
{
    // Declare a smart pointer on stack and pass it the raw pointer.
    unique_ptr<Song> song2(new Song("Starless", "King␣Crimson"));

    // Use song2...
    wstring s = song2->duration_;
    //...

} // song2 is deleted automatically here.
```

For more examples see:

- SharedPtr.cpp — Shared Pointer Example
- WeakPtr (Zip) — Weak Pointer Example

Objective-C

Objective-C History:

- Objective-C developed in **1980**'s to add **object-oriented** features to C
- Object-orientation inspired by Smalltalk (a "pure" object-oriented language)
- Licensed by NeXT (founded by Steve Jobs after leaving Apple) to develop NeXTStep operating system
- Standardised by Free Software Version as GNUstep
- Steve Jobs rejoins Apple, NeXTStep becomes the basis of Mac OS X
- Like C++, Objective-C **includes** C as a **subset**
- See Introduction to Objective-C (**Apple Developer Site**)

Defining classes

- Two files (as in C++) – the *interface* (**header file**) and **implementation**
- General form of **interface**:

```
// Imports

@interface ClassName {
  // Members
}

// Method declarations
@end
```

Implementation

- General form of **implementation**:

```
// Imports (inc. interface)

@implementation ClassName

// Implementation of methods

@end
```

Declaring methods: Syntax differences

In C (and C++):

```
    int calculateArea(int l);
```

[Note: we can define **global** functions in this way in Objective-C, but **not** **within** a **class**.]

In **Objective-C**:

```
    +(int)calculateAreaWithLength:(int)l;
```

With multiple arguments:

```
    +(int)calculateAreaWithLength:(int)l width:(int)w;
```

Sending messages to objects

- To **send** a **message** to an **object**:

```
[Rectangle calculateArea:5];
[b deposit: 1000.50];
```

- Or with **multiple arguments**:

```
[Rectangle calculateAreaWithLength:5 width:2];
```

- You **don't** need to know the **type** of something to **send** it a **message** – if the object can respond, it will
- The **Objective-C** equivalent of `NULL` in C/C++ is `nil`
- `nil` will accept any messages (see Messages example)

Messages example: Header File and a Class

Bob.h Header File

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

@interface Bob : NSObject

-(void) doSomething;

@end
```

Bob.m a Bob Class

```
#import "Bob.h"

@implementation Bob

-(void) doSomething {
        NSLog(@"Hello");
}
@end
```

Messages example cont.: Main

Test.m — Main (Test)

```
#import "Bob.h"

int main() {
        Bob *b = [[Bob alloc] init];

        [b doSomething];

        [b release];
}
```

Compling Objective-C

Compling Objective-C (Linux)

- Initially, you must execute the command to **configure** your **environment**:

```
./usr/share/GNUstep/Makefiles/GNUstep.sh
```

- Once configured, use the gcc compiler, for example::

```
gcc -o Test Test.m Bob.m  -I `gnustep-config --variable
    =GNUSTEP_SYSTEM_HEADERS` -L `gnustep-config --
    variable=GNUSTEP_SYSTEM_LIBRARIES` -lgnustep-base -
    fconstant-string-class=NSConstantString -
    D_NATIVE_OBJC_EXCEPTIONS
```

- or use a Makefile.

Compling Objective-C Cont.

Compling Objective-C (Mac OS X)

Command Line:

- Initially, you must **set** your PATH **environment** variable to point to OS X Frameworks:

```
PATH=$PATH:/System/Library/Frameworks
```

- Once configured, use the gcc compiler, for example::

```
gcc Test.m  Bob.m  -o Test -ObjC -framework
    Foundation
```

- or use a Makefile.

XCode : Use Apple's Xcode IDE.
XCode Tutorial: Create Our First XCode Project

Memory allocation

- To **allocate memory** for an **object** and **initialize** it:

```
Fraction *frac = [[Fraction alloc] init];
```

- Note the **nested** messages and use of **pointers** (**always** used for **objects** in **Objective-C**)
- `init` is the equivalent of a **constructor** in C++, but are just regular methods
- `init` returns a **pointer** to **itself**
- Memory is **released** after use via:

```
[frac release];
```

- This is a **very** simplified view of **memory management** in **Objective-C**

BankAccount example

BankAccount.h

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

@interface BankAccount : NSObject {
        float balance;
        NSString* name;
}

-(void) setName: (NSString*) n;
-(void) setBalance: (float) b;
-(BankAccount*) initWithName:(NSString*)n;
-(void) deposit: (float) b;
-(void) withdraw: (float) b;
-(float) balance;
-(NSString*) name;
@end
```

BankAccount example cont.

BankAccount.m

```
#import "BankAccount.h"

@implementation BankAccount
-(BankAccount*) initWithName:(NSString*)n {
        self = [super init];

        if (self) {
                [self setName: n];
                [self setBalance: 0.0];
        }

        return self;
}

-(void) setName: (NSString*) n {
        name = n;
}

-(void) setBalance: (float) b {
        balance = b;
}
```

BankAccount example cont.

BankAccount.m

```
#import ”BankAccount.h”

@implementation BankAccount
-(BankAccount*) initWithName:(NSString*)n {
        self = [super init];

        if (self) {
                [self setName: n];
                [self setBalance: 0.0];
        }

        return self;
}

-(void) setName: (NSString*) n {
        name = n;
}

-(void) setBalance: (float) b {
        balance = b;
}
```

BankAccount example cont.

BATest.m cont.

```
#import "BankAccount.h"

int main() {
        BankAccount *b = [[BankAccount alloc] initWithName:
            @"Dave"];

        [b deposit: 1000.50];
        NSLog( @"Balance␣is␣%.2f", [b balance]);
        NSLog( @"%@", [b name] );

        [b withdraw: 50.23];
        NSLog( @"Balance␣is␣%.2f", [b balance]);
        NSLog( @"%@", [b name] );

        [b release];
}
```

Compilation: See Makefile

Inheritance

- Can inherit from existing classes as in C++

```
@interface MySubClass : MyBaseClass
```

- Can only have **one base** (super) **class** (**similar** to Java, **different** to C++)
- Should **initialize** the **super class**

```
-(MySubClass*) init: {
  self = [super init];
  // MySubClass initialization
  return self;
}
```

- All methods can be **overridden** (as in Java) – **no need for** `virtual` (**as in C++**)

Categories

- **Categories** are an **alternative** to **inheritance**, that allow us to add **extra** functionality to **existing classes**
- **Categories** can be used even when you don't have the source code for the original class
- Syntax is:

```
@interface BankAccount (BankAccountAdditions)
```

Categories: BankAccountAdditions example

BankAccountAdditions.h

```
 #import <Foundation/NSString.h>
#import "BankAccount.h"

@interface BankAccount (BankAccountAdditions)
-(NSString *)balanceEnquiryString;
@end
```

BankAccountAdditions.m

```
#import "BankAccountAdditions.h"

@implementation BankAccount (BankAccountAdditions)
-(NSString *)balanceEnquiryString {
        return [NSString stringWithFormat: @"%@'s␣balance␣is
             ␣%.2f", [self name], [self balance]];
}
@end
```

Categories: BankAccountAdditions example cont.

BAAdditionsTest.m

```
#import "BankAccount.h"
#import "BankAccountAdditions.h"

int main() {
        NSAutoreleasePool *myPool = [[NSAutoreleasePool
            alloc] init];

        BankAccount *b = [[BankAccount alloc] initWithName:
            @"Dave"];

        [b deposit: 1000.50];
        NSLog( @"%@", [b balanceEnquiryString]);

        [b withdraw: 50.23];
        NSLog( @"%@", [b balanceEnquiryString]);

        [b release];
        [myPool drain];
}
```

Compilation: See Makefile

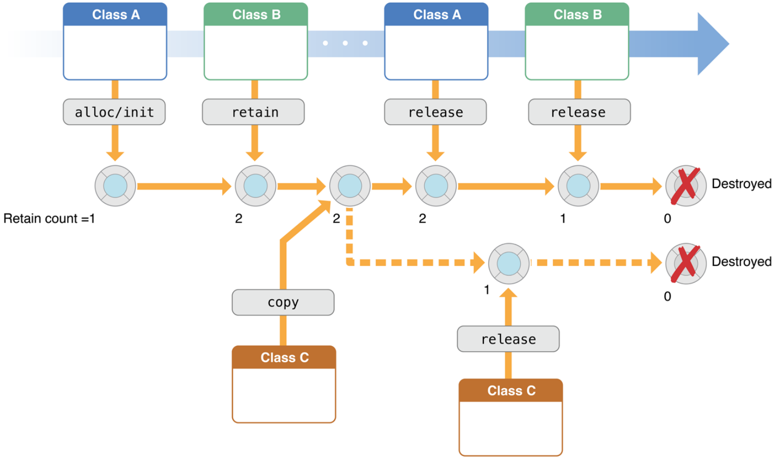
Manual memory management in Objective-C

- You **own** any object you create (`alloc`)
- You **can take ownership** of an object using `retain`
- When you no longer need it, you **must relinquish ownership** of an object you own (`release`)
- You **must not relinquish ownership** of an object you do not own
- http://bit.ly/bpjyoT
- `@autorelease` defers release – e.g. when returning from a function

```
- (NSString *)fullName {
  NSString *string = [[[NSString alloc]
      initWithFormat:@"%@␣%@", self.firstName,
      self.lastName] autorelease];
  return string;
}
```

- `@autoreleasepool {}` marks when the memory should be

Example


Class A
Class B
Class A
Class B
alloc/init
retain
release
release
Retain count =1
2
2
2
1
0
Destroyed
copy
Class C
1
release
Class C
0
Destroyed

Objective-C memory management example

MyClass.h

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

@interface MyClass
+(NSMutableString *)getString;
@end
```

MyClass.m

```objc
 #import "MyClass.h"

@implementation MyClass
+(NSMutableString *)getString {
        NSMutableString* s = [[[NSMutableString alloc]
            initWithString:@"Hello"] autorelease];
        return s;
}
@end
```

Objective-C memory management example cont.

test.m

```
#import "Foundation/Foundation.h"
#import "MyClass.h"

int main() {
        NSAutoreleasePool *myPool = [[NSAutoreleasePool alloc] init];

        NSMutableString *s = [MyClass getString];

        NSLog( @"Retain␣count␣is␣%d", [s retainCount]);
        NSLog( @"%@", s );

        [s retain];

        NSLog( @"Retain␣count␣is␣%d", [s retainCount]);
        NSLog( @"%@", s );

        [myPool drain];

        NSLog( @"Retain␣count␣is␣%d", [s retainCount]);
        NSLog( @"%@", s );

        [s release];
}
```

Compilation: See Makefile

Summary and on to the labs

- Be able to define new dynamic classes and C++'s (cleaner) use of pointers
  - Understand the limitations of pointers and how these may be addressed through smart pointers
  - Understand the principle of reference counting as a mechanism for garbage collection
- Be able to use C+11's smart pointers
- Compile C++11 programs
- Understand the differences in approach between C++ and Objective-C:
  - Message passing
  - Dynamic binding
  - Inheritance
- Be able to write and compile simple classes in Objective-C
- Understand the purpose of categories and explain the difference from inheritance