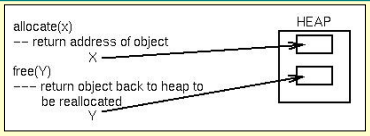
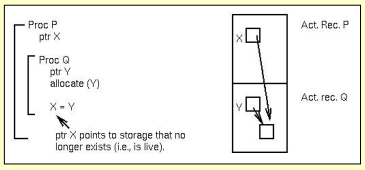
Heap storage

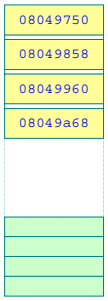


Dynamic allocation and stacks are generally incompatible.



C Run Time allocation

```
Call # 1
Address of memPtr (on stack): bffffac4
 malloc'ed memory (on heap) : 08049750

Call # 2
Address of memPtr (on stack): bffffaa4
 malloc'ed memory (on heap) : 08049858

Call # 3
Address of memPtr (on stack): bffffa84
 malloc'ed memory (on heap) : 08049960

Call # 4
Address of memPtr (on stack): bffffa64
 malloc'ed memory (on heap) : 08049a68
```

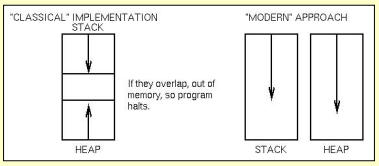


UNCA CSCI 333 / 431

Stack and heap location

Pointer X points to stack storage in procedure Q's activation record that no longer is live (exists) when procedure Q terminates. Such a reference is called a dangling reference.

Dynamic storage is usually a separate structure in C, Ada, Pascal ...



Allocation in C++ -- common structure

Here's a class to use in our examples

```
template <class Elem> class Link {
public:
  Elem element;       // Value for this node
  Link *next;         // Pointer to next node in list
  Link(const Elem& elemval, Link* nextval = (Link*)0)
    { element = elemval;  next = nextval; }
  Link(Link* nextval = (Link*)0) { next = nextval; }
};
```

UNCA CSCI 333 / 431

Looking at allocation in C

```
#define MAXCALL 3

void f(int ncall) {
  char *memPtr ;                                  // Stack allocation
  memPtr = (char *)malloc((size_t) 256) ;         // Heap allocation
  printf("\nCall # %d\n", ncall) ;
  printf("Address of memPtr (on stack): %08x\n", &memPtr) ;
  printf(" malloc'ed memory (on heap) : %08x\n", memPtr) ;
  if (ncall <= MAXCALL)
    f(ncall+1) ;
}

int main(int argc, char *argv[]) {
  f(1) ;
  return 0;
}
```

UNCA CSCI 333 / 431

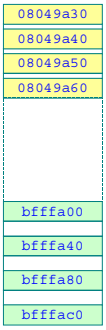
Allocation in C++ -- static vs. dynamic

```
#define MAXCALL 3

void f(int ncall) {
  Link<int>  staticL(200) ;          // Record stored on stack
  Link<int> *dynamoL ;               // Record stored in heap
  dynamoL = new Link<int>(100) ;
  printf("\nCall # %d\n", ncall) ;
  printf("Address of staticL (on stack): %08x\n", &staticL) ;
  printf("Address of dynamoL  (on heap): %08x\n", dynamoL) ;
  if (ncall <= MAXCALL)
    f(ncall+1) ;
}

int main(int argc, char *argv[]) {
  f(1) ;
  return 0;
}
```

UNCA CSCI 333 / 431

C++ Run Time Allocation

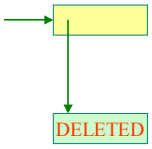
```
Call # 1
Address of staticL (on stack): bffffac0
Address of dynamoL  (on heap): 08049a30

Call # 2
Address of staticL (on stack): bffffa80
Address of dynamoL  (on heap): 08049a40

Call # 3
Address of staticL (on stack): bffffa40
Address of dynamoL  (on heap): 08049a50

Call # 4
Address of staticL (on stack): bffffa00
Address of dynamoL  (on heap): 08049a60
```



UNCA CSCI 333 / 431

Garbage pointers in C++

```
Link<int> *BadStackPointer(void) {
  Link<int>  staticL(200) ;
  Link<int> *dynamoL ;
  dynamoL = new Link<int>(100, &staticL) ;
  return dynamoL ;
}
```



UNCA CSCI 333 / 431

C++ {de,con}structor

Constructor called
- Implicitly on "declaration" within static scope
- Explicitly by new

Destructor called
- Implicitly on exit of static scope
- Explicitly by delete

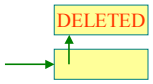
```
void f(void) {
  BigTable<float>  StaticBT(300) ;
  BigTable<float> *DynamoBT ;
  DynamoBT = new BigTable<float>(500) ;
  delete DynamoBT ;
}
```

Classes which allocate dynamic storage need destructors

UNCA CSCI 333 / 431

Garbage pointers in C++

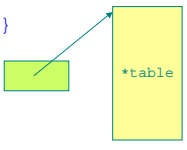
```
Link<int> *BadHeapPointer(void) {
  Link<int> *dynamoL1, *dynamoL2 ;
  dynamoL2 = new Link<int>(200) ;
  dynamoL1 = new Link<int>(100, dynamoL2) ;
  delete dynamoL2 ;
  return dynamoL1 ;
}
```



UNCA CSCI 333 / 431

C++ class with {de,con}structor

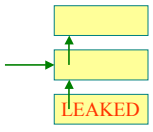
```
template <class Elem> class BigTable {
private:
  int  tsize;        // Size of table
  Elem *table;       // Table pointer
public:
  BigTable(int size = 100)
    { tsize = size ; table = new Elem[tsize]; }
  ~BigTable()
    { delete[] table ; }
  … other methods
};
```



UNCA CSCI 333 / 431

Memory leak in C++

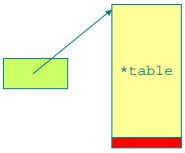
```
Link<int> *LeakMemory(void) {
  Link<int> *DynamoL1, *DynamoL2, *DynamoL3 ;
  DynamoL3 = new Link<int>(300) ;
  DynamoL2 = new Link<int>(200, DynamoL3) ;
  DynamoL1 = new Link<int>(100, DynamoL2) ;
  return DynamoL2 ;
}
```



UNCA CSCI 333 / 431

Memory overrun in C++

```
template <class Elem> class BigTable {
private:
  int  tsize;        // Size of table
  Elem *table;       // Table pointer
public:
  BigTable(int size = 100)
    { tsize = size ; table = new Elem[tsize]; }
  ~BigTable()
    { delete[] table ; }
  SetLast(Elem& newLast)
    {table[tsize] = newLast ; }
  … other methods
};
```



UNCA CSCI 333 / 431

Sequential fit allocation

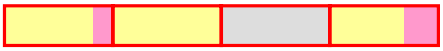
- Extra fields track important parameters
  - In allocated blocks
    - Size of allocated block
    - State of preceding block
  - In free blocks
    - Doubly linked list of allocated blocks
    - Size of free block (at end)

Used in G++ malloc

UNCA CSCI 333 / 431

Internal Fragmentation

- Memory lost inside an allocated block
  - Occurs with fixed-sized allocation is bigger than the data block



UNCA CSCI 333 / 431

Allocation strategies

Many slots are big enough, which to use?

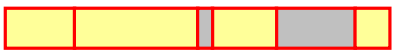
- Best fit
  - Use the smallest
- Worse fit
  - Use the biggest
- First fit
  - Use the first

For each method,
you can find a situation where only that method works

UNCA CSCI 333 / 431

External Fragmentation

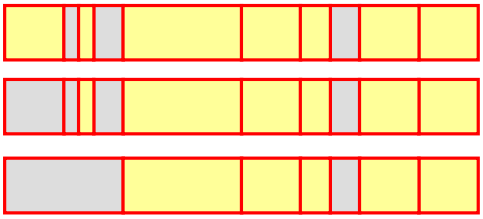
- External fragmentation
  - Memory lost outside an allocated block
    - Deleting data may leave a small hole
    - Allocating data within a large hole leaves a small hole



UNCA CSCI 333 / 431

Buddy allocation

- Memory allocated in blocks of size 2^k
  - Flip the k'th bit to find your buddy
    - 0x08ff8a00 and 0x8ff8b00 are buddies of size 256!
  - When two buddies are freed
    - A larger free block is formed



UNCA CSCI 333 / 431

Page-based bucket allocation

Buckets
- Consist of a set of pages
- Each pages contains blocks of a fixed size
  - Except some blocks may need several pages

Used in operating systems
- Where page tables may contain bucket identification

If you know the address of a block
- You know its size
- *No wasted space for block size information*

UNCA CSCI 333 / 431

Roots of Garbage Collection

Garbage collection begins with a set of roots
- Pointers stored on the stack
  - In multi-threaded applications
    - roots may be stored on several thread stacks
- Pointers stored in global data

Collection proceeds as a search from the roots
- Pointers lead to records/structures
- Records may contain new pointers

GC algorithm must be able to recognize pointers
- In local and global data
- In records

Java Platform Performance: Strategies and Tactics Appendix A

UNCA CSCI 333 / 431

Allocation in Compaq tru64 Unix kernel

```
woodfin% vmstat -M
bucket#  element_size  elements_in_use  elements_free
     0            16            11310          11730
     1            32             3691            661
     2            64             3003            325
     3           128             3589            891
     4           256              423            473

    18           192             2314            542
    19           320             1868            332
    20           384              240            243
    21           448              247            203
    22           576             1755            681
    23           640               30             18
    24           704               18             15
```

Garbage collection: Mark-sweep algorithm

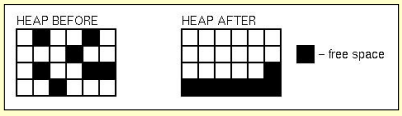
First assume fixed size blocks of size k. (Later blocks of size n*k.)
- This is the LISP case.

Two simple algorithms follow. (This is only an introduction to this topic. Many other algorithms exist)

Algorithm 1: Fixed size blocks;
Two pass mark-sweep algorithm:
1. Keep a linked list (free list) of objects available to be allocated.
2. Allocate objects on demand.
3. Ignore free commands.
4. When out of space, perform garbage collection:
   Pass 1. Mark every object still allocated.
   Pass 2. Every unmarked objects added to free list of available blocks.

Garbage collection goal

Process to reclaim memory.(Solve Fragmentation problem.)



Algorithm: You can do garbage collection if you know where every pointer is in a program. If you move the allocated storage, simply change the pointer to it.
- This is true in LISP, ML, Java, Prolog
- Not true in C, C++, Pascal, Ada

Heap storage errors

Error conditions:

Dangling reference - Cannot occur. Each valid reference will be marked during sweep pass 1.

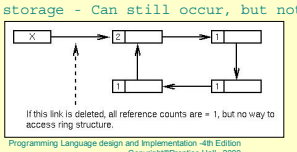
Inaccessible storage - Will be reclaimed during sweep pass 2.

Fragmentation - Does not occur; fixed size objects.

Garbage collection: reference counts

Algorithm 2:
1. Associate a reference counter field, initially 0, with every allocated object.
2. Each time a pointer is set to an object, up the reference counter by 1.
3. Each time a pointer no longer points to an object, decrease the reference counter by 1.
4. If reference counter ever is to 0, then free object.

Error conditions:
Dangling reference - Cannot occur. Reference count is 0 only when nothing points to it.
Fragmentation - Cannot occur. All objects are fixed size.
Inaccessible storage - Can still occur, but not easily.



Reference counts in file systems

In Unix, files are "addressed" by inode number
File reference counts determine if the file has a name
When the reference count reaches 0,
the file's data is deleted.

A similar facility exists in NTFS 5.0

```
woodfin% echo "A new file" > Ref1
woodfin% cp Ref1 Copy
woodfin% ln Ref1 Ref2
woodfin% ls -li
23884 -rw-r--r--   1 brock    system        11 Nov 10 16:31 Copy
23853 -rw-r--r--   2 brock    system        11 Nov 10 16:31 Ref1
23853 -rw-r--r--   2 brock    system        11 Nov 10 16:31 Ref2
woodfin% rm Ref1
woodfin% ls -li
23884 -rw-r--r--   1 brock    system        11 Nov 10 16:31 Copy
23853 -rw-r--r--   1 brock    system        11 Nov 10 16:31 Ref2
```

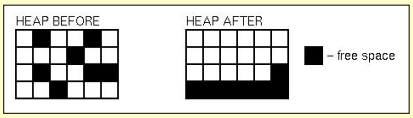
UNCA CSCI 333 / 431

Memory compaction: Variable-size elements

With variable sized blocks, the previous two algorithms will not work.

In the left heap below, a block of size 3 cannot be allocated, even though 7 blocks are free.

If the allocated blocks are moved to the start of the heap (compaction) and the free blocks collected, then a block of size up to 7 can be allocated.



Compaction algorithm

For variable sized blocks, in pass 2 of the mark-sweep algorithm:
⇒ Move blocks to top of storage
⇒ Need to reset pointers that point to the old storage location to now point to the new storage.

How to do this?
Use tombstones and two storage areas, an A area and a B area:
1. Fill area A.
2. When A fills, do mark-sweep algorithm.
3. Move all allocated storage to start of B area. Leave marker in A area so other pointers pointing to this object can be modified.
4. After everything moved, area A can discarded. Allocate in B and later compact from B back to A.

LISP storage

LISP is first language that makes heavy use of heap storage.
Storage reclaimed automatically by LISP environment when out of space.

Uses space efficiently. Each LISP item is a fixed size object allocated in the heap.

As example shows, although based on a heap, LLISP still needs a stack for execution.

Example: Trace execution of:
(defun f1(x y z) (cons x (f2 y z)))
(defun f2(v w) (cons v w))