Pointers, structs, and linked lists

CS210 - Fall 2023

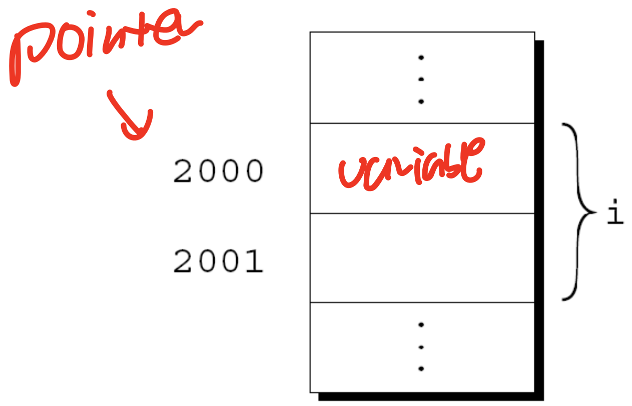
Vasiliki Kalavri
vkalavri@bu.edu

A pointer is a variable that contains the address of a variable

- Pointers are very common in C programs
  - they let us write concise and efficient code
  - they are sometimes the only means of expressing a computation

- You need to use extra care when using pointers
  - you might end up pointing to an unexpected location
  - you can easily write incomprehensible code

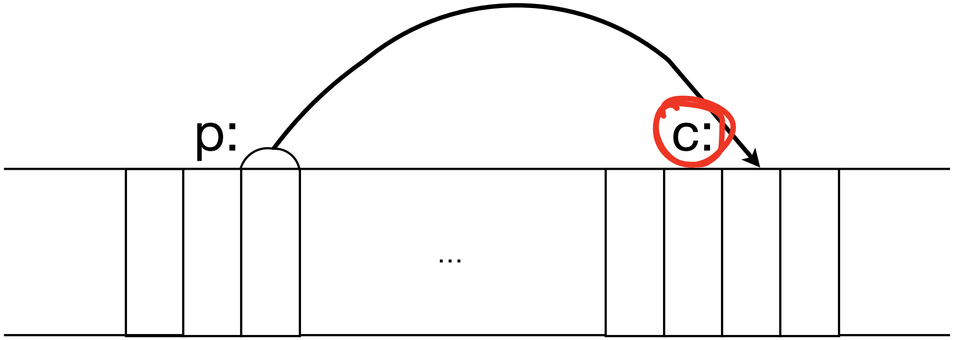
- Each variable in a program occupies one or more bytes of memory.
- The address of the first byte is said to be the address of the variable.
- In this figure, the address of the variable `i` is `2000`.



Visualization is extremely helpful

A pointer is a group of cells (usually 2 or 4) that store an address

$$p \;=\; \&c$$



Each cell has a unique ***address.***

The Address and Indirection Operators

- To find the address of a variable, we use the & (***address***) operator.
- To gain access to the object that a pointer points to, we use the * (***indirection***) operator.

```
int x=1, y=2, z[10];

int *ip; // ip is a pointer to an integer
ip = &x; // ip is pointing to x
y = *ip; // y is now 1
*ip = 0; // x is now 0
ip = &z[0]; // ip now points to z[0]
```

선언 ↓ 정의 가리킴 y=2

x=0

ip=&z 일걸.

Pointers point to specific types[1]

- C requires that every pointer variable point only to objects of a particular type (the ***referenced type***):

- There are no restrictions on what the referenced type may be.

정의
정의
정의

```
int *p;      /* points only to integers   */
double *q;   /* points only to doubles    */
char *r;     /* points only to characters */
```

[1]*except for (void *)*

```
int x=1, y=2, z[10];
```

정의

```
int *ip; // ip is a pointer to an integer
ip = &x; // ip is pointing to x
```

x 가리킴

```
*ip = *ip + 10;
```

x 주소값 +10

```
y = *ip + 1;
```

자동적으로 11

```
*ip +=1;
(*ip)++;
```

x랑 y값 같음

```
printf("x=%d, y=%d, *ip = %d, ip=%p\n",
             x, y, *ip, ip);

/* '++' and '*' are right-associative */
++*ip;
printf("*ip = %d, ip=%p\n", *ip, ip);

*ip++;
printf("*ip = %d, ip=%p\n", *ip, ip);
```

```c
int x=1, y=2, z[10];

int *ip; // ip is a pointer to an integer
ip = &x; // ip is pointing to x

*ip = *ip + 10;
y = *ip + 1;
*ip +=1;
(*ip)++;

printf("x=%d, y=%d, *ip = %d, ip=%p\n",
          x, y, *ip, ip);

/* '++' and '*' are right-associative */
++*ip;
printf("*ip = %d, ip=%p\n", *ip, ip);

*ip++;
printf("*ip = %d, ip=%p\n", *ip, ip);
```

x=11

y=12 x=12

x=13

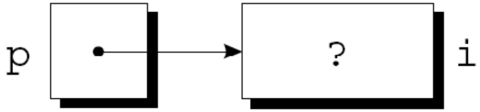
?

```
x=13, y=12, *ip = 13, ip=0x7ffee51636e8
*ip = 14, ip=0x7ffee51636e8
*ip = 0, ip=0x7ffee51636ec
```

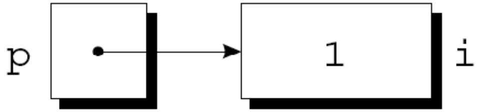
13 12 13

Pointer Assignment

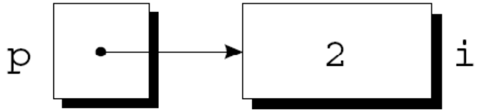
```
p = &i;
```




```
i = 1;
```

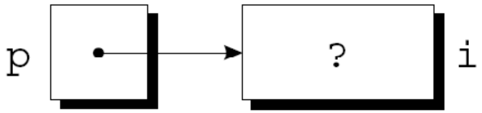



```
*p = 2;
```

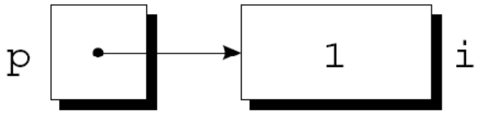


Pointer Assignment

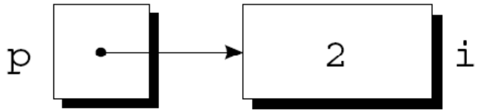
`p = &i;`



`i = 1;`

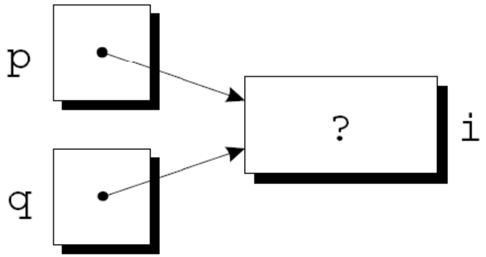


`*p = 2;`



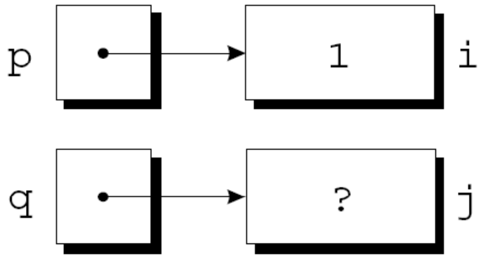
`q = p;`

`q` now points to the same place as `p`:

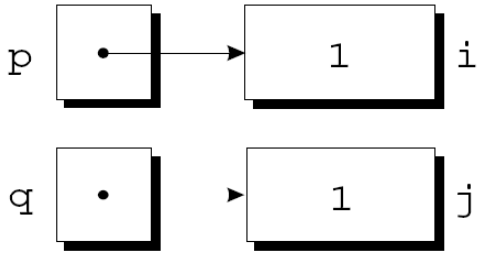


Pointer Assignment

```
p = &i;
q = &j;
i = 1;
```




```
*q = *p;
```



What will this program print?

```c
#include <stdio.h>

void swap(int x, int y) {
  int temp;

  temp = x;
  x = y;
  y = temp;
}

int main() {

  int a = 2, b = 3;
  swap(a, b);
  printf("%d, %d\n", a, b);

  return 0;
}
```

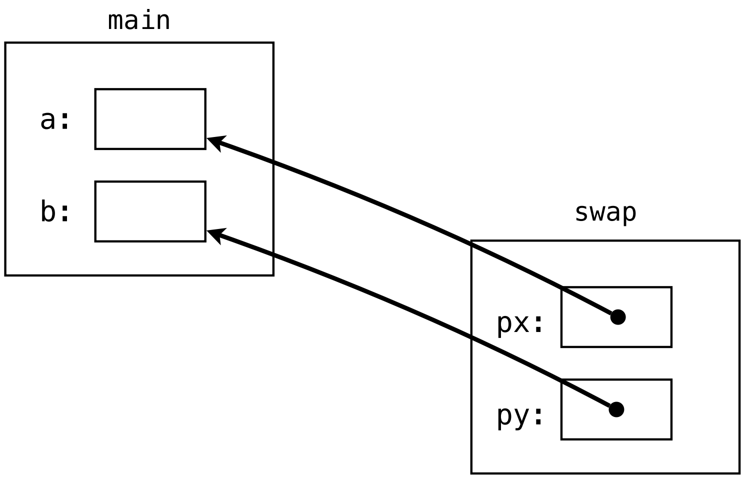
What will this program print?

`swap()` cannot modify the values of a and b because they are passed **by value**.

```c
#include <stdio.h>

void swap(int x, int y) {
  int temp;

  temp = x;
  x = y;
  y = temp;
}

int main() {

  int a = 2, b = 3;
  swap(a, b);
  printf("%d, %d\n", a, b);

  return 0;
}
```

Pointers as arguments

Pointer arguments let a function access and modify objects of its caller function




```c
void swap(int *px, int *py) {
    int temp;

    temp = *px;
    *px = *py;
    *py = temp;
}

int main() {

   int a = 2, b = 3;
   swap(&a, &b);
…
}
```

Pointers as arguments

Arguments in calls of `scanf` are pointers:

```
int i;
…
scanf("%d", &i);
```

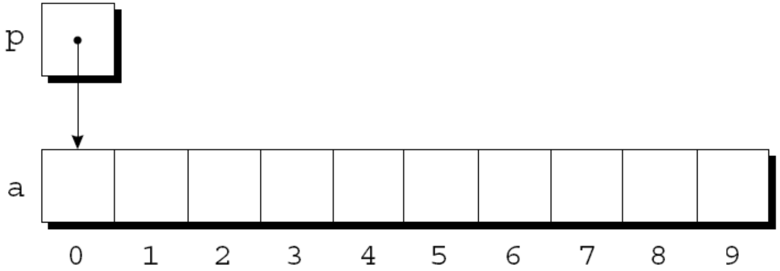
Without the `&`, `scanf` would be supplied with the *value* of `i`.

```c
/* Finds the largest and smallest elements in an array */

void max_min(int a[], int n, int *max, int *min)
{
  int i;

  *max = *min = a[0];
  for (i = 1; i < n; i++) {
    if (a[i] > *max)
      *max = a[i];
    else if (a[i] < *min)
      *min = a[i];
  }
}
```

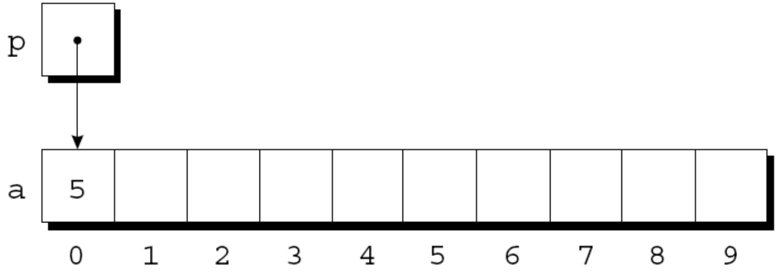
```c
int main(void)
{
  …
  max_min(b, N, &big, &small);
  …
}
```

Pointers to arrays

```
int a[10], *p;
p = &a[0];
```




```
*p = 5;
```



Pointers as Return Values

- If `a` is an array, then `&a[i]` is a pointer to element `i` of `a`.
- It's sometimes useful for a function to return a pointer to one of the elements in an array.
- A function that returns a pointer to the middle element of `a`, assuming that `a` has `n` elements:

```
int *find_middle(int a[], int n) {
  return &a[n/2];
}
```

Pointers as Return Values

- If `a` is an array, then `&a[i]` is a pointer to element `i` of `a`.
- It's sometimes useful for a function to return a pointer to one of the elements in an array.
- A function that returns a pointer to the middle element of `a`, assuming that `a` has `n` elements:

Functions can return pointers

```
int *find_middle(int a[], int n) {
  return &a[n/2];
}
```

Pointers as Return Values

What is wrong with this code?

```
int *f(void)
{
  int i;
  …
  return &i;
}
```

Pointers as Return Values

What is wrong with this code?

```c
int *f(void)
{
  int i;
  …
  return &i;
}
```

Never return a pointer to an *automatic* local variable!

The variable `i` won't exist after `f` returns.

Structs

A struct is a collection of one or more variables of possibly different types

- We use structs to organize complex data
  - they allow us to manage related variables as one unit

```
struct flight {
 int time;        // departure time of the flight
 int available;   // number of seats currently available on the flight
 int capacity;    // maximum seat capacity of the flight
};
```

Declaring Structure Variables

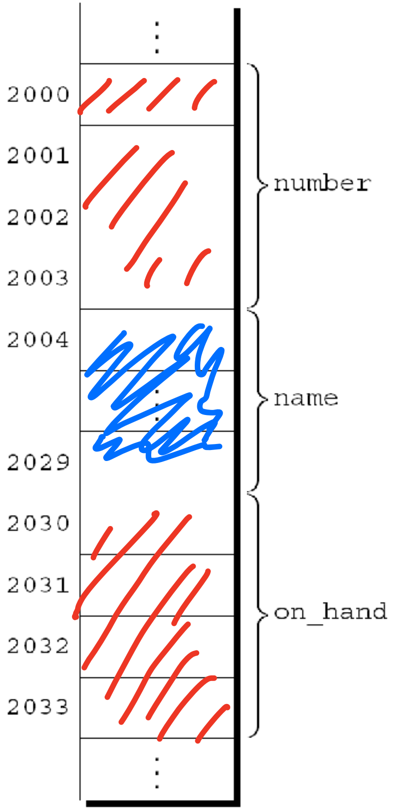
- The members of a structure are stored in memory in the order in which they're declared.
- Appearance of `part1`
- Assumptions:
  - `part1` is located at address 2000.
  - Integers occupy four bytes.
  - `NAME_LEN` has the value 25.
  - There are no gaps between the members.

```
struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1, part2;
```

24

~~25~~

4



Working with structs

```
int main() {

  struct flight f1;

  f1.time = 180;
  f1.available = 120;
  f1.capacity = 220;

  printf("f1 time: %d\n", f1.time);
  …
}
```

Working with structs

```
int main() {
  struct flight f1;

  f1.time = 180;
  f1.available = 120;
  f1.capacity = 220;

  printf("f1 time: %d\n", f1.time);
  …
}
```

Declaring struct variables

Working with structs

```c
int main() {
  struct flight f1;

  f1.time = 180;
  f1.available = 120;
  f1.capacity = 220;

  printf("f1 time: %d\n", f1.time);
  …
}
```

Declaring struct variables

Accessing and modifying struct members

Pointers to structs

```
int main() {

   struct flight f1;

   f1.time = 180;
   f1.available = 120;
   f1.capacity = 220;

   struct flight *pf;
   pf = &f1;

   (*pf).available--;
   printf("f1 available: %d\n", f1.available);
   …
}
```

Pointers to structs

```
int main() {

    struct flight f1;

    f1.time = 180;
    f1.available = 120;
    f1.capacity = 220;

    struct flight *pf;
    pf = &f1;

    (*pf).available--;
    printf("f1 available: %d\n", f1.available);
    …
}
```

pf is a pointer to a struct of type `struct flight`

Pointers to structs

```c
int main() {

   struct flight f1;

   f1.time = 180;
   f1.available = 120;
   f1.capacity = 220;

   struct flight *pf;
   pf = &f1;

   (*pf).available--;
   printf("f1 available: %d\n", f1.available);
   …
}
```

pf is a pointer to a struct of type `struct flight`

Accessing a member through the pointer

The -> Operator

- Accessing a member of a structure using a pointer is so common that C provides a special operator for this purpose.
- This operator, known as ***right arrow selection,*** is a minus sign followed by >.

```
struct flight *pf;

pf = &f1;
pf->available--;

printf("f1 available: %d\n", pf->available);
```

619

Arrays of structs

Arrays may have structures as their elements, and structures may contain arrays and structures as members.

```
struct flight flights[MAX_FLIGHTS_PER_CITY];
```

Arrays of structs

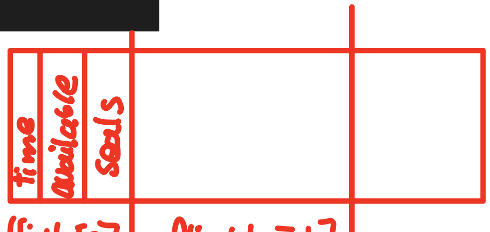
Arrays may have structures as their elements, and structures may contain arrays and structures as members.

```
struct flight flights[MAX_FLIGHTS_PER_CITY];
```

Accessing a member within a structure requires a combination of subscripting and member selection

```
flights[0].available = 0;
```

member



flight [0] flight [1]

Structs can have other structs and pointers to structs as members

```
struct flight_schedule {
  city_t destination;                         // destination city name
  struct flight flights[MAX_FLIGHTS];         // array of flights to the city
  struct flight_schedule *next;               // link list next pointer
  struct flight_schedule *prev;               // link list prev pointer
};
```

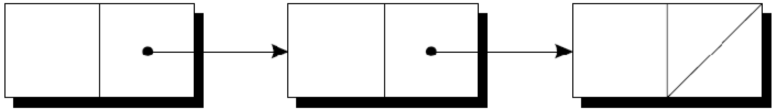
Example: reset a flight schedule

```
void flight_schedule_reset(struct flight_schedule *fs) {

  fs->destination[0] = 0;

  for (int i=0; i<MAX_FLIGHTS_PER_CITY; i++) {
    fs->flights[i].time = TIME_NULL;
    fs->flights[i].available = 0;
    fs->flights[i].capacity = 0;
  }

  fs->next = NULL;
  fs->prev = NULL;
}
```

Linked lists

Linked Lists

- A ***linked list*** consists of a chain of structures (called ***nodes***), with each node containing a pointer to the next node in the chain:
- The last node in the list contains a null pointer.



Declaring a Node Type

- To set up a linked list, we'll need a structure that represents a single node.
- A node structure will contain data (an integer in this example) plus a pointer to the next node in the list:

```
struct node {
  int value;          /* data stored in the node  */
  struct node *next;  /* pointer to the next node */
};
```

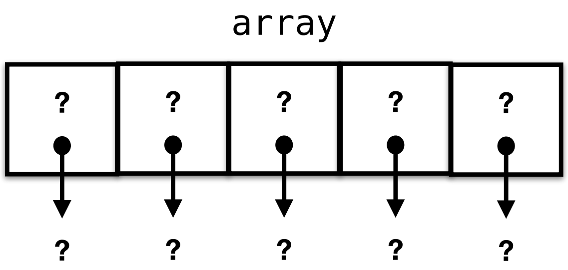
Initializing a linked list

```c
// Initializes a linked list
void list_initialize(struct node array[], int n) {

  // takes care of empty array case
  if (n==0) return;

  // connect the list
  for (int i=0; i<n-1; i++) {
    array[i].value = 0;
    array[i].next = &array[i+1];
  }
  array[n-1].next = NULL;
}
```

```c
// Initializes a linked list
void list_initialize(struct node array[], int n) {

  // takes care of empty array case
  if (n==0) return;

  // connect the list
  for (int i=0; i<n-1; i++) {
    array[i].value = 0;
    array[i].next = &array[i+1];
  }
  array[n-1].value = 0;
  array[n-1].next = NULL;
}
```

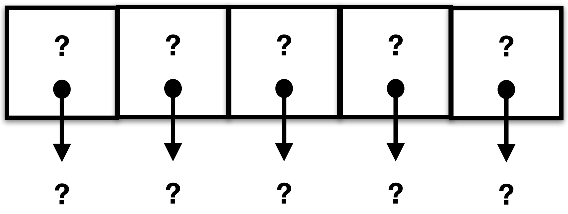
```c
// Initializes a linked list
void list_initialize(struct node array[], int n) {

  // takes care of empty array case
  if (n==0) return;

  // connect the list
  for (int i=0; i<n-1; i++) {
    array[i].value = 0;
    array[i].next = &array[i+1];
  }
  array[n-1].value = 0;
  array[n-1].next = NULL;
}
```

last one

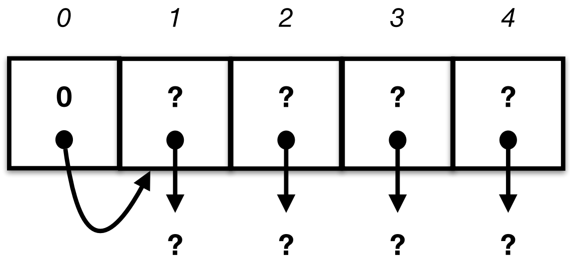


```c
// Initializes a linked list
void list_initialize(struct node array[], int n) {

  // takes care of empty array case
  if (n==0) return;

  // connect the list
  for (int i=0; i<n-1; i++) {
    array[i].value = 0;
    array[i].next = &array[i+1];
  }
  array[n-1].value = 0;
  array[n-1].next = NULL;
}
```

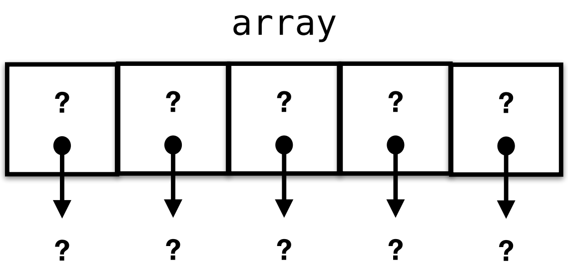
array

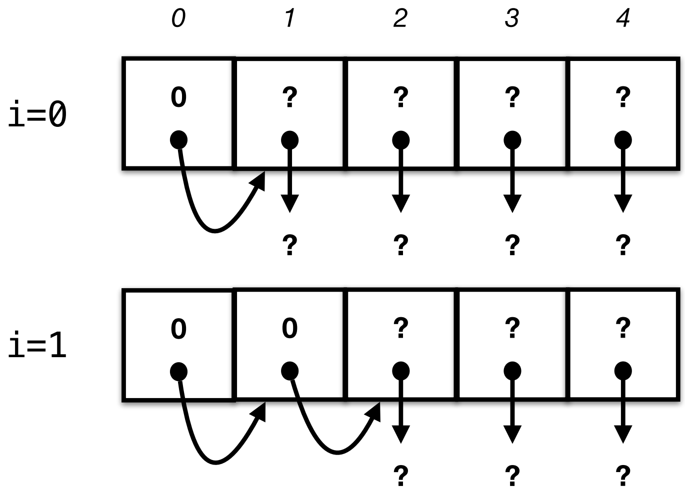


i=0

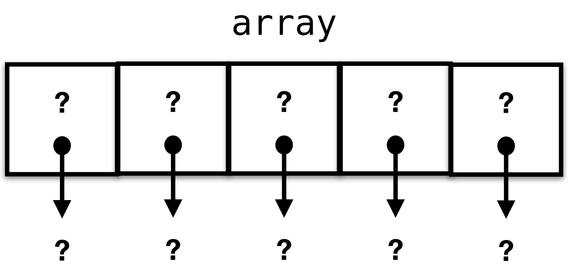


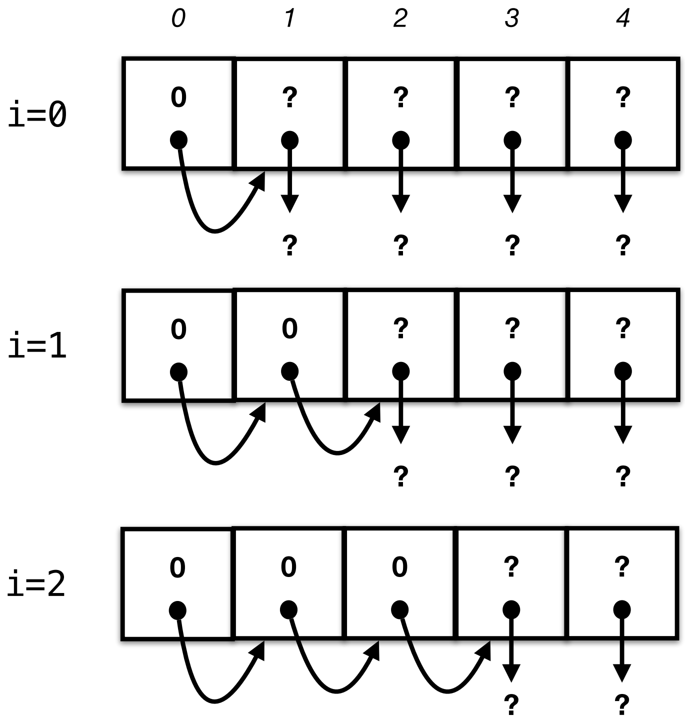
```c
// Initializes a linked list
void list_initialize(struct node array[], int n) {

  // takes care of empty array case
  if (n==0) return;

  // connect the list
  for (int i=0; i<n-1; i++) {
    array[i].value = 0;
    array[i].next = &array[i+1];
  }
  array[n-1].value = 0;
  array[n-1].next = NULL;
}
```



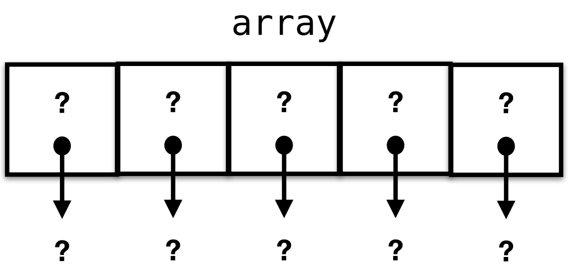


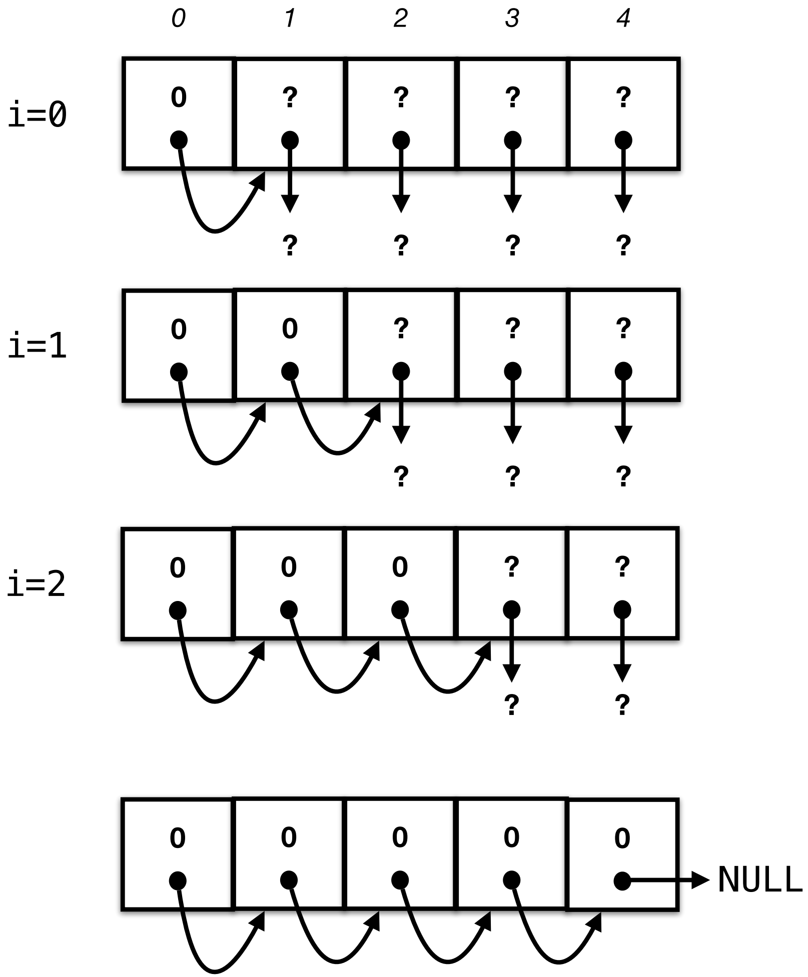
```c
// Initializes a linked list
void list_initialize(struct node array[], int n) {

  // takes care of empty array case
  if (n==0) return;

  // connect the list
  for (int i=0; i<n-1; i++) {
    array[i].value = 0;
    array[i].next = &array[i+1];
  }
  array[n-1].value = 0;
  array[n-1].next = NULL;
}
```





```c
// Initializes a linked list
void list_initialize(struct node array[], int n) {

  // takes care of empty array case
  if (n==0) return;

  // connect the list
  for (int i=0; i<n-1; i++) {
    array[i].value = 0;
    array[i].next = &array[i+1];
  }
  array[n-1].value = 0;
  array[n-1].next = NULL;
}
```





Initializing a linked list

```c
#include <stdio.h>

int main() {

  struct node list[5];

  list_initialize(list, 5);
  list[0].value = 1;
  list[0].next->value = 42;

  printf("fist value: %d, second value: %d, last value: %d\n",
         list[0].value, list[1].value, list[4].value);

}
```

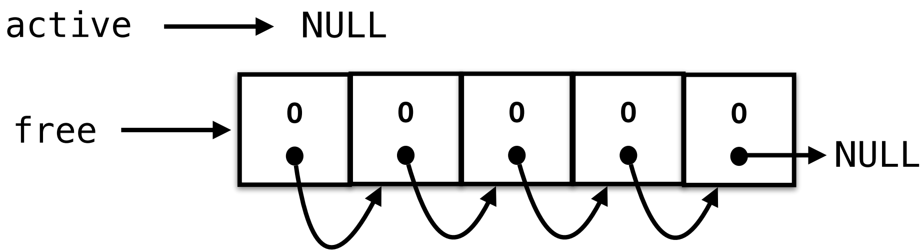
1 42 0

Searching a Linked List

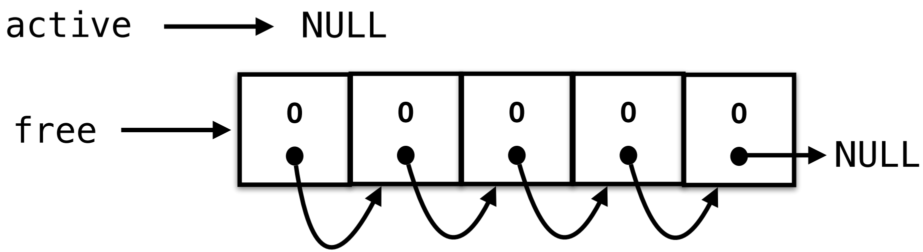
- A loop that visits the nodes in a linked list, using a pointer variable `p` to keep track of the "current" node:

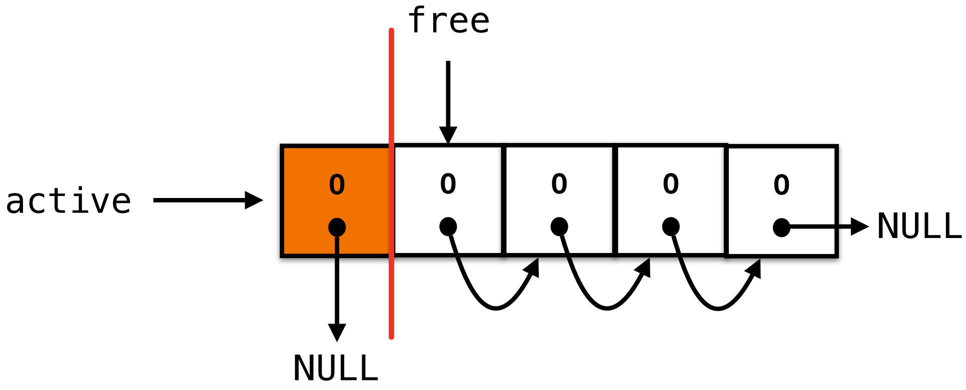
```
for (p = first; p != NULL; p = p->next)
   …
```

```
// searches the list for the number n
struct node *search_list(struct node *list, int n)  {
  while (list != NULL && list->value != n)
    list = list->next;
  return list;
}
```

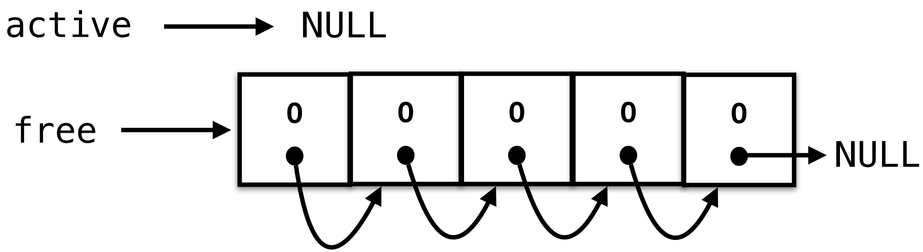


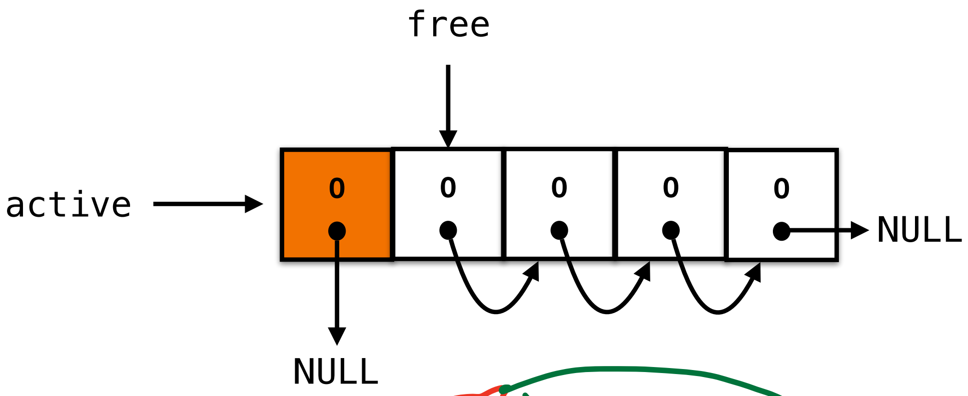
You can use pointers to move nodes from one list to another

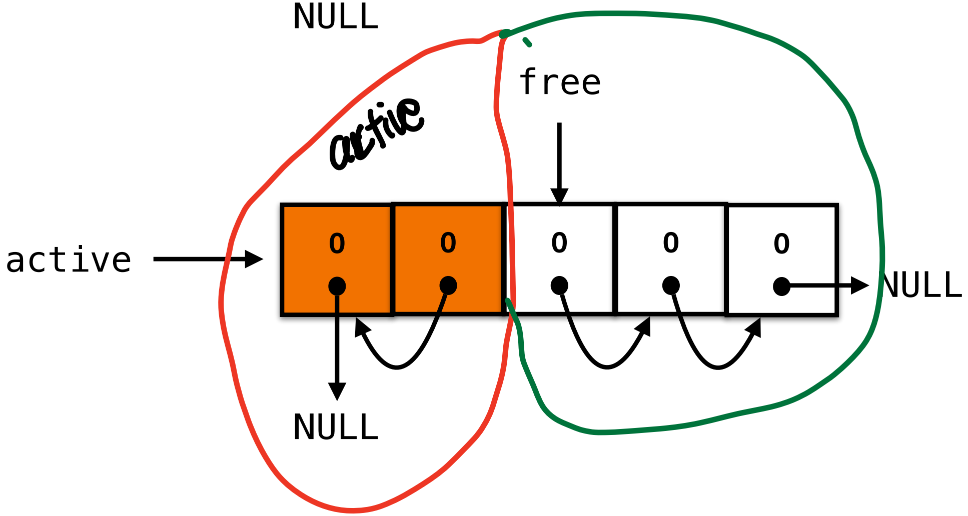




You can use pointers to move nodes from one list to another





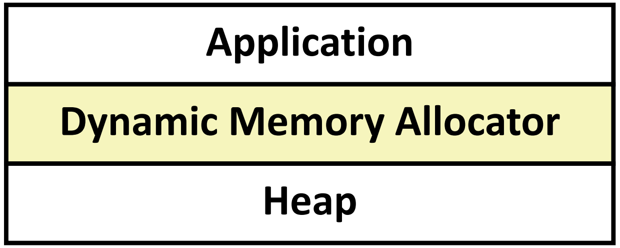


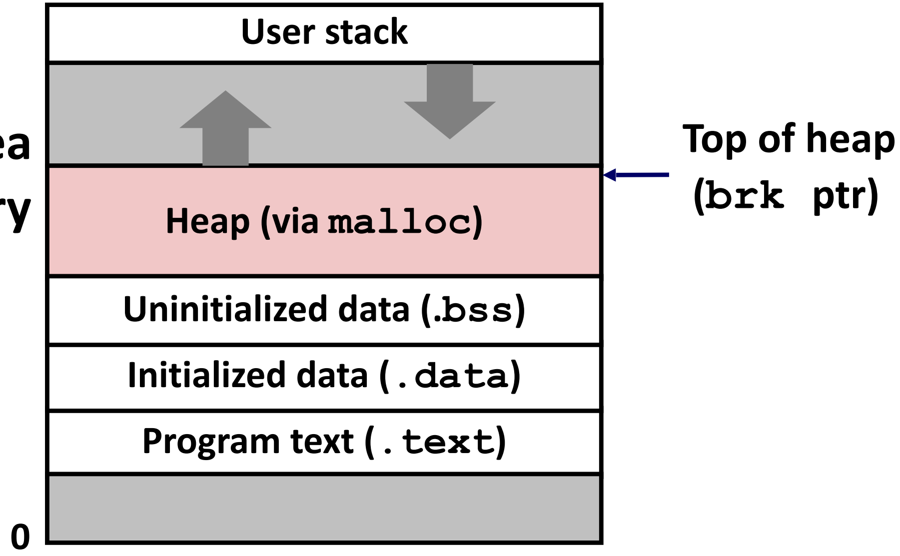
You can use pointers to move nodes from one list to another

Dynamic memory allocation

Dynamic Memory Allocation

- **Programmers use *dynamic memory allocators* (such as `malloc`) to acquire VM at run time.**
  - For data structures whose size is only known at runtime.
- **Dynamic memory allocators manage an area of process virtual memory known as the *heap*.**





Dynamic Memory Allocation

- **Allocator maintains heap as collection of variable sized *blocks*, which are either *allocated* or *free***
- **Types of allocators**
  - ***Explicit allocator***: application allocates and frees space
    - E.g., `malloc` and `free` in C
  - ***Implicit allocator:*** application allocates, but does not free space
    - E.g. garbage collection in Java, ML, and Lisp
- **Will discuss simple explicit memory allocation today**

The `malloc` Package

```
#include <stdlib.h>
```

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

- Successful:
  - Returns a pointer to a memory block of at least **`size`** bytes aligned to an 8-byte (x86) or 16-byte (x86-64) boundary
  - If **`size == 0`**, returns NULL
- Unsuccessful: returns NULL (0) and sets **`errno`**

The `malloc` Package

Need to check successful

```
#include <stdlib.h>
```

`void *malloc(size_t size)`

- Successful:
  - Returns a pointer to a memory block of at least **`size`** bytes aligned to an 8-byte (x86) or 16-byte (x86-64) boundary
  - If **`size == 0`**, returns NULL
- Unsuccessful: returns NULL (0) and sets **`errno`**

`void free(void *p)`

- Returns the block pointed at by **`p`** to pool of available memory
- **`p`** must come from a previous call to **`malloc`** or **`realloc`**

The `malloc` Package

`#include <stdlib.h>`

`void *malloc(size_t size)`

- Successful:
  - Returns a pointer to a memory block of at least **`size`** bytes aligned to an 8-byte (x86) or 16-byte (x86-64) boundary
  - If **`size == 0`**, returns NULL
- Unsuccessful: returns NULL (0) and sets **`errno`**

`void free(void *p)`

- Returns the block pointed at by **`p`** to pool of available memory
- **`p`** must come from a previous call to **`malloc`** or **`realloc`**

Other functions

- **`calloc`**: Version of **`malloc`** that initializes allocated block to zero.
- **`realloc`**: Changes the size of a previously allocated block.
- **`sbrk`**: Used internally by allocators to grow or shrink the heap

malloc Example

```c
#include <stdio.h>
#include <stdlib.h>

void foo(int n) {
    int i, *p;

    /* Allocate a block of n ints */
    p = (int *) malloc(n * sizeof(int));
    if (p == NULL) {
        perror("malloc");
        exit(0);
    }

    /* Initialize allocated block */
    for (i=0; i<n; i++)
        p[i] = i;


    /* Return allocated block to the heap */
    free(p);
}
```

different computer int size might be different

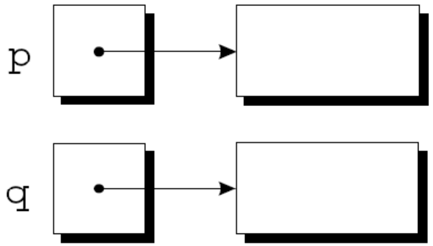
int convert OM

Deallocating Storage

- **Example:**

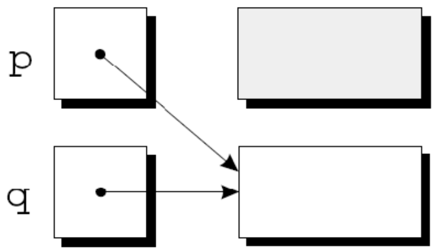
```
p = malloc(…);
q = malloc(…);
p = q;
```

- **A snapshot after the first two statements have been executed:**



Deallocating Storage

- **After `q` is assigned to `p`, both variables now point to the second memory block:**



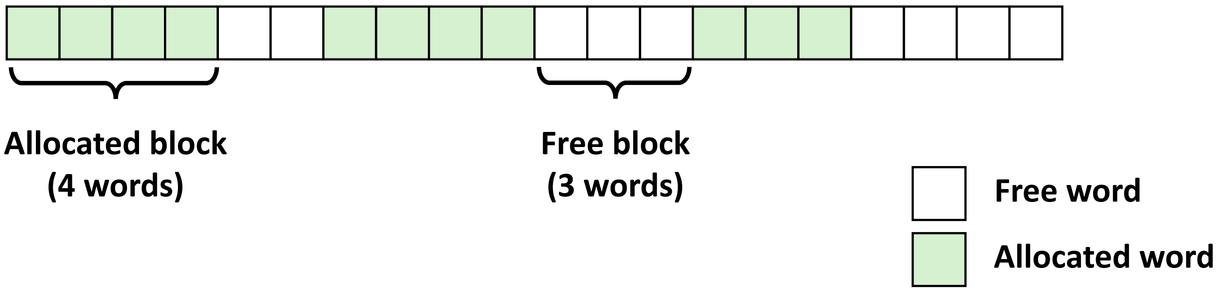
- **There are no pointers to the first block, so we'll never be able to use it again.**

Deallocating Storage

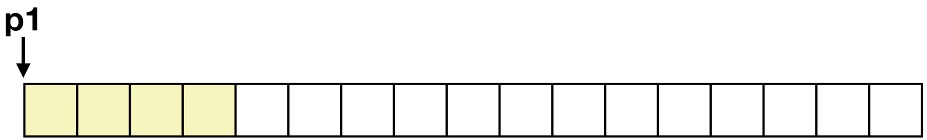
- A block of memory that's no longer accessible to a program is said to be ***garbage*.**
- A program that leaves garbage behind has a ***memory leak*.**
- Some languages provide a ***garbage collector*** that automatically locates and recycles garbage, but C doesn't.
- Instead, each C program is responsible for recycling its own garbage by calling the `free` function to release unneeded memory.

Assumptions Made in This Lecture

- **Memory is word addressed.**
- **Words are int-sized.**



Allocation Example (double-word aligned)



`p1 = malloc(4)`

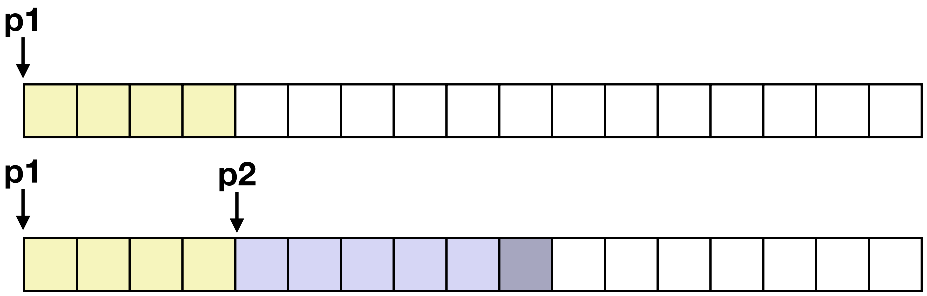
`p2 = malloc(5)`

`p3 = malloc(6)`

`free(p2)`

`p4 = malloc(2)`

Allocation Example (double-word aligned)




```
p1 = malloc(4)
```

```
p2 = malloc(5)
```

```
p3 = malloc(6)
```

```
free(p2)
```

```
p4 = malloc(2)
```

Allocation Example (double-word aligned)

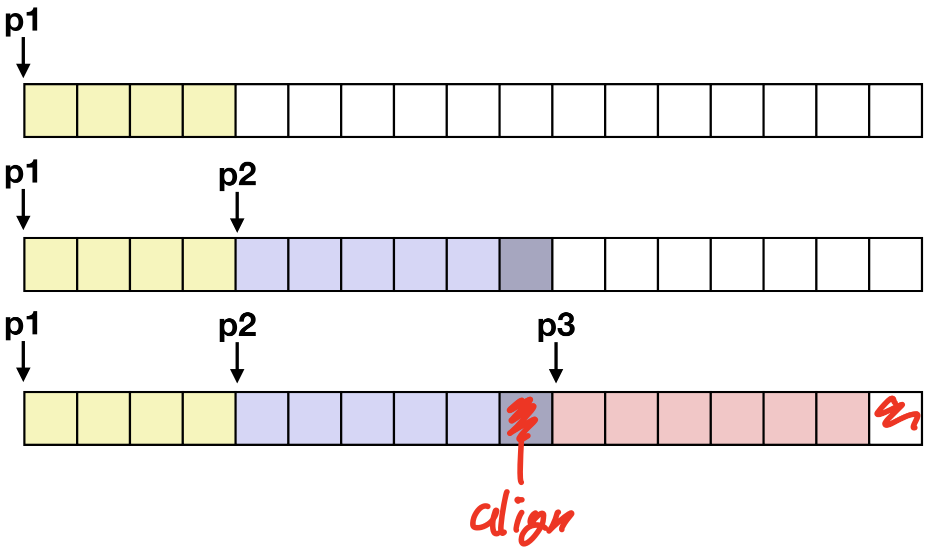
`p1 = malloc(4)`

`p2 = malloc(5)`

`p3 = malloc(6)`

`free(p2)`

`p4 = malloc(2)`



Allocation Example (double-word aligned)

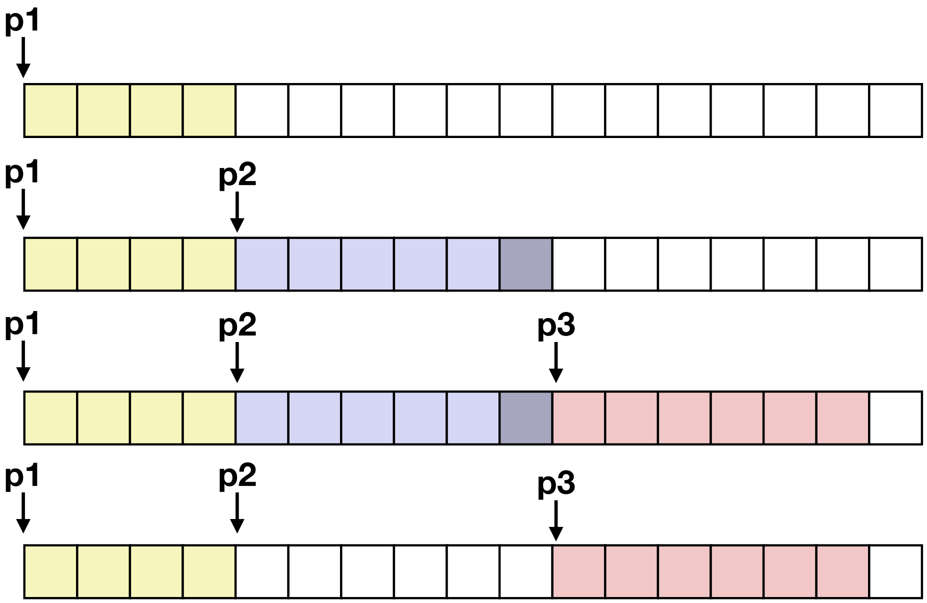
```
p1 = malloc(4)
```

```
p2 = malloc(5)
```

```
p3 = malloc(6)
```

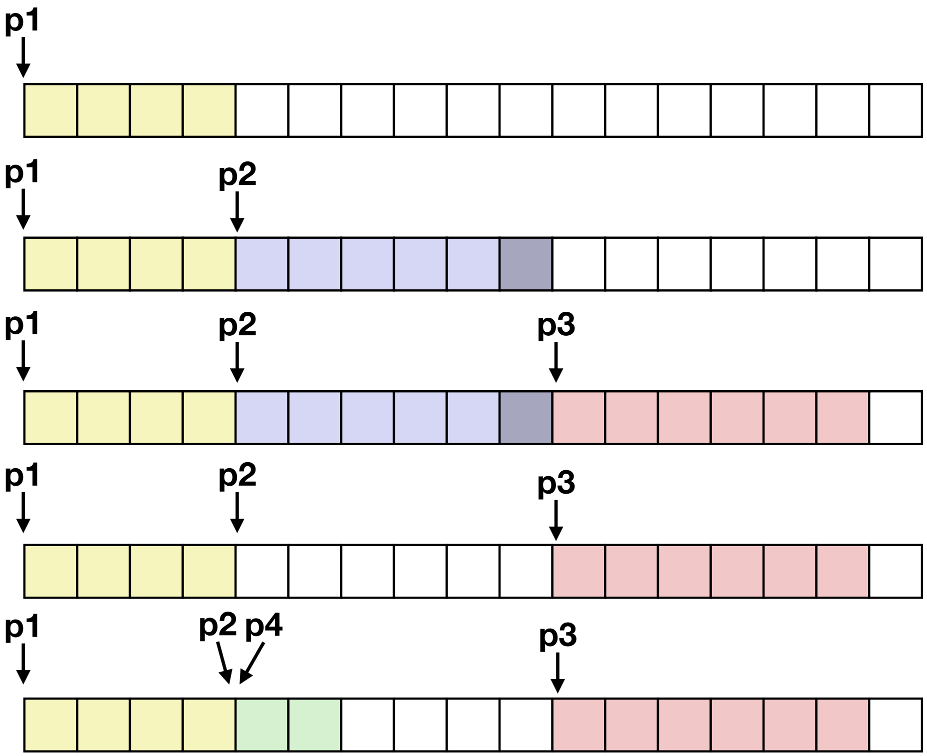
```
free(p2)
```

```
p4 = malloc(2)
```



Allocation Example (double-word aligned)

```
p1 = malloc(4)
p2 = malloc(5)
p3 = malloc(6)
free(p2)
p4 = malloc(2)
```



The “Dangling Pointer” Problem

- **Using `free` leads to a new problem: *dangling pointers*.**
- **`free(p)` deallocates the memory block that `p` points to, but doesn’t change `p` itself.**
- **If we forget that `p` no longer points to a valid memory block, chaos may ensue:**

```
char *p = malloc(4);
…
free(p);
…
strcpy(p, "abc");    /*** WRONG ***/
```

- **Modifying the memory that `p` points to is a serious error.**