This course has already ended.

« 1 Memory organization | Course materials | 3 Valgrind »

ELEC-A7100 / 5. Dynamic Memory / 2 Dynamic memory management

Fifth round tasks as a `ZIP file.`

Usually in the software projects there are situations where we are at the programming stage and cannot know how much memory the program needs in advance. In this case, we need a **dynamic memory** where size can be defined during the execution of the program. Dynamic memory is allocated by operating system upon request and generally it will always be free. This section gives introduction to computer memory organization, dynamic memory management and **valgrind** (tool is used to check memory leaks in the program).

Note: In this section, the tasks use valgrind tool to check memory leaks. If valgrind memory leaks or warnings are present in the code, only half of the total task point is given. If both valgrind and warnings exist, then you will get only quarter of the total task score. Valgrind information is provided later in this section.

Dynamic memory management¶

Allocating and releasing memory¶

New memory from heap can be allocated using the **malloc** function call, that is defined in **stdlib.h** header (like the other memory management functions discussed below).

In programming, sometimes the memory must be dynamically allocated:

- The required size of allocated memory is not known before running the program, because it might depend on the user input etc.
- The function is required to modify / write to some data (usually session-dependent data that was allocated elsewhere)
- The maximum size of allocated memory might be known, but there should be no waste memory allocated.

The exact definition for the function is `void *malloc(size_t size)`. malloc() takes one parameter that defines how many bytes of memory is allocated. The function returns a pointer to the allocated memory. It is possible that the return value is **NULL** (i.e. 0), which means that the memory allocation failed. You should always check that the return value is not **NULL** before starting to use the allocated memory. The allocated memory is uninitialized, so you cannot assume anything about its initial content.

The `void*` -pointer in malloc return value is a generic pointer, that is otherwise as any other pointer, but cannot be directly dereferenced or used, because no type has been specified. Neither can the pointer arithmetics be applied with void* pointer. However, it is easy to assign (without explicit typecast) a generic pointer to any typed pointer variable, after which it can be used normally. The below example shows how this happens with the malloc return value.

After memory is successfully allocated using the malloc() call and the returned pointer assigned to a typed pointer variable, it can be used as any other pointer. For example, if malloc() was used to allocate an array of certain type, the normal array operators can be used to modify and access the content of the array.

The example below shows how **malloc** - function is used.

When the memory is successfully booked on line 7, start address of the reserved memory area is placed in the pointer variable **table**. In this example, space is reserved for `100` **int** data type. Line 12 copies contents to **table** array.

```
1   #include <stdlib.h>
2   #include <stdio.h>
3
4   int main(void)
5   {
6       int *table; // uninitialized at this point
7       table = malloc(100 * sizeof(int));
8       if (table == NULL) {
9           return -1;  // memory allocation failed
10      }
11      int i;
12      for (i = 0; i < 100; i++) {
13          table[i] = 100 - i;
14      }
15
16      for (i = 0; i < 100; i++) {
17          printf("%d ", table[i]);
18          if (!(i % 20))
19              printf("\n");
20      }
21
22      // allocated memory is not needed anymore, must be freed
23      free(table);
24  }
```

It is important to take the data type size into account when allocating memory. In this case, we are allocating space for 100 integers. Therefore the size of the allocated memory needs to be multiplied by the data type size, using the **sizeof** operator (on line 7). Eventhough the array operator is not visible in the table declaration, we are effectively using a dynamically allocated array. In the above, after allocating space for an array of 100 integers from heap, the array can be manipulated using normal array operators (as on line 13).

In the end, the allocated memory is released using the **free** function call. The call takes one parameter, the pointer to the allocated memory. The allocated memory should always be released after it is not needed. Otherwise a memory leak would follow, meaning that your program would slowly consume increasing amount of memory, leaving less memory for the other processes in the system. When program terminates, the allocated memory is released by the operating system, but memory leaks can be a problem in long-running processes, for example in network servers.

An earlier allocated memory can be resized using the **realloc** function. The exact definition of the function is `void *realloc(void ptr, size_t size)`. `ptr` is the pointer to the earlier allocated memory, **size** is the new size (either smaller or larger than before). As with malloc, the function returns pointer to the allocated memory. The returned pointer may be different than what was given in the ptr parameter.

Logically this call is equivalent to 1) allocating a new memory space of given size; 2) copying data from the earlier allocated memory space to new one; 3) releasing the earlier allocated memory space.

The example below shows how *realloc* works.

```
1   #include <stdlib.h>
2   #include <stdio.h>
3
4   int main(void)
5   {
6       int *table; // uninitialized at this point
7       table = malloc(100 * sizeof(int));
8       if (table == NULL) {
9           return -1;  // memory allocation failed
10      }
11      int i;
12      for (i = 0; i < 100; i++) {
13          table[i] = 100 - i;
14      }
15
16      int *newtable = realloc(table, 200 * sizeof(int));
17
18      if (!newtable) {
19          free(table);  // realloc failed, old pointer still valid
20          return -1; // error occured
21      }
22      else { // read 100 more numbers to array, which size was increased
23          for(i = 100; i<200; i++) {
24              newtable[i] = 100 - i;
25          }
26
27          for (i = 0; i < 200; i++) {
28              printf("%d ", newtable[i]);
29              if (!(i % 20))
30                  printf("\n");
31          }
32
33          // must free memory before main-function ends
34          free(newtable);  // realloc succeeded, old pointer was released before
35      }
36
37      return 0;
38  }
```

The above program allocates 100 bytes of memory on line 7 using `malloc` function and stores it in **table** pointer. On line 16, 100 bytes memory is increased to 200 bytes memory and stored in **newtable** pointer using `realloc` function. If `realloc` fails, then the already reserved memory is released and it has to be released explicitly using **free** function (on line 19).

realloc function can also be called in such a way that the previous memory area address is given `NULL`. In this case, it is equivalent to **malloc** function.

In addition to above functions, there is another function called `calloc` that allocates a block of memory for an array of **n** elements, each of them size **bytes** long, and initializes all its bits of the allocated memory to zero. The exact definition of the calloc function is `void * calloc(size_t count, size_t size);`. Thus, for example, `calloc` could be called like following `table = calloc(100, sizeof(int));`.

« 1 Memory organization | Course materials | 3 Valgrind »