Programming report
Week 3 Assignments C++

Jaime Betancor Valado
Christiaan Steenkist
Remco Bos
Diego Ribas Gomes

October 19, 2016

Assignment 40, Pointer and index-using implementations

We are asked to provide a function called nowhere using pointers and explain of how that function and strcpy function works.

How strcpy works:

To the function a C-string is passed as a pointer src pointing to a const char. A new pointer variable (pointer dst pointing to a char) which is empty as default takes the character where src is pointing to as first. The while loop iterates over all characters where src is pointing at. Both pointer variables are incremented and the characters of src are copied one by one to the dst pointer variable. The incrementation of the pointer accesses the next character in the memory block. So first the first character in memory is copied to the first memory place of dst, then the second, etc. The loop terminates after the end of the C-string is reached, i.e. when 'Ø' is copied to dst.

How nowhere works:

A C-string is passed to the pointer variable src like the strcpy function. And a character(or multiple characters) is chosen by the user to see if it is nowhere in the C-string. This is done by checking if the character in memory given by the user is equal to a character in the C-string. Like strcpy the while loop terminates after 'Ø' is reached, and in this case it also terminates if the character is found, hence it is not nowhere in the C-string. If the character given by the user is nowhere in the C-string, an if-statement is used to check if the 'Ø' is reached, hence end of the C-string and the character is nowhere in the C-string.

Why pointer-implementation is preferred:

The pointer-implementation is preferred, because it is more efficient. If indices are used, then the program first goes to the index and then looks at the memory. A pointer goes directly to the memory and then performs its duty.

Code Listings

Listing 1: `nowhere.cc`

```
void nowhere(char *character, char const *src)
{
    while((*character =! *src++))
    {
        if (*src == '\0')
            {
              std::cout << "Character is nowhere!\n";
            }
      }
}
```

Assignment 41, Some questions

We are asked to comment about variants of new/delete and provide some examples.

1.new:

To allocate primitive types or objects; eg:

Listing 2: `example.cc`

```
    int *v1 = new int;       // an int pointer variable
        //points to memory allocated by operator new
    string *s1 = new string;// a class-type object
        //is allocated
```

2. delete:

To release the memory of a single element allocated using new; eg:

Listing 3: `example.cc`

```
delete v1;
delete s1;
```

3. new[]:

To allocate dynamic arrays, whose lifetime may exceed the lifetime of the function in which they were created; eg:

Listing 4: `example.cc`

```
int *intarr = new int[20];  // allocates 20 ints
string *stringarr = new string[10]; // allocates
            // 10 class-type objects 'string'
```

4. delete[]:

To call the destructor for each element in the array and return the memory pointed at by the pointer to the common pool; eg:

Listing 5: `example.cc`

```
delete[] intarr;
delete[] stringarr;
```

5. operator new(sizeInBytes):

To allocate raw memory, a block of memory for unspecified purpose; eg:

Listing 6: `example.cc`

```
char *chPtr =
static_cast<char *>(operator new(numberOfBytes));
    // the raw memory returned by new is a
    //void *, here assigned to a char * variable
```

6. operator delete:

To return the raw memory allocated by operator new; eg:

Listing 7: `example.cc`

```
    operator delete(chPtr);
```

7. placement new:

To initialize an object or value into an existing block of memory; eg:

Listing 8: `example.cc`

```
    Type *new(void *memory) Type(arguments);
        //memory is a block of memory of at
        //least sizeof(Type) bytes and Type(arguments)
        //is any constructor of the class Type;
```

(Memory allocated this way is returned by explicitly calling the object's destructor)

Assignment 42, Destructors

We are asked to prevent memory leaks in our implemented string class by submitting a destructor.

Code Listings

Listing 9: `strings.h`

```
#ifndef STRINGS_H
#define STRINGS_H

#include<iostream>

class Strings
{
    char *d_str;
    size_t d_size = 0;

    void addCapacity(size_t increment);

    public:
        Strings(size_t argc, char **argv);
        Strings(char **environ);
        ~Strings();

        size_t size();
        char *str();
        char *at(size_t index);
        char const *at(size_t index) const;

        void addString(std::string newString);
        void addString(char *charArray);
        void setSize(size_t size);
        void setStr(char *str);
};

#endif
```

Listing 10: `destructor.c`

```
#include "strings.ih"

Strings::~Strings()
{
    delete[] d_str;
}
```

Assignment 43, Double pointers

We are asked to modify our string class using double pointers because the algorithm of exercie 33 being too inefficient.