CPSC 213

Introduction to Computer Systems

Summer Session 2019, Term 2

Unit 1c – Jul 9, 11

Structs and Dynamic Allocation

Overview

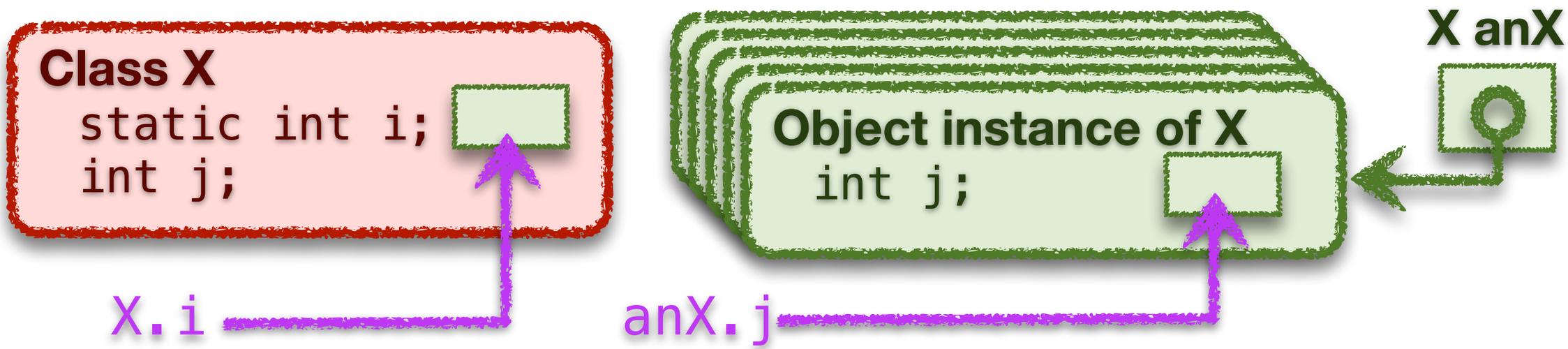
- Reading
  - Companion: 2.4.4-2.5
- Reference
  - Textbook: 3.9.1, 9.9, 3.10
- Learning Objectives
  - read and write C code that includes structs
  - compare Java classes/objects with C structs
  - explain the difference between static and non-static variables in Java and C
  - explain why ISAs have both base-plus-offset and indexed addressing modes by showing how each is useful for implementing specific features of programming languages like C and Java
  - distinguish static and dynamic computation for access to members of a static struct variable in C
  - distinguish static and dynamic computation for access to members of a non-static struct variable in C
  - translate C struct-access code into assembly language
  - count memory references required to access struct elements
  - compare Java dynamic allocation the C's explicit-delete approach by identifying relative strengths and weaknesses
  - identify and correct dangling-pointer and memory leak bugs in C caused by improper use of free()
  - write C code that uses techniques that avoid dynamic allocation as a way to minimize memory-allocation bugs
  - write C code that uses reference counting as a way to minimize memory-allocation bugs
  - explain why Java code does not have dangling-reference errors but can have memory-leak errors

So Far …

The Simplest Variables in Any Language

- Static Variables
  - the compiler allocates them; i.e., their memory address is a constant
  - ADVANTAGES:
  - DISADVANTAGES:
- Scalars and Arrays
  - a scalar is a variable that stores a single value
  - an array stores multiple values named by a single variable and an index
    - the array data can be allocated either statically or dynamically
    - the array variable can either BE the array (static) or can store a pointer to it (dynamic)
    - the index is generally a dynamic value
- Now for some other types of variables …

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 programer
  - 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

```
struct D  d0;
```

Static Memory Layout

```
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`**

```
struct D* d1;
```

Static Memory Layout

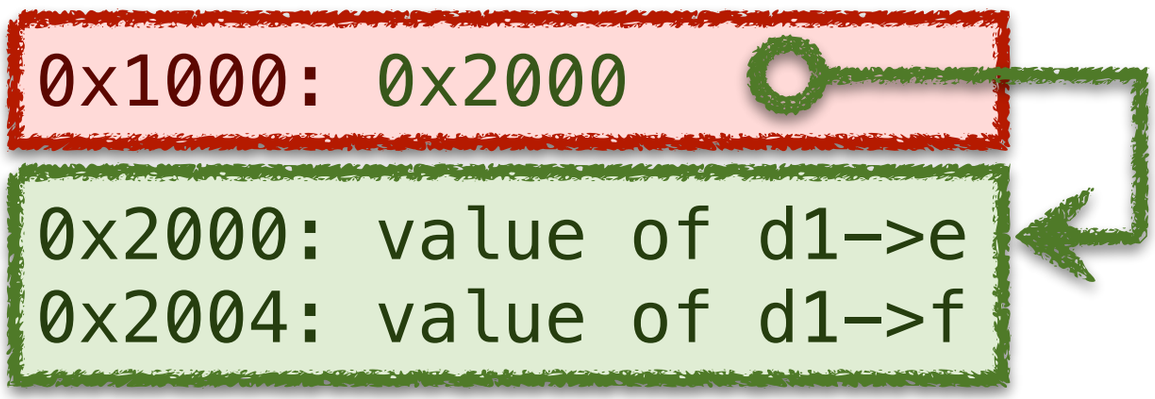
```
0x1000: value of d1
```

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

- runtime allocation of dynamic struct

```
void foo() {
  d1 = 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 load/store base plus offset instructions

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

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

The 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 instruction byte to store it
  - and so, we will store offset / 4 in the instruction
- The Revised ISA

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

Memory Addressing Modes

- Scalars

```
i = a
```

  - address in register (`r1`)
  - access memory at address in register

```
ld (r1), r0
```

- Arrays

```
i = a[j]
```

  - base address in register (`r1`) r1 stores address of a
  - ***dynamic*** index in register (`r2`) r2 stores value of j
  - access memory at base plus index times element-size (4)

```
ld (r1,r2,4), r0
```

- Struct Members (Instance Variables)

```
i = a.j
```

```
i = b->k
```

  - base address in register (`r1`) r1 stores address of a or value of b
  - ***static***/constant offset (X) X is offset to j/k from beginning of struct
  - access memory at base plus offset

```
ld X(r1), r0
```

CPSC 213

Introduction to Computer Systems

Summer Session 2019, Term 2

Unit 1c – Jul 9, 11

Structs and Dynamic Allocation

Structs Declared Inside of Other Structs

- The struct variable is
  - the variable that stores the struct for static structs or
  - the variable that stores a pointer to the struct or dynamic structs
- Struct variables can be declared inside of other structs

```
struct W {
  int a;
  int j;
};
```

```
struct A {
  int a;
  int b;
};
```

```
struct Y {
  int i;
  struct A* s;
  int j;
};
```

```
struct X {
  int i;
  struct A s;
  int j;
};
```

  - what are the *offsets* to the `j`'s?
- A struct variable of type X can be declared inside a struct X

```
struct Z {
  int i;
  struct Z *s;
  int j;
};
```

Size and Alignment

- Alignment rules apply inside of a struct
  - each instance variable will be aligned according to its type size
  - structs are aligned according to their largest instance variable type
- Structs can mix types - but *padding* might be added

```
struct X {
  char a;
  char b;
  int i;
};
```

```
struct Y {
  char a;
  int i;
  char b;
};
```

  - what are the *sizes* of the structs above? (i.e. sizeof(struct X))
- Similarly, struct members can be arrays

```
struct X {
  int i;
  int a[10];
  int j;
};
```

```
struct Y {
  int  i;
  int* a;
  int  j;
};
```

  - what are the *offsets* to the **j**'s?

Arrays of Structs

- Array of structs

```
struct S a[10];
```

- Array of pointers to structs

```
struct S* b[10];
```

- Or combine struct and array declaration
  - struct name is optional (but recommended)

```
struct {
  int i[2];
  char c;
} c[10];
```

```
struct {
  int i[2];
  char c;
} *d[10];
```

```
struct S {
  int i[2];
  char c;
};
```

Aside: What's this?

```
int *e[10];
```

Struct Values

- Structs can be copied (by value!)

```
struct S s0;
struct S s1;
s0 = s1;
```

```
struct S {
  int i[2];
  char c;
};
```

- Structs can be returned (by value!)

```
struct S makeS() {
  struct S s = { { 0, 1 }, 'c'};
  return s;
}
```

- Be careful with pointers inside structs!

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` procedure
- In C

```
void *malloc (n)
```

  - `n` is the number of bytes to allocate
  - `void*`
    - is a pointer that is automatically cast to/from any other pointer
    - can not be dereferenced directly
- Use `sizeof` to determine number of bytes to allocate
  - `sizeof(x)` statically computes # bytes in type or variable
    - why can't you use `sizeof()` to compute the size of a dynamic array?

```
struct Foo *f = malloc(sizeof(struct Foo))
```

Deallocation in C and Java

- 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
- **Memory Leak**
  - when dynamically allocated data is not deallocated when it is no longer needed
  - size of program gradually increases … available memory leaks away
- How is memory deallocated in Java?
- Deallocation in C
  - programs must explicitly deallocate memory by calling the free procedure
  - free releases the memory immediately, with no check to see if its still in
- The Heap
  - malloc and free work together to manage an abstraction called the *heap*
  - the *heap* is a large section memory from which malloc allocates objects
  - all objects are stored in the *heap*

The Problem with Explicit Delete

```
struct Foo* f = malloc(sizeof(struct Foo))
free(f);
```

You should add this after the free (but its not enough)

```
f = 0;
```

- What `free(f)` does
  - deallocates “object” at address `f`
  - so that this memory can be reused by subsequent call to `malloc`
- What `free(f)` does not do
  - it doesn’t change the value of f
    - what would have to be true of the language to change the value of `x`
- Therefore
  - after a call `free(f)`
  - the variable f still points at the freed object
  - **other variables may point there too …**

Considering Explicit Delete

- Let's look at this example

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

How can you be sure that this memory is eventually freed?

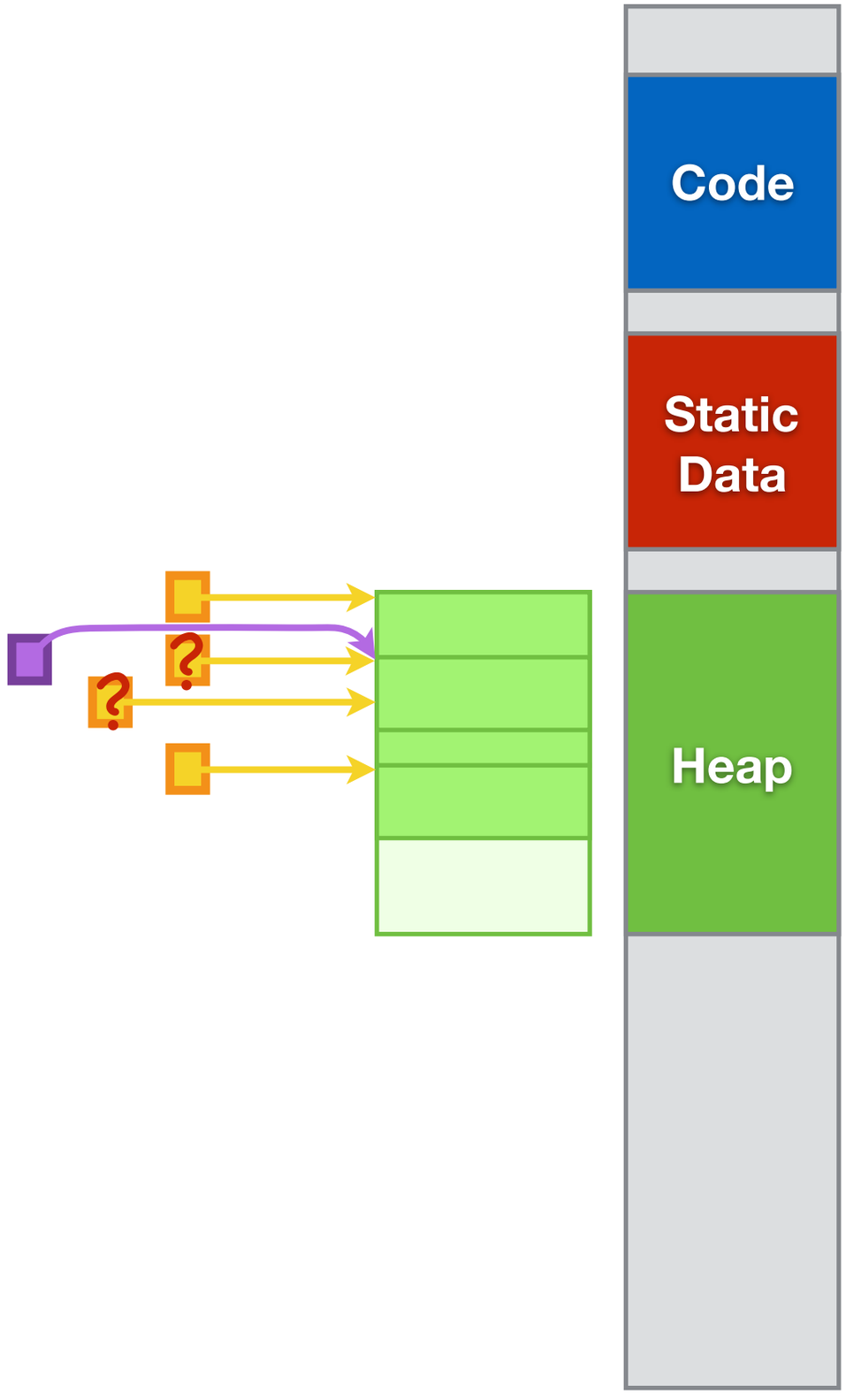
```
void foo() {
  struct MBuf *mb = receive();
  bar(mb);
  free(mb);
  mb = 0;
}
```

Is it safe to free it here?

- what bad things can happen?

What actually happens in malloc/free?

- Organization of memory
  - statically allocated
    - code
    - static data
  - the rest of memory
    - unused or dynamically allocated
- Malloc/Free
  - implemented in library
  - manage a chunk of memory called the *Heap*
  - keep track of what's allocated or free
- Malloc `a = malloc(16);`
  - find a free memory
  - mark it allocated
  - return pointer to it
    - keeping track of its size
- Free `free(a);`
  - use pointer to determine allocated size
  - mark referenced memory as free



The Problem with Explicit Delete

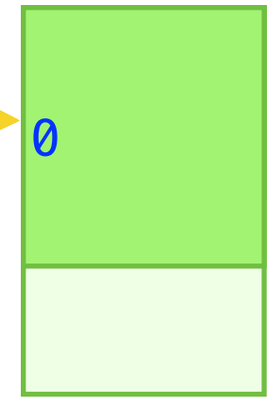
- Lets extend the example to see
  - what might happen in bar()
  - and why a subsequent call to baz() would expose a serious bug

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

void foo() {
  struct MBuf *mb = receive();
2 bar(mb);
4 free(mb);
  baz();
}

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

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

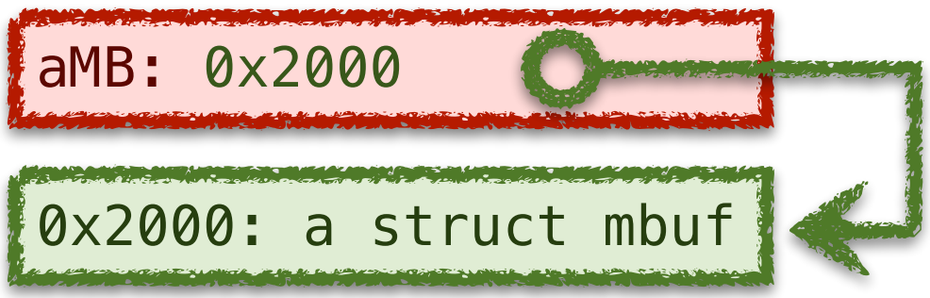


This statement writes to unallocated (or re-allocated) memory.

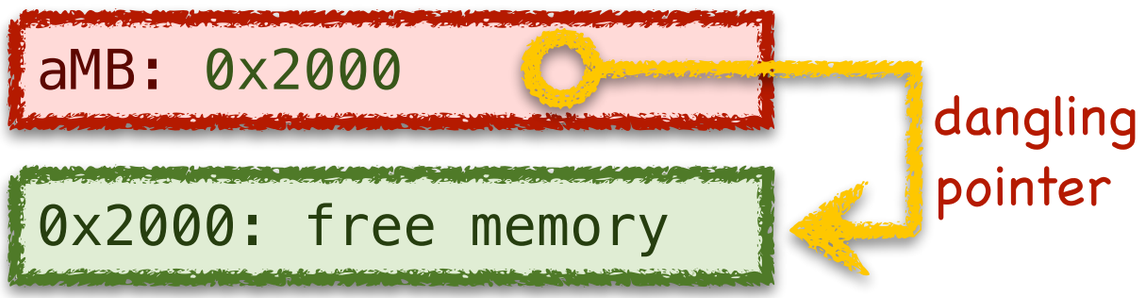
Dangling Pointers

- 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

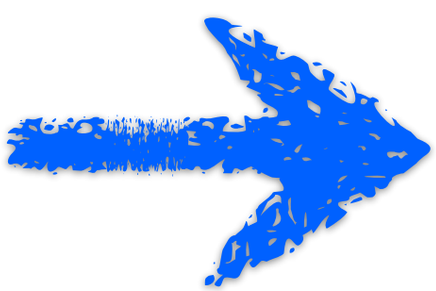
- Understand the problem
  - when allocation and free appear in different places in your code
  - for example, when a procedure returns a pointer to something it allocates
- Avoid the problem cases, if possible
  - restrict dynamic allocation/free to single procedure or module, if possible
  - don't 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

Co-locate Allocation and Deallocation

- If procedure returns value of dynamically allocated object
  - allocate that object in ***caller*** and pass pointer to it to ***callee***
  - good if caller does both malloc / free itself

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

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



```
void receive(struct MBuf *mBuf) {
  ...      Delegate problem to caller
}

void foo() {
  struct MBuf *mb = malloc(sizeof(struct MBuf);
  receive(mb);
  free(mb);      Transfer malloc to foo
}
```

Another Example – Strings

- A string “like this” is in C
  - an array of characters
  - the end of the string is indicated by the first 0 (null) in the array
  - so, every string has a
    - maximum length (the length of the array - 1)
    - and a length determined by the position of the first null
- The standard C library has many operations on strings
  - e.g., strlen(s) returns the length of a string
- Lets consider
  - how best to create a new string
  - that is a copy of an existing string

String Copy Version 1

```
// in the string library
char *copy(char *s) {
  int len = strlen(s);
  char *d = malloc(len + 1);
  for (int i=0; i<len+1; i++)
    d[i] = s[i];
  return d;
}

// in your application program
void foo(char *s) {
  char *d = copy(s);
  printf("%s\n", d);
  free(d);
}

void bar() {
  foo("Hello, World!");
}
```

- What don't you like?
- How do improve this code?

String Copy Version 2

```
// in the string library
void copy(char *d, char *s) {
  int len = strlen(s);
  for (int i=0; i < len+1; i++)
    d[i] = s[i];
}

// in your application program
void foo(char *s) {
  int len = strlen(s);
  char *d = malloc(len + 1);
  copy(d, s);
  printf ("%s\n", d);
  free(d);
}
```

- Pros?
- Cons?

String Copy Version 3

```
// in the string library
void copy(char *d, char* s, int dSize) {
  int sLen = strlen(s);
  if (sLen > dSize - 1)
    sLen = dSize - 1;
  for (int i=0; i < sLen; i++)
    d[i] = s[i];
  d[sLen] = 0;
}

// in your application program
void foo(char *s) {
  int len = strlen(s);
  char* d = malloc(len + 1);
  copy(d, s, len+1);
  printf ("%s\n", d);
  free(d);
}
```

- Pros?
- Cons?
- stdlib
  - strdup, strcpy, strlcpy/strncpy

But, sometimes we need more

- If a reference needs to be passed among multiple modules
  - then each module knows when it needs and doesn't need reference
  - but, what do you need to know in order to correctly free referent?
  - why is this bad?
  - can we do better?

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

```
void foo() {
  struct MBuf *mb = receive();
  bar(mb);
  free(mb);  // ???
  mb = 0;
}
```

Consider interaction between foo & bar

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

void foo() {
  struct MBuf *mb = receive();
  bar(mb);
  // maybe do something else with mb
  free(mb) ??? is it okay to do this ???
}

struct MBuf *aMB = 0;

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

Reference Counting

- Struct can store a *reference count*
  - that records the number of variables that store pointers to the struct
  - maintained explicitly by code the programmer adds (i.e., this is *manual reference counting*)
- Allocation of reference-counted struct
  - set the reference count to 1
  - because the caller has a reference
- Saving additional references
  - call `rc_keep_ref` to increment the reference count
- Just before a reference is removed
  - call `rc_free_ref` to decrement reference count
- Never call `free` directly
  - `free_ref` calls `free` automatically when the reference count goes to zero
- Implementation
  - this is something you would have to implement yourself
  - for example, need a sort of super class that implements `rc_keep_ref()` and `rc_free_ref()`
  - more shortly …

- The example code then uses reference counting like this

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

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

struct MBuf *aMB = 0;

void bar(struct MBuf *mb) {
  // free existing reference in aMB
  if(aMB != 0)
    rc_free_ref(aMB);
  rc_keep_ref(aMB);
  aMB = mb;
}
```

mb->ref_count==1

mb==aMB; mb->ref_count==2

mb==aMB; mb->ref_count==1

if aMB currently points to an MBuf, changing its value removes a reference to that MBuf. As this is the last reference to the MBuf, it is freed from the heap.

aMB!=mb; aMB->ref_count==1

free (aMB) is called

mb->ref_count==2

aMB==mb; aMB->ref_count==2

Implementing Reference Counting

refcount.h

```
void *rc_malloc   (int nbytes);   alternative malloc with room for ref_count
void  rc_keep_ref (void *p);      call when reference added
void  rc_free_ref (void *p);      call when reference removed
```

refcount.c

```
void *rc_malloc(int nbytes) {
   int *ref_count = malloc(nbytes + 8);
   *ref_count = 1;
   return ((void *)ref_count) + 8;
}

void rc_keep_ref(void *p) {
   int *ref_count = p - 8;
   (*ref_count)++;
}

void rc_free_ref(void *p) {
   int *ref_count = p - 8;
   (*ref_count)--;
   if(*ref_count == 0)
      free(ref_count);
}
```

SIDE NOTES: including header files, compiling multiple files, and valgrind.

Reference Counting Example

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
  - reference cycles are not a problem
    - cycles & reference counting => memory leak
    - cycles are collected by garbage collector

```
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 advantages of reference counting?

- 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 serous 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://www.ibm.com/developerworks/library/j-leaks/

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;
}
```