Imperial College
London

Software Engineering 1: Introduction to Computing

6 – Raw memory (part b: dynamic memory)

Max Cattafi (m.cattafi@imperial.ac.uk)

In *part a* memory was always statically allocated. In other words the memory allocated for arrays and other variables was already determined at compile time.

In general we would like to be able to allocate the needed amount of memory as determined while the program runs (i.e. at runtime), for instance depending on the size of the input or the space needed to perform a certain algorithm.

We already know how to this by making use of C++ vectors. But how are vectors implemented?

In these notes we will consider some implementative details of dynamic memory in C++.

Pointers and dynamic memory allocation

Let's begin with an example. The following program illustrates several examples of operations involving pointers and dynamic memory allocation.

```
1  #include <iostream>
2
3  struct point{
4      double x;
5      double y;
6  };
7
8  int main(){
9
10     int a = 15;
```

```
11      std::cout << "a: " << a << std::endl;
12      int* pa;
13      pa = &a;
14      *pa = 3;
15      std::cout << "a (after the change through pa): " << a << std::endl;
16      // so far everything is like in previous examples:
17      // pointer pa points to the memory address of variable a
18      // the value of a can be changed through pa
19
20      // the following lines show how to perform something similar
21      // without using a variable declared in the usual way
22      // and using instead dynamic memory allocation
23
24      int* pb;
25      // *pb = 3;
26      // this can't be done at this stage: pb is not pointing
27      // to a memory address allocated for us
28      // (it is not even pointing anywhere defined)
29
30      pb = new int;
31      // instruction "new int" (dynamically)
32      // allocates memory for an int variable
33      // and returns the address of this memory areas
34      // which in this case is then assigned to pb
35
36      *pb = 3;
37      // now this can be done, the value is stored
38      // in the dynamically allocated memory cell
39
40      std::cout << "*pb: " << *pb << std::endl;
41
42      // when dynamically allocate memory
43      // is not needed anymore
44      // it needs to be manually deallocated with "delete"
45      // (dynamic memory that is not properly deallocated
46      // is a "memory leak" and it can cause the program
47      // to slow down and crash)
48
49      delete pb;
50
51      // similar to above but with a structured type (point)
52
53      point* ppoint;
54      ppoint = new point;
55      ppoint->x = 0.1;
56      ppoint->y = -0.1;
57      std::cout << "ppoint->x: " << ppoint->x << " ppoint->y: " << ppoint->y << std::endl;
58      delete ppoint;
59
60      // we can allocate dynamic memory also for arrays
61      // including information about the physical size
62      // (maximum number of elements of the array)
63      // in this case for instance we read the size
64      // from the user so it is only known at runtime
65      // (while in static arrays this information
66      // needs to be known at compile time)
67
68      int* parr;
69      int maxsize, lsize = 0;
70      std::cout << "enter amount of memory to allocate for array:" << std::endl;
71      std::cin >> maxsize;
72      parr = new int[maxsize];
```

```
73
74      parr[0] = 3;
75      lsize++;
76      parr[1] = 4;
77      lsize++;
78      for(int i = 0; i < lsize; i++){
79          std::cout << "parr[" << i << "]: " << parr[i] << std::endl;
80      }
81
82      // dynamic memory areas for arrays
83      // need to be deallocated with delete[]
84      // (not just delete)
85      delete[] parr;
86
87      return 0;
88  }
```

Simulating a vector of integers

The aim of this exercise is to get an insight into how something similar to a C++ vector can be built using dynamic arrays (and do some practice with dynamic memory allocation).

We want to write a C++ program which reads from a text file a sequence of integers, stores it in a dynamic "growing" array and then prints it back.

We implement the "growing" by using the following algorithm:

Initially, little or no memory is allocated (for simplicity, you can allocate a dynamic array of size 1).

Note that the dynamic array in which we are storing the data is always identified by a pointer (and always the same pointer) which points to the memory allocated for it.

At each iteration of the loop (which is reading from the file and storing into the dynamic array) we keep track of:

- The maximum size of the array (the allocated memory).
- The number of elements actually stored in the array (the logical size).

When the logical size is less than the allocated memory, the element can be just stored as usual at the relevant index.

However when a new allocation makes the number of elements greater than the maximum size of the array, a *reallocation* needs to take place as follows:

- Allocate another dynamic memory area with a greater maximum size (an auxiliary pointer will point to this memory location).
- Copy all the elements from the current array to the new memory area.
- Deallocate the memory area allocated for the current array.
- Get the pointer to the previous array to point to the new memory area pointed by the auxiliary pointer.
- Store the new item as usual at the relevant index.

The algorithm described above is similar to what happens when we call `push_back` to enter an element in a vector.

How much memory should be allocated for the new dynamic memory area? It would probably make little sense to allocate the same number of cells + 1, as it is likely that this would require a reallocation again at the next insertion (and as you can imagine the process of reallocation is quite computationally expensive). On the other hand we don't want to allocate too much memory and risk leaving it unused (it might end up being as memory inefficient as using a static array).

Some typical implementations of this algorithm in C++ vectors double the allocated memory every time a reallocation occurs[1] and you can adopt this same strategy.

Therefore for example if initially only one cell is allocated and the memory is doubled at each reallocation, and if there are 9 numbers in the file, this would involve 4 reallocations.

[1]We know that we can find out the logical size of a vector by using member function `size()`. If instead we would like to know the physical size, we can use member function `capacity()`. Knowing this we could write a program to verify whether the physical size of a vector is really doubled at each reallocation.