Jaap van der Leest
Timon van der Berg
October 21, 2017

C++ Course
Assignment 6

Exercise 42

Problem statement. *What are the variants of new/delete? For each of the variants provide a (short!) example in which the used new/delete is appropriate and provide a short explanation why it is appropriately used.*
Solution.

new/delete

`new` is used to allocate memory for a primitive type or object. When allocating for an object it will call the object constructor. Example of `new` use:

```
int *ptr = new int;
```

This allocation is appropriate since we want to allocate memory for a single int.

`delete` is used to deallocate memory that was allocated using `new`. If called on an object (not a primitive type) it will also call that objects' destructor. Example of `delete` use:

```
std::string *ptr = new std::string;
delete ptr;
```

This is appropriate because `delete` is used on memory allocated using `new`.

new[], delete[]

`new[]` is used to allocate memory for arrays. Like `new` it is type-safe: the type of the element has to be declared. Like `new`, it calls constructors. An example of using `new`:

```
int *aoi = new int[20];
```

`delete[]` is used to delete memory allocated using `new[]`. Unlike `new`, new[] saves the size of the array it allocates. `delete[]` uses this to delete the array. Destructors are called[1]. An example of `delete[]` usage:

```
string *strp = new string[550];
delete[] strp;
```

This is appropriate because `delete[]` is used on an array allocated using `new[]`.

operator new, operator delete

`operator new` is used to allocate raw bytes of memory. To actually use this memory, a static cast is required. An example of using `operator new`:

```
size_t *sp = static_cast<size_t *>(operator new(5 * sizeof(size_t)));
```

[1] If the array contains a primitive type no destructors are called. Therefore an array of pointers require manual destruction of whatever is pointed to.

This is appropriate because `operator new` is used to allocate raw memory. Here we first calculate the number of bytes needed for 5 `size_t` variables. We then allocate the memory. `operator new` does not care for types.

`operator delete` is used to deallocate memory that was allocated using `operator new`. Like `operator new`, `operator delete` does not care for types. Because `operator new` saves the number of bytes allocated, `operator delete` knows how much memory to deallocate. `operator delete` does not call any destructors. An example of using `operator delete`:

```
string *sp = static_cast<string *>(operator new(5 * sizeof(string)));
operator delete(sp)
```

This is appropriate because we are using operator delete to deallocate memory that was allocated using operator new.

placement new

placement `new` is found in `<memory>` and overloads `new`. Placement `new` is used to place objects in previously allocated memory. An example of using placement `new`:

```
string *sp = static_cast<string *>( operator new(15 * sizeof(string)));
new sp string("Donald Knuth");
```

This is appropriate because we are using operator new on memory of the correct type that was previously allocated. We have placed a single string in this memory, leaving room for 14 more.

Exercise 43

Problem statement. *Fix the memory leak in the 'Strings' class.*
Solution. Because our own implementation of 'Strings' was not perfect, we instead modified the official solution provided in the answers of set 5.

strings.h

```
#ifndef INCLUDED_STRINGS_
#define INCLUDED_STRINGS_

#include <iosfwd>

class Strings
{
    size_t d_size;
    std::string *d_str;

    public:
        struct POD
        {
            size_t      size;
            std::string *str;
        };

        Strings();
        ~Strings();
        Strings(int argc, char *argv[]);
        Strings(char *environLike[]);
        Strings(std::istream &in);

        void swap(Strings &other);

        size_t size() const;
        std::string const *data() const;
```

```
28          POD release();
29
30          std::string const &at(size_t idx) const;    // for const-objects
31          std::string &at(size_t idx);                // for non-const objects
32
33          void add(std::string const &next);          // add another element
34
35      private:
36          void fill(char *ntbs[]);                    // fill prepared d_str
37
38          std::string &safeAt(size_t idx) const;      // private backdoor
39          std::string *enlarge();
40          void destroy();
41
42          static size_t count(char *environLike[]);   // # elements in env.like
43
44  };
45
46  inline size_t Strings::size() const         // potentially dangerous practice:
47  {                                           // inline accessors
48      return d_size;
49  }
50
51  inline std::string const *Strings::data() const
52  {
53      return d_str;
54  }
55
56  inline std::string const &Strings::at(size_t idx) const
57  {
58      return safeAt(idx);
59  }
60
61  inline std::string &Strings::at(size_t idx)
62  {
63      return safeAt(idx);
64  }
65
66
67  #endif
```

strings5.cc

```
1   #include "strings.ih"          // using namespace std;
2
3   Strings::~Strings()
4   {
5       delete[] d_str;            // 0 pointer allowed
6   }
```

Exercise 44

Problem statement. *gi* **Solution.** go

Exercise 45

Problem statement. *gi* **Solution.** go

Exercise 46

Problem statement. *gi* **Solution.** go

Exercise 47

Problem statement. *gi* **Solution.** go

Exercise 48

Problem statement. *gi* **Solution.** go