Lecture 12 - Dynamic Memory Management

Meng-Hsun Tsai
CSIE, NCKU

12.1 Dynamic Memory Allocation

Dynamic memory Allocation

- C's data structures, including arrays, are normally fixed in size.
- Fixed-size data structures can be a problem, since we're forced to choose their sizes before executing a program.
- Fortunately, C supports ***dynamic memory allocation:*** the ability to allocate memory during program execution.
- Using dynamic memory allocation, we can design data structures that grow (and shrink) as needed.

IMSLab Since 2010

Dynamic memory Allocation (cont.)

- Dynamic memory 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 memory allocation is done by calling a memory allocation function.

Memory Allocation Functions

- The `<stdlib.h>` header declares three memory allocation functions:

  `malloc`—Allocates a block of memory but doesn’t initialize it.

  `calloc`—Allocates a block of memory and clears it.

  `realloc`—Resizes a previously allocated block of memory.

- These functions return a value of type `void *` (a “generic” pointer).

Null Pointers

- If a memory allocation function can't locate a memory block of the requested size, it returns a ***null pointer*.**
- A null pointer is a special value that can be distinguished from all valid pointers.
- After we've stored the function's return value in a pointer variable, we must test to see if it's a null pointer.

Null Pointers (cont.)

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

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

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

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

IMSLab Since 2010

Null Pointers (cont.)

- Pointers test true or false in the same way as numbers.
- All non-null pointers test true; only null pointers are false.
- Instead of writing

  ```
  if (p == NULL) …
  ```

  we could write

  ```
  if (!p) …
  ```

- Instead of writing

  ```
  if (p != NULL) …
  ```

  we could write

  ```
  if (p) …
  ```

12.2 Dynamically Allocated Strings

Dynamically Allocated Strings

- Dynamic memory 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.

IMSLab Since 2010

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

- Prototype for the `malloc` function:

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

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

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

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

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

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

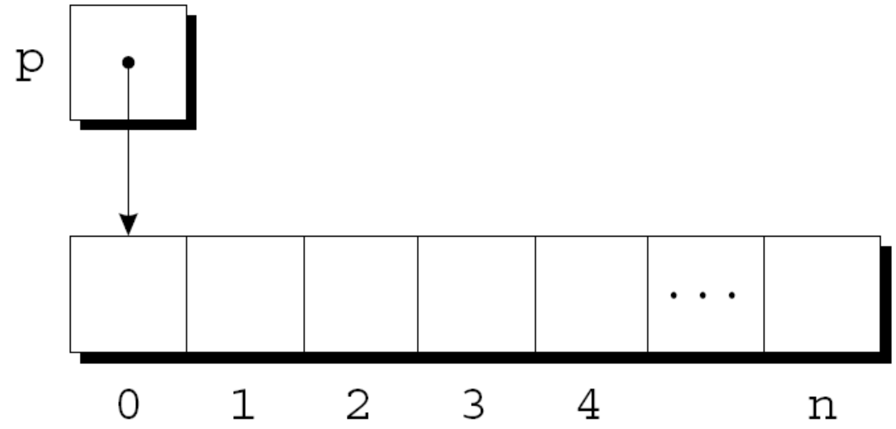
- Some programmers prefer to cast `malloc`'s return value, although the cast is not required:

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

IMSLab Since 2010

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

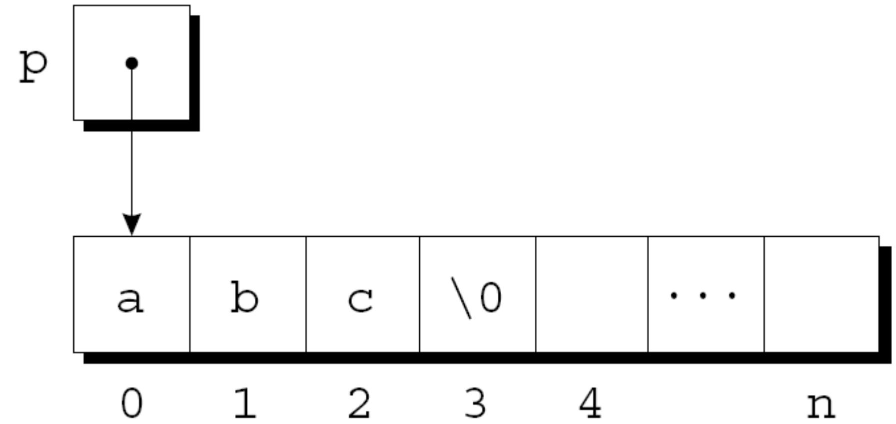
- Memory allocated using `malloc` isn't cleared, so `p` will point to an uninitialized array of `n` + 1 characters:



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

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

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



Using Dynamic memory Allocation in String Functions

- Dynamic memory 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 memory Allocation in String Functions (cont.)

```
char *concat(const char *s1, const char *s2)
{
  char *result;

  result = malloc(strlen(s1) + strlen(s2) + 1);
  if (result == NULL) {
    printf("Error: malloc failed in concat\n");
    exit(EXIT_FAILURE);
  }
  strcpy(result, s1);
  strcat(result, s2);
  return result;
}
```

Using Dynamic memory Allocation in String Functions (cont.)

- A call of the `concat` function:

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

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

Using Dynamic memory Allocation in String Functions (cont.)

- Functions such as `concat` that dynamically allocate memory must be used with care.
- When the string that `concat` returns is no longer needed, we'll want to call the `free` function to release the space that the string occupies.
- If we don't, the program may eventually run out of memory.

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

- The `remind2.c` program is based on the `remind.c` program of Lecture 10, 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.

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

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

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

remind2.c

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

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

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

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

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

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

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

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

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

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

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

    num_remind++;
  }

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

  return 0;
}
```

12.3 Dynamically Allocated Arrays

Dynamically Allocated Arrays

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

```
int *a;
```

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

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

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

IMSLab Since 2010

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

- We can now ignore the fact that `a` is a pointer and use it instead as an array name, thanks to the relationship between arrays and pointers in C.
- For example, we could use the following loop to initialize the array that `a` points to:

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

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

The `calloc` Function

- The `calloc` function is an alternative to `malloc`.
- Prototype for `calloc`:

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

- Properties of `calloc`:
  - Allocates space for an array with `nmemb` elements, each of which is `size` bytes long.
  - Returns a null pointer if the requested space isn't available.
  - Initializes allocated memory by setting all bits to 0.

The `calloc` Function (cont.)

- A call of `calloc` that allocates space for an array of `n` integers:

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

- By calling `calloc` with 1 as its first argument, we can allocate space for a data item of any type:

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

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

The `realloc` Function

- The `realloc` function can resize a dynamically allocated array.
- Prototype for `realloc`:

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

- `ptr` must point to a memory block obtained by a previous call of `malloc`, `calloc`, or `realloc`.
- `size` represents the new size of the block, which may be larger or smaller than the original size.

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

IMSLab Since 2010

The `realloc` Function (cont.)

- Properties of `realloc`:
  - When it expands a memory block, `realloc` doesn't initialize the bytes that are added to the block.
  - If `realloc` can't enlarge the memory block as requested, it returns a null pointer; the data in the old memory block is unchanged.
  - If `realloc` is called with a null pointer as its first argument, it behaves like `malloc`.
  - If `realloc` is called with 0 as its second argument, it frees the memory block.

The `realloc` Function (cont.)

- We expect `realloc` to be reasonably efficient:
  - When asked to reduce the size of a memory block, `realloc` should shrink the block "in place."
  - `realloc` should always attempt to expand a memory block without moving it.
- If it can't enlarge a block, `realloc` will allocate a new block elsewhere, then copy the contents of the old block into the new one.
- Once `realloc` has returned, be sure to update all pointers to the memory block in case it has been moved.

12.4 Deallocating Memory

Deallocating memory

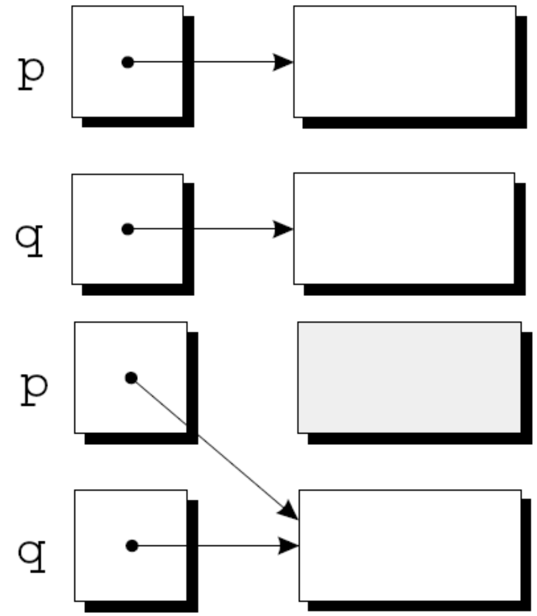
- `malloc` and the other memory allocation functions obtain memory blocks from a memory pool known as the ***heap***.
- Calling these functions too often—or asking them for large blocks of memory—can exhaust the heap, causing the functions to return a null pointer.
- To make matters worse, a program may allocate blocks of memory and then lose track of them, thereby wasting space.

Deallocating memory (cont.)

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

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

- After `q` is assigned to `p`, both variables now point to the second memory block.
- There are no pointers to the first block, so we'll never be able to use it again.



Deallocating memory (cont.)

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

The `free` Function

- Prototype for `free`:

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

- `free` will be passed a pointer to an unneeded memory block:

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

- Calling `free` releases the block of memory that `p` points to.

The "Dangling Pointer" Problem

- Using `free` leads to a new problem: ***dangling pointers*.**
- `free(p)` deallocates the memory block that `p` points to, but doesn't change `p` itself.
- If we forget that `p` no longer points to a valid memory block, chaos may ensue:

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

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

The "Dangling Pointer" Problem (cont.)

- Dangling pointers can be hard to spot, since several pointers may point to the same block of memory.
- When the block is freed, all the pointers are left dangling.

A Quick Review to This Lecture

- The `<stdlib.h>` header declares three memory allocation functions which return void pointers (`void *`) to allocated space or return ***null pointers*** if allocation is failed:

  - `malloc` allocates a block of `size` bytes but doesn't initialize it.

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

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

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

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

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

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

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

A Quick Review to This Lecture (cont.)

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

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

```
p = malloc(…);
free(p);
```

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