Advanced Uses of Pointers

Index

01. Dynamical Storage Allocation

- Introduction
- Memory Allocation
- Deallocating Storages

02. Linked List

- Introduction
- Operator
- Instructions
- Pointer to

01

Dynamical Storage Allocation

Dynamic Storage Allocation

Introduction

C's data structures, including arrays, are normally fixed in size

Fixed-size data structures can be a problem, since we're forced to choose their sizes when writing a program

Fortunately, C supports dynamic storage allocation: the ability to allocate storage during program execution

Using dynamic storage allocation, we can design data structures that grow (and shrink) as needed

Dynamic storage allocation is used most often for strings, arrays, and structures

Dynamic storage allocation is done by calling a memory allocation function

Dynamic Storage Allocation

Memory Allocation Function

The <stdlib.h> header declares three memory allocation functions

- malloc
  - Allocates a block of memory but doesn’t initialize it
- calloc
  - Allocates a block of memory and clears it
- realloc
  - Resizes a previously allocated block of memory

These functions return a value of type void * (a "generic" pointer)

```
void *a = NULL;
int *b = NULL;
a = b;
```

```
void *a;
int *b = NULL;
b = (int *)a;
```

Dynamic Storage Allocation

Memory Allocation Function

Null pointers

- If a memory allocation function can't locate a memory block of the requested size, it returns a null pointer
- A null pointer is a special value that can be distinguished from all valid pointers
- After we've stored the function's return value in a pointer variable, we must test to see if it's a null pointer
- An example of testing malloc's return value

```
p = malloc(10000);
if (p == NULL) {
    /* allocation failed; take appropriate action */
}
```

macro

```
if ((p = malloc(10000)) == NULL) {
    /* allocation failed; take appropriate action */
}
```

Dynamic Storage Allocation

Memory Allocation Function

Pointers test true or false in the same way as numbers

All non-null pointers test true; only null pointers are false

Instead of writing

```
if (p == NULL) …          if (p != NULL) …
```

we could write

```
if (!p) …                 if (p) …
```

Dynamic Storage Allocation

Memory Allocation Function

Dynamically Allocated Strings

- Dynamic storage allocation is often useful for working with strings
- Strings are stored in character arrays, and it can be hard to anticipate how long these arrays need to be
- By allocating strings dynamically, we can postpone the decision until the program is running

Dynamic Storage Allocation

Memory Allocation Function

Malloc

- Prototype for the malloc function

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

- malloc allocates a block of size bytes and returns a pointer to it
- size_t is an unsigned integer type defined in the library
- A call of malloc that allocates memory for a string of n characters

```
char *p; p = malloc(n + 1);
```

- Each character requires one byte of memory; adding 1 to n leaves room for the null character
- Some programmers prefer to cast malloc's return value, although the cast is not required

```
p = (char *) malloc(n + 1);
```

Dynamic Storage Allocation

Memory Allocation Function

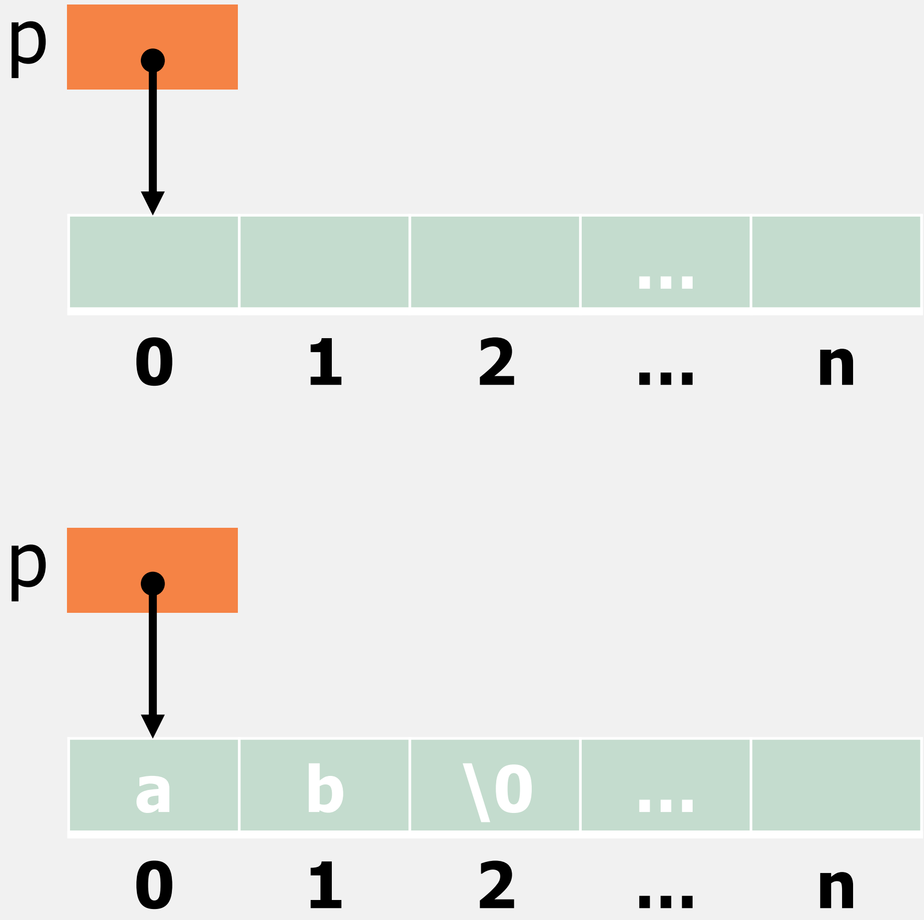
Malloc

- Memory allocated using malloc isn't cleared, so p will point to an uninitialized array of n + 1 characters

- Calling strcpy is one way to initialize this array

```
strcpy(p, "ab");
```

- The first four characters in the array will now be a, b, and \0



Dynamic Storage Allocation

Memory Allocation Function

Dynamic storage allocation makes it possible to write functions that return a pointer to a "new" string

Consider the problem of writing a function that concatenates two strings without changing either one

The function will measure the lengths of the two strings to be concatenated, then call malloc to allocate the right amount of space for the result

Dynamic Storage Allocation

Memory Allocation Function

A call of the concat function

```
p = concat("ab", "def");
```

After the call, p will point to the string "abdef", which is stored in a dynamically allocated array

```
char *concat(const char *s1, const char *s2)
{
  char *result;

  result = malloc(strlen(s1) + strlen(s2) + 1);
  if (result == NULL) {
    printf("Error: malloc failed in concat\n");
    exit(EXIT_FAILURE);
  }
  strcpy(result, s1);
  strcat(result, s2);
  return result;
}
```

Dynamic Storage Allocation

Memory Allocation Function

Functions such as concat that dynamically allocate storage must be used with care

When the string that concat returns is no longer needed, we'll want to call the free function to release the space that the string occupies

If we don't, the program may eventually run out of memory

Dynamic Storage Allocation

Memory Allocation Function

Write a function named my_malloc that serves as a "wrapper" for malloc n bytes and check to make sure that malloc doesn't return a null pointer, and then returns the pointer from malloc

```
void *p;
p = malloc(n);
```

```
Please input the byte number: 100
Memory allocation done
```

```
Please input the byte number: 5000000000
Memory allocation failed
```

Dynamic Storage Allocation

Memory Allocation Function

```c
void *my_malloc(long n)
{
  void *p;
  p = malloc(sizeof(long) * n);
  if (p == NULL)
  {
    printf("Memory allocation failed\n");
    exit(EXIT_FAILURE);
  }
  else
  {
    printf("Memory allocation done\n");
  }
  return p;
}
```

```c
int main()
{
  long n;
  int *p;

  printf("\nPlease input the byte number: ");
  scanf("%ld", &n);


  p = (int *)my_malloc(n);

}
```

Dynamic Storage Allocation

Memory Allocation Function

Dynamically allocated arrays have the same advantages as dynamically allocated strings

The close relationship between arrays and pointers makes a dynamically allocated array as easy to use as an ordinary array

Although malloc can allocate space for an array, the calloc function is sometimes used instead, since it initializes the memory that it allocates

The realloc function allows us to make an array "grow" or "shrink" as needed

Dynamic Storage Allocation

Malloc for an array

- We can now ignore the fact that a is a pointer and use it instead as an array name, thanks to the relationship between arrays and pointers in C
- For example, we could use the following loop to initialize the array that a points to

```
for (i = 0; i < n; i++)
    a[i] = 0;
```

We also have the option of using pointer arithmetic instead of subscripting to access the elements of the array

Dynamic Storage Allocation

Memory Allocation Function

Calloc

- The calloc function is an alternative to malloc
- Prototype for calloc

```
void *calloc(size_t nmemb, size_t size);
```

Properties of calloc

- Allocates space for an array with nmemb elements, each of which is size bytes long
- Returns a null pointer if the requested space isn't available
- Initializes allocated memory by setting all bits to 0

Dynamic Storage Allocation

Memory Allocation Function

A call of calloc that allocates space for an array of n integers

```
a = calloc(n, sizeof(int));
```

By calling calloc with 1 as its first argument, we can allocate space for a data item of any type

```
struct point { int x, y; } *p;

p = calloc(1, sizeof(struct point));
```

Dynamic Storage Allocation

Memory Allocation Function

Realloc

- ➢ The realloc function can resize a dynamically allocated array
- ➢ Prototype for realloc

```
void *realloc(void *ptr, size_t size);
```

ptr must point to a memory block obtained by a previous call of malloc, calloc, or realloc

size represents the new size of the block, which may be larger or smaller than the original size

Dynamic Storage Allocation

Memory Allocation Function

Properties of realloc

- When it expands a memory block, realloc doesn't initialize the bytes that are added to the block
- If realloc can't enlarge the memory block as requested, it returns a null pointer; the data in the old memory block is unchanged
- If realloc is called with a null pointer as its first argument, it behaves like malloc
- If realloc is called with 0 as its second argument, it frees the memory block

Dynamic Storage Allocation

Memory Allocation Function

We expect realloc to be reasonably efficient:

- When asked to reduce the size of a memory block, realloc should shrink the block "in place"
- realloc should always attempt to expand a memory block without moving it

If it can't enlarge a block, realloc will allocate a new block elsewhere, then copy the contents of the old block into the new one

Once realloc has returned, be sure to update all pointers to the memory block in case it has been moved

Dynamic Storage Allocation

Deallocating Storages

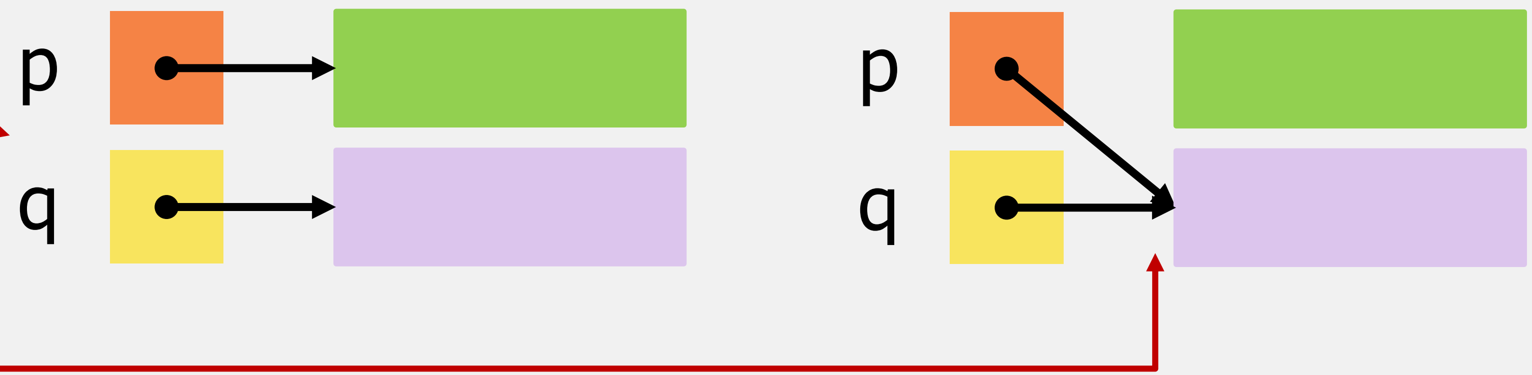
malloc and the other memory allocation functions obtain memory blocks from a storage pool known as the heap

Calling these functions too often—or asking them for large blocks of memory—can exhaust the heap, causing the functions to return a null pointer

To make matters worse, a program may allocate blocks of memory and then lose track of them, thereby wasting space

```
p = malloc(…);
q = malloc(…);
```

```
p = q;
```



Dynamic Storage Allocation

Deallocating Storages

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

Dynamic Storage Allocation

Deallocating Storages

Free function

- Prototype

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

- free will be passed a pointer to an unneeded memory block

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

- Calling free releases the block of memory that p points to

Dynamic Storage Allocation

Dangling pointer problem

- Using free function at inappropriate timing
- 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
- Dangling pointers can be hard to spot, since several pointers may point to the same block of memory
- When the block is freed, all the pointers are left dangling

Dynamic Storage Allocation

Deallocating Storages

Write two functions

- duplicate as following to create a copy of a string

```
#include <string.h>
*str = NULL;
size_t len = 0;
getline(&str, &len, stdin);
```

```
The input string: This is test

The duplicate string: This is test
```

- create_array as following which should return a pointer to a int array with n members and each of members is initialized to initial_value

```
int *create_array(int n, int initial_value);
```

```
Please input the array length and initial value (n/v): 10/5
The array is: 5 5 5 5 5 5 5 5 5 5
```

```
Please input the array length and initial value (n/v): 5000000000000000000/10
Memory allocation failed
```

Dynamic Storage Allocation

Deallocating Storages

```c
char *duplicate(const char *s)
{
  char *temp = malloc(strlen(s) + 1);

  if (temp == NULL)
    return NULL;

  strcpy(temp, s);
  return temp;
}
```

```c
int main()
{
  char *p, *str = NULL;
  size_t len = 0;

  printf("\nThe input string: ");

  getline(&str, &len, stdin);

  p = (char *)duplicate(str);

  if(p)
  {
    printf("\nThe duplicate string: %s\n", p);
  }
  else
  {
    printf("Memory allocation failed\n");
    exit(EXIT_FAILURE);
  }

}
```

Dynamic Storage Allocation

Deallocating Storages

```c
int *create_array(int n, int initial_value)
{
  int *arr, *p;

  arr = (int *)malloc(n * sizeof(int));
  if (arr != NULL)
    for (p = arr; p < arr + n; p++)
      *p = initial_value;
  return arr;
}
```

```c
int main()
{
  int *arr, num, initial_value;

  printf("\nPlease input the array length and initial value (n/v): ");
  scanf("%d/%d", &num, &initial_value);
  arr = create_array(num, initial_value);
  if(arr)
  {
      printf("The array is: ");
      for(int i = 0; i < num; i++)
        printf("%d ", arr[i]);
      printf("\n");
  }
  else
  {
      printf("Memory allocation failed\n");
      exit(EXIT_FAILURE);
  }

}
```

02

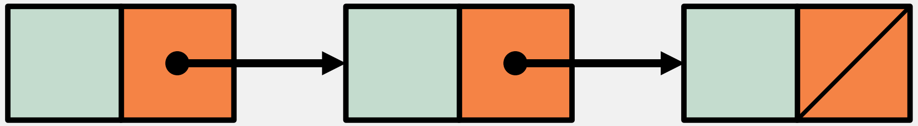
Linked List

Linked List

Introduction

Dynamic storage allocation is especially useful for building lists, trees, graphs, and other linked data structures

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

Linked List

Introduction

A linked list is more flexible than an array

- we can easily insert and delete nodes in a linked list, allowing the list to grow and shrink as needed

On the other hand, we lose the "random access" capability of an array

- Any element of an array can be accessed in the same amount of time
- Accessing a node in a linked list is fast if the node is close to the beginning of the list, slow if it’s near the end

Linked List
Introduction

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 */
};
```

- node must be a tag, not a typedef name, or there would be no way to declare the type of next

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

```
test.c:7:4: error: unknown type name 'Node'
    Node *next;   /* pointer to the next node */
    ^~~~
```

Creating a node type

- a variable that always points to the first node in the list

```
struct node *first = NULL;
```

- Setting first to NULL indicates that the list is initially empty
- As we construct a linked list, we'll create nodes one by one, adding each to the list
- Steps involved in creating a node
  - Allocate memory for the node
  - Store data in the node
  - Insert the node into the list

Creating a node type

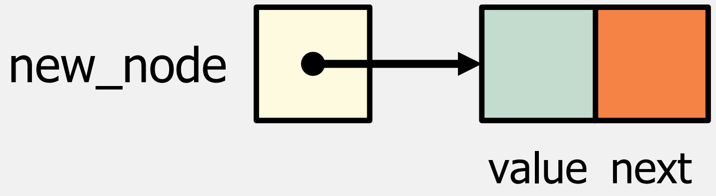
- When we create a node, we'll need a variable that can point to the node temporarily

```
struct node *new_node;
```

- We'll use malloc to allocate memory for the new node, saving the return value in new_node

```
new_node = malloc(sizeof(struct node));
```

- new_node now points to a block of memory just large enough to hold a node structure



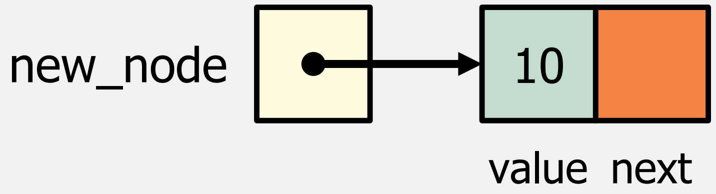
Linked List
Introduction

Creating a node type

- Next, we'll store data in the value member of the new node

```
(*new_node).value = 10;
```

- The resulting picture



- The parentheses around *new_node are mandatory because the . operator would otherwise take precedence over the * operator

Linked List

Operator

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 >
- Using the -> operator, we can write

```
new_node->value = 10;
```

- instead of

```
(*new_node).value = 10;
```

- The -> operator produces an lvalue, so we can use it wherever an ordinary variable would be allowed
- A scanf example

```
scanf("%d", &new_node->value);
```

- The & operator is still required, even though new_node is a pointer

Linked List

Operator

Suppose that f and p are declared as follows, which of the following statements are legal?

```
struct {
  union{
    char a, b;
    int c;
  } d;
  int e[5];
} f, *p = &f;
```

(a) p->b = ' ';
(b) p->e[3] = 10;
(c) (*p).d.a = '*';
(d) p->d->c = 20;

(a) illegal
(b) legal
(c) legal
(d) illegal

Linked List

Instructions

Inserting a node

- One of the advantages of a linked list is that nodes can be added at any point in the list
- However, the beginning of a list is the easiest place to insert a node
- Suppose that new_node is pointing to the node to be inserted, and first is pointing to the first node in the linked list
- It takes two statements to insert the node into the list
  - The first step is to modify the new node's next member to point to the node that was previously at the beginning of the list

```
new_node->next = first;
```

  - The second step is to make first point to the new node

```
first = new_node;
```

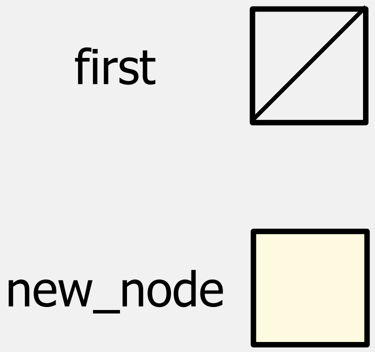
Linked List

Instructions

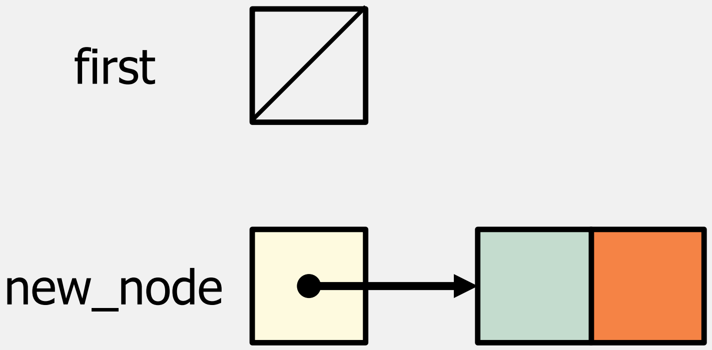
Inserting a node

- Let's trace the process of inserting two nodes into an empty list
- We'll insert a node containing the number 10 first, followed by a node containing 20

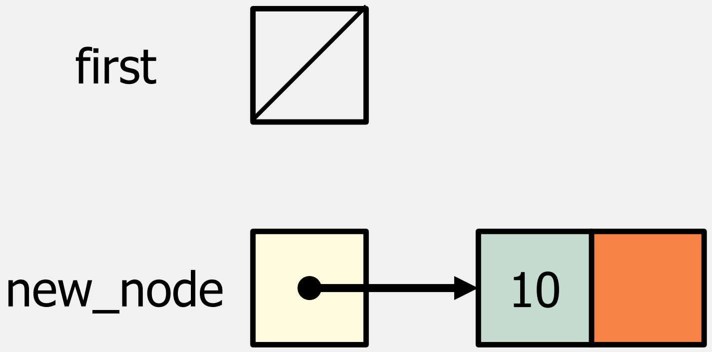
`first = NULL;`



`new_node = malloc(sizeof(struct node));`



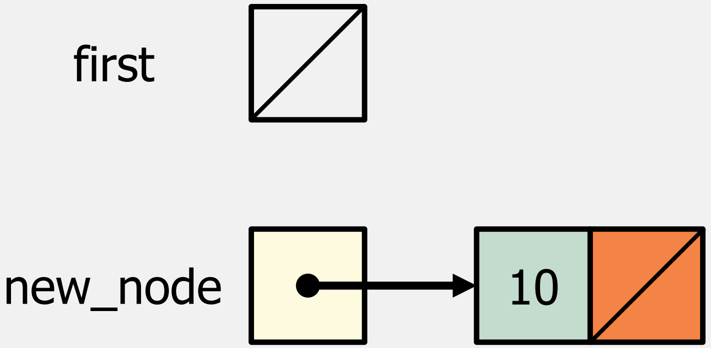
`new_node->value = 10;`



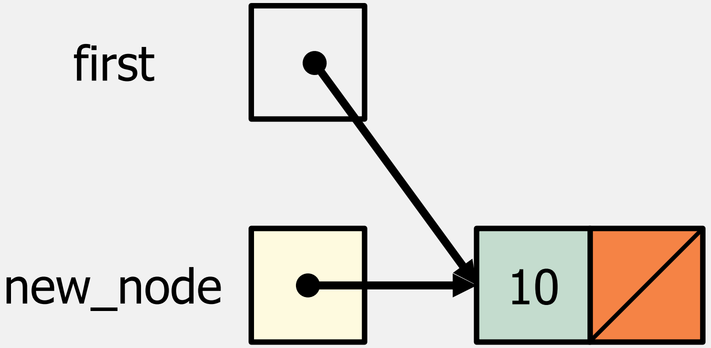
Linked List
Instructions

Inserting a node

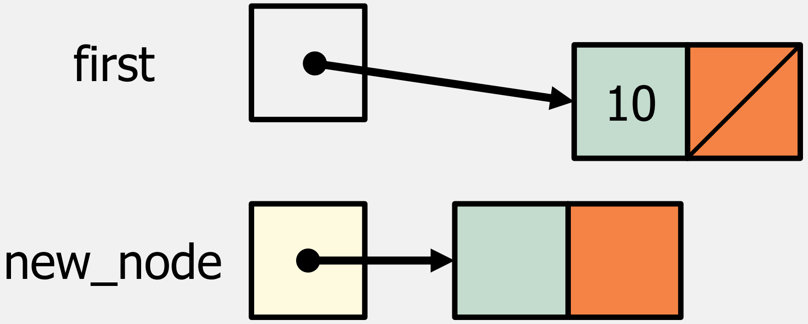
```
new_node->next = first;
```




```
first = new_node;
```




```
new_node = malloc(sizeof(struct node));
```

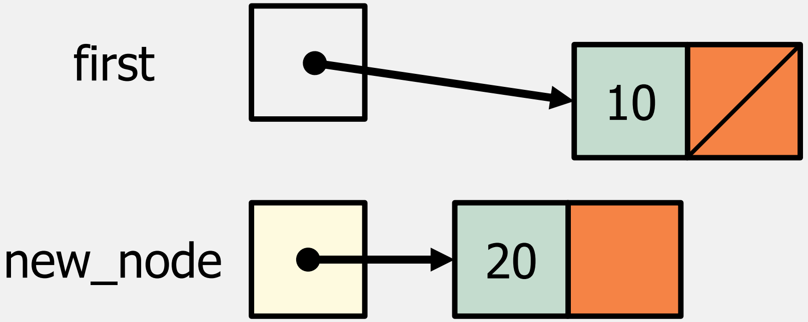


Linked List

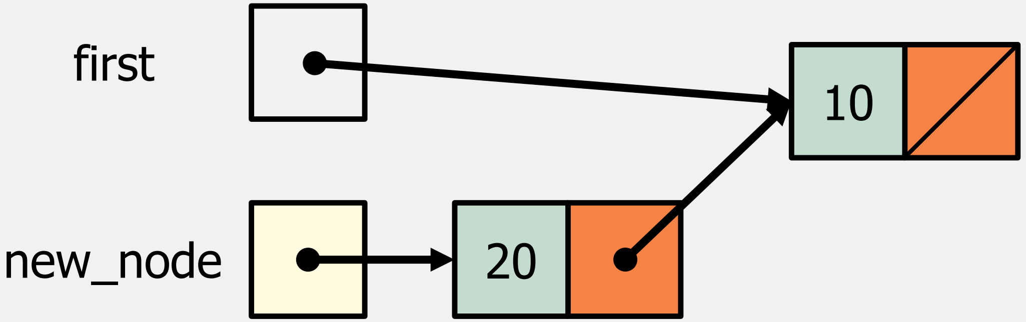
Instructions

Inserting a node

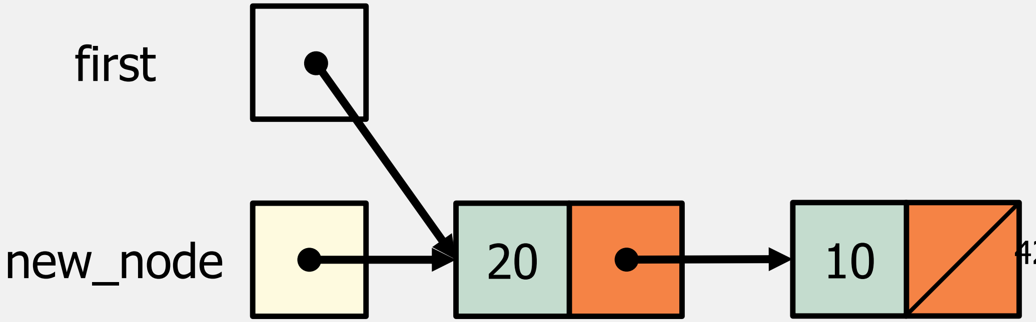
new_node->value = 20;



new_node->next = first;



first = new_node;



Linked List
Instructions

A function that inserts a node containing n into a linked list, which pointed to by list

```
struct node *add_to_list(struct node *list, int n)
{
    struct node *new_node;

    new_node = malloc(sizeof(struct node));
    if (new_node == NULL) {
      printf("Error: malloc failed in add_to_list\n");
      exit(EXIT_FAILURE);
    }
    new_node->value = n;
    new_node->next = list;
    return new_node;
}
```

Linked List

Instructions

Note that add_to_list returns a pointer to the newly created node (now at the beginning of the list)

When we call add_to_list, we'll need to store its return value into first

```
first = add_to_list(first, 10);
first = add_to_list(first, 20);
```

Linked List
Instructions

A function that uses add_to_list to create a linked list containing numbers entered by the user

```
struct node *add_node(void)
{
	struct node *first = NULL;
	int n;

	printf("Enter a series of integers (0 to terminate): ");
	for (;;) {
	  scanf("%d", &n);
	  if (n == 0)
	    return first;
	  first = add_to_list(first, n);
	}
}
```

The numbers will be in reverse order within the list

Linked List
Instructions

Searching a linked list

- Although a while loop can be used to search a list, the for statement is often superior
- 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)
    …
```

- A loop of this form can be used in a function that searches a list for an integer n

Linked List

Instructions

Searching a linked list

- If it finds n, the function will return a pointer to the node containing n; otherwise, it will return a null pointer
- An initial version of the function

```
struct node *search_list(struct node *list, int n)
{
        struct node *p;

        for (p = list; p != NULL; p = p->next)
          if (p->value == n)
            return p;
        return NULL;
}
```

Linked List

Instructions

Searching a linked list

- One alternative is to eliminate the p variable, instead using list itself to keep track of the current node

```
struct node *search_list(struct node *list, int n)
{
      for (; list != NULL; list = list->next)
        if (list->value == n)
          return list;
      return NULL;
}
```

- Since list is a copy of the original list pointer, there's no harm in changing it within the function

Linked List

Instructions

```c
struct node *add_to_list(struct node *list, int n)
{
  struct node *new_node;
  new_node = malloc(sizeof(struct node));
    if (new_node == NULL) {
      printf("Error: malloc failed in add_to_list\n");
      exit(EXIT_FAILURE);
    }
  new_node->value = n;
  new_node->next = list;
  return new_node;
}
 struct node *search_list(struct node *list, int n)
 {
     for (; list != NULL; list = list->next)
       if (list->value == n)
         return list;
     return NULL;
 }
struct node *search_list_change(struct node *list, int n, int new_n)
{
    for (; list != NULL; list = list->next)
      if (list->value == n)
      {
        list->value = new_n;
        return list;
      }
    return NULL;
}
```

```c
struct node *first = NULL, *p = NULL;
first = add_to_list(first, 10);
first = add_to_list(first, 20);
first = add_to_list(first, 30);

printf("\nBefore Searching===========\n");
for(p = first; p != NULL; p = p->next)
  printf("%d\n", p->value);

printf("\nAfter Searching with 20===========\n");
p = search_list(first, 20);

for(; p != NULL; p = p->next)
  printf("%d\n", p->value);

printf("\nAfter Searching and chagning===========\n");
p = search_list_change(first, 20, 25);

for(p = first; p != NULL; p = p->next)
  printf("%d\n", p->value);
```

```
Before Searching===========
30
20
10

After Searching with 20===========
20
10

After Searching and chagning===========
30
25
10
```

Linked List

Instructions

Searching a linked list

- Another alternative

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

- Since list is NULL if we reach the end of the list, returning list is correct even if we don't find n

Linked List

Instructions

Searching a linked list

- This version of search_list might be a bit clearer if we used a while statement

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

Linked List

Instructions

Deleting a node from linked list

- A big advantage of storing data in a linked list is that we can easily delete nodes
- Deleting a node involves three steps
  - Locate the node to be deleted
  - Alter the previous node so that it "bypasses" the deleted node
  - Call free to reclaim the space occupied by the deleted node
- Step 1 is harder than it looks, because step 2 requires changing the previous node
- There are various solutions to this problem

Linked List
Instructions

Deleting a node from linked list

- The "trailing pointer" technique involves keeping a pointer to the previous node (prev) as well as a pointer to the current node (cur)
- Assume that list points to the list to be searched and n is the integer to be deleted
- A loop that implements step 1

```
for (cur = list, prev = NULL;
     cur != NULL && cur->value != n;
     prev = cur, cur = cur->next)
  ;
```

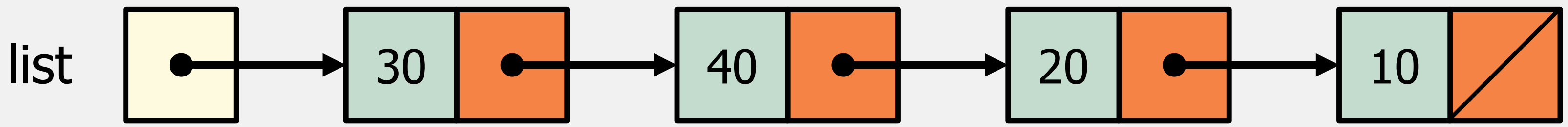
- When the loop terminates, cur points to the node to be deleted and prev points to the previous node

Linked List

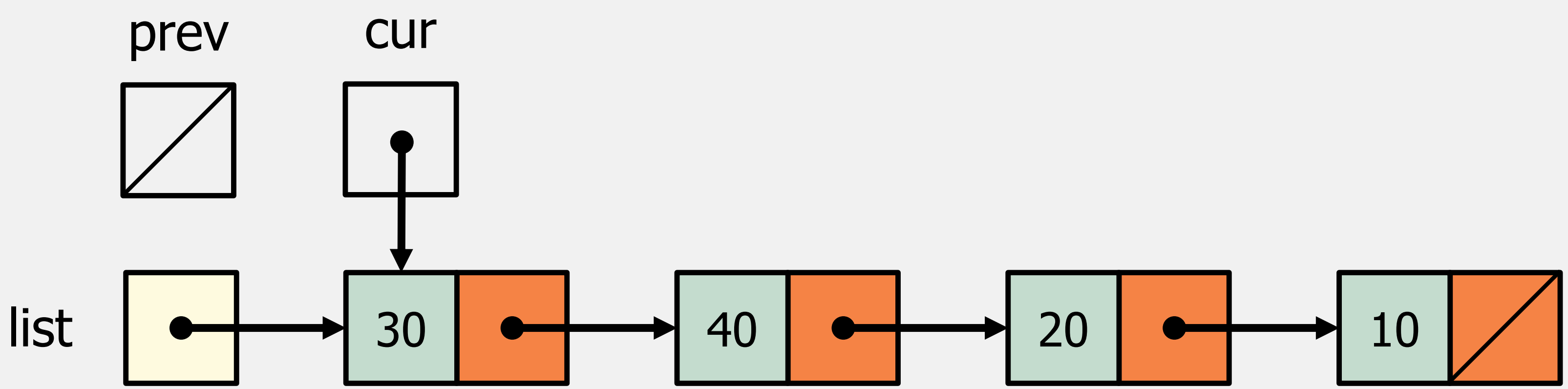
Instructions

Deleting a node from linked list

- Assume that list has the following appearance and n is 20



- After cur = list, prev = NULL has been executed

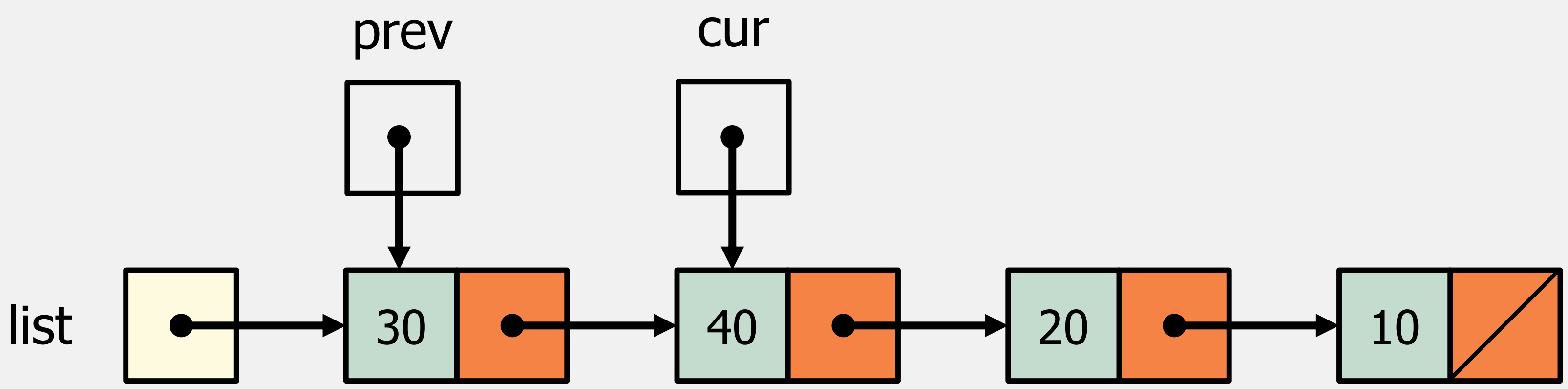


Linked List

Instructions

Deleting a node from linked list

- The test cur != NULL && cur->value != n is true, since cur is pointing to a node and the node doesn’t contain 20
- After prev = cur, cur = cur->next has been executed

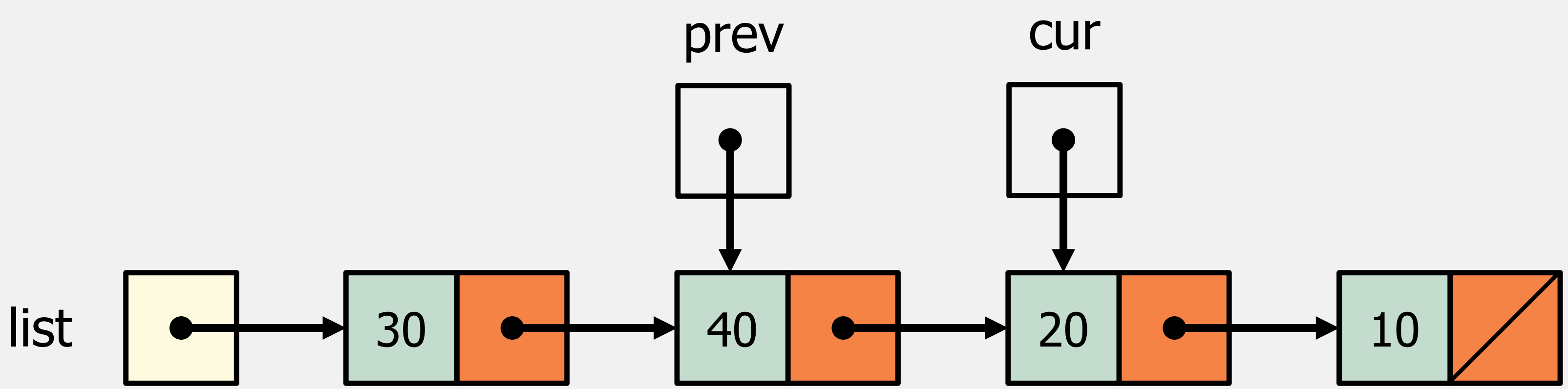


Linked List

Instructions

Deleting a node from linked list

- The test cur != NULL && cur->value != n is again true, so prev = cur, cur = cur->next is executed once more



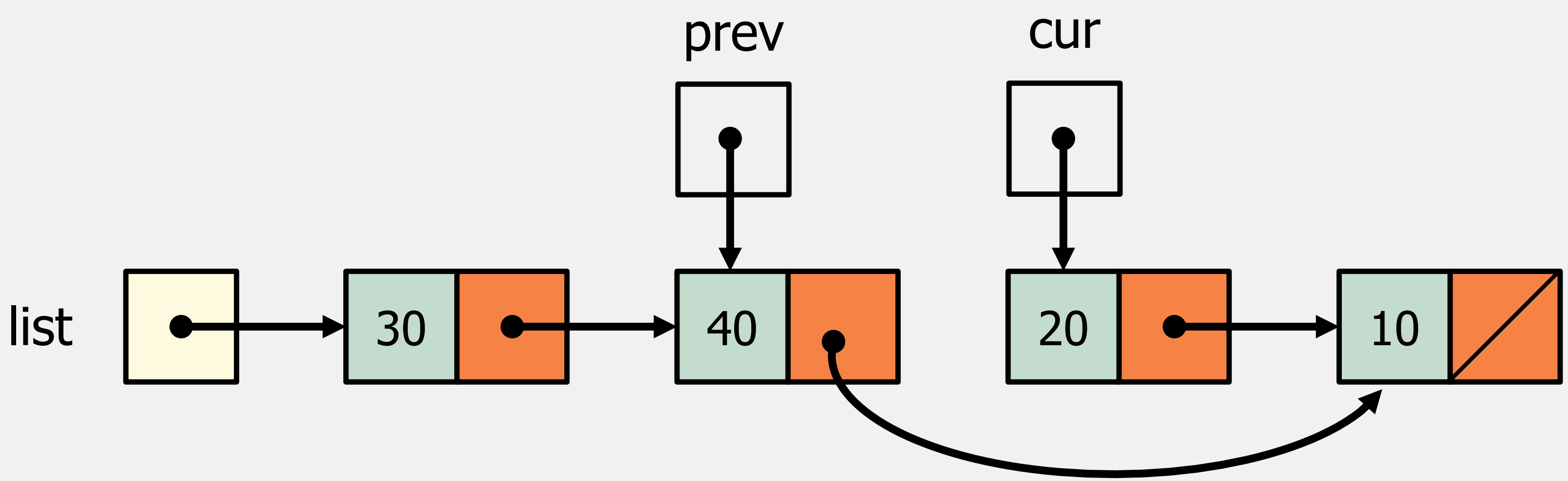
- Since cur now points to the node containing 20, the condition cur->value != n is false and the loop terminates

Linked List
Instructions

Deleting a node from linked list

- Next, The statement makes the pointer in the previous node point to the node after the current node

prev->next = cur->next;



- Finally, releasing the memory occupied by the current node

free(cur);

Linked List

Instructions

```
struct node *delete_from_list(struct node *list, int n)
{
  struct node *cur, *prev;

  for (cur = list, prev = NULL;
       cur != NULL && cur->value != n;
       prev = cur, cur = cur->next)
    ;
  if (cur == NULL)
    return list;              /* n was not found */
  if (prev == NULL)
    list = list->next;        /* n is in the first node */
  else
    prev->next = cur->next;  /* n is in some other node */
  free(cur);
  return list;
}
```

Linked List

Instructions

Ordered lists

- When the nodes of a list are kept in order—sorted by the data stored inside the nodes—we say that the list is ordered
- Inserting a node into an ordered list is more difficult, because the node won't always be put at the beginning of the list
- However, searching is faster: we can stop looking after reaching the point at which the desired node would have been located

Linked List
Pointer to

Pointer

- The concept of "pointers to pointers" also pops up frequently in the context of linked data structures
- In particular, when an argument to a function is a pointer variable, we may want the function to be able to modify the variable
- Doing so requires the use of a pointer to a pointer
- The add_to_list function is passed a pointer to the first node in a list; it returns a pointer to the first node in the updated list

Pointer

- Getting add_to_list to modify first requires passing add_to_list a pointer to first

```
void add_to_list(struct node **list, int n)
{
    struct node *new_node;

    new_node = malloc(sizeof(struct node));
    if (new_node == NULL) {
      printf("Error: malloc failed in add_to_list\n");
      exit(EXIT_FAILURE);
    }
    new_node->value = n;
    new_node->next = *list;
    *list = new_node;
}
```

```
struct node *add_to_list(struct node *list, int n)
{
    struct node *new_node;

    new_node = malloc(sizeof(struct node));
    if (new_node == NULL) {
      printf("Error: malloc failed in add_to_list\n");
      exit(EXIT_FAILURE);
    }
    new_node->value = n;
    new_node->next = list;
    return new_node;
}
```

Linked List

Pointer to

Pointer

- When the new version of add_to_list is called, the first argument will be the address of first

```
add_to_list(&first, 10);
```

- Since list is assigned the address of first, we can use *list as an alias for first
- In particular, assigning new_node to *list will modify first

Linked List

Pointer to

Rewrite the database using linked list

```
Enter operation code: i
Enter part number: 528
Enter part name: Disk drive
Enter quantity on hand: 10

Enter operation code: s
Enter part number: 528
Part name: Disk drive
Quantity on hand: 10

Enter operation code: s
Enter part number: 914
Part not found.
```

```
Enter operation code: i
Enter part number: 914
Enter part name: Printer cable
Enter quantity on hand: 5

Enter operation code: u
Enter part number: 528
Enter change in quantity on hand: -2

Enter operation code: s
Enter part number: 528
Part name: Disk drive
Quantity on hand: 8
```

Linked List

Pointer to

```
Enter operation code: p
Part Number   Part Name                    Quantity on Hand
    528       Disk drive                           8
    914       Printer cable                        5

Enter operation code: q
```

Linked List

Pointer to

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

#define NAME_LEN 25

struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
  struct part *next;
};

struct part *databases = NULL;

struct part *find_part(int number);
void insert(void);
void search(void);
void update(void);
void delete(void);
void print(void);
int read_line(char str[], int n);
```

```c
int main(void)
{
  char code;

  for (;;) {
    printf("Enter operation code: ");
    scanf(" %c", &code);
    while (getchar() != '\n')
      ;
    switch (code) {
      case 'i': insert();
                break;
      case 's': search();
                break;
      case 'u': update();
                break;
      case 'd': delete();
                break;
      case 'p': print();
                break;
      case 'q': return 0;
      default:  printf("Illegal code\n");
    }
    printf("\n");
  }
}
```

Linked List

Pointer to

```c
void search(void)
{
  int number;
  struct part *p;

  printf("Enter part number: ");
  scanf("%d", &number);
  p = find_part(number);
  if (p != NULL) {
    printf("Part name: %s\n", p->name);
    printf("Quantity on hand: %d\n", p->on_hand);
  } else
    printf("Part not found.\n");
}
void update(void)
{
  int number, change;
  struct part *p;

  printf("Enter part number: ");
  scanf("%d", &number);
  p = find_part(number);
  if (p != NULL) {
    printf("Enter change in quantity on hand: ");
    scanf("%d", &change);
    p->on_hand += change;
  } else
    printf("Part not found.\n");
}
struct part *find_part(int number)
{
  struct part *p;

  for (p = databases;
       p != NULL && number > p->number;
       p = p->next)
    ;
  if (p != NULL && number == p->number)
    return p;
  return NULL;
}
```

```c
void insert(void)
{
  struct part *cur, *prev, *new_node;

  new_node = malloc(sizeof(struct part));
  if (new_node == NULL) {
    printf("Database is full; can't add more parts.\n");
    return;
  }

  printf("Enter part number: ");
  scanf("%d", &new_node->number);

  for (cur = databases, prev = NULL;
       cur != NULL && new_node->number > cur->number;
       prev = cur, cur = cur->next)
    ;
  if (cur != NULL && new_node->number == cur->number) {
    printf("Part already exists.\n");
    free(new_node);
    return;
  }

  printf("Enter part name: ");
  read_line(new_node->name, NAME_LEN);
  printf("Enter quantity on hand: ");
  scanf("%d", &new_node->on_hand);

  new_node->next = cur;
  if (prev == NULL)
    databases = new_node;
  else
    prev->next = new_node;
}
```

```c
void delete(void)
{
  struct part *cur, *prev;
  int number;

  printf("Enter part number: ");
  scanf("%d", &number);

  for (cur = databases, prev = NULL;
       cur != NULL && number != cur->number;
       prev = cur, cur = cur->next)
       ;

  if (!cur){
    printf("Part not found.\n");
    return;
  }

  if (!prev)
    databases = databases->next;
  else
    prev->next = cur->next;

  free(cur);
}
```

```c
void print(void)
{
  struct part *p;

  printf("Part Number   Part Name                  "
         "Quantity on Hand\n");
  for (p = databases; p != NULL; p = p->next)
    printf("%7d       %-25s%11d\n", p->number, p->name,
           p->on_hand);
}
int read_line(char str[], int n)
{
  int ch, i = 0;

  while (isspace(ch = getchar()))
    ;
  while (ch != '\n' && ch != EOF) {
    if (i < n)
      str[i++] = ch;
    ch = getchar();
  }
  str[i] = '\0';
  return i;
}
```