CS-034
Memory Allocation/deallocation

Thomas Doeppner
Pascal Van Hentenryck

Naming Types

```
typedef char *String;
typedef int  *IntPtr;
```

- typedef allows you to name a type

```
int main() {
   String str;
   ...
```

Naming Types

```
typedef struct Complex {
   float r;
   float i;
} Complex, *ComplexPtr;
```

```
int main() {
   Complex c;
   ...
```

no struct needed

Memory Allocation

Static allocation

- size is known at compile time
- local variables/arguments in functions
  - stack allocated
- global variables

Dynamic allocation

- size is unknown at compile time
- heap allocated

Dynamic Allocation

```
void *malloc(size_t size);
```

size of the object

unsigned int

Pointer to the object

- How do I know the size?

Dynamic Allocation

- Why not

```
int  *malloc_int();
char *malloc_char();
```

Dynamic Allocation

`malloc` is a dangerous beast

- it may return 0
  - no memory left
- always test its result

disclaimer
not always
in these slides

Memory allocation

```
int main() {
    int* p = (int *) malloc(sizeof(int));
    if (!p) {
      printf("Get new hardware\n");
      return 1;
    }
    *p = 34;
    printf("%d\n",*p);
}
```

Memory allocation

```
int main() {

   ComplexPtr c1 = (ComplexPtr) malloc(sizeof(Complex));
   ComplexPtr c2 = (ComplexPtr) malloc(sizeof(Complex));
   ComplexPtr c3 = (ComplexPtr) malloc(sizeof(Complex));

   c1->r = 1.0; c1->i = 2.3;
   c2->r = 2.0; c2->i = 4.3;

   complexAdd(c1,c2,c3);
   printf("%f %f\n",c3->r,c3->i);
}
```

Memory allocation

```
int main() {

   ComplexPtr c1 = (ComplexPtr) malloc(sizeof(Complex));
   ComplexPtr c2 = (ComplexPtr) malloc(sizeof(Complex));
   ComplexPtr c3 = (ComplexPtr) malloc(sizeof(Complex));

   c1->r = 1.0; c1->i = 2.3;
   c2->r = 2.0; c2->i = 4.3;
}
```

- Does C need operator “->”?

Memory allocation

```
int main() {

   ComplexPtr c1 = (ComplexPtr) malloc(sizeof(Complex));
   ComplexPtr c2 = (ComplexPtr) malloc(sizeof(Complex));
   ComplexPtr c3 = (ComplexPtr) malloc(sizeof(Complex));

   (*c1).r = 1.0; (*c1).i = 2.3;
   c2->r = 2.0; c2->i = 4.3;
}
```

- Does C need operator “->”?

Memory Deallocation

Memory allocation

- allocation is on the heap
- is not released by C
  – is alive after function calls

Memory deallocation

- tell C that you do not need the space anymore
- contrast to garbage collector

Memory Deallocation

```
void free(void *ptr);
```

Tell C you do not need the space pointed to by `ptr`

- like dumping your boyfriend/girlfriend
- you can’t touch “it” anymore after that although you still have “its” address

Memory Deallocation

```
int main() {
   int* p = (int *) malloc(sizeof(int));
   if (!p) {
     printf("Get new hardware\n");
     return 1;
   }
   *p = 34;
   printf("%d\n",*p);
   free(p);
}
```

Memory Deallocation Bugs

- Using a pointer after it is freed
  - the memory may be reallocated to something else
- Freeing a pointer twice
  - the memory may be reallocated to something else several times
- Freeing a pointer not allocated by malloc
  - The memory can be allocated to something else

Debugging strategies

Remove all the `free` of your program

Check array bounds and memory accesses

There are tools available

- **Purify, watch, …**
- **Be patient …**

Array allocation

```
int main() {
    int *a;
    int size;
    scanf("%d",&size);
    a = (int *) malloc(size * sizeof(int));
    int k;
    for(k = 0; k < size; k++)
        a[k] = k;
    free(a);
}
```

number of elements

size of each element

String Copy

```
char* mystrcpy(char* str,int l) {
 char* r = (char *) malloc((l+1)*sizeof(char));
 int i = 0;
 for(i = 0; i < l; i++)
   r[i] = str[i];
 r[l] = '\0';
 return r;
}
```

More Interesting Arrays

- **assume that I want an array a[18..118]**
  - **age of the population that can vote**

```
int main() {
   int *a;
   int low, up, size, k;
   scanf("%d%d",&low,&up);
   size = up - low + 1;
   a = (int *) malloc(size *sizeof(int));
   for(k = low; k <= up; k++)
      a[k-low] = k;
}
```

More Interesting Arrays

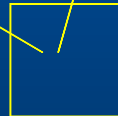
- assume that I want an array a[18..118]
  - age of the population that can vote

```
int main() {
   int *a;
   int low, up, size, k;
   scanf(”%d%d",&low,&up);
   size = up - low + 1;
   a = (int *) malloc(size *sizeof(int));
   a -= low;
   for(k = low; k <= up; k++)
      a[k] = k;
}
```

More Interesting Arrays

- assume that I want an array a[18..118]
  - age of the population that can vote

a



Can I free a?

Searching an array

```
int find(int* a,int l,int e) {
    int i = 0;
    while (i < l && a[i] != e) i++;
    if (i < l) return i;
    else return -1;
}
```

- Can I make it more efficient?

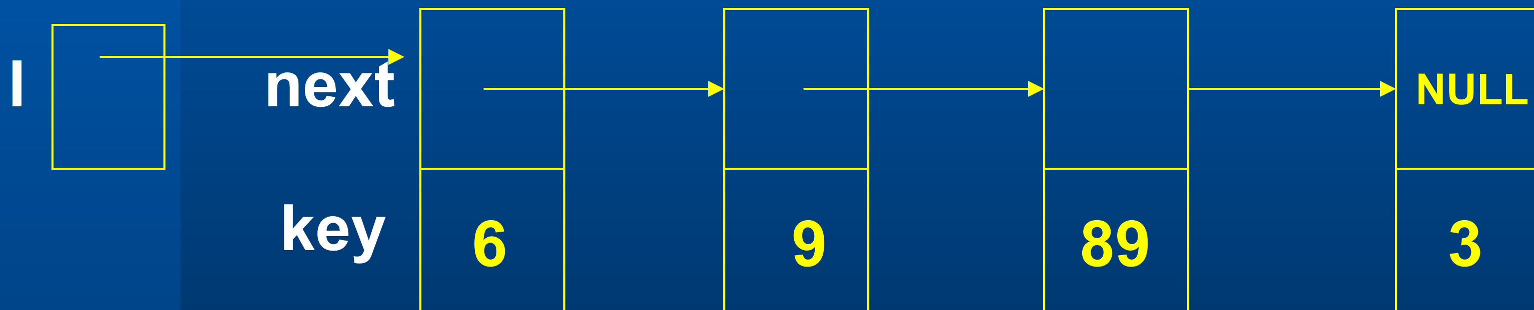
Searching an array: Sentinel

```
int find(int* a,int l,int e) {
   a[l] = e;
   int i = 0;
   while (a[i] != e) i++;
   if (i < l) return i;
   else return -1;
}
```

```
a = (int *) malloc((size+1)*sizeof(int));
```

List again, like we did last summer

```
typedef struct lnode {
   struct lnode *next;
   int key;
} ListNode, *ListNodePtr;
```

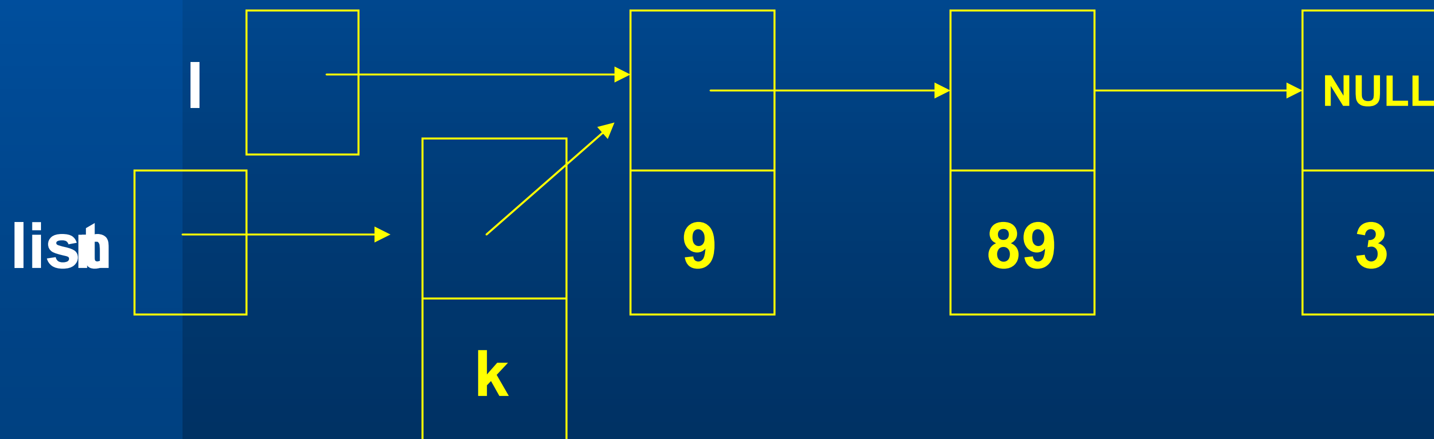


List again, like we did last summer

```
int main() {
   ListNodePtr list = 0;
   list = insertFront(list,4);
   list = insertFront(list,44);
   list = insertFront(list,444);
   list = insertFront(list,4444);
   list = insertLast(list,333);
   printf("Found: %d\n",find(list,444));
   list = delete(list,444);
   list = delete(list,4444);
   printList(list);
}
```

Insert in Front: The Java way

```
ListNodePtr insertFront(ListNodePtr l,int k) {
  ListNodePtr n = (ListNodePtr) malloc(sizeof(ListNode));
  n->key = k;
  n->next = l;
  return n;
}
```



List again, like we did last summer

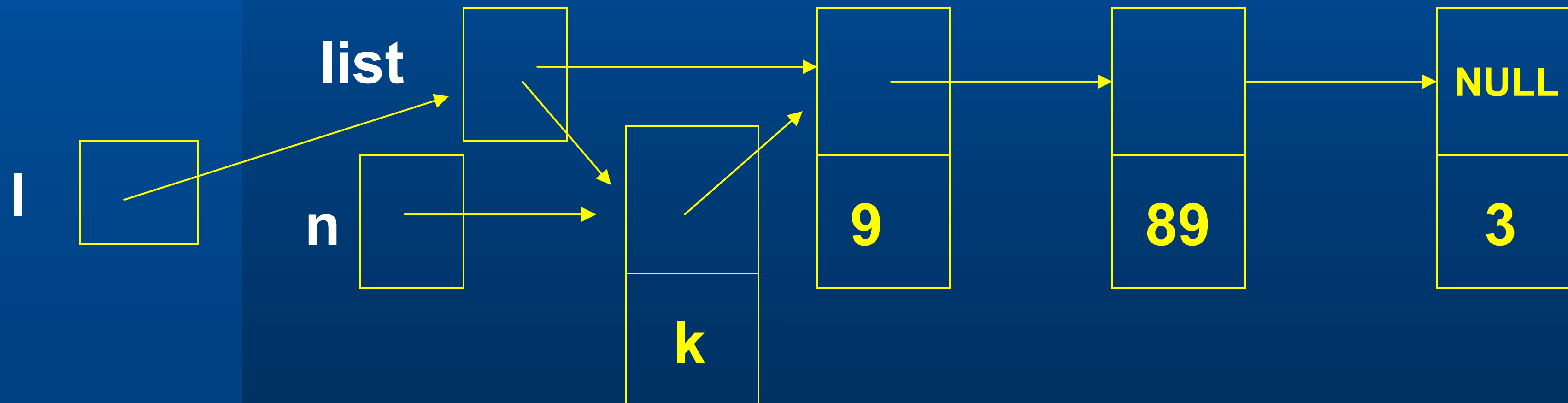
```
int main() {
   ListNodePtr list = 0;
   insertFront(&list,4);
   insertFront(&list,44);
   insertFront(&list,444);
   insertFront(&list,4444);
   insertLast(&list,333);
   printf("Found: %d\n",find(list,444));
   delete(&list,444);
   delete(&list,4444);
   printList(list);
}
```

Insert in Front: another way

```
ListNodePtr insertFront(ListNodePtr *l,int k) {
  ListNodePtr n = (ListNodePtr) malloc(sizeof(ListNode));
  n->key = k;
  n->next = l;
  *l = n;
}
```

what is my type?

struct lnode **l



Searching a list

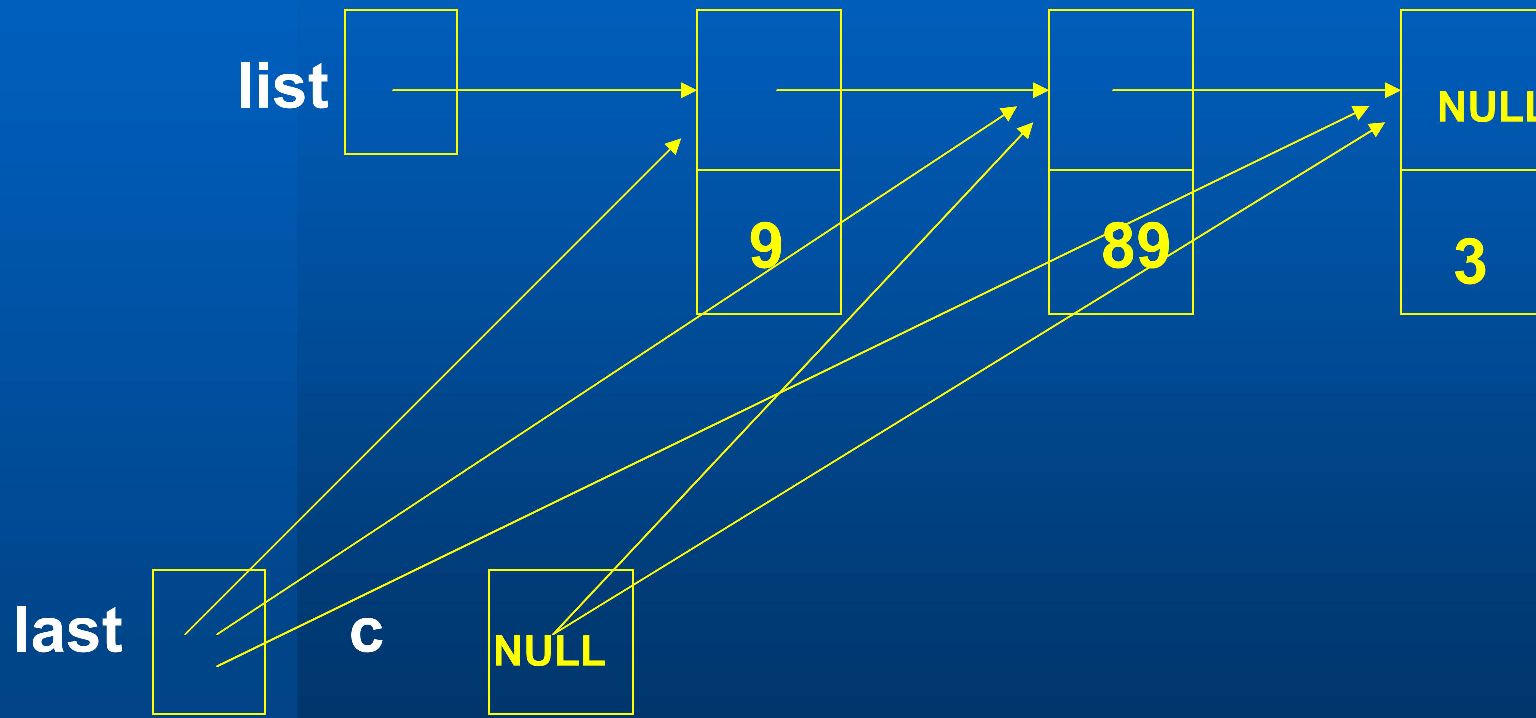
```
int find(ListNodePtr l,int k) {
  while (l != NULL) {
    if (l->key == k)
      return 1;
    l = l->next;
  }
  return 0;
}
```

```
int find(ListNodePtr l,int k) {
  while (l) {
    if (l->key == k)
      return 1;
    l = l->next;
  }
  return 0;
}
```

InsertLast: The Java Way

```
ListNodePtr insertLast(ListNodePtr l,int k) {
 ListNodePtr n =(ListNodePtr) malloc(sizeof(ListNode));
 n->key = k; n->next = NULL;
 if (l == NULL) return n;
 else {
    ListNodePtr last = l;
    ListNodePtr c = l->next;
    while (c) {
       last = c; c = c->next;
    }
    last->next = n;
    return l;
 }
}
```

Insert Last: The Java way



Insert Last: The Java Way

Basic Inconvenients

- special case for the empty list
- two pointers
  - previous
  - current element
- need to return the list

Insert Last: The Other Way

Key idea

- Keep a pointer to cells that may be updated

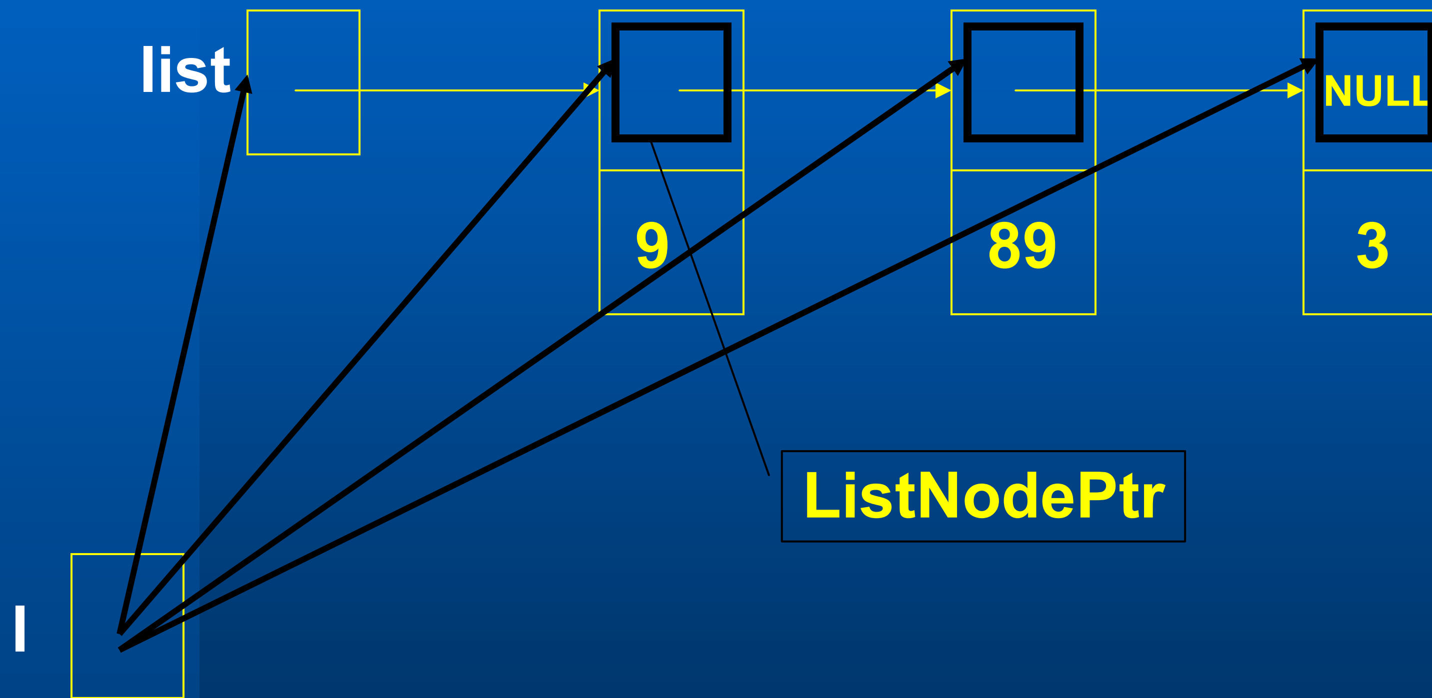
Advantages

- No special case for the empty list
- Only one pointer
- No need to return the list

InsertLast: The Other Way

```
void insertLast(ListNodePtr* l,int k) {
    ListNodePtr n = (ListNodePtr) malloc(sizeof(ListNode));
    n->key = k;
    n->next = NULL;
    while (*l)
        l = &((*l)->next);
    *l = n;
}
```

Insert Last: The other way



List: deletion

```
void delete(ListNodePtr* l,int k) {
   while (*l) {
      if ((*l)->key == k)
         (*l) = (*l)->next;
      else
         l = &((*l)->next);
   }
}
```

- Didn’t we forget something?

List: Delete and Free

```
void delete(ListNodePtr* l,int k) {
   while (*l) {
      if ((*l)->key == k) {
         ListNodePtr n = *l;
         (*l) = (*l)->next;
         free(n);
      } else
         l = &((*l)->next);
   }
}
```

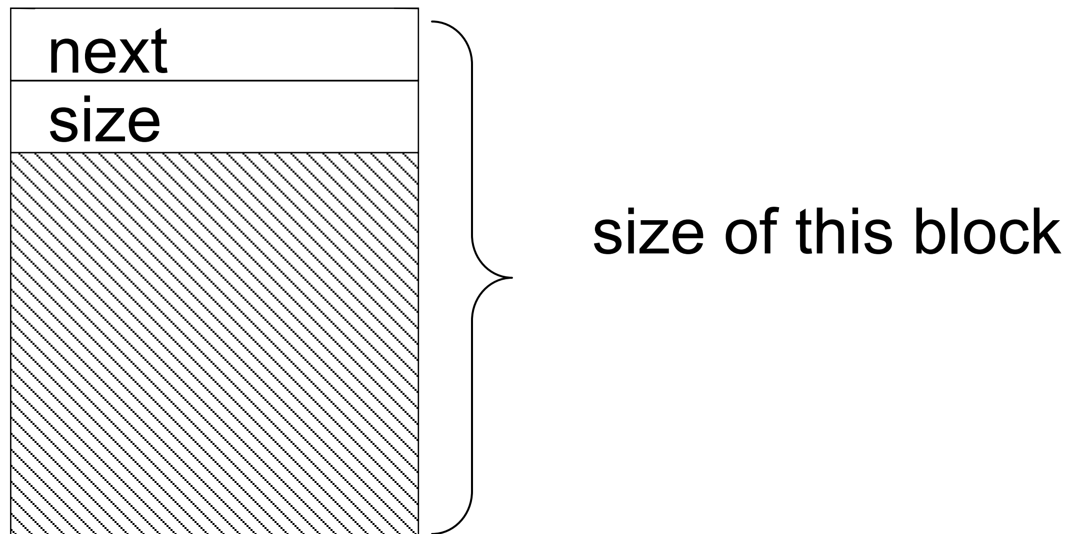
A C Program

```
int main() {
 printf("Welcome to CS-034\n");
 return 0;
}
```

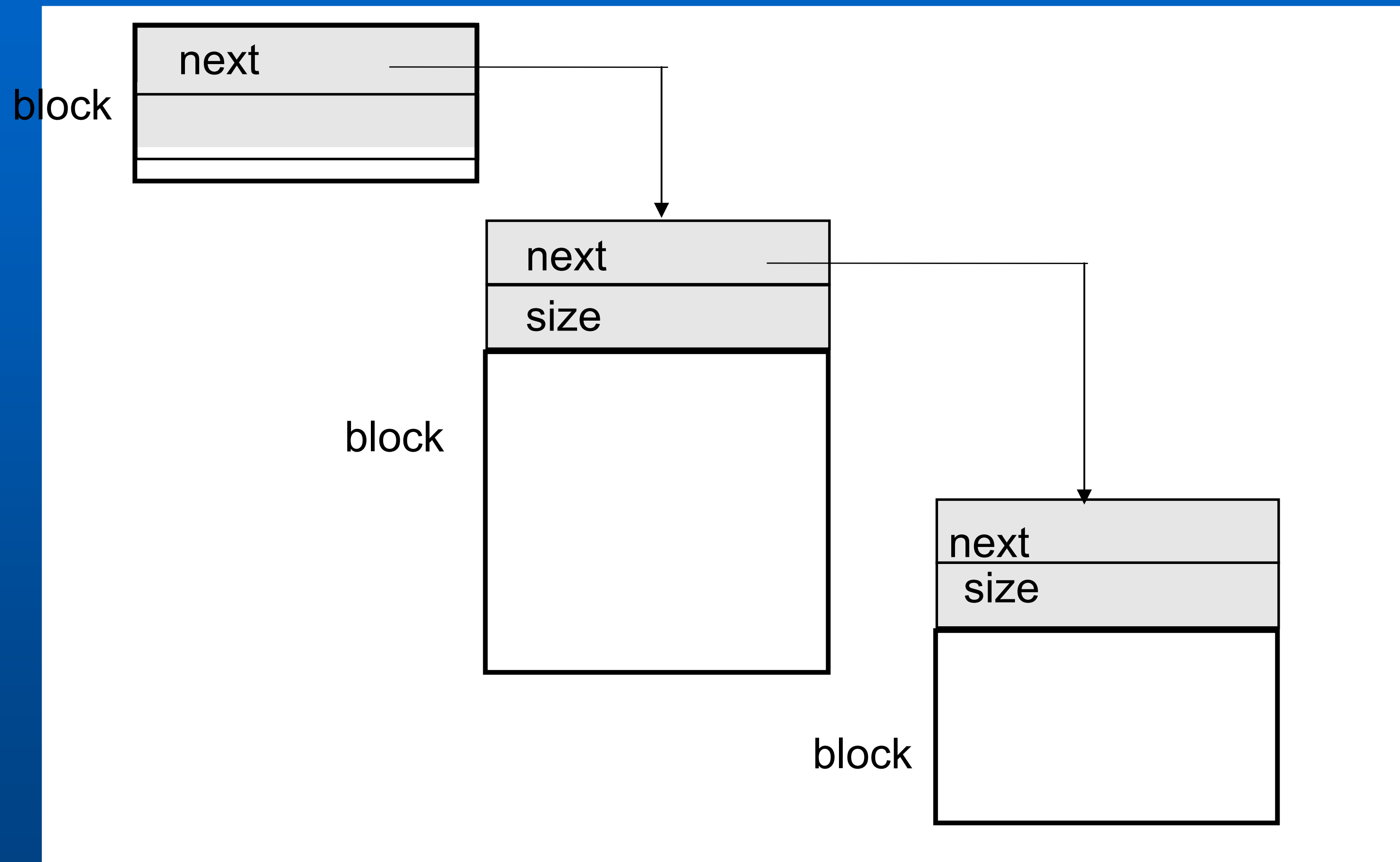
malloc and free implementation

- Free List
  - keep a list of memory spaces
- Start with one big block of memory
- When malloc is called
  - find a block large enough
  - divide it in two (in general)
  - return one of the two

malloc and free implementation

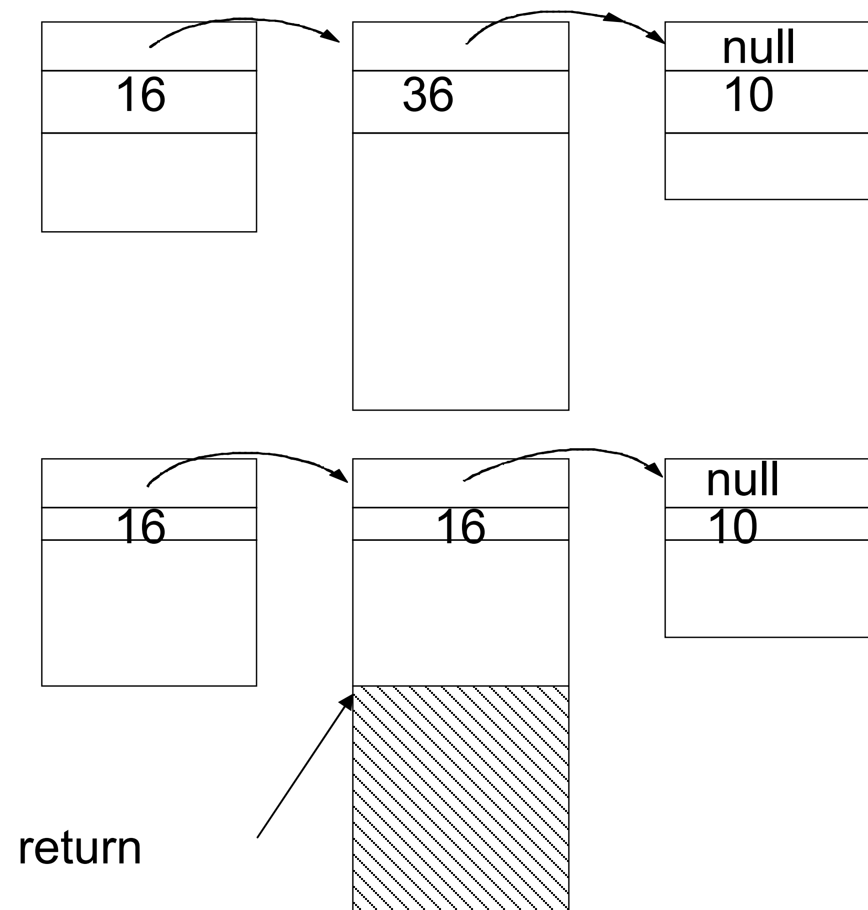


malloc and free implementation



malloc and free implementation

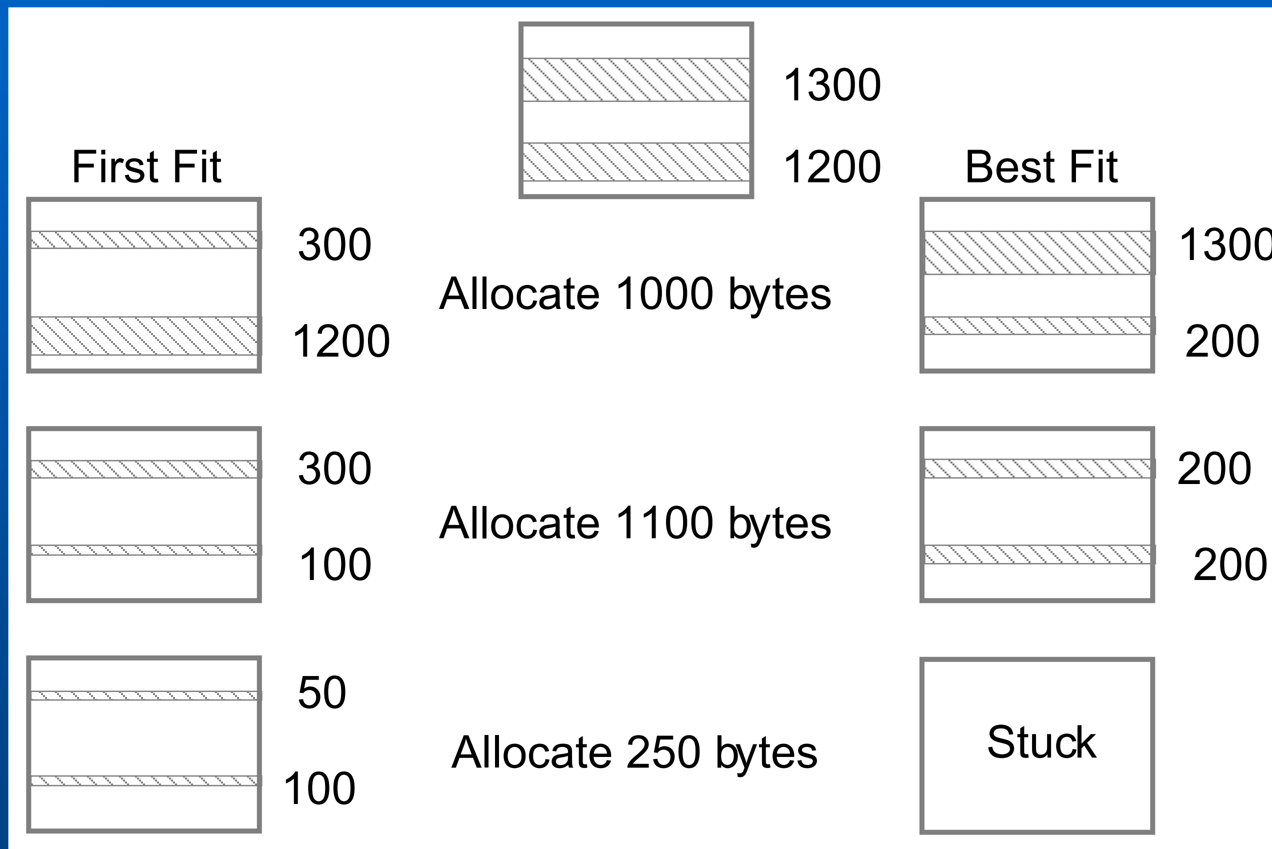
malloc(20)



malloc and free implementation

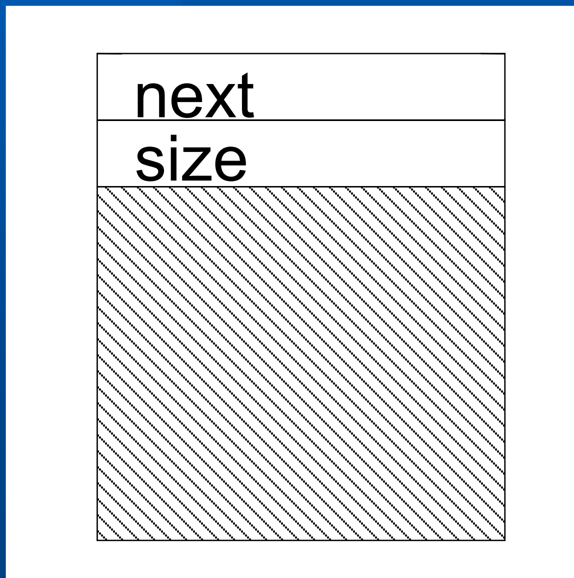
- How to find a block large enough?
  - First fit/Best fit
- First Fit
  - To allocate a block of size n,
    1. walk down the free list until you find an element of size >= n
    2. break off a chunk of size n and give back a pointer to it
    3. let the remainder, if any, in the list

malloc and free implementation



malloc and free implementation

- **In which language is `malloc` written?**
  - C of course
- **How do we declare a block?**




```
typedef struct block {
    struct block *next;
    long          size;
    long          data[0];
} Block, *BlockPtr;
BlockPtr freelist;
```

Open Arrays in C

- **clean dynamic data structures**

```
typedef struct vector {
   long          size;
   long          data[0];
} Vector, *VectorPtr;
int main() {
   …
   VectorPtr v =
     (VectorPtr) malloc(sizeof(Vector)
+size*sizeof(long));
  v->size = size; v->data[3] = 8;
```

Implementing Malloc and Free

- **Initializing the freelist**

```
void my_initMemory() {
    freelist = (BlockPtr) sbrk(BlockSize);
    freelist->size = Blocksize/sizeof(long);
    freelist->next = 0;
}
```

Implementing Malloc

```
void* my_malloc(size_t s) {
  int sl = 2 + (s-1)/sizeof(long) + 1;
  BlockPtr p = freelist; BlockPtr *ap = &freelist;
  while (p) {
    if (p->size == sl) {
      *ap = p->next;
      return &p->data[0];
    } else if (p->size > sl) {
      …
    }
    ap = &(p->next); p = p->next;
  }
  …
}
```

Implementing Malloc

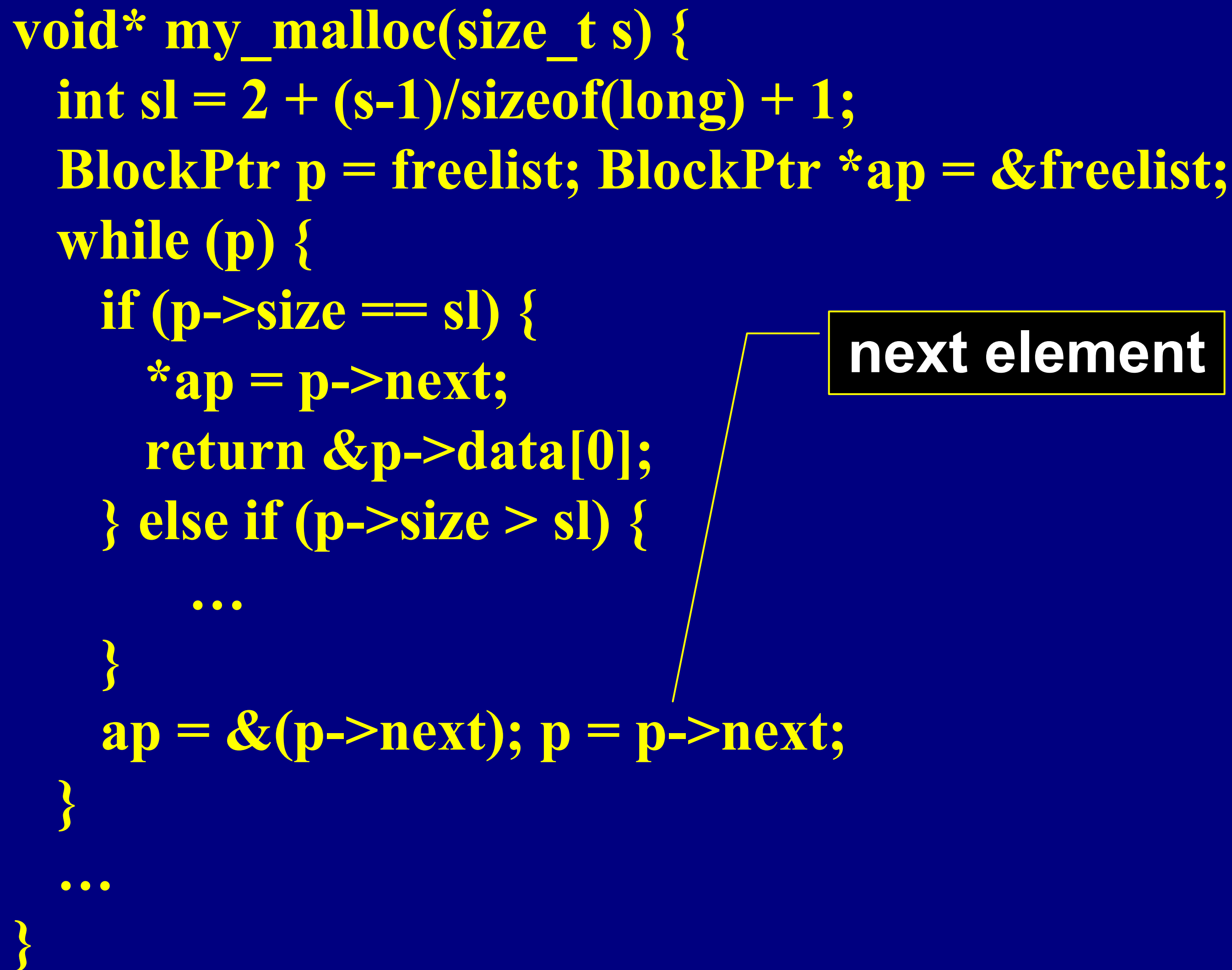
```
void* my_malloc(size_t s) {
  int sl = 2 + (s-1)/sizeof(long) + 1;
  BlockPtr p = freelist; BlockPtr *ap = &freelist;
  while (p) {
    if (p->size == sl) {
      *ap = p->next;
      return &p->data[0];
    } else if (p->size > sl) {
        …
    }
    ap = &(p->next); p = p->next;
  }
  …
}
```

return the space

Implementing Malloc

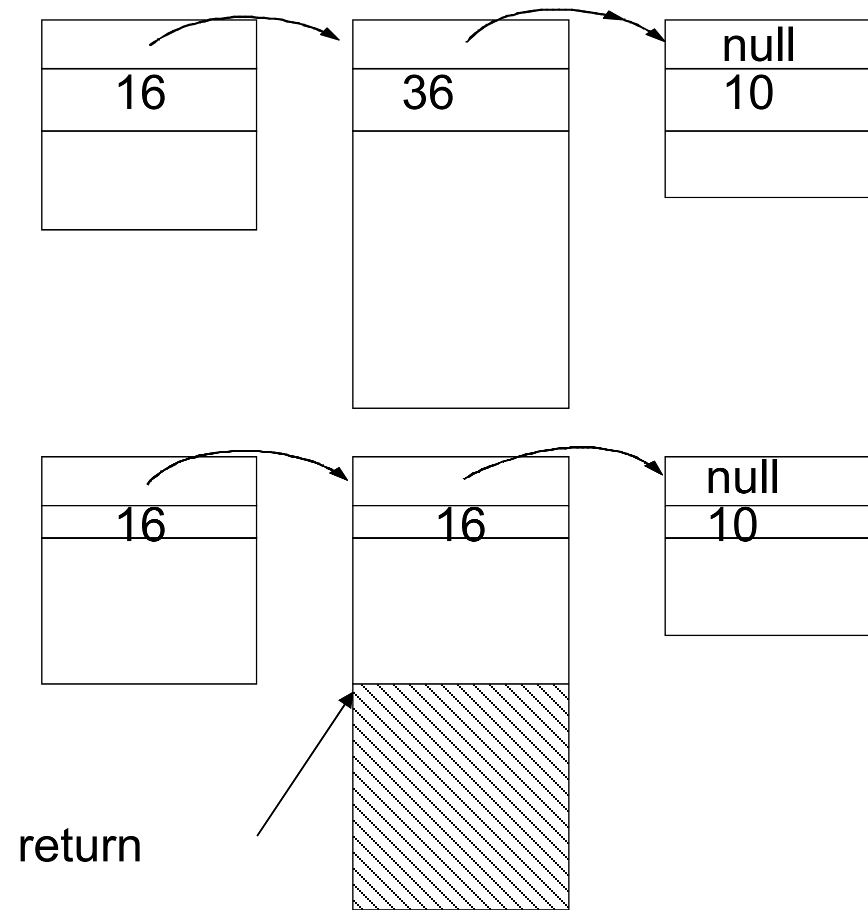
```
void* my_malloc(size_t s) {
  int sl = 2 + (s-1)/sizeof(long) + 1;
  BlockPtr p = freelist; BlockPtr *ap = &freelist;
  while (p) {
    if (p->size == sl) {
      *ap = p->next;
      return &p->data[0];
    } else if (p->size > sl) {
        …
    }
    ap = &(p->next); p = p->next;
  }
  …
}
```

Implementing Malloc



malloc and free implementation

malloc(20)



Implementing Malloc

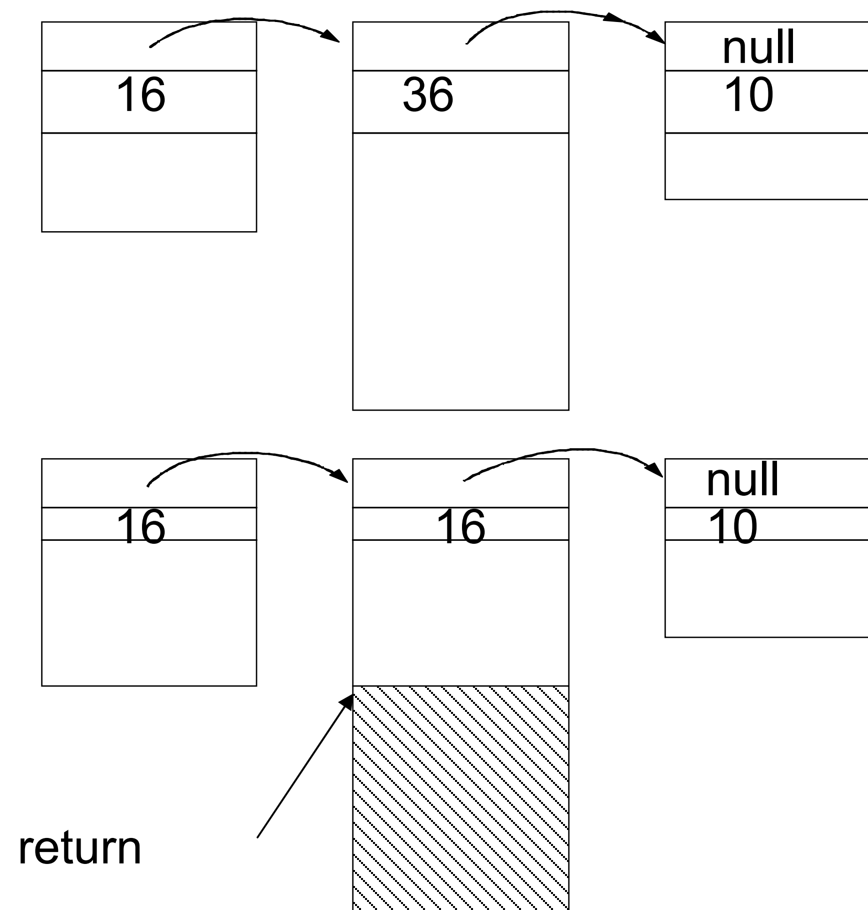
```
while (p) {
    if (p->size == sl) {
        *ap = p->next;
        return &p->data[0];
    } else if (p->size > sl) {
        p->size -= sl;
        BlockPtr mal = (BlockPtr) (&p->data[p->size-2]);
        mal->size = sl;
        return &(mal->data[0]);
    }
    ap = &(p->next);
    p = p->next;
}
```

decrease the size

new space

malloc and free implementation

malloc(20)



Implementing Free

```
void my_free(void *m) {
   BlockPtr n = (BlockPtr)((long*)m-2);
   n->next = freelist;
   freelist = n;
}
```

Questions…