Chapter 17: Advanced Uses of Pointers

Chapter 17

Advanced Uses of Pointers
Chapter 17: Advanced Uses of Pointers

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 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

• 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.
Chapter 17: Advanced Uses of Pointers

Null Pointers

• An example of testing malloc’s return value:
  
  ```c
  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:
  
  ```c
  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) ...
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:
  ```c
  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

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

  ```c
  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:

  ```c
  p = (char *) malloc(n + 1);
  ```
Chapter 17: Advanced Uses of Pointers

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:
  
  ```c
  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.
Chapter 17: Advanced Uses of Pointers

Using Dynamic Storage Allocation in String Functions

```c
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:**
  
  ```c
  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.
Chapter 17: Advanced Uses of Pointers

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

• The `remind2.c` program is based on the `remind.c` program of Chapter 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.
Chapter 17: Advanced Uses of Pointers

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

• 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**).
remind2.c

/* Prints a one-month reminder list (dynamic string version) */

#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;
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];
Chapter 17: Advanced Uses of Pointers

```c
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++;
```

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

```c
return 0;
```
int read_line(char str[], int n)
{
    int ch, i = 0;

    while ((ch = getchar()) != '\n')
        if (i < n)
            str[i++] = ch;
    str[i] = '\0';
    return i;
}
Chapter 17: Advanced Uses of Pointers

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.

• 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.
Suppose a program needs an array of \( n \) integers, where \( n \) is computed during program execution.

We’ll first declare a pointer variable:

\[
\text{int } *a;
\]

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

\[
a = \text{malloc}(n * \text{sizeof(int)});
\]

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:
  ```c
  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.
Chapter 17: Advanced Uses of Pointers

The calloc Function

• The calloc function is an alternative to malloc.

• Prototype for calloc:

  ```c
  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.
Chapter 17: Advanced Uses of Pointers

The calloc Function

- A call of `calloc` that allocates space for an array of `n` integers:
  ```c
  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:
  ```c
  struct point { int x, y; } *p;
  p = calloc(1, sizeof(struct point));
  ```
The \texttt{realloc} Function

• The \texttt{realloc} function can resize a dynamically allocated array.

• Prototype for \texttt{realloc}:
  
  \begin{verbatim}
  void *realloc(void *ptr, size_t size);
  \end{verbatim}

• \texttt{ptr} must point to a memory block obtained by a previous call of \texttt{malloc}, \texttt{calloc}, or \texttt{realloc}.

• \texttt{size} represents the new size of the block, which may be larger or smaller than the original size.
Chapter 17: Advanced Uses of Pointers

The `realloc` 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.
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

- `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

• Example:
  
  \[
  p = \text{malloc}(\ldots);
  
  q = \text{malloc}(\ldots);
  
  p = q;
  \]

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

\[
\begin{array}{c}
\text{p} \\
\text{q}
\end{array}
\]
Deallocating Storage

• After \( q \) is assigned to \( p \), both variables now point to the second memory block:

\[ \begin{array}{c}
\text{p} \\
\bullet \\
\text{q} \\
\bullet
\end{array} \]

• There are no pointers to the first block, so we’ll never be able to use it again.
Deallocating Storage

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

- Prototype for `free`:
  ```c
  void free(void *ptr);
  ```
- `free` will be passed a pointer to an unneeded memory block:
  ```c
  p = malloc(...);
  q = malloc(...);
  free(p);
  free(p);
  p = q;
  ```
- Calling `free` releases the block of memory that `p` points to.
Chapter 17: Advanced Uses of Pointers

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:

  ```c
  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

• 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.
Chapter 17: Advanced Uses of Pointers

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:

![Diagram of a linked list]

- The last node in the list contains a null pointer.
Chapter 17: Advanced Uses of Pointers

Linked Lists

• 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 */
};
```

• node must be a tag, not a typedef name, or there would be no way to declare the type of next.
Declaring a Node Type

• Next, we’ll need a variable that always points to the first node in the list:

```c
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

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

```c
struct node *new_node;
```

• We’ll use `malloc` to allocate memory for the new node, saving the return value in `new_node`:

```c
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:

\[(\ast \text{new\_node}).\text{value} = 10;\]

• The resulting picture:

```
new_node -> 10
```

• The parentheses around \(\ast \text{new\_node}\) are mandatory because the \. operator would otherwise take precedence over the \* operator.
Chapter 17: Advanced Uses of Pointers

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;
Chapter 17: Advanced Uses of Pointers

The \texttt{\&} \texttt{\textarrow{}} Operator

- The \texttt{\textarrow{}} operator produces an lvalue, so we can use it wherever an ordinary variable would be allowed.

- A \texttt{scanf} example:
  \begin{verbatim}
  scanf("\%d", \&new_node\textarrow{value});
  \end{verbatim}

- The \texttt{\&} operator is still required, even though \texttt{new\_node} is a pointer.
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:
  
  ```c
  new_node->next = first;
  ```

- The second step is to make `first` point to the new node:
  
  ```c
  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.
Chapter 17: Advanced Uses of Pointers

Inserting a Node at the Beginning of a Linked List

```c
first = NULL;

new_node = malloc(sizeof(struct node));
new_node->value = 10;
```
Chapter 17: Advanced Uses of Pointers

Inserting a Node at the Beginning of a Linked List

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

```c
first = new_node;
```

```c
new_node = malloc(sizeof(struct node));
```
Chapter 17: Advanced Uses of Pointers

Inserting a Node at the Beginning of a Linked List

```c
new_node->value = 20;

new_node->next = first;

first = new_node;
```
Chapter 17: Advanced Uses of Pointers

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:

```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;
}
```
Chapter 17: Advanced Uses of Pointers

Inserting a Node at the Beginning of a Linked List

• 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`, turns out to be tricky.
Chapter 17: Advanced Uses of Pointers

Inserting a Node at the Beginning of a Linked List

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

```c
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

- 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:
  
  ```c
  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 \( n \).
Chapter 17: Advanced Uses of Pointers

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:

```c
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:

```c
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.
Searching a Linked List

• Another alternative:

```c
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.
Chapter 17: Advanced Uses of Pointers

Searching a Linked List

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

```c
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

- 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 ($\text{prev}$) as well as a pointer to the current node ($\text{cur}$).
- Assume that $\text{list}$ points to the list to be searched and $n$ is the integer to be deleted.
- A loop that implements step 1:

  ```
  for (\text{cur} = \text{list}, \text{prev} = \text{NULL};
       \text{cur} \neq \text{NULL} \&\& \text{cur}->\text{value} \neq n;
       \text{prev} = \text{cur}, \text{cur} = \text{cur}->\text{next})
  ;
  ```

- When the loop terminates, $\text{cur}$ points to the node to be deleted and $\text{prev}$ points to the previous node.
Deleting a Node from a Linked List

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

```plaintext
list → 30 → 40 → 20 → 10
```

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

```plaintext
prev → cur

list → 30 → 40 → 20 → 10
```
Deleting a Node from a Linked List

- The test \( \text{cur} \neq \text{NULL} && \text{cur} -> \text{value} \neq n \) is true, since \( \text{cur} \) is pointing to a node and the node doesn’t contain 20.
- After \( \text{prev} = \text{cur}, \text{cur} = \text{cur} -> \text{next} \) has been executed:

```
   prev
  /   \
```

```
   cur
  /   \
```

```
list  \rightarrow  30  \rightarrow  40  \rightarrow  20  \rightarrow  10
```
Chapter 17: Advanced Uses of Pointers

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
  \[
  \text{prev->next = cur->next;}
  \]
  makes the pointer in the previous node point to the node after the current node:

![Diagram of a linked list with pointers](image-url)
Deleting a Node from a Linked List

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

  free(cur);
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

```c
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;
}
```
Chapter 17: Advanced Uses of Pointers

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.
Chapter 17: Advanced Uses of Pointers

Program: Maintaining a Parts Database (Revisited)

• The inventory2.c program is a modification of the parts database program of Chapter 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)

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

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

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

```c
struct part *inventory = NULL;
```
Program: Maintaining a Parts Database (Revisited)

• Most of the functions in the new program will closely resemble their counterparts in the original program.

• find_part and insert will be more complex, however, since we’ll keep the nodes in the inventory list sorted by part number.
Program: Maintaining a Parts Database (Revisited)

- 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.
Chapter 17: Advanced Uses of Pointers

Program: Maintaining a Parts Database (Revisited)

• 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:

```c
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:

```c
if (p != NULL && number == p->number)
    return p;
```
Chapter 17: Advanced Uses of Pointers

Program: Maintaining a Parts Database (Revisited)

• 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:

```c
for (cur = inventory, prev = NULL;
    cur != NULL && new_node->number > cur->number;
    prev = cur, cur = cur->next)
;;
```
Program: Maintaining a Parts Database (Revisited)

• 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.

• Like the original program, this version requires the `read_line` function of Chapter 16.
Chapter 17: Advanced Uses of Pointers

inventory2.c

/* Maintains a parts database (linked list version) */

#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);
Chapter 17: Advanced Uses of Pointers

/**********************************************************
* main: Prompts the user to enter an operation code, *
* then calls a function to perform the requested *
* action. Repeats until the user enters the *
* command 'q'. Prints an error message if the user *
* enters an illegal code.
*
***********************************************************/

int main(void)
{
    char code;

    for (;;) {
        printf("Enter operation code: ");
        scanf(" %c", &code);
        while (getchar() != 'n')   /* skips to end of line */
            ;
    
}
Chapter 17: Advanced Uses of Pointers

```c
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");
}```
Chapter 17: Advanced Uses of Pointers

/**************************************************************************
 * find_part: Looks up a part number in the inventory list. Returns a pointer to the node
 * containing the part number; if the part number is not found, returns NULL.
 */

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;
}
Chapter 17: Advanced Uses of Pointers

/****************************************************************************
 * insert: Prompts the user for information about a new part and then inserts
 * the part into the inventory list; the list remains sorted by part number.
 * Prints an error message and returns prematurely if the part already exists
 * or space could not be allocated for the part. *
****************************************************************************/

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 = 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;
}
**Chapter 17: Advanced Uses of Pointers**

/********************************************************************************
 * search: Prompts the user to enter a part number, then looks up the part in the database. If the part exists, prints the name and quantity on hand; if not, prints an error message.
 *********************************************************************************/

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");
}
Chapter 17: Advanced Uses of Pointers

/*******************************************************************************
 * update: Prompts the user to enter a part number. *
 * Prints an error message if the part doesn't exist; otherwise, prompts the user to enter change in quantity on hand and updates the database.

/*******************************************************************************
 */

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");
}
Chapter 17: Advanced Uses of Pointers

/**************************************************************************
 * print: Prints a listing of all parts in the database, showing the part *
 *        number, part name, and quantity on hand. Part numbers will appear in *
 *        ascending order.
 */

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

• Chapter 13 introduced the idea of a pointer to a 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.
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:

```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;
}
```
Pointers to Pointers

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

- Example:
  ```c
  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:**

```c
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;
}
```
Pointers to Pointers

• When the new version of `add_to_list` is called, the first argument will be the address of `first`:
  ```c
  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`.
Chapter 17: Advanced Uses of Pointers

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.
Chapter 17: Advanced Uses of Pointers

Function Pointers as Arguments

• A function named `integrate` that integrates a mathematical function \( f \) can be made as general as possible by passing \( f \) as an argument.

• **Prototype for `integrate`:**
  ```c
  double integrate(double (*f)(double), double a, double b);
  ```

  The parentheses around \( *f \) indicate that \( f \) is a pointer to a function.

• **An alternative prototype:**
  ```c
  double integrate(double f(double), double a, double b);
  ```
Chapter 17: Advanced Uses of Pointers

**Function Pointers as Arguments**

- A call of `integrate` that integrates the `sin` (sine) function from 0 to \( \pi / 2 \):
  
  ```c
  result = integrate(sin, 0.0, PI / 2);
  ```

- When a function name isn’t followed by parentheses, the C compiler produces a pointer to the function.

- Within the body of `integrate`, we can call the function that \( f \) points to:
  
  ```c
  y = (*f)(x);
  ```

- Writing \( f(x) \) instead of \( (*f)(x) \) is allowed.
The `qsort` 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.
Chapter 17: Advanced Uses of Pointers

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`
Chapter 17: Advanced Uses of Pointers

The `qsort` Function

- Prototype for `qsort`:
  ```c
  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.
Chapter 17: Advanced Uses of Pointers

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:

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

- **compare_parts** is a function that compares two **part** structures.
The \texttt{qsort} Function

- Writing the \texttt{compare_parts} function is tricky.
- \texttt{qsort} requires that its parameters have type \texttt{void *}, but we can’t access the members of a \texttt{part} structure through a \texttt{void *} pointer.
- To solve the problem, \texttt{compare_parts} will assign its parameters, \texttt{p} and \texttt{q}, to variables of type \texttt{struct part *}. 
Chapter 17: Advanced Uses of Pointers

The `qsort` Function

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

```c
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;
}
```
Chapter 17: Advanced Uses of Pointers

The `qsort` Function

- Most C programmers would write the function more concisely:

  ```c
  int compare_parts(const void *p, const void *q)
  {
    if (((struct part *) p)->number <
        ((struct part *) q)->number)
      return -1;
    else if (((struct part *) p)->number ==
              ((struct part *) q)->number)
      return 0;
    else
      return 1;
  }
  ```
The `qsort` Function

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

```c
int compare_parts(const void *p, const void *q) {
    return ((struct part *) p)->number -
          ((struct part *) q)->number;
}
```
Chapter 17: Advanced Uses of Pointers

The `qsort` Function

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

```c
int compare_parts(const void *p, const void *q)
{
    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:
  
  \[
  \text{void (}*\text{pf})(\text{int});
  \]

• If \( f \) is such a function, we can make \( pf \) point to \( f \) in the following way:
  
  \[
  \text{pf} = f;
  \]

• We can now call \( f \) by writing either
  
  \[
  (\text{*pf})(i);
  \]
  or
  
  \[
  \text{pf}(i);
  \]
Chapter 17: Advanced Uses of Pointers

Other Uses of Function Pointers

• An array whose elements are function pointers:

```c
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
};
```
Chapter 17: Advanced Uses of Pointers

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](); */

• We could get a similar effect with a switch statement, but using an array of function pointers provides more flexibility.
Program: Tabulating the Trigonometric Functions

• The `tabulate.c` program prints tables showing the values of the \( \cos, \sin, \) and \( \tan \) functions.
• The program is built around a function named `tabulate` that, when passed a function pointer `f`, prints a table showing the values of `f`.
• `tabulate` uses the `ceil` function.
• When given an argument `x` of `double` type, `ceil` returns the smallest integer that’s greater than or equal to `x`. 
Chapter 17: Advanced Uses of Pointers

Program: Tabulating the Trigonometric Functions

• A session with tabulate.c:
  Enter initial value: 0
  Enter final value: .5
  Enter increment: .1

<table>
<thead>
<tr>
<th>x</th>
<th>cos(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00000</td>
<td>1.00000</td>
</tr>
<tr>
<td>0.10000</td>
<td>0.99500</td>
</tr>
<tr>
<td>0.20000</td>
<td>0.98007</td>
</tr>
<tr>
<td>0.30000</td>
<td>0.95534</td>
</tr>
<tr>
<td>0.40000</td>
<td>0.92106</td>
</tr>
<tr>
<td>0.50000</td>
<td>0.87758</td>
</tr>
</tbody>
</table>
# Chapter 17: Advanced Uses of Pointers

## Program: Tabulating the Trigonometric Functions

<table>
<thead>
<tr>
<th>$x$</th>
<th>$\sin(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>0.10000</td>
<td>0.09993</td>
</tr>
<tr>
<td>0.20000</td>
<td>0.19867</td>
</tr>
<tr>
<td>0.30000</td>
<td>0.29552</td>
</tr>
<tr>
<td>0.40000</td>
<td>0.38942</td>
</tr>
<tr>
<td>0.50000</td>
<td>0.47943</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$x$</th>
<th>$\tan(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>0.10000</td>
<td>0.10033</td>
</tr>
<tr>
<td>0.20000</td>
<td>0.20271</td>
</tr>
<tr>
<td>0.30000</td>
<td>0.30934</td>
</tr>
<tr>
<td>0.40000</td>
<td>0.42279</td>
</tr>
<tr>
<td>0.50000</td>
<td>0.54630</td>
</tr>
</tbody>
</table>
Chapter 17: Advanced Uses of Pointers

**tabulate.c**

/* Tabulates values of trigonometric functions */

#include <math.h>
#include <stdio.h>

void tabulate(double (*f)(double), double first, double last, double incr);

int main(void)
{
    double final, increment, initial;

    printf("Enter initial value: ");
    scanf("%lf", &initial);

    printf("Enter final value: ");
    scanf("%lf", &final);

    printf("Enter increment: ");
    scanf("%lf", &increment);

    return 0;
}
Chapter 17: Advanced Uses of Pointers

```c
printf("\n    x        cos(x)"
    "\n    -------    -------\n")
tabulate(cos, initial, final, increment);

printf("\n    x        sin(x)"
    "\n    -------    -------\n")
tabulate(sin, initial, final, increment);

printf("\n    x        tan(x)"
    "\n    -------    -------\n")
tabulate(tan, initial, final, increment);

return 0;
}

void tabulate(double (*f)(double), double first, double last, double incr)
{
    double x;
    int i, num_intervals;

    num_intervals = ceil((last - first) / incr);
    for (i = 0; i <= num_intervals; i++) {
        x = first + i * incr;
        printf("%10.5f %10.5f\n", x, (*f)(x));
    }
}
```
Restricted Pointers (C99)

• In C99, the keyword `restrict` may appear in the declaration of a pointer:
  ```c
  int * restrict p;
  ```
  `p` is said to be a `restricted pointer`.

• The intent is that if `p` points to an object that is later modified, then that object is not accessed in any way other than through `p`.

• Having more than one way to access an object is often called `aliasing`.
Restricted Pointers (C99)

• Consider the following code:
  ```c
  int * restrict p;
  int * restrict q;
  p = malloc(sizeof(int));
  ```

• Normally it would be legal to copy `p` into `q` and then modify the integer through `q`:
  ```c
  q = p;
  *q = 0; /* causes undefined behavior */
  ```

• Because `p` is a restricted pointer, the effect of executing the statement `*q = 0;` is undefined.
Restricted Pointers (C99)

• To illustrate the use of restrict, consider the \texttt{memcpy} and \texttt{memmove} functions.

• The C99 prototype for \texttt{memcpy}, which copies bytes from one object (pointed to by \texttt{s2}) to another (pointed to by \texttt{s1}):

  \begin{verbatim}
  void *memcpy(void * restrict s1, const void * restrict s2, size_t n);
  \end{verbatim}

• The use of \texttt{restrict} with both \texttt{s1} and \texttt{s2} indicates that the objects to which they point shouldn’t overlap.
Chapter 17: Advanced Uses of Pointers

Restricted Pointers (C99)

• In contrast, restrict doesn’t appear in the prototype for memmove:
  
  ```c
  void *memmove(void *s1, const void *s2, size_t n);
  ```

• memmove is similar to memcpy, but is guaranteed to work even if the source and destination overlap.

• Example of using memmove to shift the elements of an array:
  
  ```c
  int a[100];
  ...
  memmove(&a[0], &a[1], 99 * sizeof(int));
  ```
Restricted Pointers (C99)

• Prior to C99, there was no way to document the difference between \texttt{memcpy} and \texttt{memmove}.

• The prototypes for the two functions were nearly identical:

  \begin{verbatim}
  void *memcpy(void *s1, const void *s2, size_t n);
  void *memmove(void *s1, const void *s2, size_t n);
  \end{verbatim}

• The use of \texttt{restrict} in the C99 version of \texttt{memcpy}'s prototype is a warning that the \texttt{s1} and \texttt{s2} objects should not overlap.
Restricted Pointers (C99)

- `restrict` provides information to the compiler that may enable it to produce more efficient code—a process known as *optimization*.
- The C99 standard guarantees that `restrict` has no effect on the behavior of a program that conforms to the standard.
- Most programmers won’t use `restrict` unless they’re fine-tuning a program to achieve the best possible performance.
Flexible Array Members (C99)

- Occasionally, we’ll need to define a structure that contains an array of an unknown size.
- For example, we might want a structure that stores the characters in a string together with the string’s length:

  ```c
  struct vstring {
    int len;
    char chars[N];
  };
  ```
- Using a fixed-length array is undesirable: it limits the length of the string and wastes memory.
Flexible Array Members (C99)

• C programmers traditionally solve this problem by declaring the length of `chars` to be 1 and then dynamically allocating each string:

```c
struct vstring {
    int len;
    char chars[1];
};
...
struct vstring *str =
    malloc(sizeof(struct vstring) + n - 1);
str->len = n;
```

• This technique is known as the “struct hack.”
Chapter 17: Advanced Uses of Pointers

Flexible Array Members (C99)

- The struct hack is supported by many compilers.
- Some (including GCC) even allow the chars array to have zero length.
- The C89 standard doesn’t guarantee that the struct hack will work, but a C99 feature known as the flexible array member serves the same purpose.
Flexible Array Members (C99)

• When the last member of a structure is an array, its length may be omitted:

  struct vstring {
    int len;
    char chars[]; /* flexible array member - C99 only */
  };

• The length of the array isn’t determined until memory is allocated for a vstring structure:

  struct vstring *str = malloc(sizeof(struct vstring) + n);
  str->len = n;

  sizeof ignores the chars member when computing the size of the structure.
Flexible Array Members (C99)

- Special rules for structures that contain a flexible array member:
  - The flexible array must be the last member.
  - The structure must have at least one other member.

- Copying a structure that contains a flexible array member will copy the other members but not the flexible array itself.
Flexible Array Members (C99)

• A structure that contains a flexible array member is an *incomplete type*.

• An incomplete type is missing part of the information needed to determine how much memory it requires.

• Incomplete types are subject to various restrictions.

• In particular, an incomplete type can’t be a member of another structure or an element of an array.

• However, an array may contain pointers to structures that have a flexible array member.