# 1 End of the non-OOP C++ : some pointers, dynamic allocation, miscellaneous things

## 1.1 Dynamic allocation

### 1.1.1 Why? How?

In many situations the programmer does not know when he writes a program what objects he will need. It can be that he does not know if he will need a given object, or or that he does not know the size of a required array.

The `new` operator allows to create an object at run-time. This operator takes as an operand a type, which may be followed either by a number of elements between `[ ]` or by an initial value between `( )` :

```
1  char *c = new char;
2  int *n = new int(123);
3  double *x = new double[250];
```

**The area of memory allocated by `new` is still used out of the pointer's scope!**

Bugs due to missing object deallocation are called *memory leaks*.

To free the memory, the programmer has to explicitly indicate to the computer to do so. The `delete` operator (resp. `delete[]`) takes a pointer to a single object (resp. an array of objects) as an operand and free the corresponding area of the memory ;

```
1 delete n;
2 delete[] x;
```

The variables declared in the source code, without the usage of `new` and `delete` are called **static variables**. Their existence is completely handled by the compiler who implicitly adds *invisible* `new`s and `delete`s.

The operand pointer can be equal to 0 ; in that case `delete` (or `delete[]`) does nothing. But deallocating an area which has already been deallocated has a non-defined behavior (i.e. crashes most of the time).

### 1.1.2 Dynamic arrays

The typical usage of dynamically allocated arrays is the following :

```
1  #include <iostream>
2
3  int main(int argc, char **argv) {
4    int n;
5
6    cout << "What array size? ";
7    cin >> n;
8
9    int *x = new int[n];
10   for(int k = 0; k < n; k++) x[k] = k*k;
11   delete[] x;
12 }
```