COMP 2001/2401 Course Notes: Dynamic Memory Allocation

Benjamin Yan

1 Introduction

1.1 Dynamic Memory Allocation

Dynamic memory allocation in C refers to the actions of allocating memory in a flexible manner, and the deallocation of the memory when it is no longer required by the program. The C standard library provides the following functions used for this purpose [1]:
malloc()
realloc()
calloc()
free()

When memory is dynamically allocated in a C program, a block of physical memory is reserved on the heap for use. In general, access to these memory locations would be prohibited otherwise.

1.2 Motivation

An array of integers in C can be easily created as an automatic variable, using a constant array length:

```
int arrayX[10];
```

Under the C99 Standard, it is also possible to declare an automatic array using an integer variable as the array length. The use of automatic variables is beneficial in many circumstances, since their usage is highly optimized during compilation, and the memory usage is managed automatically. It is also possible to use the same variable name in a number of different blocks in the code.

On the other hand, declaring automatic arrays has certain caveats. Since automatic variables are created along with their function blocks on the stack, the variables cannot persist beyond their scope, unless they are declared as static, in which case they persist through the life of the program. In either case, the programmer may require greater flexibility in the lifetime of the allocated memory. It is also impossible to declare an automatic array if the size cannot be known before run-time. Further, it is relatively tedious for the programmer to expand or reduce the size of an automatic array, as each element of the array must be copied to another variable. These caveats can be overcome by using dynamically allocated memory instead.

1.3 Physical Memory

Depending what type of variable created (global, static, automatic, or dynamic), each type of variable is located in a different section of the physical memory. In simple terms, global and static variables are generally stored in the data segment, automatic variables are stored in the stack frames corresponding to their scope, and dynamically allocated variables are stored in the heap. The data segment generally lies in the lower memory addresses, below the heap. The heap lies between the data segment and stack, and the heap and stack grow in opposite directions, toward each other[6]. A simple program is used below to demonstrate the various types of variables in C, and an outline of the corresponding memory layout is shown in Figure 1.

```
#include <stdlib.h>

int gArray[5] = {0,1,2,3,4};

int main(){
  static int sArray[10];
  char       aArray[10];
  int*       mArray = malloc(sizeof(int)*10);
  double*    cArray = calloc(sizeof(double)*5);
  mArray            = realloc(arrayX, sizeof(int)*20);
  return 0;
}
```

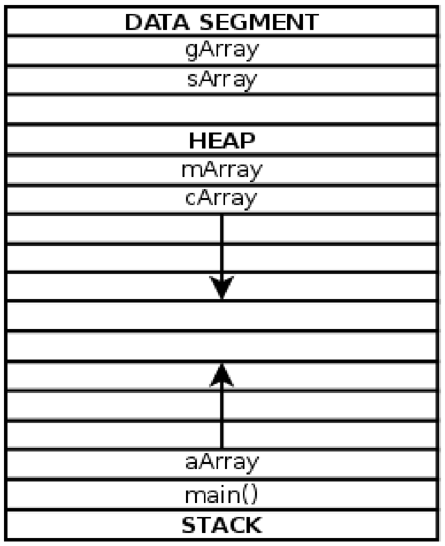

Figure 1: Memory model of various types of variables

2 Using Dynamic Memory Allocation

To use dynamic memory allocation in C, the C standard library must be imported using:

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

2.1 malloc()

The most basic form of dynamic memory allocation is performed using the malloc()function, the prototype of which is given as[4]:

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

The input to the function is effectively an integer denoting the number of bytes required, and the function returns a pointer to the beginning of the allocated memory block, if the allocation was successful. Otherwise, a null pointer is returned. Since the function returns a void type pointer, it can be casted to whatever data type is required. An example of the function usage is given below:

```
int* arrayX = malloc(sizeof(int)*10);
```

In this example, an integer array of size 10 is allocated to the arrayX variable, as we have done in section 1.2. The sizeof()function is used to ensure that we allocate the correct memory size, which is, in general, machine dependent. The values in each element of the array are uninitialized, so they are effectively random bits, until they are explicitly assigned a value.

2.2 realloc()

Once a memory block has been allocated using malloc(), the memory allocation can be resized using the realloc()function. It is also possible to use realloc()in the same manner as malloc(), by passing in a null pointer. The realloc()function prototype is given as[5]:

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

The ptr variable should either be a null pointer, or a pointer to a currently allocated memory block. Otherwise, an error should be raised. If the reallocation fails, a null pointer is returned by the function, and the memory block at ptr should remain unchanged. If the reallocation succeeds, the function returns the pointer to the new memory block, and the values of the bits in the memory blocks are preserved up to the smaller of the new and old sizes. If the new size is greater, the value of the additional bits are unknown. An example of the use of realloc()is given below:

```
int* arrayX = malloc(sizeof(int)*10);
for (int i=0; i<10; i++)
  arrayX[i]=0;
arrayX = realloc(arrayX, sizeof(int)*20);
```

Here we use malloc()to create an initial integer array of size 10, as before. Subsequently, we assign the value 0 to each of the elements in arrayX, and reallocate the variable to another memory block that is sized for 20 integers. Assuming that memory reallocation is successful, arrayX now points to a size 20 integer array, the first 10 elements of which have the value 0.

2.3 calloc()

In the case that the programmer wants to create a dynamic array with all bit values set to 0, one can use the calloc()function instead, and the prototype is given as[2]:

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

In contrast to the function call for malloc(), the above prototype requires two inputs, where num denotes the number of elements in the array, and size denotes the memory size of each element. Again, the function returns a pointer to the memory block of a successful allocation, or a null pointer if the allocation fails. In the following example, we use calloc()to create a size 10 array of integers with all values set to 0.

```
int* arrayX = calloc(10, sizeof(int));
```

2.4 free()

To actively control the lifetime of dynamic variables, and to avoid memory leaks, the memory blocks allocated using malloc(), realloc(), or calloc(), can be deallocated using the free()function. The function is defined as[3]:

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

Here, the ptr variable should be a pointer to a previously allocated memory block, using any of the allocation functions above. If a null pointer is passed to the function, nothing is done. If ptr is not a pointer to a previously allocated location, the operation causes undefined behaviour, and generally causes the program to crash. After the function call, the ptr variable is unchanged, and still points to the original location, so it is generally good practice to set ptr to NULL after calling free(ptr). The usage of free()is shown in the next example:

```
int*  arrayX = malloc(sizeof(int)*10);
char* arrayY = calloc(sizeof(char)*15);
arrayX       = realloc(arrayX, sizeof(int)*20);
free(arrayX);
free(arrayY);
arrayX = NULL;
arrayY = NULL;
```

References

[1] Various Authors. *C dynamic memory allocation*. 2013. URL: http://en.wikipedia.org/wiki/C_dynamic_memory_allocation.

[2] Various Authors. *cplusplus.com: calloc*. URL: http://www.cplusplus.com/reference/cstdlib/calloc/.

[3] Various Authors. *cplusplus.com: free*. URL: http://www.cplusplus.com/reference/cstdlib/free/.

[4] Various Authors. *cplusplus.com: malloc*. URL: http://www.cplusplus.com/reference/cstdlib/malloc/.

[5] Various Authors. *cplusplus.com: realloc*. URL: http://www.cplusplus.com/reference/cstdlib/realloc/.

[6] Narendra Kangralkar. *Memory Layout of C Programs*. 2011. URL: http://www.geeksforgeeks.org/memory-layout-of-c-program/.