

# CPSC 213

## Introduction to Computer Systems

*Unit 1c*

***Instance Variables and Dynamic Allocation***

# Reading

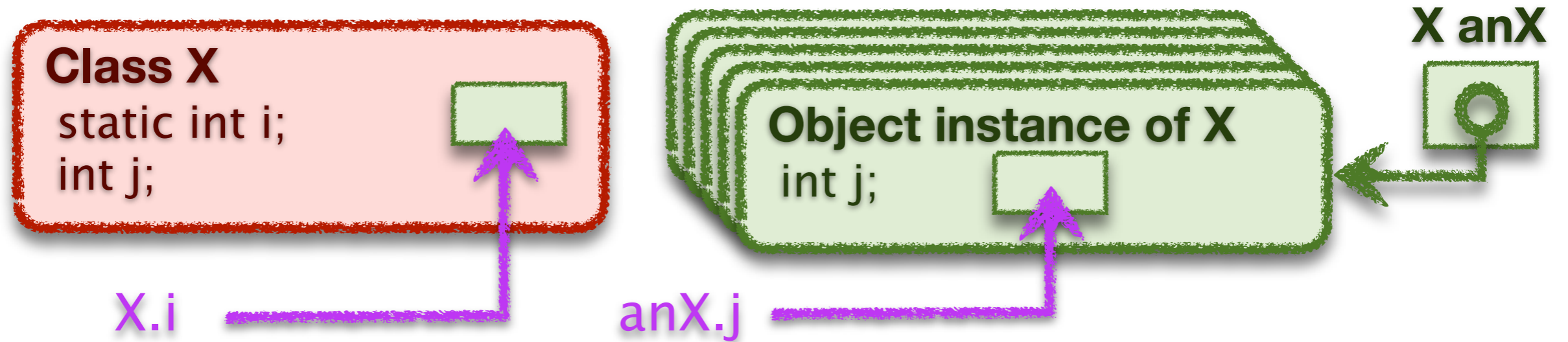
## ▶ Companion

- 2.4.4-2.4.6

## ▶ Textbook

- 2ed: 3.9.1, 9.9.1-9.9.2, 3.10
- 1ed: 3.9.1, 10.9.1-10.9.2, 3.11

# Instance Variables



- ▶ Variables that are an instance of a class or struct
  - created dynamically
  - many instances of the same variable can co-exist
- ▶ Java vs C
  - Java: **objects** are instances of non-static variables of a **class**
  - C: **structs** are named variable groups, instance is also called a struct
- ▶ Accessing an instance variable
  - requires a reference to a particular object (pointer to a struct)
  - then variable name chooses a variable in that object (struct)

# Structs in C (S4-instance-var)

```
struct D {  
  int e;  
  int f;  
};
```



```
class D {  
  public int e;  
  public int f;  
}
```

## ▶ A struct is a

- collection of variables of arbitrary type, allocated and accessed together

## ▶ Declaration

- similar to declaring a Java class without methods
- name is “struct” plus name provided by programmer

- static

```
struct D d0;
```

- dynamic

```
struct D* d1;
```

## ▶ Access

- static

```
d0.e = d0.f;
```

- dynamic

```
d1->e = d1->f;
```

# Struct Allocation

```
struct D {  
    int e;  
    int f;  
};
```

- ▶ Static structs are allocated by the compiler

## Static Memory Layout

```
struct D d0;
```

```
0x1000: value of d0.e  
0x1004: value of d0.f
```

- ▶ Dynamic structs are allocated at runtime

- the variable that stores the struct pointer may be static or dynamic
- the struct itself is allocated when the program calls **malloc**

## Static Memory Layout

```
struct D* d1;
```

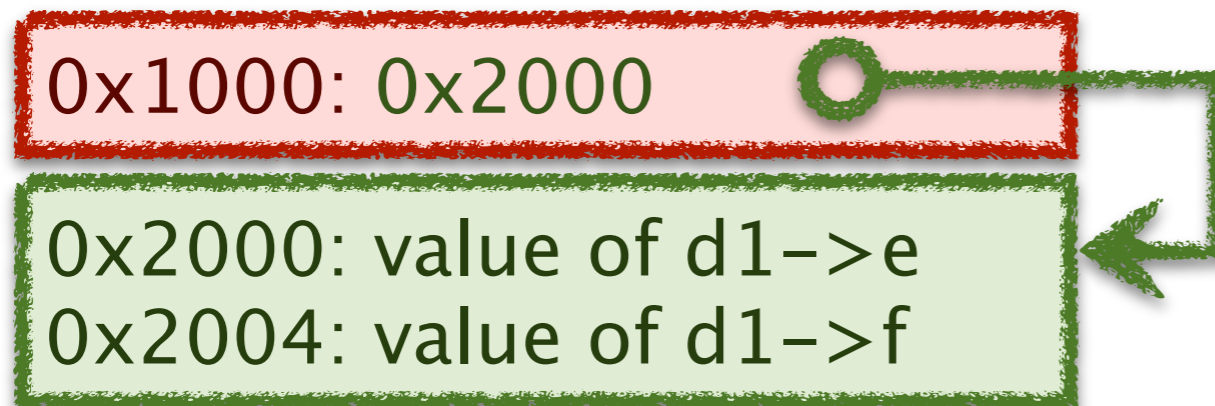
```
0x1000: value of d1
```

```
struct D {  
    int e;  
    int f;  
};
```

- runtime allocation of dynamic struct

```
void foo () {  
    d1 = (struct D*) malloc (sizeof(struct D));  
}
```

- assume that this code allocates the struct at address 0x2000



# Struct Access

```
struct D {  
  int e;  
  int f;  
};
```

## ▶ Static and dynamic differ by an extra memory access

- dynamic structs have dynamic address that must be read from memory
- in both cases the offset to variable from base of struct is static

```
d0.e = d0.f;
```

```
m[0x1000] ← m[0x1004]
```

```
r[0] ← 0x1000  
r[1] ← m[r[0]+4]  
m[r[0]] ← r[1]
```

```
d1->e = d1->f;
```

```
m[m[0x1000]+0] ← m[m[0x1000]+4]
```

```
r[0] ← 0x1000  
r[1] ← m[r[0]]  
r[2] ← m[r[1]+4]  
m[r[1]] ← r[2]
```

load d1

```
struct D {  
  int e;  
  int f;  
};
```

```
d0.e = d0.f;
```

```
r[0] ← 0x1000  
r[1] ← m[r[0]+4]  
m[r[0]] ← r[1]
```

```
ld $0x1000, r0 # r0 = address of d0  
ld 4(r0), r1 # r1 = d0.f  
st r1, (r0) # d0.e = d0.f
```

```
d1->e = d1->f;
```

```
r[0] ← 0x1000  
r[1] ← m[r[0]]  
r[2] ← m[r[1]+4]  
m[r[1]] ← r[2]
```

load d1

```
ld $0x1000, r0 # r0 = address of d1  
ld (r0), r1 # r1 = d1  
ld 4(r1), r2 # r2 = d1->f  
st r2, (r1) # d1->e = d1->f
```

## ▶ The revised load/store base plus offset instructions

- dynamic base address in a register plus a static offset (displacement)

```
ld 4(r1), r2
```



# The Revised Load-Store ISA

## ▶ Machine format for base + offset

- note that the offset will in our case always be a multiple of 4
- also note that we only have a single hex digit in instruction to store it
- and so, we will store offset / 4 in the instruction

## ▶ The Revised ISA

Name	Semantics	Assembly	Machine
<i>load immediate</i>	$r[d] \leftarrow v$	ld \$v, rd	0d-- vvvvvvvv
<i>load base+offset</i>	$r[d] \leftarrow m[r[s] + (o = p * 4)]$	ld o(rs), rd	1psd
<i>load indexed</i>	$r[d] \leftarrow m[r[s] + 4 * r[i]]$	ld (rs,ri,4), rd	2sid
<i>store base+offset</i>	$m[r[d] + (o = p * 4)] \leftarrow r[s]$	st rs, o(rd)	3spd
<i>store indexed</i>	$m[r[d] + 4 * r[i]] \leftarrow r[s]$	st rs, (rd,ri,4)	4sdi

# Dynamic Allocation


# Dynamic Allocation in C and Java

- ▶ Programs can allocate memory dynamically
  - allocation reserves a range of memory for a purpose
  - in Java, instances of classes are allocated by the **new** statement
  - in C, byte ranges are allocated by call to **malloc** function
- ▶ Wise management of memory requires deallocation
  - memory is a scarce resource
  - deallocation frees previously allocated memory for later re-use
  - Java and C take different approaches to deallocation
- ▶ How is memory deallocated in Java?
- ▶ Deallocation in C
  - programs must explicitly deallocate memory by calling the **free** function
  - **free** frees the memory immediately, with no check to see if its still in use

# Considering Explicit Delete

## ▶ Let's look at this example

```
struct MBuf * receive () {  
    struct MBuf* mBuf = (struct MBuf*) malloc (sizeof (struct MBuf));  
    ...  
    return mBuf;  
}  
  
void foo () {  
    struct MBuf* mb = receive ();  
    bar (mb);  
    free (mb);  
}
```



- is it safe to free mb where it is freed?
- what bad thing can happen?

## ▶ Let's extend the example to see

- what might happen in bar()
- and why a subsequent call to bat() would expose a serious bug

```
struct MBuf * receive () {  
    struct MBuf* mBuf = (struct MBuf*) malloc (sizeof (struct MBuf));  
    ...  
    return mBuf;  
}
```

```
void foo () {  
    struct MBuf* mb = receive ();  
    bar (mb);  
    free (mb);  
}
```

```
void MBuf* aMB;
```

```
void bar (MBuf* mb) {  
    aMB = mb;  
}
```

```
void bat () {  
    aMB->x = 0;  
}
```

This statement writes to unallocated (or re-allocated) memory.



# Dangling Pointers

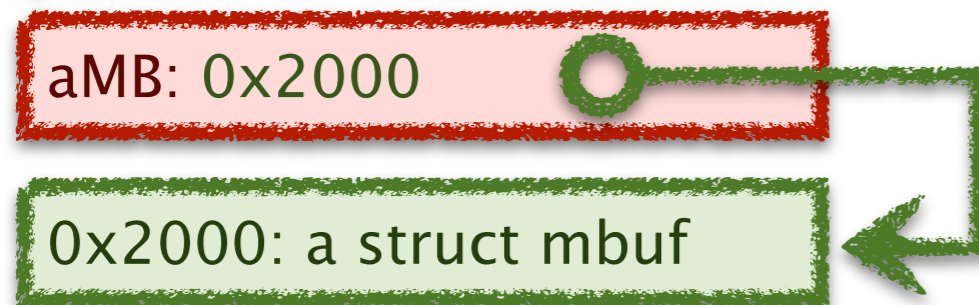
## ▶ A dangling pointer is

- a pointer to an object that has been freed
- could point to unallocated memory or to another object

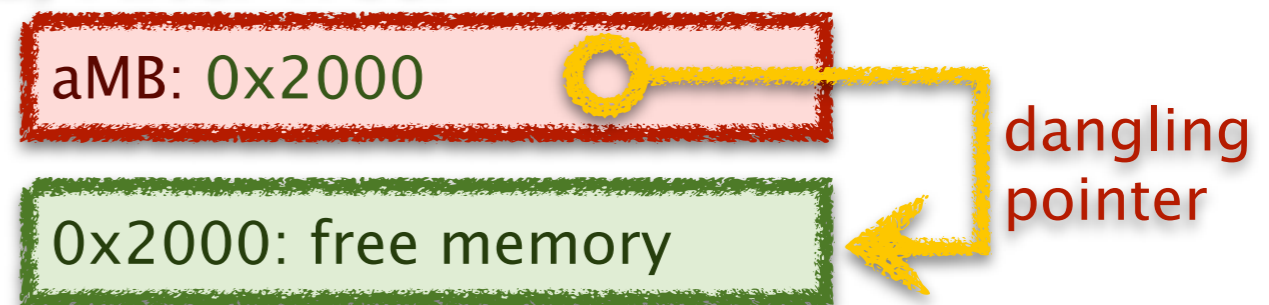
## ▶ Why they are a problem

- program thinks its writing to object of type X, but isn't
- it may be writing to an object of type Y, consider this sequence of events

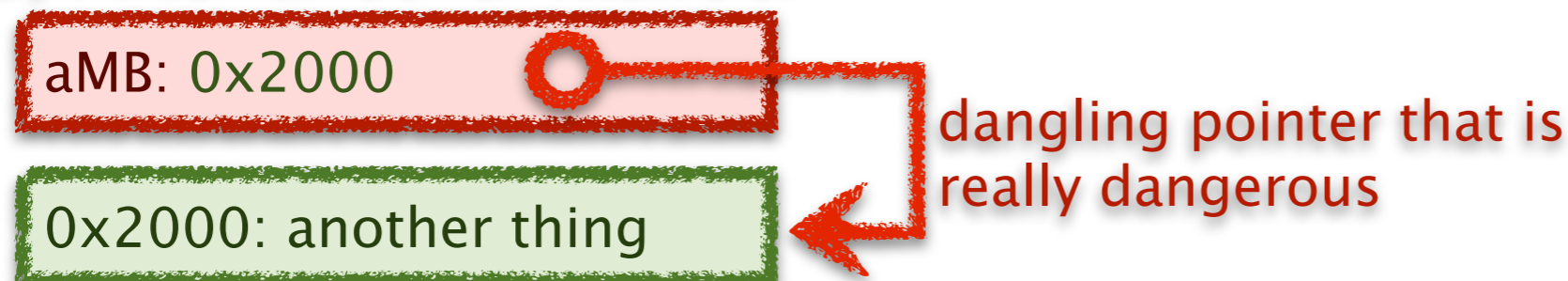
(1) Before free:



(2) After free:



(3) After another malloc:



# Avoiding Dangling Pointers in C

## ▶ Understand the problem

- when allocation and free appear in different places in your code
- for example, when a procedure returns a pointer to something it allocates

## ▶ Avoid the problem cases, if possible

- restrict dynamic allocation/free to single procedure, if possible
- don't write procedures that return pointers, if possible
- use local variables instead, where possible
  - we'll see later that local variables are automatically allocated on call and freed on return

## ▶ Engineer for memory management, if necessary

- define rules for which procedure is responsible for deallocation, if possible
- implement explicit reference counting if multiple potential deallocators
- define rules for which pointers can be stored in data structures
- use coding conventions and documentation to ensure rules are followed

# Avoiding dynamic allocation

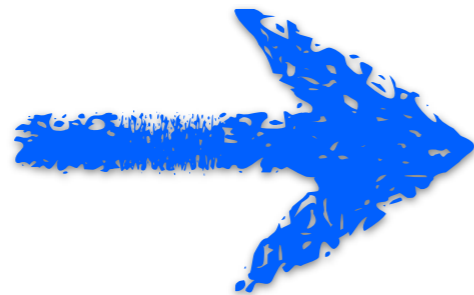
- ▶ If procedure returns value of dynamically allocated object
  - allocate that object in *caller* and pass pointer to it to *callee*
  - good if caller can allocate on stack or can do both malloc / free itself

```
struct MBuf * receive () {  
    struct MBuf* mBuf = (struct MBuf*) malloc (sizeof (struct MBuf));  
    ...  
    return mBuf;  
}
```

```
void foo () {  
    struct MBuf* mb = receive ();  
    bar (mb);  
    free (mb);  
}
```

```
void receive (struct MBuf* mBuf) {  
    ...  
}
```

```
void foo () {  
    struct MBuf mb;  
    receive (&mb);  
    bar (mb);  
}
```





# Reference Counting

- ▶ Use reference counting to track object use
  - any procedure that stores a reference increments the count
  - any procedure that discards a reference decrements the count
  - the object is freed when count goes to zero

```
struct MBuf* malloc_Mbuf () {  
    struct MBuf* mb = (struct MBuf* mb) malloc (sizeof (struct MBuf));  
    mb->ref_count = 1;  
    return mb;  
}
```

```
void keep_reference (struct MBuf* mb) {  
    mb->ref_count ++;  
}
```

```
void free_reference (struct MBuf* mb) {  
    mb->ref_count --;  
    if (mb->ref_count==0)  
        free (mb);  
}
```

- ▶ The example code then uses reference counting like this

```
struct MBuf * receive () {
    struct MBuf* mBuf = malloc_Mbuf ();
    ...
    return mBuf;
}

void foo () {
    struct MBuf* mb = receive ();
    bar (mb);
    free_reference (mb);
}

void MBuf* aMB = 0;

void bar (MBuf* mb) {
    if (aMB != 0)
        free_reference (aMB);
    aMB = mb;
    keep_reference (aMB);
}
```

# Garbage Collection

- ▶ In Java objects are deallocated implicitly
  - the program never says free
  - the runtime system tracks every object reference
  - when an object is unreachable then it can be deallocated
  - a *garbage collector* runs periodically to deallocate unreachable objects
- ▶ Advantage compared to explicit delete
  - no dangling pointers

```
MBuf receive () {  
    MBuf mBuf = new MBuf ();  
    ...  
    return mBuf;  
}  
  
void foo () {  
    MBuf mb = receive ();  
    bar (mb);  
}
```

# Discussion

- ▶ What are the advantages of C's explicit delete
- ▶ What are the advantages of Java's garbage collection
- ▶ Is it okay to ignore deallocation in Java programs?

# Memory Management in Java

## ▶ Memory leak

- occurs when the garbage collector fails to reclaim unneeded objects
- memory is a scarce resource and wasting it can be a serious bug
- its huge problem for long-running programs where the garbage accumulates

## ▶ How is it possible to create a memory leak in Java?

- Java can only reclaim an object if it is unreachable
- but, unreachability is only an approximation of whether an object is needed
- an unneeded object in a hash table, for example, is never reclaimed

## ▶ The solution requires engineering

- just as in C, you must plan for memory deallocation explicitly
- unlike C, however, if you make a mistake, you can not create a dangling pointer
- in Java you remove the references, Java reclaims the objects

## ▶ Further reading

- [http://java.sun.com/docs/books/performance/1st\\_edition/html/JPAAppGC.fm.html](http://java.sun.com/docs/books/performance/1st_edition/html/JPAAppGC.fm.html)

# Ways to Avoid Unintended Retention

**ENRICHMENT:** You are not required to know this

## ▶ imperative approach with *explicit reference annulling*

- explicitly set references to NULL when referent is longer needed
- add close() or free() methods to classes you create and call them explicitly
- use try-finally block to ensure that these *clean-up* steps are always taken
- ***these are imperative approaches; drawbacks?***

## ▶ declarative approach with *reference objects*

- refer to objects without requiring their retention
- store object references that the garbage collector can reclaim

```
WeakReference<Widget> weakRef = new WeakReference<Widget>(widget);  
Widget widget                = weakRef.get() // may return NULL
```

### • different levels of reference stickiness

- soft           discarded only when new allocations put pressure on available memory
- weak           discarded on next GC cycle when no stronger reference exists
- phantom       unretrievable (get always returns NULL), used to register with GC reference queue

# Using Reference Objects

**ENRICHMENT:** You are not required to know this

## ▶ Creating a reclaimable reference

- the Reference class is a template that be instantiated for any reference
- store instances of this class instead of the original reference

```
void bar (MBuf mb) {  
    aMB = new WeakReference<Mbuf>(mb);  
}
```

- allows the garbage collector to collect the MBuf even if aMB points to it

## ▶ This does not reclaim the weak reference itself

- while the GC will reclaim the MBuf, it can't reclaim the WeakReference
- the problem is that aMB stores a reference to WeakReference
- not a big issue here, there is only one
- but, what if we store a large collection of weak references?

# Using Reference Queues

**ENRICHMENT:** You are not required to know this

## ▶ The problem

- reference objects will be stored in data structures
- reclaiming them requires first removing them from these data structures

## ▶ The reference queue approach

- a reference object can have an associated reference queue
- the GC adds reference objects to the queue when it collects their referent
- your code scans the queue periodically to update referring data structures

```
ReferenceQueue<MBuf> refQ = new ReferenceQueue<MBuf> ();

void bar (MBuf mb) {
    aMB = new WeakReference<Mbuf> (mb,refQ);
}

void removeGarbage () {
    while ((WeakReference<Mbuf> ref = refQ.poll()) != null)
        // remove ref from data structure where it is stored
        if (aMB==ref)
            aMB = null;
}
```