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 are forced to choose their sizes when writing a program
- Fortunately, `C` supports ***dynamic storage allocation***, i.e., 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
  - By allocating strings dynamically, we can postpone the *length-of-the-string* decision until the program is running
  - 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 does not 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 *` (i.e., a "*generic*" pointer)

Null Pointers

- If a memory allocation function can not 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 have stored the function's return value in a pointer variable, we must test to see if it is 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) …
```


Using `malloc` to Allocate Memory for a String

- Prototype for the `malloc` function

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

- `malloc` allocates a block of `size` *bytes* and returns a pointer to it
- `size_t` is an *unsigned integer* type defined in the library

Using `malloc` to Allocate Memory for a String

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

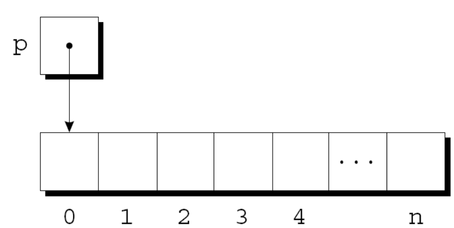
```
char *p;
p = malloc(n + 1);
```

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

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


Using `malloc` to Allocate Memory for a String

- Memory allocated using `malloc` *is not cleared*, so `p` will point to an *uninitialized array* of `n + 1` characters

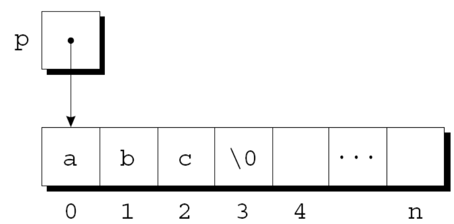


Using `malloc` to Allocate Memory for a String

- 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

- Consider 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)
  { 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*

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 will want to call the `free` function to release the space that the string occupies
- If we do not, the program may eventually run out of memory

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

- The **`remind2.c`** program is based on the **`remind.c`** program of 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, instead of using *two-dimensional array of characters*, we will use *a one-dimensional array of pointers to dynamically allocated strings*

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>  /* First new line */
#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]; /* Second new line */
/* Originally was:
  char reminders[MAX_REMIND][MSG_LEN+3];
*/
  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]; /* Third new line */
/*    Originally was:
      strcpy(reminders[j], reminders[j-1]);
*/
```



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

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

    num_remind++;
  }

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

  return 0;
}
```

```
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;
}
```


Using `malloc` to Allocate Storage for an Array

- Dynamically allocated arrays have the same advantages as dynamically allocated strings
- Suppose a program needs an array of `n` integers, where `n` is computed during program execution
- we will first declare a pointer variable

  `int *a;`
- Once the value of `n` is known, the program can call `malloc` to allocate space for the array

  `a = malloc(n * sizeof(int));`
- *Always use the* `sizeof` *operator to calculate the amount of space required for each element*

Using **malloc** to Allocate Storage for an Array

- 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

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


The **calloc** Function

- The calloc function is very similar to malloc, but it initializes (*clears*) the memory that it allocates
- 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 is not 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 *single* data item
  - The size of this item is specified in the second argument

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

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


The `realloc` Function

- The `realloc` function can resize a dynamically allocated array, i.e., make an array "grow" or "shrink" as needed
- 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

- Properties of `realloc`
  - 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
  - If `realloc` can not enlarge the memory block as requested,
    - it returns a null pointer;
    - the data in the old memory block is unchanged
  - When it expands a memory block, `realloc` does ***not*** initialize the new bytes that are added to the 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 not 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

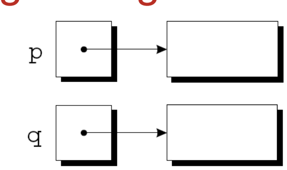
Deallocating Storage

- `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

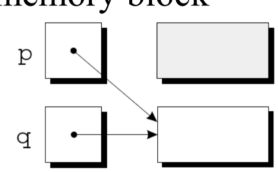
Deallocating Storage

- Example

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



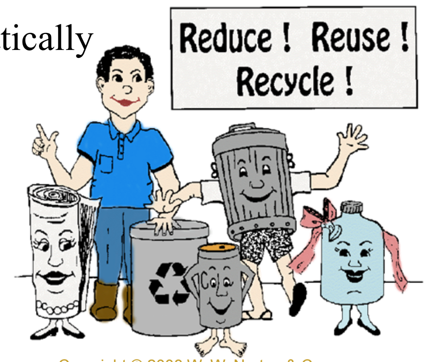
- After `q` is assigned to `p`, both variables now point to the second memory block



- There are no pointers to the first block, so we will never be able to use it again

Deallocating Storage

- A block of memory that is 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`** ***does not***
- 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 does not change `p` itself
- If we forget that `p` no longer points to a valid memory block, chaos may arise

```
char *p = malloc(4);
…
free(p);
…
strcpy(p, "abc");    /*** WRONG ***/
```

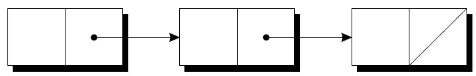
- Modifying the memory that `p` points to is a serious error

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

Linked Lists

- Dynamic storage allocation is especially useful for building lists, trees, graphs, and other linked data structures
- A ***linked list*** consists of a chain of structures (called ***nodes***), with each node containing a pointer to the next node in the chain



- The last node in the list contains a null pointer

Linked Lists

- 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 is near the end

Declaring a Node Type

- To set up a linked list, we will 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 will 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 will 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 will concentrate on the first two steps for now

Creating a Node

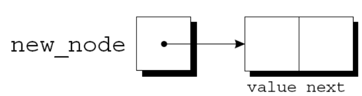
- When we create a node, we will need a variable that can point to the node temporarily

```
struct node *new_node;
```

- we will 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

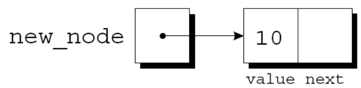


Creating a Node

- Next, we will store data in the `value` member of the new node

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

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



The `->` Operator

- Accessing a member of a structure using a pointer is so common
  - `C` provides a special operator for this purpose
- This operator, known as ***right arrow selection***, is a *minus sign* followed by *right arrow*, ***i.e.,*** `->`
- Using the `->` operator, we can write

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

instead of

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

The -> Operator

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

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

- The `&` operator is still required, even though `new_node` is a pointer
- This `scanf` can be written as

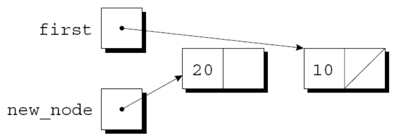
```
scanf("%d", &(*new_node).value);
```

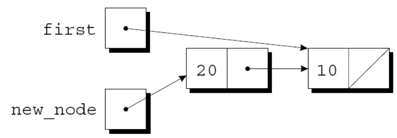

Inserting a Node at the Beginning of a Linked List

- One of the advantages of a linked list is that nodes can be added at *any* point in the list
  - At the beginning of a list
  - At any other location other than the beginning of the list
- 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

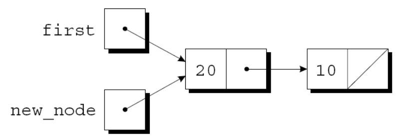
Inserting a Node at the Beginning of a Linked List

- It takes two statements to insert the node into the list
  1. 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;`





  2. Make `first` to point to the new node
     `first = new_node;`
     These statements work even if the list is empty

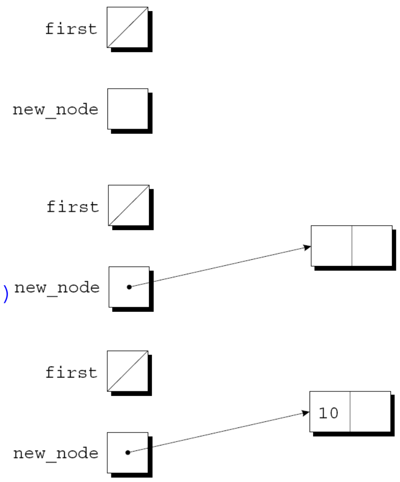


Inserting a Node at the Beginning of a Linked List

- Let us trace the process of inserting two nodes into an empty list
- we will insert a node containing the number `10` first, followed by a node containing `20`

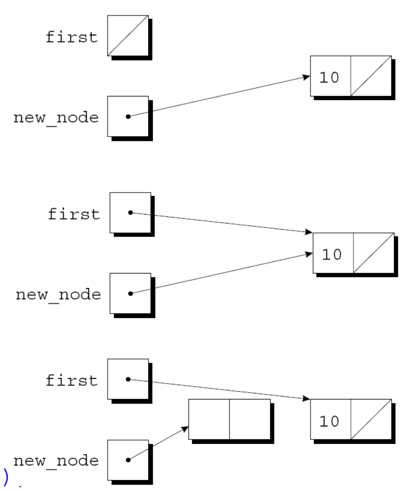
Inserting a Node at the Beginning of a Linked List

```
first = NULL;

struct node *new_node;

new_node =
  malloc(sizeof(struct node));

new_node->value = 10;
```



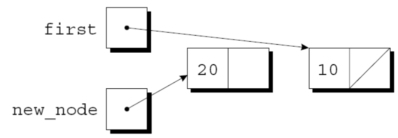
Inserting a Node at the Beginning of a Linked List

```
new_node->next = first;

first = new_node;

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

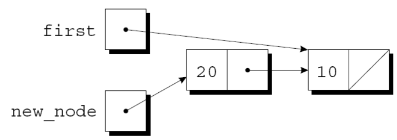


Inserting a Node at the Beginning of a Linked List

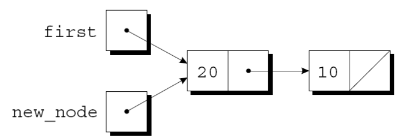
```
new_node->value = 20;
```




```
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;
}
```

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 will 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 (*will be addressed later in the chapter*)

Inserting a Node at the Beginning of a Linked List

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

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

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

- The numbers will be in reverse order within the list