Pointers
Dynamic memory allocation

Anna Simon

September 23, 2015

Summary of the pointers (so far)

notation	description
& nValue	address-of operator (get the memory address of `nValue`)
int *myValue	declaration of a pointer
int *myValue = &nValue	declares a pointer and assigns it the address of nValue
*myValue	dereference operator (accesses the value the pointer is pointing to)
*myValue = nValue2	assigns a value of nValue2 to the variable the pointer is pointing to
myValue = &nValue2	redirects the pointer to a different variable
myValue	returns the address the pointer is pointing to

Pointers and const

Pointers can be declared as constant.

- const pointer

```
1 int nValue = 5;
2 int *const pPointer = &nValue;
3 int const* pPtr = &nValue;
```

const pointer must be initialized to a value at declaration and its value cannot be changed.
The pointer will always point to the same address, but the value being pointed to can be changed.

```
1 *pPointer = 6;
```

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int var1 = 0, var2 = 0;
7      int *const ptr = &var1;
8      ptr = &var2;        //No!
9      *ptr = var2;        //OK
10     cout << *ptr << endl;
11
12     return 0;
13 }
```

Pointers and const

- pointer to a constant

```
1 int nValue = 5;
2 const int *pnPtr = &nValue;
```

a pointer to a constant variable treats the variable as constant when it is accessed through the pointer.
It can be redirected to a different variable.

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int var1 = 0;
7      const int *ptr = &var1;
8      *ptr = 1;          //No!
9      cout << *ptr << endl;
10
11     return 0;
12 }
```

Pointers and const

- const pointer to a const value

```
const int nValue;
const int *const pPointer = &
    nValue;
```

cannot be redirected and cannot change the value it is pointing to.

```
#include <iostream>
using namespace std;

int main()
{
    int var1 = 0,var2 = 0;
    const int *const ptr = &var1;
    *ptr = 1; //No!
    ptr = &var2; //No!
    cout << *ptr << endl;

    return 0;
}
```

Dynamic memory allocation

```
1 int *pPointer = NULL;
2 pPointer = new int;
```

This will allocate memory for an integer and write the address of that allocation into the pPointer. The variable has no identifier. Thus, can be accessed only by the pointer.

```
1 *pPointer = 2;
```

The latter line assigns a value of 2 to the memory allocation the pPointer is pointing to.

```
1 delete pPointer;
```

This frees the memory allocation. pPointer still exists, but we cannot use it, as the memory can be now used by a different program! Using it may modify variables from another application!

```
1 pPointer = NULL;
```

After freeing the memory using `delete`, set the pointer to NULL (or point it to a new memory location).

Dynamic memory allocation example

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int* pAge = NULL;
7      pAge = new int;
8
9      cout << "How old are you?" <<
           endl;
10     cin >> *pAge; //writes users
           input in the memory
           allocation indicated by
           pAge
11
12     cout << "You are " << *pAge <<
           " years old." << endl;
13
14     delete pAge;
15     pAge = NULL;
16
17     return 0;
18 }
```

- After the memory is freed it can be used for another variable (or by a different program).
- This way we control (and reduce) memory use.

Memory leaks

```
1 void doSomething()
2 {
3     int *pnValue = new int;
4 }
```

- the pointer is not deleted at the end of the function, but it still goes out of scope when the function block ends.
- now, there is no reference to the dynamically allocated integer. This is called a **memory leak**.
- now there is no way to access that integer, it cannot be used, changed or deleted. It is just using up memory.
- this could slow down the whole machine or even cause the system to crash.

Dynamic memory allocation and arrays

- The same notation can be used with arrays. In this case the size of the array does not need to be defined as a constant
- The difference is `new[]` and `delete[]` are followed by [ ]
- There is no need to specify the size of the array to be deleted. Compiler know is. Plus it reduces chances of programmer's error.

```
int nSize = 12;
int *pnArray = new int[nSize]; // note: nSize does not need to be constant!
pnArray[4] = 7;
delete[] pnArray;
```

```
int main()
{
    int size;

    cout << "Array size: ";
    cin >> size;

    int* array = new int[size]; // Dynamically allocated "array" of size 'size'

    for(int i = 0; i < size; i++)
    {
        array[i] = i+1;
        cout << array[i] << " ";
    }

    delete [] array;

    return 0;
}
```

Problem 1

The file `grades.dat` contains a header with the number of students followed by a table cointaining the students' ID and point grade on the scale to 100. Write a code that will utilize a dynamic memory allocation to create an array to store the grades. The code should:

- read the grades into an array,
- find the lowest and the higest grade and calculate the average grade for the course,
- return the numbers of students that received an A (above 94 points) and students that failed the course (less than 60 points)

Don't forget to delete the allocated memory when you're done using it! Use `sprintf()` to print out the result to the screen.