Introduction to Computers and Programming

Lecture 11 – Chap 17

Advanced Uses of Pointers

Tien-Fu Chen

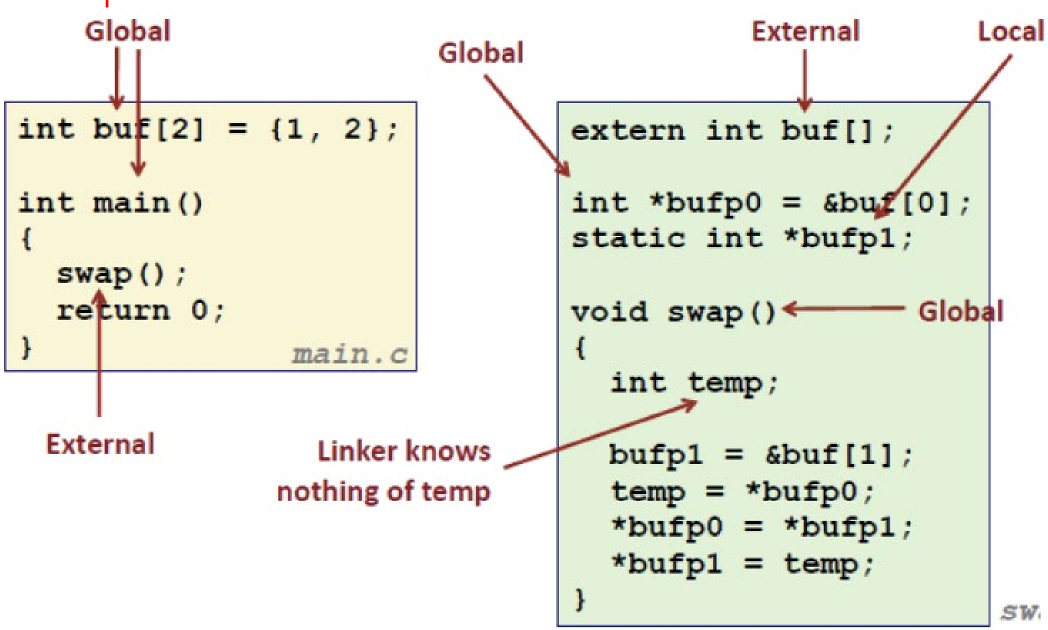
Dept. of Computer Science and Information Engineering

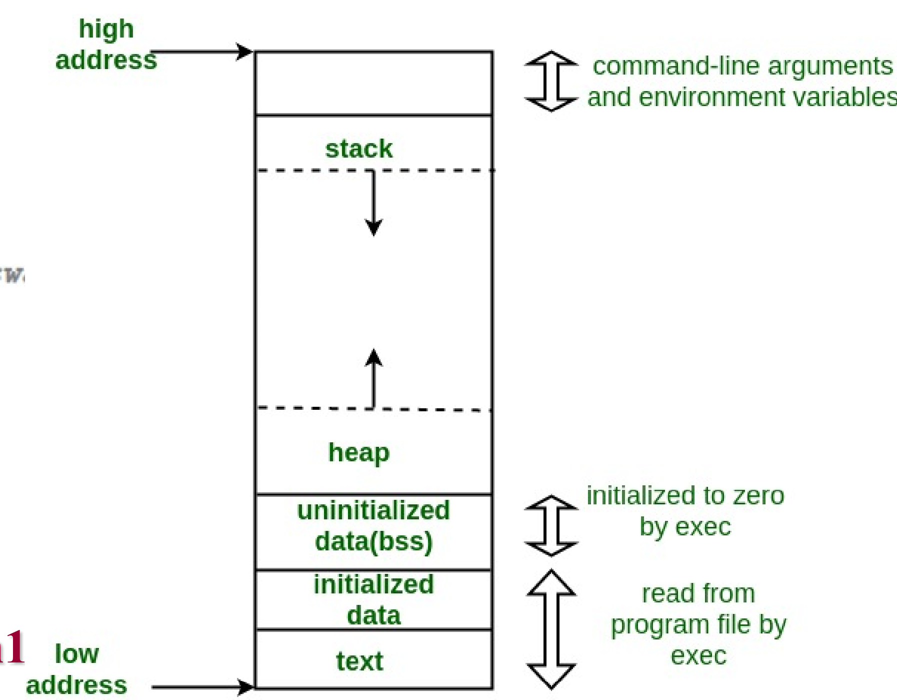
National Yang Ming Chiao Tung Univ.

Dynamic Storage Allocation

Dynamically allocating Storage

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






```
char *p = "this is a book";
```

Dynamic Storage Allocation

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

- 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. ***malloc() free()***

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.***
- 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

- 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) …
```

Using `malloc` to Allocate Memory for a String

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

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

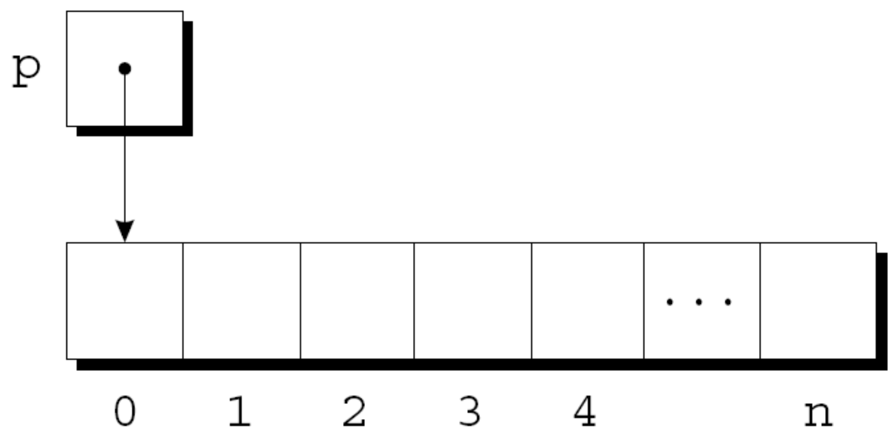
  `p` is a `char *` variable.

- 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

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

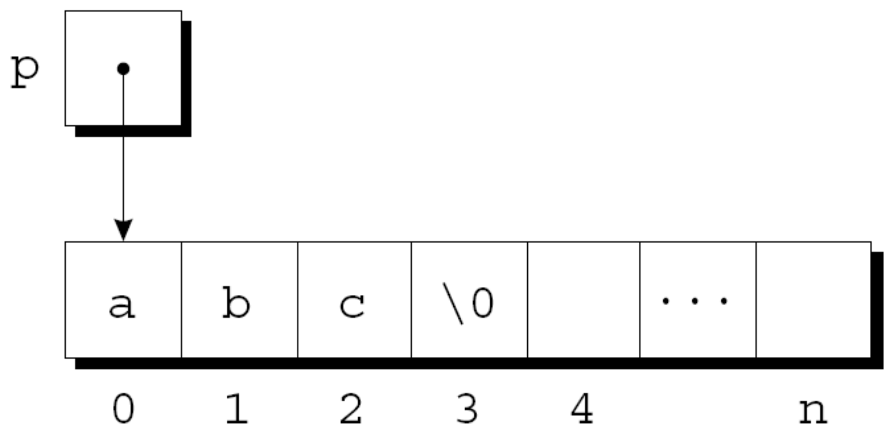


Using `malloc` to Allocate Memory for a String

- 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

```
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

- 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.

- When the string that `concat` returns is no longer needed, we'll want to call the `free` function to release.
- If we don't, the program may eventually run out of memory.

Using `malloc` to Allocate Storage for an 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.

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

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

- 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.

The `realloc` Function

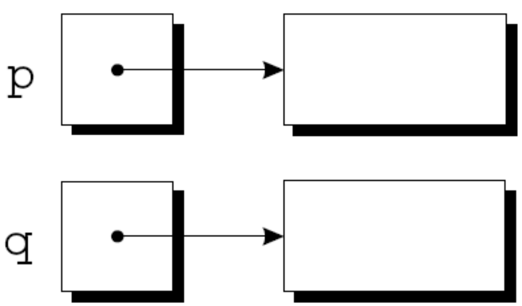
- We expect `realloc` to be reasonably efficient:
  - When asked to reduce the size of a memory block, `realloc` should shrink the block "in place."
  - `realloc` should always attempt to expand a memory block without moving it.
- If it can't enlarge a block, `realloc` will allocate a new block elsewhere, 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.

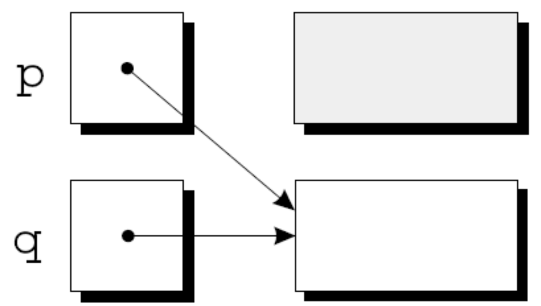
Deallocating Storage

- Example:

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

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





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.

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.

Dangling pointer errors involving heap and stack

Heap based

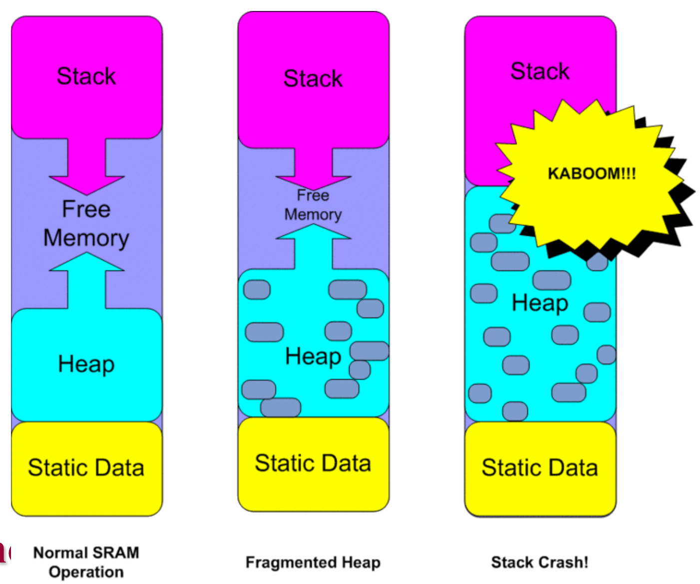
```
int *p, *q, *r;
p = malloc(8);
...
q = p;
...
free(p);
r = malloc(8);
...
... = *q;
```

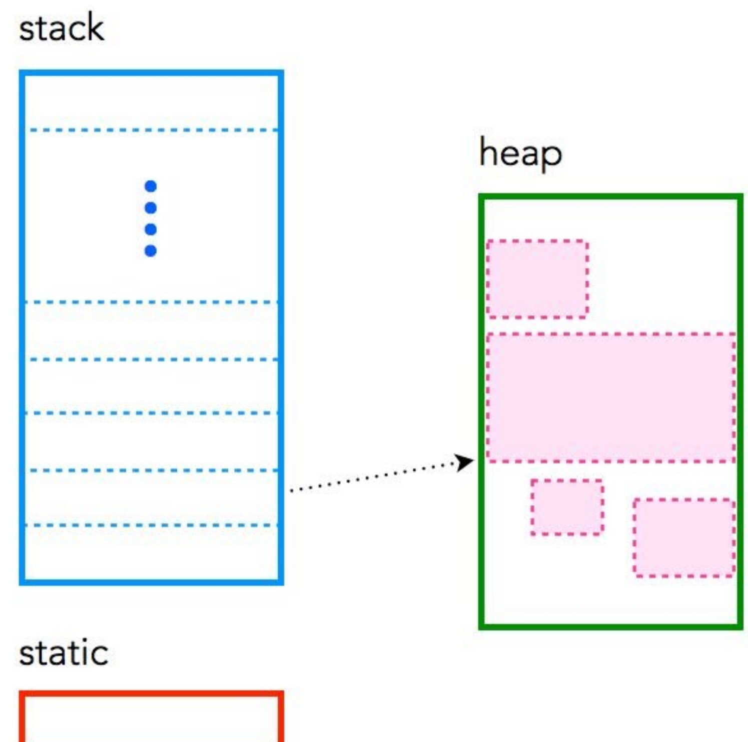
Stack based

```
int* q;
void foo() {
  int a;
  q = &a;
}
int main() {
  foo();
  ... = *q;
}
```

Memory IN C

- static: global variable storage, permanent for the entire run of the program.
- stack: local variable storage (automatic, continuous memory).
- heap: dynamic storage (large pool of memory, not allocated in contiguous order).



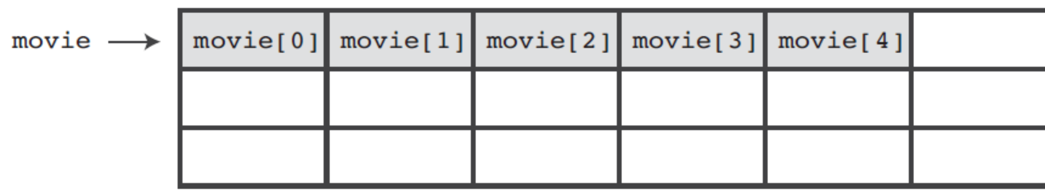


Linked list

Allocating structures in a block v.s. allocating them individually

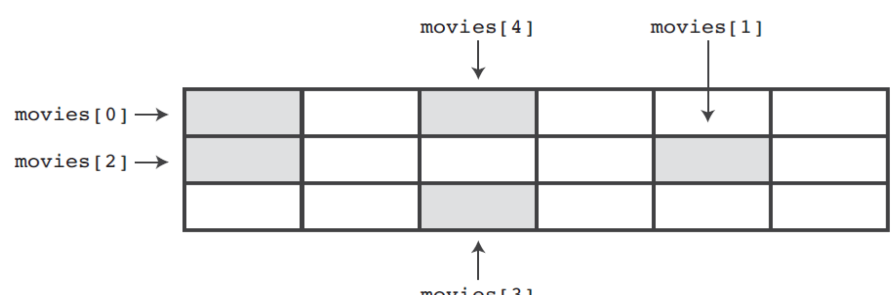
- Allocating structures in a array block

```
struct film * movie;

movie = (struct film *)
malloc(5*sizeof(struct film);
```



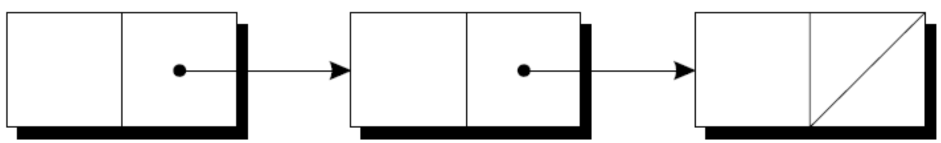
- Allocating structures individually

```
struct film * movies[s];
for (i = 0; i < 5; i++)
     movies[i] = (struct films *)
          malloc(sizeof(struct films));
```



Linked Lists

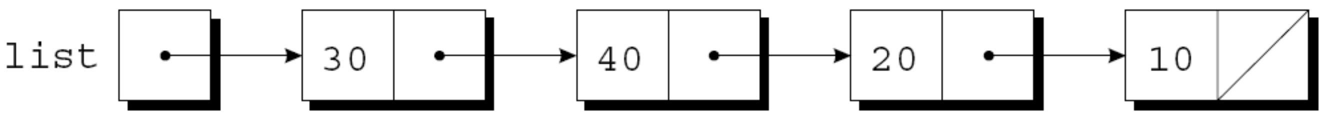
- Lists, trees, graphs are linked data structures: use dynamic storage allocation to build.
- A ***linked list*** consists of a chain of structures (called ***nodes***), with each node containing a pointer to the next node in the chain:



- The last node in the list contains a null pointer.

Compare array with linked lists

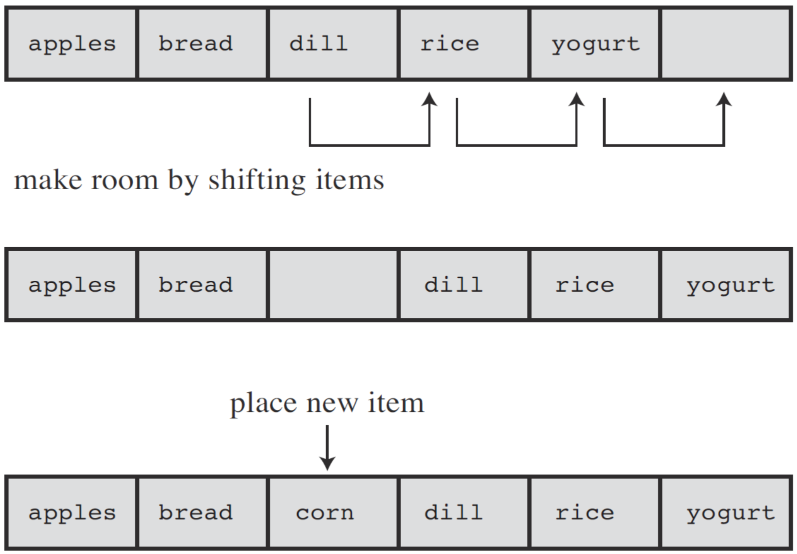
- Good: A linked list is more flexible than an array:
  - we can easily insert and delete nodes in a linked list, allowing the list to grow and shrink as needed.
- Bad: we lose the "random access" capability of an array:
  - Any element of an array can be accessed in same amount of time.



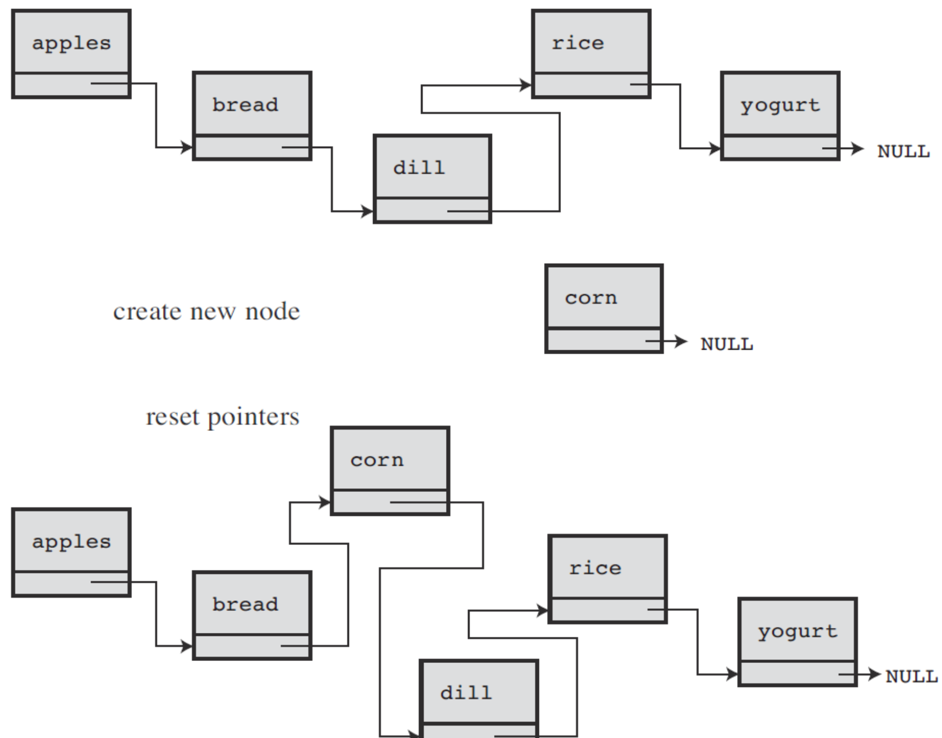
Data Form	Pros	Cons
Array	Directly supported by C. Provides random access. at compile time.	Size determined Inserting and deleting elements is time consuming
Linked list	Size determined during runtime. Inserting and deleting elements is quick.	No random access. User must provide programming support.

Insert a new data

- Insert in an array



- Insert into a linked list



Declaring a Node Type

- A node structure will contain data (an integer in this example) plus a pointer to the next node in the list:

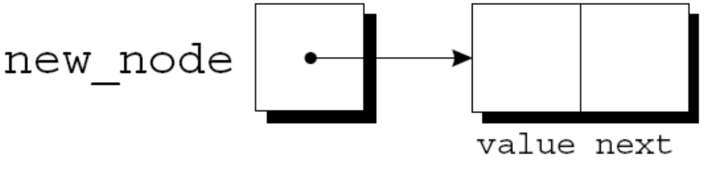
```
struct node {
  int value;            /* data stored in the node  */
  struct node *next;  /* pointer to the next node */
};
struct node A,B;


typedef struct node{
  int value;           /* data stored in the node  */
  struct node *next;  /* pointer to the next node */
} mynode;
mynode A, B;
```

Creating a Node

- When we create a node, we'll need a variable that can point to the node temporarily:
- We'll use `malloc` to allocate memory for the new node:

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

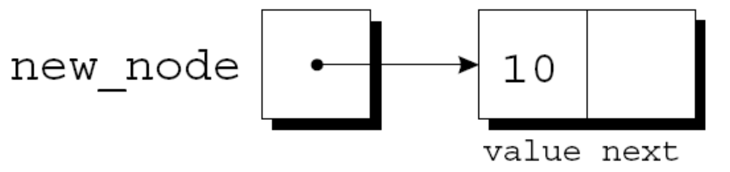
- `new_node` now points to a block of memory just large enough to hold a `node` structure:



Accessing a Node

- store data in the `value` member of the new node:

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



A `scanf` example:

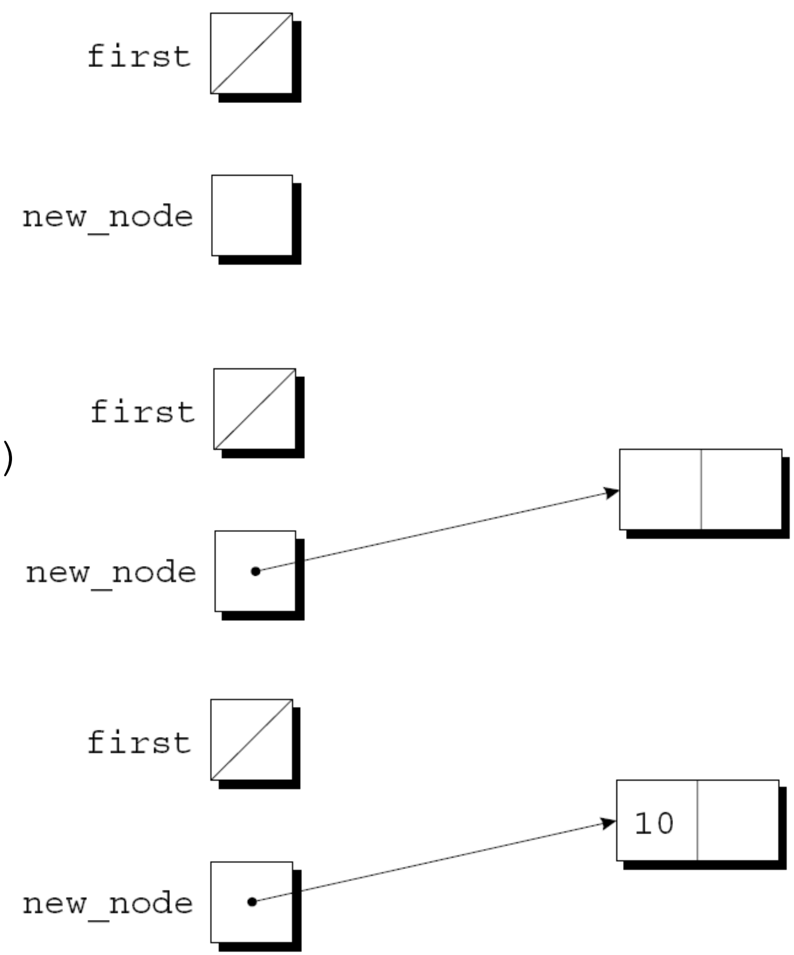
```
scanf("%d", &new_node->value);
```

Inserting a Node at the Beginning of a Linked List

```
first = NULL;
```

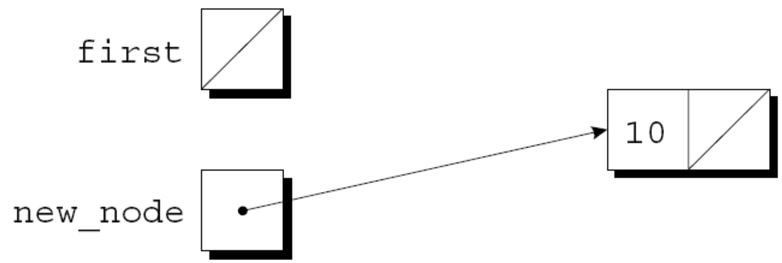
```
new_node =
  malloc(sizeof(struct node))
```

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

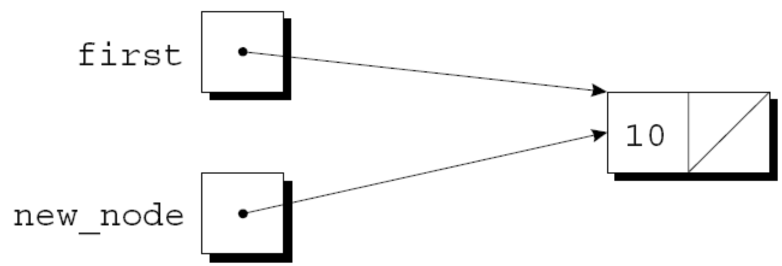


Inserting a Node before the Beginning

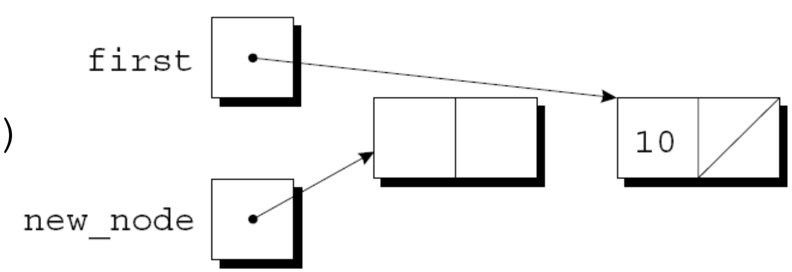
```
new_node->next = first;
```




```
first = new_node;
```

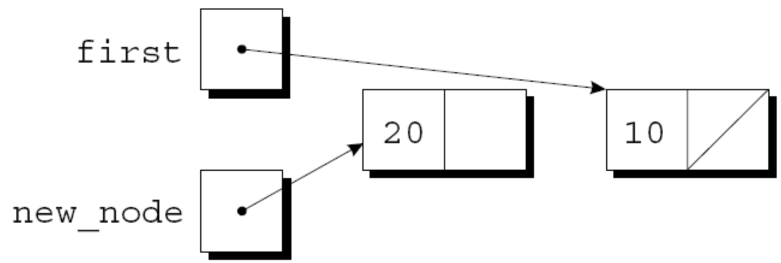



```
new_node =
  malloc(sizeof(struct node))
```

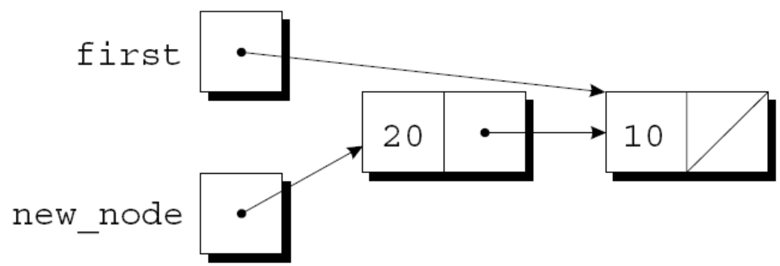


The first pointing a Linked List

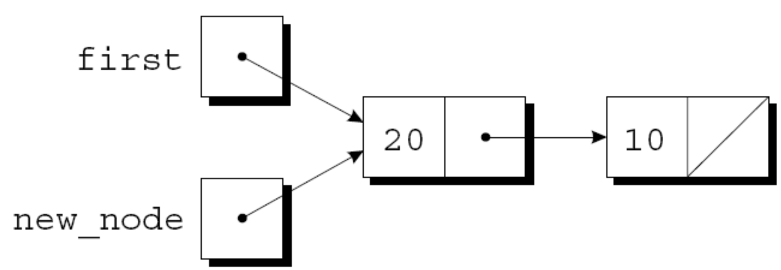
```
new_node->value = 20;
```




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




```
first = new_node;
```



A generic function to handle insertion

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

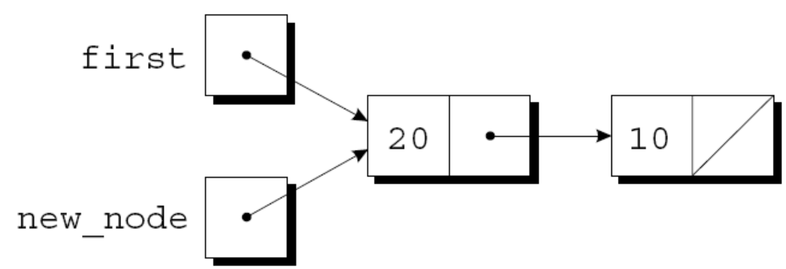
```
struct node *add_to_list(struct node *list, int n)
{
  struct node *new_node;

  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;
}
```

Insert to first by the insertion function

- 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);
```

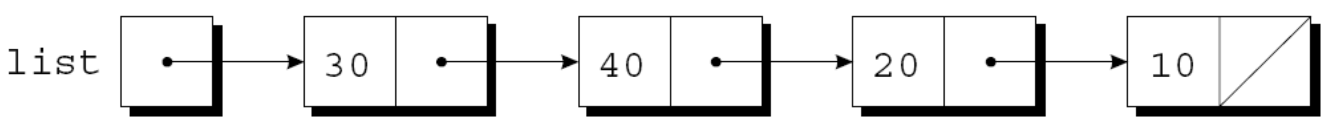
- Getting `add_to_list` to update `first` directly, rather than return a new value for `first`



Traverse the linked list

- The numbers will be in reverse order within the list.

```
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);
    }
  }
```



Searching a Linked List – v1

- Search function finds `n` and returns a pointer to the node containing `n`; If not, return a null pointer.
- Search 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;
}
```

Searching a Linked List – v2

- 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 local copy of the original list pointer, there's no harm to change it.

Searching a Linked List – v3

- 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`.

Searching a Linked List – v4

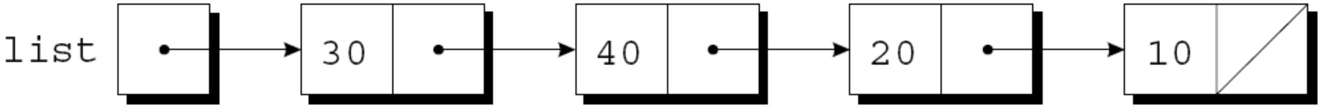
- This version of `search_list` might be a bit clearer if we used a `while` statement:

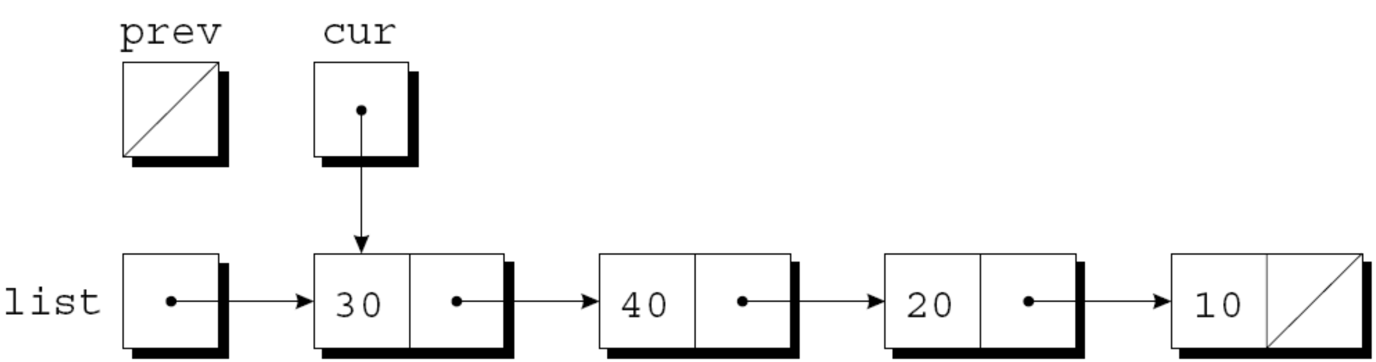
```
struct node *search_list(struct node *list, int n)
{
  while (list != NULL && list->value != n)
    list = list->next;
  return list;
}
```

Deleting a Node from a Linked List

- Two pointers:
  - a "trailing pointer" previous node (`prev`)
  - a pointer to the current node (`cur`)

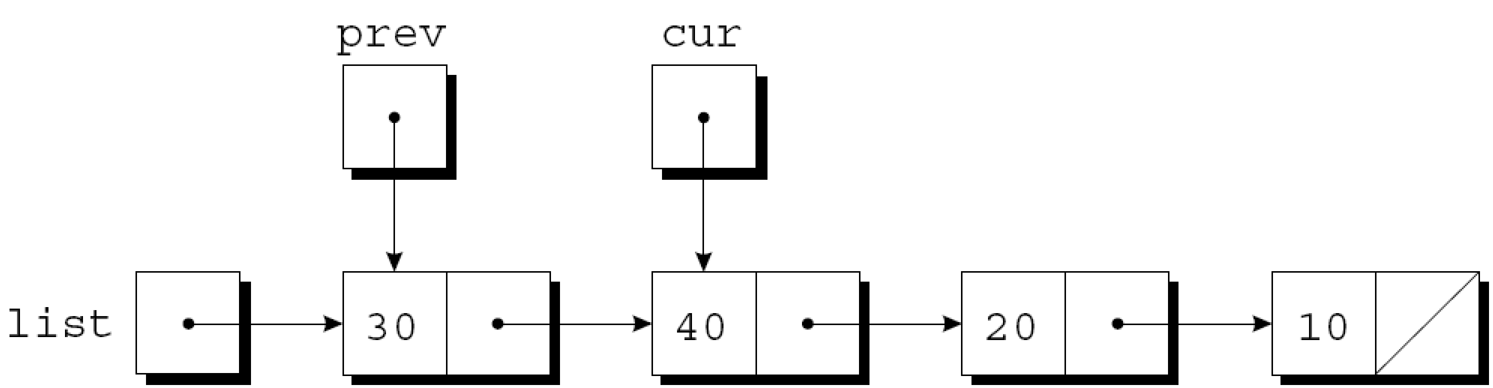
```
for (cur = list, prev = NULL;
     cur != NULL && cur->value != n;
     prev = cur, cur = cur->next)
  ;
```





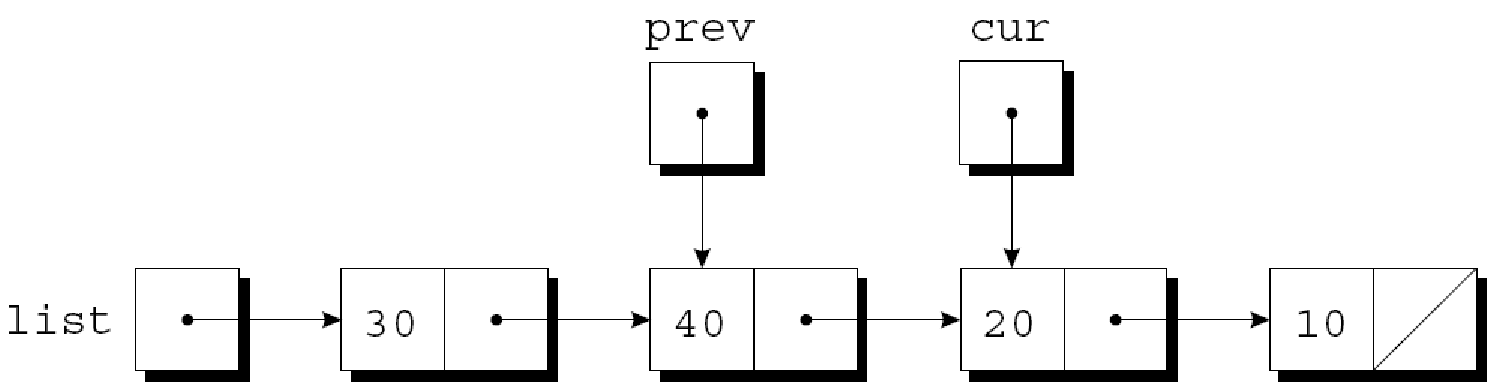
Deleting a Node of 20

- 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:



Deleting a Node of 20

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



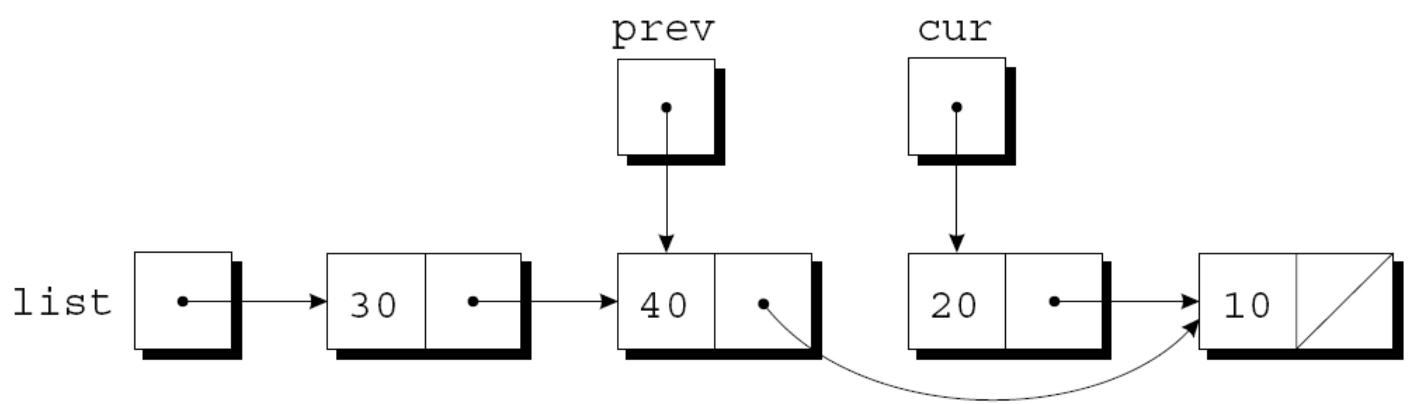
- Since `cur` now points to the node of 20, the condition `cur->value != n` is false and the loop terminates.

Deleting a Node from a Linked List

- The statement

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

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



- Finally, release the memory occupied by the current node:

  ```
  free(cur);
  ```

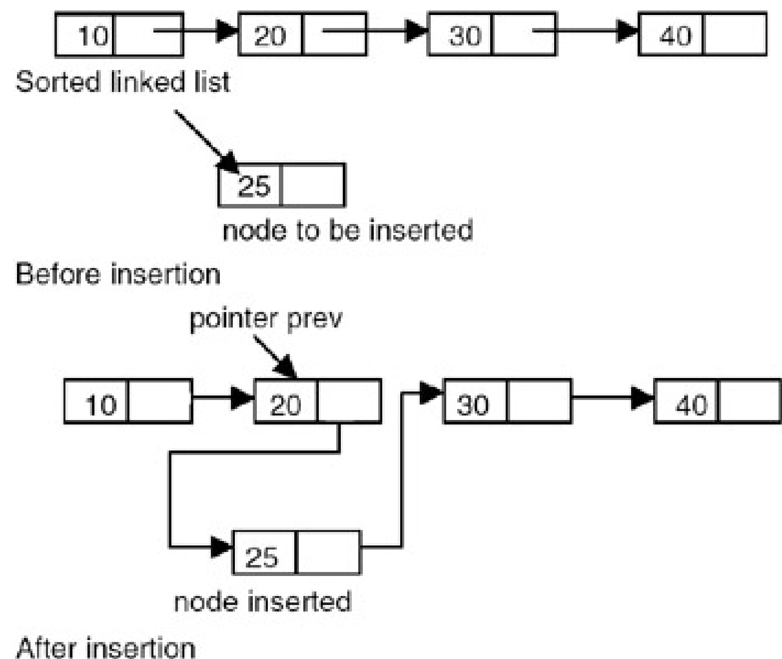
Deleting a Node from a Linked List

```
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;
}
```

Insert new node into a sorted list

- Also use two pointers:

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

   new = add_to_list(list, n);
   for (cur = list, prev = NULL;
        cur != NULL && cur->value < n;
        prev = cur, cur = cur->next)
     ;
 /* n is before the first node */
  if (list == cur) return new;

  new->next = cur;
  prev->next = new;
/* n is after the first one*/

   return list;
}
```



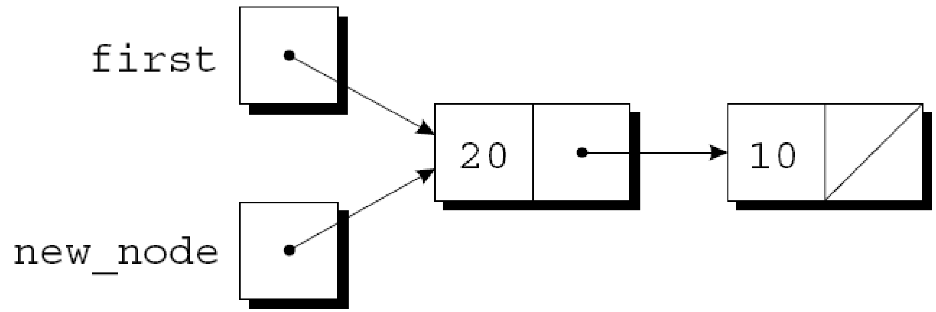
Pointers to Pointers

Function pointers

Problem with original `add_to_list`

- The function is passed a pointer to the first node; it assigns `new_node` to `list` instead of returning `new node`.
- it modifies the first node to the updated list:

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

  new_node = malloc(sizeof(struct node));
  new_node->value = n;
  new_node->next = list;
  list = new_node;
}
```

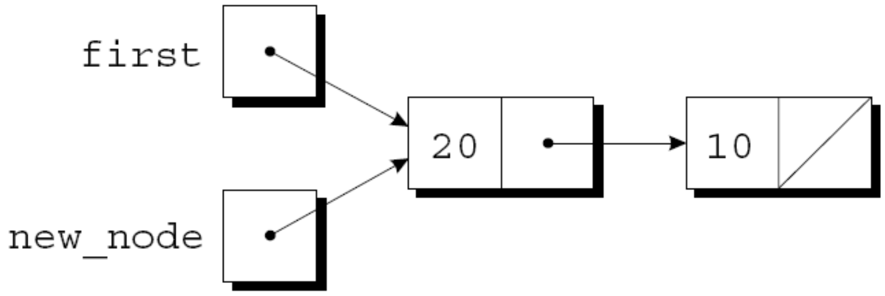


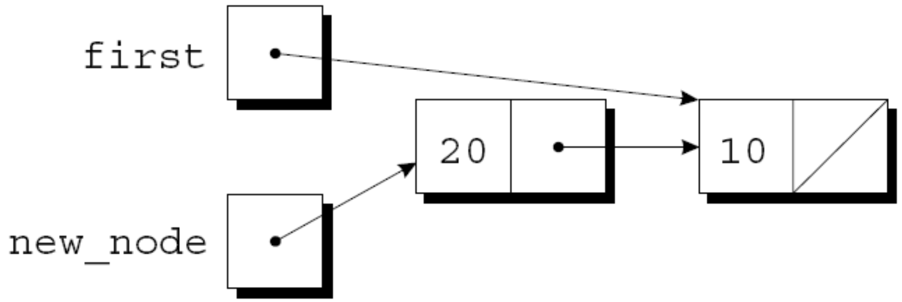
Pointers to Pointers

- Example of caller:

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

- At the point of the call, `first` is copied into `list`.
- If the function changes the value of `list`, making it point to the new node, `first` is not affected.

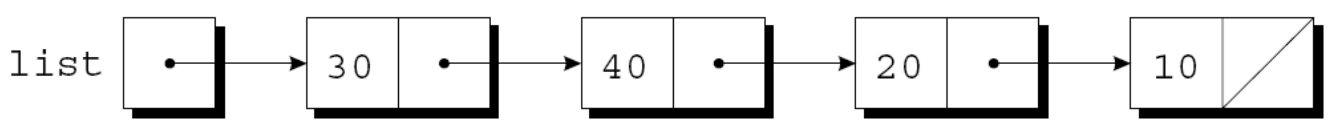




Using Pointers to Pointers for update

- Getting `add_to_list2` to modify `first` requires passing `add_to_list2` a *pointer* to `first`:

```
void add_to_list2(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;
}
```



Pointers to Pointers

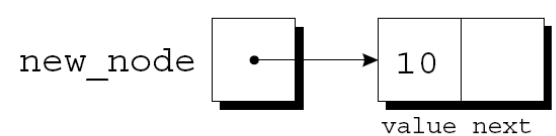
- When the new version of `add_to_list` is called, the first argument will be the address of `first`:

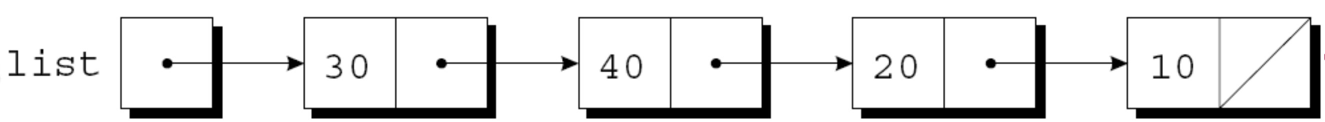
  ```
  add_to_list2(&first, 10);
  ```

- 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`.

Another way: keep first and last pointers

```
struct node *alloc_node(int n)
{
  struct node *new_node;

   new_node = malloc(sizeof(struct node));
   new_node->value = n;
   new_node->next = NULL;
   return new_node;
}
void add_to_first(struct node **first, int n)
{
   struct node *new_node = alloc_node(n);

   new_node->next = *list;
   *list = new_node;
}
void add_to_last(struct node **last, int n)
{
   struct node *new_node =
       alloc_node(n);

   last->next = new_node;
   *last = new_node;
}
```





Function Pointers

- float (*add)();

// this is a legal declaration for the function pointer

- float *add();

// this is an illegal declaration for the function pointer

- A function pointer can also point to another function
  - it holds the address of another function.

```
float add (int a, int b);
  // function declaration
float (*a)(int, int);
  // declaration of a pointer to a function
a=add;
  // assigning address of add() to 'a' pointer
```

Function Pointers as Arguments

- A function `integrate` that integrates a function `f` can be made general by passing `f` as an argument.
- The parentheses around `*f` indicate that `f` is a pointer to a function.

```
double integrate(double (*f)(double),
                 double a, double b);
```

An alternative:

```
double integrate(double f(double),
                 double a, double b);
```

Function Pointers as Arguments

- A call of `integrate` that integrates the `sin` (sine) function from 0 to $\pi/2$:

```
result = integrate(sin, 0.0, PI / 2);
```

- Within the body of `integrate`, we can call the function that `f` points to:

```
y = (*f)(x);
```

- Writing `f(x)` instead of `(*f)(x)` is allowed.

Other Uses of Function Pointers

- A variable that can store a pointer to a function with an `int` parameter and a return type of `void`:

```
void (*pf)(int);
```

- If `f` is such a function, we can make `pf` point to `f` in the following way:

```
pf = f;
```

- We can now call `f` by writing either

```
(*pf)(i);
```

or

```
pf(i);
```

Jump table: Other Uses of Function Pointers

- An array whose elements are function pointers:

```
void (*file_cmd[])(void) = {
     new_cmd,
     open_cmd,
     close_cmd,
     save_cmd,
     print_cmd,
     exit_cmd
    };
```

- A call of the function stored in position `n` of the `file_cmd` array:

```
(*file_cmd[n])();  /* or file_cmd[n](); */
```