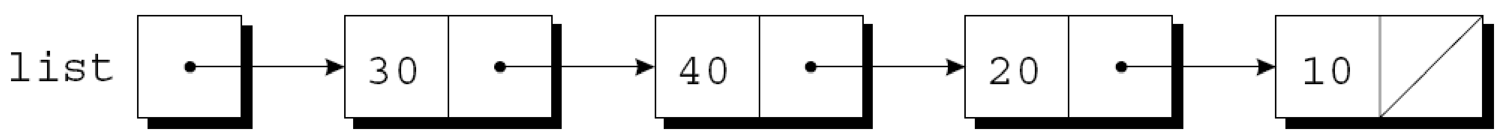
Lecture 17 - Advanced Uses of Pointers

Meng-Hsun Tsai
CSIE, NCKU



Dynamic Storage Allocation

- 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 before executing 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 (cont.)

- Dynamic storage allocation is used most often for strings, arrays, and structures.
- Dynamically allocated structures can be linked together to form lists, trees, and other data structures.
- Dynamic storage allocation is done by calling a memory allocation function.

Memory Allocation Functions

- 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).

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.

Null Pointers (cont.)

- An example of testing `malloc`'s return value:

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

- `NULL` is a macro (defined in various library headers) that represents the null pointer.
- Some programmers combine the call of `malloc` with the `NULL` test:

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

Null Pointers (cont.)

- 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) …
  ```

  we could write

  ```
  if (!p) …
  ```

- Instead of writing

  ```
  if (p != NULL) …
  ```

  we could write

  ```
  if (p) …
  ```

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.

Using `malloc` to Allocate Memory for a String

- 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.

Using `malloc` to Allocate Memory for a String (cont.)

- A call of `malloc` that allocates memory for a string of `n` characters:

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

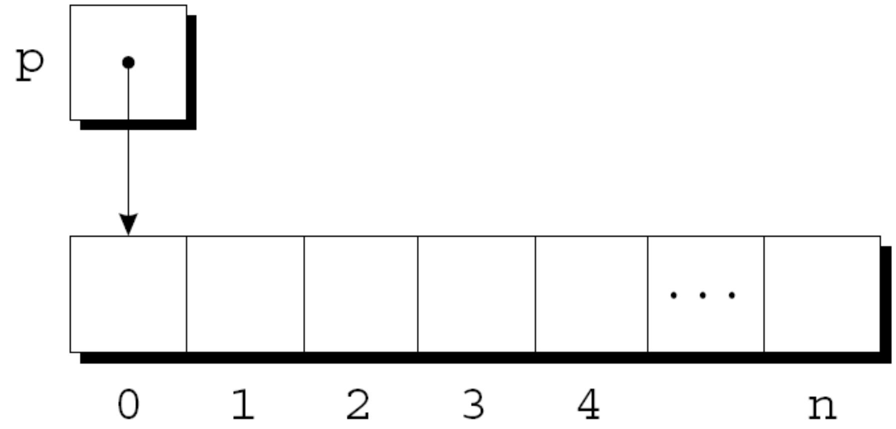
- Each character requires one byte of memory; adding 1 to `n` leaves room for the null character.

- Some programmers prefer to cast `malloc`'s return value, although the cast is not required:

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

Using `malloc` to Allocate Memory for a String (cont.)

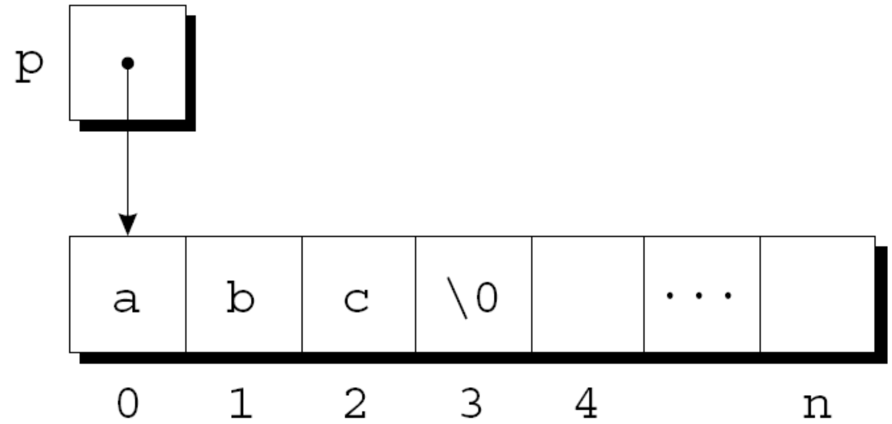
- Memory allocated using `malloc` isn't cleared, so `p` will point to an uninitialized array of `n` + 1 characters:



- Calling `strcpy` is one way to initialize this array:

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

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



Using Dynamic Storage Allocation in String Functions

- 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.

Using Dynamic Storage Allocation in String Functions (cont.)

```
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;
}
```

Using Dynamic Storage Allocation in String Functions (cont.)

- A call of the `concat` function:

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

- After the call, `p` will point to the string `"abcdef"`, which is stored in a dynamically allocated array.

Using Dynamic Storage Allocation in String Functions (cont.)

- 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.

Program: Printing a One-Month Reminder List (Revisited)

- The `remind2.c` program is based on the `remind.c` program of Lecture 13, which prints a one-month list of daily reminders.
- The original `remind.c` program stores reminder strings in a two-dimensional array of characters.
- In the new program, the array will be one-dimensional; its elements will be pointers to dynamically allocated strings.

IMSLab Since 2010

Program: Printing a One-Month Reminder List (Revisited) (cont.)

- Advantages of switching to dynamically allocated strings:
  - Uses space more efficiently by allocating the exact number of characters needed to store a reminder.
  - Avoids calling `strcpy` to move existing reminder strings in order to make room for a new reminder.
- Switching from a two-dimensional array to an array of pointers requires changing only eight lines of the program (shown in **bold**).

Program: Printing a One-Month Reminder List (Revisited) (cont.)

remind2.c

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

#define MAX_REMIND 50   /* maximum number of reminders */
#define MSG_LEN 60      /* max length of reminder message */

int read_line(char str[], int n);
int main(void)
{
  char *reminders[MAX_REMIND];
  char day_str[3], msg_str[MSG_LEN+1];
  int day, i, j, num_remind = 0;
```

```
char reminders[MAX_REMIND][MSG_LEN+3];
```

Program: Printing a One-Month Reminder List (Revisited) (cont.)

```
for (;;) {
  if (num_remind == MAX_REMIND) {
    printf("-- No space left --\n");
    break;
  }
  printf("Enter day and reminder: ");
  scanf("%2d", &day);
  if (day == 0)
    break;
  sprintf(day_str, "%2d", day);
  read_line(msg_str, MSG_LEN);

  for (i = 0; i < num_remind; i++)
    if (strcmp(day_str, reminders[i]) < 0)
      break;
  for (j = num_remind; j > i; j--)
    reminders[j] = reminders[j-1];
```

```
strcpy(reminders[j],
   reminders[j-1]);
```

Program: Printing a One-Month Reminder List (Revisited) (cont.)

```
    reminders[i] = malloc(2 + strlen(msg_str) + 1);
    if (reminders[i] == NULL) {
      printf("-- No space left --\n");
      break;
    }

    strcpy(reminders[i], day_str);
    strcat(reminders[i], msg_str);

    num_remind++;
  }

  printf("\nDay Reminder\n");
  for (i = 0; i < num_remind; i++)
    printf(" %s\n", reminders[i]);

  return 0;
}
```

Dynamically Allocated Arrays

- The close relationship between arrays and pointers makes a dynamically allocated array as easy to use as an ordinary array.
- Suppose a program needs an array of `n` integers, where `n` is computed during program execution.
- We'll first declare a pointer variable:

```
int *a;
```

- Once the value of `n` is known, the program can call `malloc` to allocate space for the array:

```
a = malloc(n * sizeof(int));
```

- Always use the `sizeof` operator to calculate the amount of space required for each element.

Using `malloc` to Allocate Storage for an Array (cont.)

- 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.

The `calloc` Function

- 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.

The `calloc` Function (cont.)

- 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));
```

The `realloc` Function

- 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.

```
q = realloc(p, 20000);
```

The `realloc` Function (cont.)

- 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.

The `realloc` Function (cont.)

- 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.

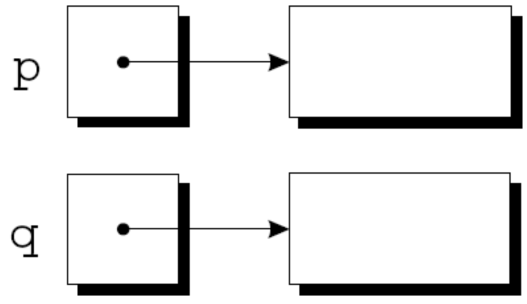
Deallocating Storage

- `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.

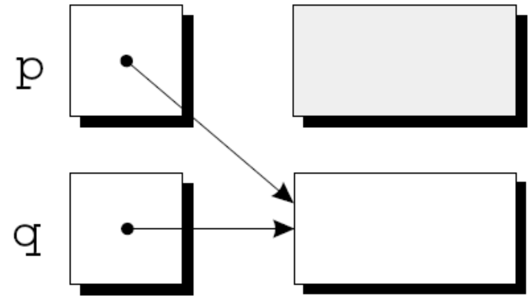
Deallocating Storage (cont.)

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

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



- 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 (cont.)

- 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.

The **free** Function

- Prototype for `free`:

```
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.

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 ***/
```

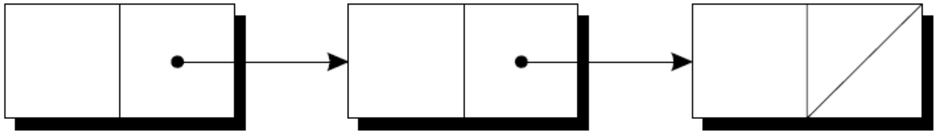
- Modifying the memory that `p` points to is a serious error.

The "Dangling Pointer" Problem (cont.)

- 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.

Linked Lists

- 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 Lists (cont.)

- 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.

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 */
};
```

Declaring a Node Type (cont.)

- Next, we’ll need 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.

Creating a Node

- As we construct a linked list, we'll create nodes one by one, adding each to the list.
- Steps involved in creating a node:
  1. Allocate memory for the node.
  2. Store data in the node.
  3. Insert the node into the list.
- We'll concentrate on the first two steps for now.

Creating a Node (cont.)

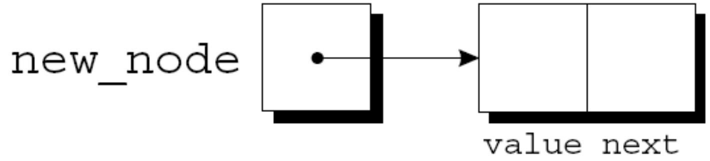
- When we create a node, we'll need a variable that can point to the node temporarily:

```
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:

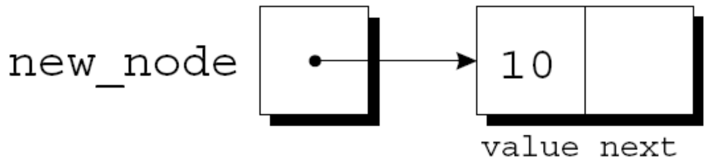


Creating a Node (cont.)

- 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.

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 (cont.)

- 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.

Inserting a Node at the Beginning of a Linked List

- Suppose that `new_node` is pointing to the node to be inserted, and `first` is pointing to the first node in the linked 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;
```

- These statements work even if the list is empty.

Inserting a Node at the Beginning of a Linked List (cont.)

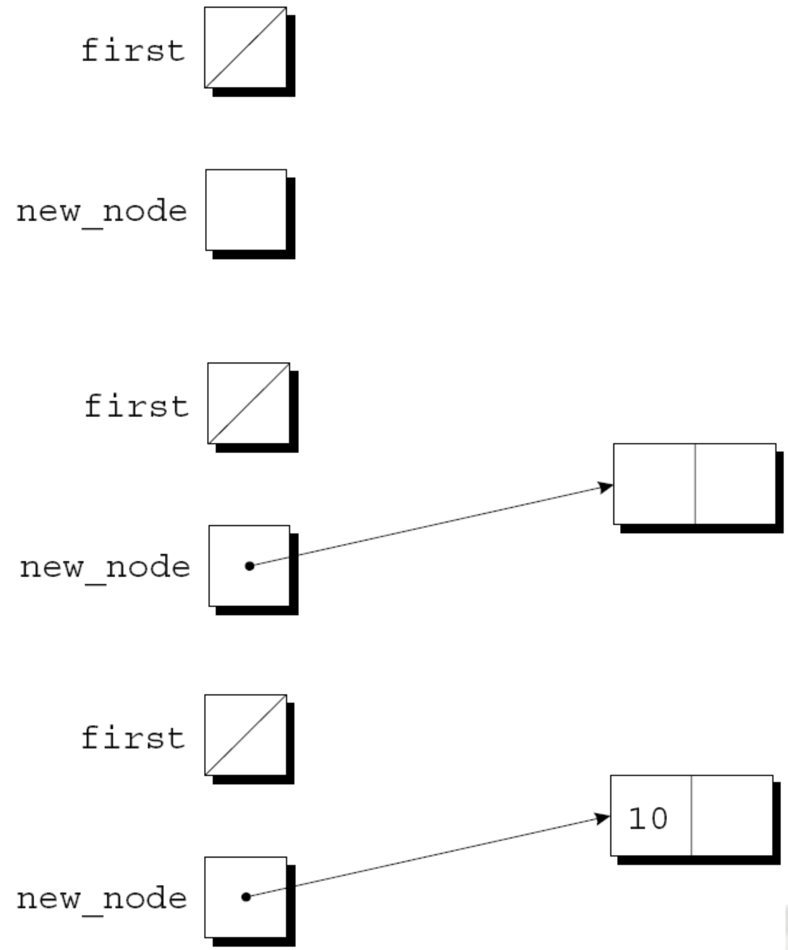
- Let’s trace the process of inserting two nodes into an empty list.
- We’ll insert a node containing the number 10 first, followed by a node containing 20.

Inserting a Node at the Beginning of a Linked List (cont.)

```
first = NULL;
```

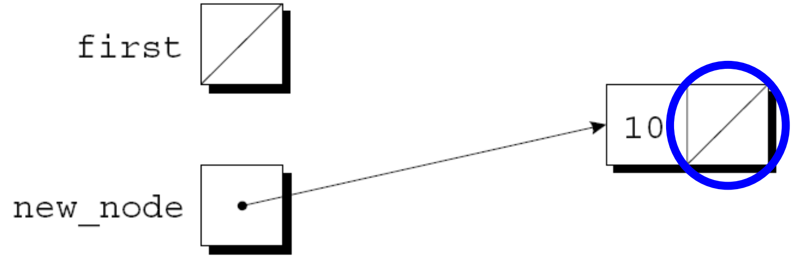
```
new_node =
  malloc(sizeof(struct node));
```

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

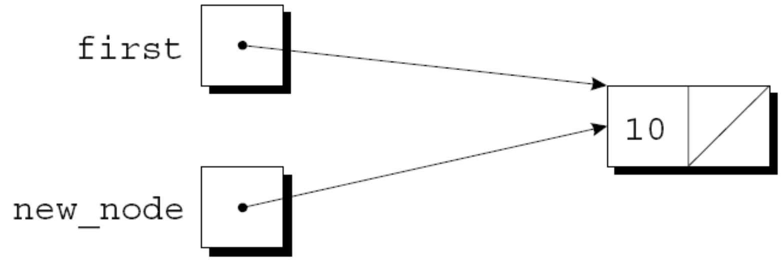


Inserting a Node at the Beginning of a Linked List (cont.)

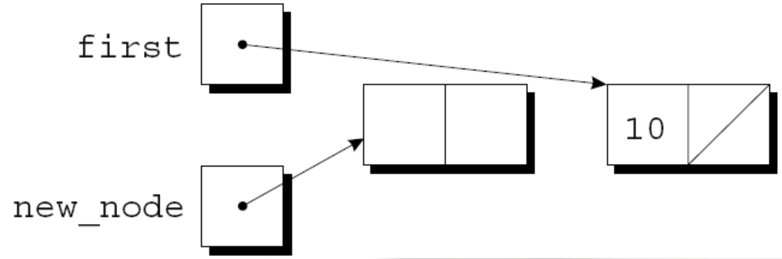
```
new_node->next = first;
```




```
first = new_node;
```

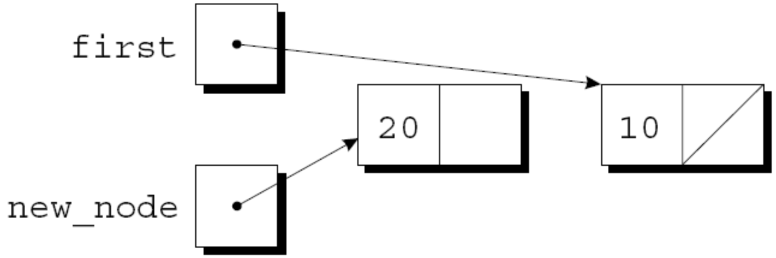



```
new_node =
  malloc(sizeof(struct node));
```

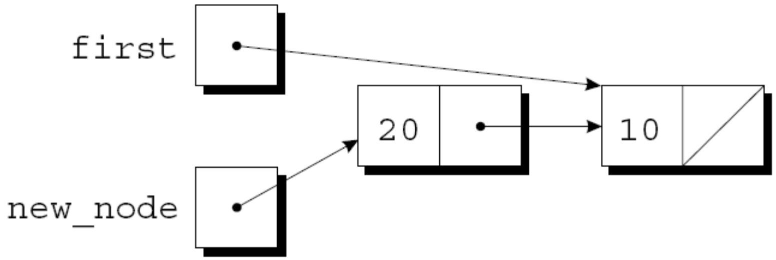


Inserting a Node at the Beginning of a Linked List (cont.)

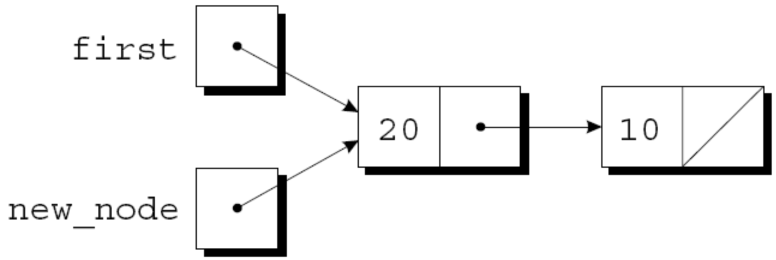
```
new_node->value = 20;
```




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




```
first = new_node;
```



IMSLab since 2010

Inserting a Node at the Beginning of a Linked List (cont.)

- 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;
}
```

Inserting a Node at the Beginning of a Linked List (cont.)

- 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);
```

IMSLab Since 2010

Inserting a Node at the Beginning of a Linked List (cont.)

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

```
struct node *read_numbers(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.

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)
  …
```

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

```
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;
}
```

Deleting a Node from a 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:
  1. Locate the node to be deleted.
  2. Alter the previous node so that it "bypasses" the deleted node.
  3. 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.

Deleting a Node from a Linked List (cont.)

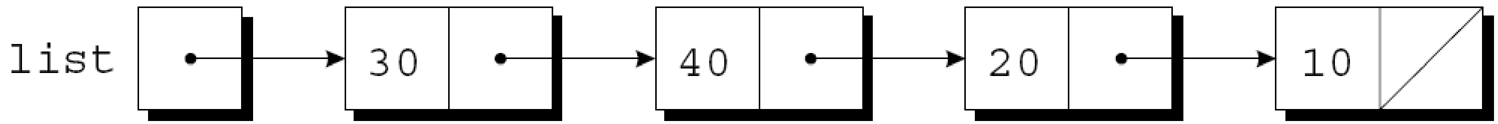
- We can keep a pointer to the previous node (`prev`) as well as a pointer to the current node (`cur`).
- Assume that `list` points to the list to be searched and `n` is the integer to be deleted.
- A loop that implements step 1:

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

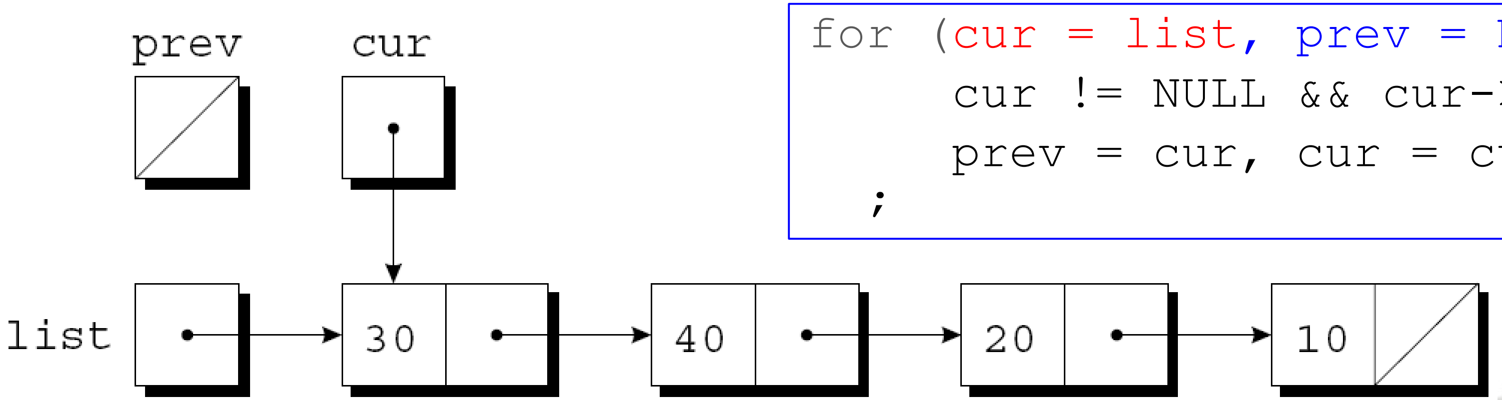
- When the loop terminates, `cur` points to the node to be deleted and `prev` points to the previous node.

Deleting a Node from a Linked List (cont.)

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



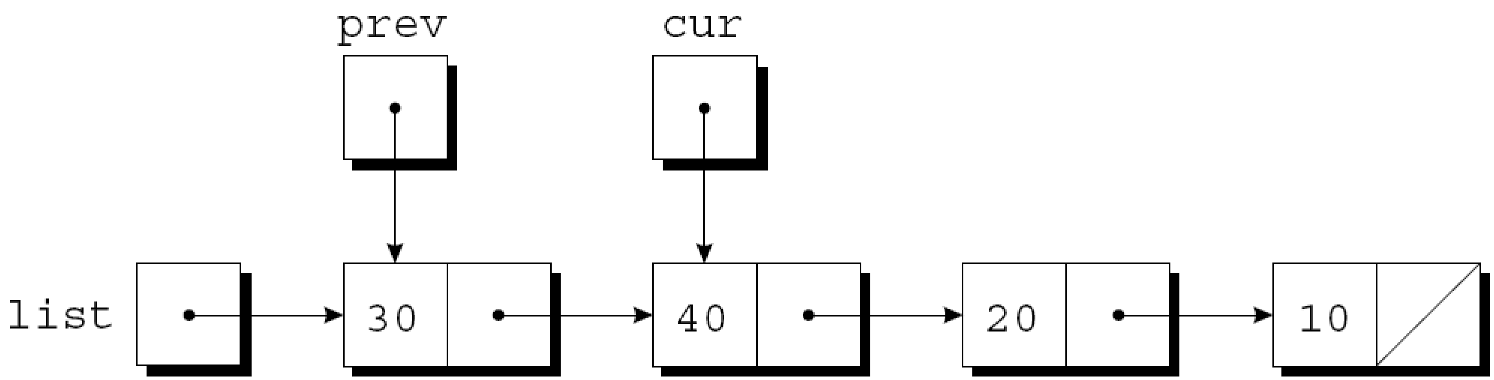
- After `cur = list, prev = NULL` has been executed:




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

Deleting a Node from a Linked List (cont.)

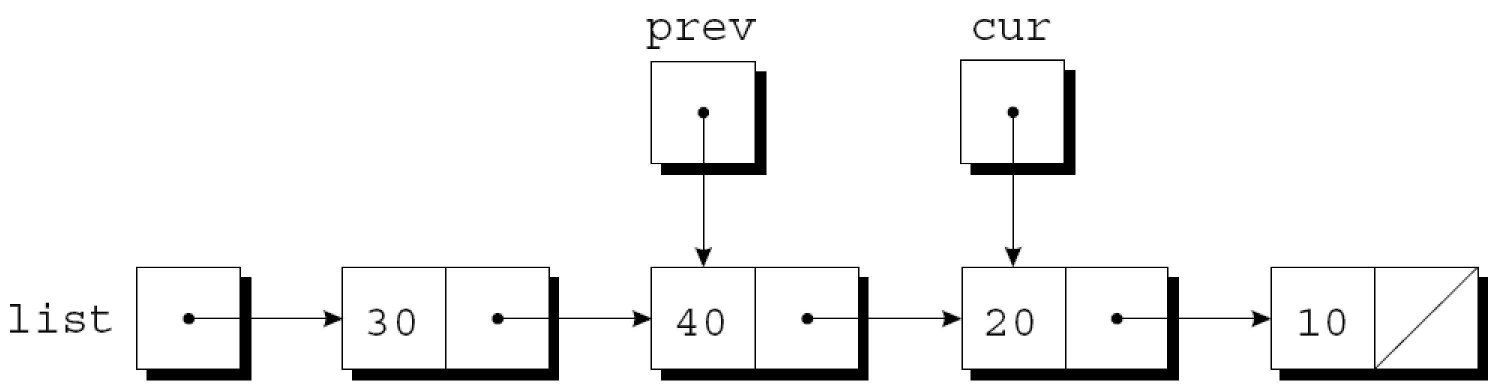
- The test `cur != NULL && cur->value != n` is true, since `cur` is pointing to a node and the node doesn't contain 20.
- After `prev = cur, cur = cur->next` has been executed:




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

Deleting a Node from a Linked List (cont.)

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



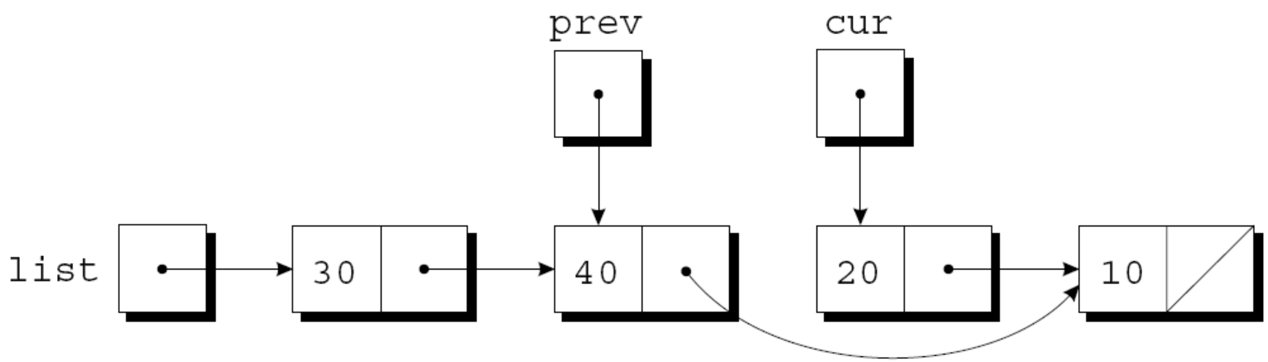
- Since `cur` now points to the node containing 20, the condition `cur->value != n` is false and the loop terminates.

Deleting a Node from a Linked List (cont.)

- At step 2, the following statement

  ```
  prev->next = cur->next;
  ```

  makes the pointer in the previous node point to the node *after* the current node:



- Step 3 is to release the memory occupied by the current node:

  ```
  free(cur);
  ```

Deleting a Node from a Linked List (cont.)

- The `delete_from_list` function uses the strategy just outlined.
- When given a list and an integer `n`, the function deletes the first node containing `n`.
- If no node contains `n`, `delete_from_list` does nothing.
- In either case, the function returns a pointer to the list.
- Deleting the first node in the list is a special case that requires a different bypass step.

Deleting a Node from a Linked List (cont.)

```
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;
}
```

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.

Program: Maintaining a Parts Database (Revisited)

- The `inventory2.c` program is a modification of the parts database program of Lecture 16, with the database stored in a linked list this time.
- Advantages of using a linked list:
  - No need to put a limit on the size of the database.
  - Database can easily be kept sorted by part number.
- In the original program, the database wasn't sorted.

Program: Maintaining a Parts Database (Revisited) (cont.)

- The `part` structure will contain an additional member (a pointer to the next node):

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

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

- `inventory` will point to the first node in the list:

```
struct part *inventory = NULL;
```

Program: Maintaining a Parts Database (Revisited) (cont.)

- `find_part` and `insert` will be more complex, however, since we'll keep the nodes in the inventory list sorted by part number.
- In the original program, `find_part` returns an index into the `inventory` array.
- In the new program, `find_part` will return a pointer to the node that contains the desired part number.
- If it doesn't find the part number, `find_part` will return a null pointer.

Program: Maintaining a Parts Database (Revisited) (cont.)

- Since the list of parts is sorted, `find_part` can stop when it finds a node containing a part number that's greater than or equal to the desired part number.
- `find_part`'s search loop:

```
for (p = inventory;
     p != NULL && number > p->number;
     p = p->next)
  ;
```

- When the loop terminates, we'll need to test whether the part was found:

```
if (p != NULL && number == p->number)
  return p;
```

Program: Maintaining a Parts Database (Revisited) (cont.)

- The original version of `insert` stores a new part in the next available array element.
- The new version must determine where the new part belongs in the list and insert it there.
- It will also check whether the part number is already present in the list.
- A loop that accomplishes both tasks:

```
for (cur = inventory, prev = NULL;
     cur != NULL && new_node->number > cur->number;
     prev = cur, cur = cur->next)
  ;
```

Program: Maintaining a Parts Database (Revisited) (cont.)

- Once the loop terminates, `insert` will check whether `cur` isn't `NULL` and whether `new_node->number` equals `cur->number`.
  - If both are true, the part number is already in the list.
  - Otherwise, `insert` will insert a new node between the nodes pointed to by `prev` and `cur`.
- This strategy works even if the new part number is larger than any in the list.

Program: Maintaining a Parts Database (Revisited) (cont.)

inventory2.c

```
#include <stdio.h>
#include <stdlib.h>
#include "readline.h"
#define NAME_LEN 25

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

struct part *inventory = NULL;   /* points to first part */

struct part *find_part(int number);
void insert(void);
void search(void);
void update(void);
void print(void);
```

```
struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} inventory[MAX_PARTS];
```

```
int num_parts = 0;   /* number of parts */
```

```
int find_part(int number);
```

Program: Maintaining a Parts Database (Revisited) (cont.)

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

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

Program: Maintaining a Parts Database (Revisited) (cont.)

```
struct part *find_part(int number)
{
  struct part *p;

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

```
int find_part(int number)
{
  int i;

  for (i = 0; i < num_parts; i++)
    if (inventory[i].number == number)
      return i;
  return -1;
}
```

Program: Maintaining a Parts Database (Revisited) (cont.)

```
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);
```

```
if (num_parts == MAX_PARTS) {
    printf("Database is full; can't add more parts.\n");
    return;
  }
```

Program: Maintaining a Parts Database (Revisited) (cont.)

```
for (cur = inventory, 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)
  inventory = new_node;
else
  prev->next = new_node;
}
```

```
if (find_part(part_number) >= 0) {
  printf("Part already exists.\n");
  return;
}
```

```
// insert at the end
inventory[num_parts].number = part_number;
num_parts++;
```

Program: Maintaining a Parts Database (Revisited) (cont.)

```
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");
}
```

```
i = find_part(number);
if (i >= 0) {
```

IMSLab since 2010

Program: Maintaining a Parts Database (Revisited) (cont.)

```
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");
}
```

```
i = find_part(number);
if (i >= 0) {
```

```
inventory[i].on_hand += change;
```

Program: Maintaining a Parts Database (Revisited) (cont.)

```
void print(void)
{
  struct part *p;
  printf("Part Number   Part Name                  "
         "Quantity on Hand\n");
  for (p = inventory; p != NULL; p = p->next)
    printf("%7d       %-25s%11d\n", p->number, p->name,
           p->on_hand);
}
```

```
int i;
for (i = 0; i < num_parts; i++)
```

A Quick Review to This Lecture

- The <stdlib.h> header declares three memory allocation functions which return void pointers (`void *`) to allocated space or return ***null pointers*** if allocation is failed:
  - `malloc` allocates a block of `size` bytes but doesn't initialize it.

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

```
p = malloc(10000);
if (p == NULL){ /* failed*/ }
```

  - `calloc` allocates an array with `nmemb` elements, each of which is `size` bytes long, and clears it.

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

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

  - `realloc` resizes a block (where `ptr` points to) to a new size `size`

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

A Quick Review to This Lecture (cont.)

- Each C program is responsible for recycling its own garbage by calling the `free` function to release unneeded memory.

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

```
p = malloc(…);
free(p);
```

- When the block is freed, all the pointers are left dangling.