Advanced uses of pointers

CS210 - Fall 2023

Vasiliki Kalavri

C Programming: A Modern Approach, Ch. 17.5-17.7

Null Pointers

malloc 실패시 Null

- 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

- 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

- 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) …
  ```

— Hea

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.

Strig

malloc

Using **malloc** to Allocate Memory for a String

char *p

n+1

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

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

'\0'

n

  `p` is a `char *` variable.

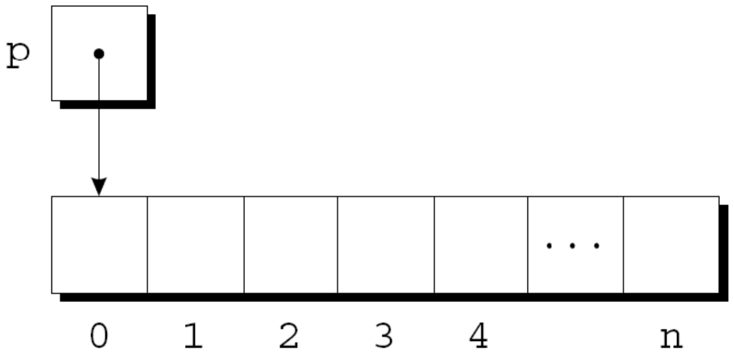
- Each character requires one byte of memory; adding 1 to `n` leaves room for the null character.

- 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 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

```
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;
}
```

String \n

null

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.

Using Dynamic Storage Allocation in String Functions

- 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.

Dynamically Allocated Arrays

- 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.

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;
```

int a [100]

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

```
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

- 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;
```

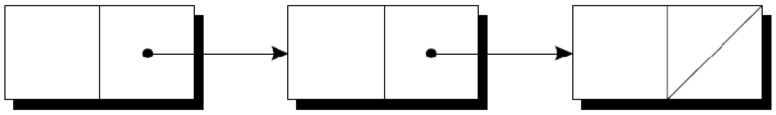
- We also have the option of using pointer arithmetic instead of subscripting to access the elements of the array.

Revisiting linked lists

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.

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:

int or other part of data.

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

Declaring a Node Type

- 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.

stuct node

Creating a Node

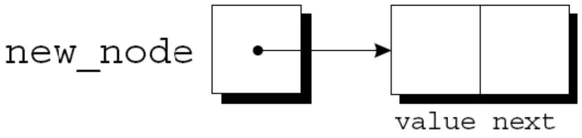
- When we create a node, we'll need a variable that can point to the node temporarily:

```
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:

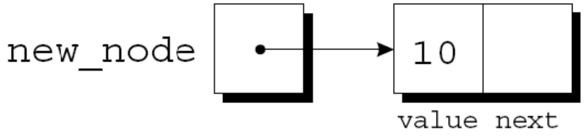


Creating a Node

- 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.

Inserting a Node at the Beginning of a Linked List

- 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.

Inserting a Node at the Beginning of a 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;
```

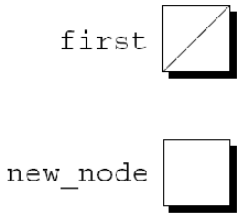
- These statements work even if the list is empty.

Inserting a Node at the Beginning of a Linked List

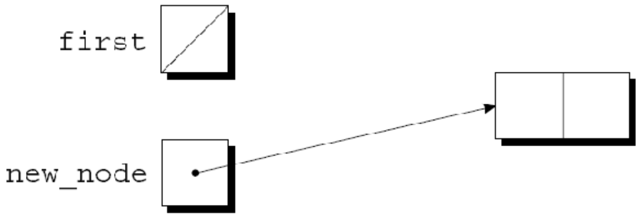
- Let’s trace the process of inserting two nodes into an empty list.
- We’ll insert a node containing the number 10 first, followed by a node containing 20.

Inserting a Node at the Beginning of a Linked List

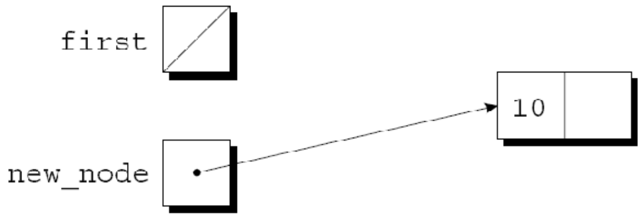
```
first = NULL;
```




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

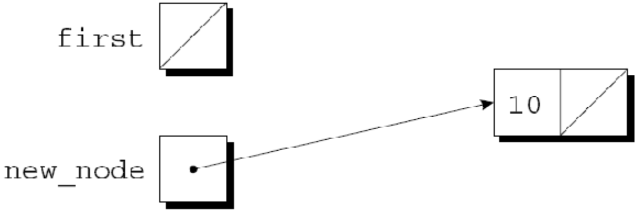



```
new_node->value = 10;
```

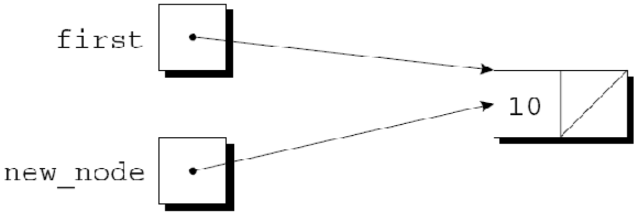


Inserting a Node at the Beginning of a Linked List

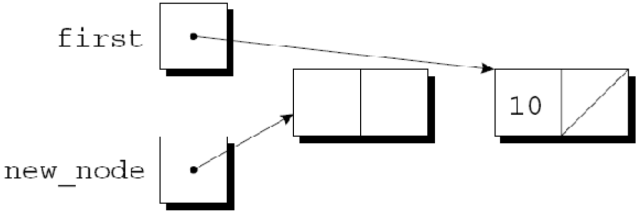
```
new_node->next = first;
```




```
first = new_node;
```

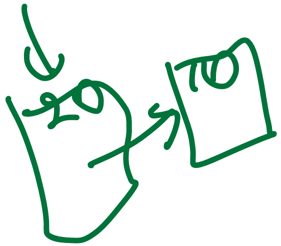


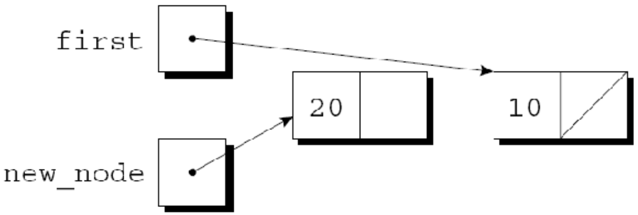

```
new_node =
  malloc(sizeof(struct node));
```



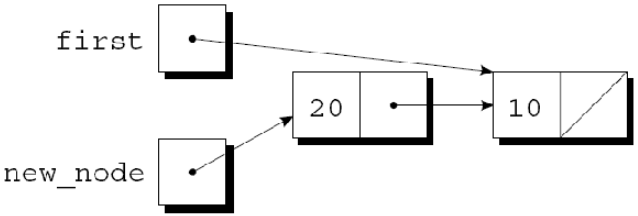
Inserting a Node at the Beginning of a Linked List

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

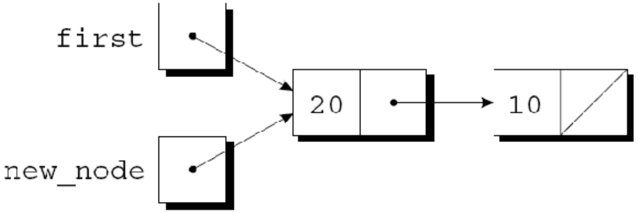





```
new_node->next = first;
```




```
first = new_node;
```



Inserting a Node at the Beginning of a Linked List

- 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;
}
```

Pointers to Pointers

- 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:

```
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;
}
```

Pointers to Pointers

- Modifying `add_to_list` so that it assigns `new_node` to `list` instead of returning `new_node` doesn't work.

- Example:

```
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.

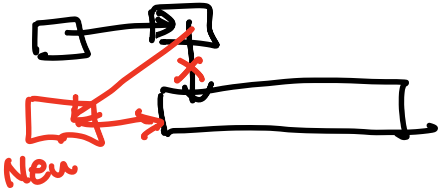
Pointers to Pointers

- 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;
}
```

pointer to a pointer.



Pointers to Pointers

- 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`.

Searching a Linked List

- 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`.

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:

return a pointer

```
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

- There are many other ways to write `search_list`.
- One alternative is to eliminate the `p` variable, instead using `list` itself to keep track of the current node:

get rid of p value.

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

copy of the original pointer

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

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;
}
```

modifying the copy of it.

Don't make the garbage.

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.

- There are various solutions to this problem.

Deleting a Node from a Linked List

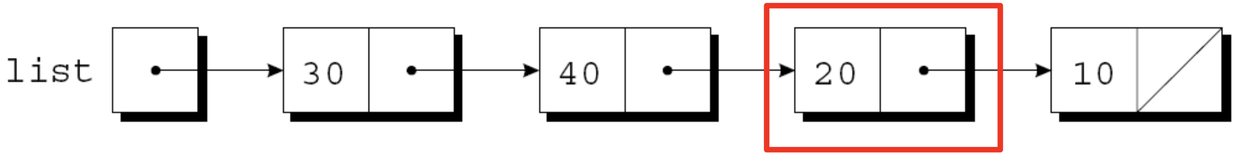
- The "trailing pointer" technique involves keeping a pointer to the previous node (`prev`) as well as a pointer to the current node (`cur`).
- Assume that `list` points to the list to be searched and `n` is the integer to be deleted.
- A loop that implements step 1:

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

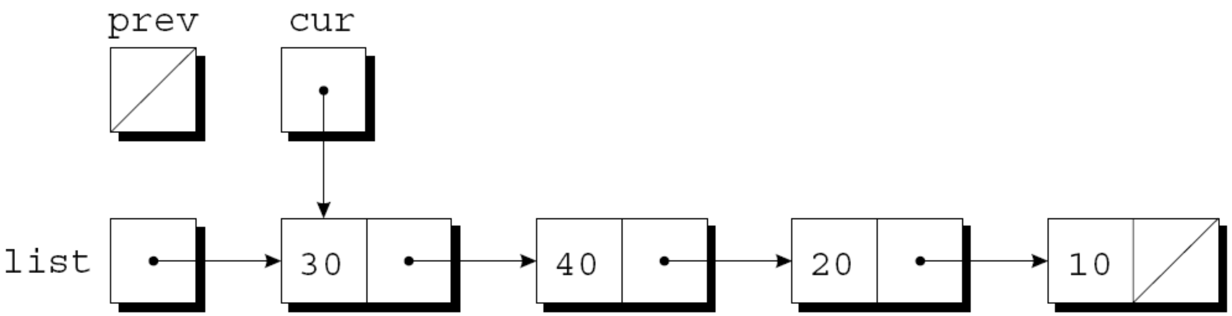
- When the loop terminates, `cur` points to the node to be deleted and `prev` points to the previous node.

Deleting a Node from a Linked List

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



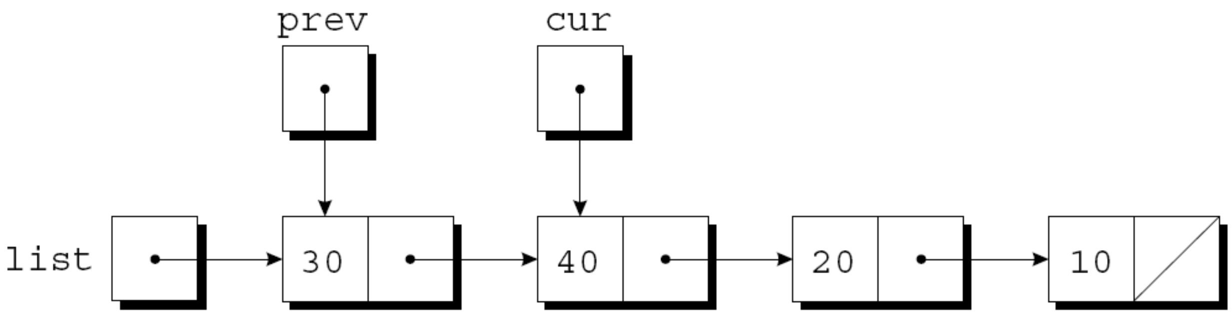
- After `cur = list, prev = NULL` has been executed:



Deleting a Node from a Linked List

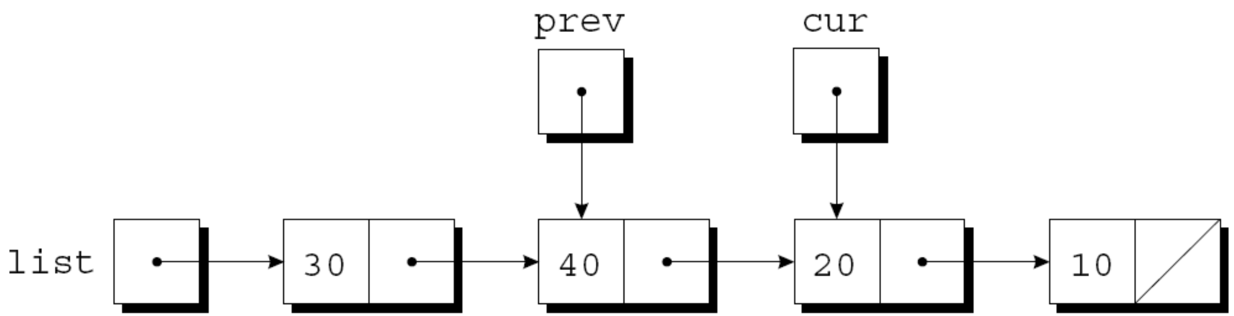
- The test `cur != NULL && cur->value != n` is true, since `cur` is pointing to a node and the node doesn't contain 20.

- After `prev = cur, cur = cur->next` has been executed:



Deleting a Node from a 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.

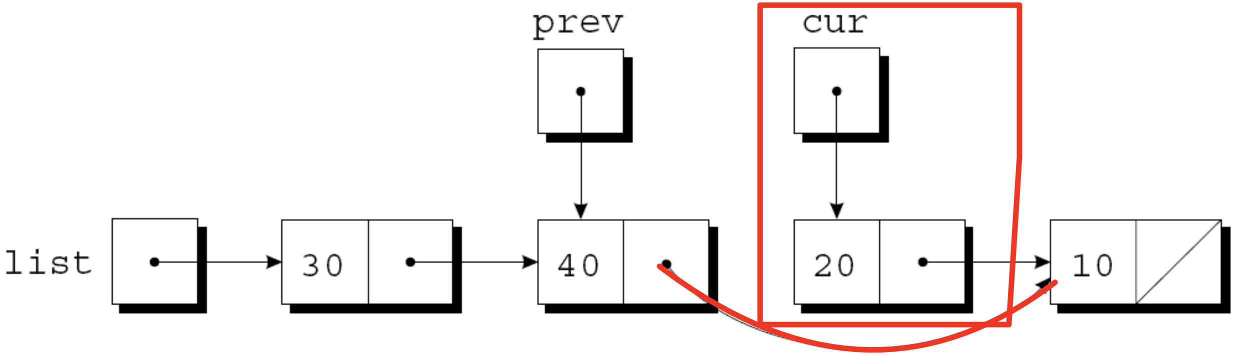
Deleting a Node from a Linked List

- Next, we'll perform the bypass required by step 2.

- The statement

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

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



Deleting a Node from a Linked List

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

```
free(cur);
```

don't forget.

Deleting a Node from a Linked List

- 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

```
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;
}
```

Pointers to Functions

- C doesn't require that pointers point only to *data;* it's also possible to have pointers to *functions.*

- Functions occupy memory locations, so every function has an address.

- We can use function pointers in much the same way we use pointers to data.

- Passing a function pointer as an argument is fairly common.

The `qsort` Function

sorting function.

- Some of the most useful functions in the C library require a function pointer as an argument.
- One of these is `qsort`, which belongs to the `<stdlib.h>` header.
- `qsort` is a general-purpose sorting function that's capable of sorting any array.

The `qsort` Function

- `qsort` must be told how to determine which of two array elements is “smaller.”

- This is done by passing `qsort` a pointer to a ***comparison function.***

- When given two pointers `p` and `q` to array elements, the comparison function must return an integer that is:
  - *Negative* if `*p` is “less than” `*q`
  - *Zero* if `*p` is “equal to” `*q`
  - *Positive* if `*p` is “greater than” `*q`

The `qsort` Function

- Prototype for `qsort`:

```
void qsort(void *base, size_t nmemb, size_t size,
  int (*compar)(const void *, const void *));
```

- `base` must point to the first element in the array (or the first element in the portion to be sorted).

- `nmemb` is the number of elements to be sorted.

- `size` is the size of each array element, measured in bytes.

- `compar` is a pointer to the comparison function.

The `qsort` Function

- When `qsort` is called, it sorts the array into ascending order, calling the comparison function whenever it needs to compare array elements.

- A call of `qsort` that sorts the `inventory` array of Chapter 16:

```
qsort(inventory, num_parts,
      sizeof(struct part), compare_parts);
```

- `compare_parts` is a function that compares two `part` structures.

The `qsort` Function

- Writing the `compare_parts` function is tricky.
- `qsort` requires that its parameters have type `void *`, but we can't access the members of a `part` structure through a `void *` pointer.
- To solve the problem, `compare_parts` will assign its parameters, `p` and `q`, to variables of type `struct part *`.

The `qsort` Function

- A version of `compare_parts` that can be used to sort the `inventory` array into ascending order by part number:

```
int compare_parts(const void *p, const void *q)
{
  const struct part *p1 = p;
  const struct part *q1 = q;

  if (p1->number < q1->number)
    return -1;
  else if (p1->number == q1->number)
    return 0;
  else
    return 1;
}
```

-1 0 1

2 different version

The `qsort` Function

- `compare_parts` can be made even shorter by removing the `if` statements:

```
int compare_parts(const void *p, const void *q)
{
  return ((struct part *) p)->number -
         ((struct part *) q)->number;
}
```

The `qsort` Function

- A version of `compare_parts` that can be used to sort the `inventory` array by part name instead of part number:

```
int compare_parts(const void *p, const void *q)
{             string compare
  return strcmp(((struct part *) p)->name,
                ((struct part *) q)->name);
}
```

Other Uses of Function Pointers

- Although function pointers are often used as arguments, that’s not all they’re good for.
- C treats pointers to functions just like pointers to data.
- They can be stored in variables or used as elements of an array or as members of a structure or union.
- It’s even possible for functions to return function pointers.

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);
```

Other Uses of Function Pointers

- An array whose elements are function pointers:

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

Other Uses of Function Pointers

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

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

or

- We could get a similar effect with a `switch` statement, but using an array of function pointers provides more flexibility.

A C compiler

A. translate C source code into machine-specific code

B. ensures that your program is free of bugs

C. Compiles a list of all the ways your code can be improved

D. all of the above

E none of the above

which of the following is true regarding the '*' symbol in C:

A. '*' is used in the declaration of a pointer

B. '*' is used as the indirection operator

C. '*' is used as the multiplication operator

D all of the above

E none of the above

Given this code:

```
long long x=-1;
unsigned long long y=x;
```

The underlying byte value of y will be the same as that of x

byte

True

Considering a 4-bit 2's complement representation, the result of 1111+1111 in decimal is

-2?

1|1110   1110   -2

842

-8+4+2 6

In this declaration:

```
int **p;
```

A: p is a function pointer

B: p is a pointer to a pointer to an int

C: p is a pointer to an int

D: p is a function that returns a pointer to an int