Dynamical Memory Allocation

Marcus Birkenkrahe

November 26, 2024

Advanced use of pointers

- Using *dynamical memory allocation*, a program can obtain blocks of memory as needed during execution.
- We'll discuss dynamically allocated strings and arrays with `malloc` and their deallocation with `free`.
- These structures can be linked together to form lists, trees, and other flexible data structures.
- We introduce *linked lists*, the most fundamental linked data structure.
- Advanced pointer use also includes pointers to functions: Some of C's library functions expect function pointers as arguments - one of these is `qsort`, which can sort any array fast (we won't cover this).
- As a motivation booster, the number of programmers who master these concepts is no larger than 10% world-wide. And you can get there in a 200-level undergraduate class!
- Differently put, in one of the densest, most complete books on C programming, you're now past page 400 (few explorers get that far).

Dynamic storage allocation

- C data structures are normally fixed in size:

    1. The number of elements of an array is fixed after compilation.
    2. In C99, a VLA length is determined at run time but it remains fixed for the array's lifetime.

- This is a problem because we're forced to choose the size when writing a program - we cannot change it without rewriting and recompiling the code.

Memory allocation functions

- There are three, all defined in `<stdlib.h>`:

  1. `malloc`, which allocates a memory block without initialization.
  2. `calloc`, which allocates memory and clears it.
  3. `realloc`, which resizes a previously allocated block of memory.

- Of these, `malloc` is most used and most efficient.
- When we call `malloc` to request memory, the function has no idea what type of data we plan to store there - it cannot return a pointer to an ordinary type (`int` or `char`).
- Instead, the `malloc` returns a value of type `void *`, a generic pointer, which is just a memory address, to be specified later.

Null pointers

- If `malloc` cannot locate a block of memory large enough to satisfy our request, it returns a *null pointer*, a pointer to nothing, `NULL`.
- After calling `malloc`, we must test if its `return` value is `NULL`:

```
p = malloc(10000); // reserve 10k bytes, store address in pointer p
if (p == NULL)
  /* allocation failed, take appopriate action */
```

- You can also combine these in one statement:

```
if ( (p = malloc(10000)) == NULL)
    /* allocation failed, take appopriate action */
```

- What is "appropriate action"?

  1. Abort the program.

2. Issue warning, keep a log, save what can be saved, and continue.

- Remember that numbers can be `TRUE` or `FALSE` - any number that's not `0` is `TRUE`, and only `0` is `FALSE`.
- In the same way, all non-null pointers test `TRUE`, and only `NULL` tests `FALSE`.
- So you could write `if (!p)` instead of `if (p==NULL)`, and `if (p)` instead of `if (p != NULL)` (don't do it).

Dynamically allocated strings with `malloc`

- Strings are stored in character arrays and it can be hard to anticipate their length.
- By allocating a string dynamically, we can postpone the decision until the program is running.
- The `malloc` function has the following *prototype*:

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

1. It returns a generic pointer `void *` to the start of a block
2. It allocates a block of `size` bytes.
3. The block `size` has type `size_t` to be machine independent.
4. `size_t` is an unsigned integer type defined in the C library.
5. So you can think of `size` as an ordinary integer `int`.

- To allocate space for a string of `n` characters:

```
char *p;   // declare character pointer (pointer to string)
p = malloc(n + 1); // reserve n+1 bytes for the string
```

1. `p` is a character pointer, a `char *` variable.
2. A `char` value requires 1 byte of storage: `sizeof(char) = 1`.
3. You need to leave one byte free for the null character `'\0'`.
4. The generic `void *` pointer from `malloc` is converted to `char *`.

- You can also make the *cast* from `void *` to `char *` explicit:

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

- To initialize `p`, you can use `strcpy`:

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

- Now the first four characters in the array will be `a`, `b`, `c`, and `'\0`.
- What is `sizeof(p)`? Is that the length of the string stored in `p`?

  > No. `sizeof(p)` is the size of the pointer, not the data that `p` points to. To find the length of the string, print `strlen(p)`.

Practice: Dynamically Allocating and Managing Strings (v1)

Write a C program that does the following:

1. Prompts the user to enter their name.
2. Dynamically allocates memory to store the name.
3. Copies the entered name into the allocated memory.
4. Prints a greeting message using the name stored in the dynamically allocated memory.
5. Frees the allocated memory before exiting.

Example Output:

```
Enter your name: Marcus
Hello, Marcus!
```

Hints:

- Use `fgets` to get any string (including whitespace) instead of `scanf`.
- Use `malloc` to allocate memory for the string.
- Remember to allocate space for the null terminator `\0`.
- Check memory allocation success with `NULL`.
- Use `strcpy` to copy the user input into the allocated memory.
- Use `free` to release the allocated memory.

Solution

- Sample input:

```
echo "Marcus Birkenkrahe" > name
cat name
```

```
Marcus Birkenkrahe
```

- Code (remove `main` and `includes` templates) v1: with `gets`

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

int main() {
  char temp[100];  // Temporary buffer for user input
  char *name;      // Pointer for dynamically allocated memory

  printf("Enter your full name: ");
  gets(temp);
  printf("%s",temp);

  // Dynamically allocate memory for the input
  name = (char *)malloc(strlen(temp) + 1);  // +1 for null terminator
  if (name == NULL) {
    printf("Memory allocation failed!\n");
    return 1;
  }

  // Copy the input into the dynamically allocated memory
  strcpy(name, temp);

  // Print a greeting
  printf("\nHello, %s!\n", name);

  // Free the allocated memory
  free(name);
  return 0;
}
```

```
Enter your full name: 1000
Hello, 1000!
```

- Code (remove `main` and `includes` templates) v2: with `fgets`

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

int main() {
  char temp[100];  // Temporary buffer for user input
  char *name;      // Pointer for dynamically allocated memory

  printf("Enter your full name: ");
  if (fgets(temp, sizeof(temp), stdin) != NULL) {
    // Remove the newline character, if present
    size_t len = strlen(temp);
    if (len > 0 && temp[len - 1] == '\n') {
temp[len - 1] = '\0';
printf("%s",temp);
    }

    // Dynamically allocate memory for the input
    name = (char *)malloc(len + 1);  // +1 for null terminator
    if (name == NULL) {
printf("Memory allocation failed!\n");
return 1;
    }

    // Copy the input into the dynamically allocated memory
    strcpy(name, temp);

    // Print a greeting
    printf("\nHello, %s!\n", name);

    // Free the allocated memory
    free(name);
  } else {
    printf("Error reading input!\n");
  }
```

```
  return 0;
}
```

```
Enter your full name: 1000
Hello, 1000!
```

Practice: Using Command-Line Arguments with Dynamically Allocated Strings (v2)

Write a C program that does the following:

1. Accepts the user's full name (in quotes) as command-line argument.
2. Dynamically allocates memory to store the name.
3. Copies the command-line argument into the allocated memory.
4. Prints a greeting message using the name stored in the dynamically allocated memory.
5. Frees the allocated memory before exiting.

Example Usage:

```
$ ./main Marcus Birkenkrahe
Hello, Marcus Birkenkrahe!
```

Hints:

- Use `main(int argc, char *argv[])` to handle command-line arguments.
- `argc` represents the number of arguments passed to the program.
- `argv[1]` holds the first command-line argument after the program name.
- Use `malloc` to allocate memory for the string.
- Remember to allocate space for the null terminator ('\0')
- Use `strcpy` to copy the cmd-line argument into the allocated memory.
- Use `free` to release the allocated memory.

Solution:

- Code without checks: source code `main.c`

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

int main(int argc, char *argv[]) {
  char *name;  // Pointer for dynamically allocated memory

  // Allocate memory for the string
  name = (char *)malloc(strlen(argv[1]) + 1);  // +1 for null terminator
  // Copy the command-line argument into allocated memory
  strcpy(name, argv[1]);

  // Print a greeting
  printf("Hello, %s!\n", name);

  // Free the allocated memory
  free(name);

  return 0;
}
```

- Test:

```
gcc main.c -o main
./main "Marcus Birkenkrahe"


Marcus Birkenkrahe
```

- Code with checks: source code `main2.c`

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

int main(int argc, char *argv[]) {
```

```
  char *name;  // Pointer for dynamically allocated memory

  // Check if the user provided a name
  if (argc < 2) {
    printf("Usage: %s <name>\n", argv[0]);
    return 1;  // Exit the program
  }

  // Allocate memory for the string
  name = (char *)malloc(strlen(argv[1]) + 1);  // +1 for null terminator
  if (name == NULL) {
    printf("Memory allocation failed!\n");
    return 1;  // Exit the program
  }

  // Copy the command-line argument into allocated memory
  strcpy(name, argv[1]);

  // Print a greeting
  printf("Hello, %s!\n", name);

  // Free the allocated memory
  free(name);

  return 0;
}
```

- Test:

  ```
  gcc main2.c -o main2
  ./main2 "Marcus Birkenkrahe"
  ```

Using dynamic storage allocation in string functions

- You can now write functions that return a pointer to a `new` string that didn't exist before the function was called.
- Example: concatenate two strings without changing eigher one. The C standard library has `strcat` but it modifies one of them:

```
char one[50] = "Hello, "; // destination string with space left
char *two = "world!";  // source string

printf("%s\n",strcat(one,two));
```

```
Hello, world!
```

- Why does this code not work? ("Segmentation fault

```
char *one = "Hello, ";
char *two = "world!";

printf("%s\n",strcat(one,two));
```

  Answer: The program tries to copy the string literal `two` into a string literal (constant) `one` - that's not allowed because string literals are stored in *read-only memory*.

- The `concat` function does this:

  1. measure length of the two strings to be concatenated with `strlen`
  2. call `malloc` to allocate the right amount of space
  3. copy first string into new space using `strcpy`
  4. concatenate second string using `strcat`

- Code:

```
// concat: concatenate two strings into a new string
// returns: character pointer to start of new string
// params: two constant character pointers
char *concat(const char *s1, const char *s2);

int main()
{
  printf("%s\n", concat("Hello, ","world!"));
  return 0;
}
```

```
char *concat(const char *s1, const char *s2)
{
  char *result; // uninitialized character pointer

  size_t len_s1 = strlen(s1); // compute length of s1
  size_t len_s2 = strlen(s2); // compute length of s2

  result = malloc(len_s1 + len_s2 + 1); // allocate result

  strcpy(result, s1);
  strcat(result, s2);

  return result;
}
```

```
Hello, world!
```

- Notice that the string `result` now occupies memory. When it is no longer needed, we'll want to `free` it or the program might otherwise run out of memory.

Dynamically allocated arrays

- Strings are arrays, and dynamically allocated arrays have the same advantages as dynamically allocated strings: You can wait until run-time to decide an array's size.
- C lets you allocate space for an array during execution and then access the array through a pointer to its first element.
- Sometimes, `calloc` is used instead of `malloc` since it initializes the memory that it allocates. `realloc` lets us shrink or grow the array.

Using `malloc` to allocate storage for an array

- To allocate an array of `n` integers where `n` is to be computed during run-time, we

  1. declare an integer pointer variable `int *`

2. allocate memory with `malloc` using `sizeof(int)`
3. initialize the array (can use pointer arithmetic)
4. `free` the array memory when we're done using `free(3)`.

- Code:

```
#define N 5 // define size of array

int *a; // declare integer pointer variable
int *b;

a = malloc(N * sizeof(int)); // reserve memory
b = malloc(N * sizeof(int));

// initialize array with subscripts
for (int i = 0; i < N; i++) {
  a[i] = 1;
  printf("%d ",a[i]);
 }; puts("");

// initialize array with pointer arithmetic
for(int *p = b; p < b + N; p++) {
  (*b) = 1;
  printf("%d ", *b);
 }

// free memory
free(a);
printf("\na[0]: %d ", *a);
printf("\nb[0]: %d ", b[0]);
free(b);
printf("\n%d ", *b);
```

```
1 1 1 1 1
1 1 1 1 1
a[0]: 86905179
b[0]: 1
-465853445
```

- Sometimes, you'll also see a casting operator `(int *)` before the `malloc` function - because it returns a `void *` pointer by default. You might see this when code is shared between C and C++.

```
int *a, N;
a = (int *) malloc(N * sizeof(int)); // reserve memory
```

Safeguarding `malloc` with `fprintf`

- To ensure dynamic memory allocation is successful, it is good practice to safeguard against failures using conditional checks.

- The `fprintf` function can be used to display an error message to `stderr` (the standard error stream) when `malloc` returns `NULL`, preventing undefined behavior in the event of memory allocation failure. Below is an example of how to use `fprintf` for this purpose:

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

int main() {
  int *array = malloc(10 * sizeof(int));
  if (array == NULL) {
    fprintf(stderr, "Error: Memory allocation failed\n");
    return 1;
  }

  // Proceed with normal operations after successful allocation
  for (int i = 0; i < 10; i++) {
    array[i] = i * 2;
  }

  // Print the array
  for (int i = 0; i < 10; i++) {
    printf("%d ", array[i]);
  }
  printf("\n");

  // Free the allocated memory
  free(array);
```

```
  return 0;
}
```

```
0 2 4 6 8 10 12 14 16 18
```

- In this example, the program checks if `malloc` returns `NULL` and exits with an error message if the allocation fails. This approach makes the program more robust and easier to debug in environments with constrained memory.

Deallocating storage with `free`

- `malloc` and the other memory allocation functions obtain memory blocks from the *heap*. Calling them too often may exhaust it.
- The program may allocated memory and then lose track of it, wasting space:

```
#define N 5

// reserve space for two memory blocks
int *p = malloc(N + sizeof(int));
int *q = malloc(N + sizeof(int));

printf("%p != %p\n",p,q);

p = q; // both pointers point to the 2nd memory block

printf("%p = %p\n",p,q);
```

```
0x5d5d8a5c92a0 != 0x5d5d8a5c92c0
0x5d5d8a5c92c0 = 0x5d5d8a5c92c0
```

- The first memory block is no longer accessible to the program, it's called *garbage*. The program has a *memory leak*.
- Some languages clean up their garbage (*garbage collection*), but C does not: You're responsible for doing that with `free`.

The "dangling pointer" problem

- The call `free(p)` deallocates the memory block that `p` points to, but it does not change `p` itself - it's a *dangling pointer*, and we must not use it unless it is reinitialized:

```
char *p = malloc(4); // string with three letters + null character

strcpy(p,"abc"); // initialize p with a string
printf("%s at %p\n", p, &p);

free(p); // deallocate the memory that p points to
printf("%s\n", p); // pointer is dangling, not NULL

strcpy(p,"abc"); // this is "undefined behavior" = crash danger
printf("%s at %p\n", p, &p);
```

```
abc at 0x7ffd862b6fd0
]abc at 0x7ffd862b6fd0
```

- It looks as if the last `strcpy` command worked but the only reason why it seems that way is because the memory block has not been overwritten.

Practice: Randomly Initialized Dynamic Arrays Using Command-Line Arguments

Write a C program that:

1. Dynamically allocates an array to store `n` integers, where `n` is provided as a command-line argument.
2. Initializes the array with random numbers between 1 and 100 using the `rand()` function.
3. Computes the sum of all elements in the array using pointer arithmetic.
4. Prints the array and the computed sum.
5. Safeguards against memory allocation failure with `fprintf`.
6. Frees the allocated memory after computation.

Solution Code

- Code:

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

int main(int argc, char *argv[]) {
  if (argc != 2) {
    fprintf(stderr, "Usage: %s <size_of_array>\n", argv[0]);
    return 1;
  }

  // Convert command-line argument to integer
  int n = atoi(argv[1]);
  if (n <= 0) {
    fprintf(stderr, "Error: Array size must be a positive integer.\n");
    return 1;
  }

  int sum = 0;

  // Allocate memory for the array
  int *array = malloc(n * sizeof(int));
  if (array == NULL) {
    fprintf(stderr, "Error: Memory allocation failed\n");
    return 1;
  }

  // Seed the random number generator
  srand(time(NULL));

  // Initialize array with random numbers and compute sum
  printf("Array elements: ");
  for (int *p = array; p < array + n; p++) {
    *p = rand() % 100 + 1; // Random number between 1 and 100
    sum += *p;
    printf("%d ", *p); // Print element
  }
```

```
  printf("\n");

  // Print the computed sum
  printf("Sum of array elements: %d\n", sum);

  // Free the allocated memory
  free(array);

  return 0;
}
```

- To run the program, compile it and provide the size of the array as a command-line argument:

```
gcc main.c -o main
./main 10
```

```
10
```