Programming report
Week 5 Assignments C++

Jaime Betancor Valado
Christiaan Steenkist
Remco Bos
Diego Ribas Gomes

October 17, 2016

Assignment 40, `strcpy`

We were tasked with comparing a pointer and index implementations of `strcpy`

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 `'\0'` 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 `'\0'` 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 `'\0'` is reached, hence end of the C-string and the character is nowhere in the C-string.

0.1 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: `example.cc`

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

Assignment 41, new and delete

Here are the various new and delete operators that can be used.

0.2 `new`

To allocate primitive types or objects;

```
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
```

0.3 `delete`

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

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

0.4 `new[]`

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

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

0.5 `delete[]`

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

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

0.6 operator new(sizeInBytes)

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

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

0.7 operator delete

To return the raw memory allocated by operator new;

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

0.8 placement new

To initialize an object or value into an existing block of memory; (Memory allocated this way is returned by explicitly calling the object's destructor)

```
  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;
```

Assignment 42, Strings destructor

We were tasked with making a destructor for the Strings class.

Code listings

Listing 2: 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 3: `destructor.cc`

```
#include "strings.ih"

void ~Strings::Strings()
{
  delete d_str;
}
```

Assignment 43, double pointers

We were tasked with changing the `Strings` from single pointer to double pointer.

Code listings

Listing 4: `strings.h`

```
#ifndef STRINGS_H
#define STRINGS_H

#include<iostream>

class Strings
{
  char **d_str = new char*[1];
  size_t d_size = 0;
  size_t d_capacity = 1;

  void reserve(size_t size);

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

    void setStr(char **str);
    void setSize(size_t size);
    void resize(size_t size);

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

    void addString(std::string newString);
    void addString(char *charArray);
    size_t calcCapacity(size_t size);
};

#endif
```

Listing 5: `at1.cc`

```
#include "strings.ih"

char *Strings::at(size_t index)
{
  return d_str[index];
}
```

Listing 6: `at2.cc`

```
#include "strings.ih"

char const *Strings::at(size_t index) const
{
  return d_str[index];
}
```

Listing 7: `addstring1.cc`

```
#include "strings.ih"

void Strings::addString(string newString)
{
  reserve(d_size + 1);
  d_str[d_size] = &newString[0];
  ++d_size;
}
```

Listing 8: `addstring2.cc`

```
#include "strings.ih"
```

```
void Strings::addString(char *charArray)
{
  reserve(d_size + 1);
  d_str[d_size] = charArray;
  ++d_size;
}
```

Listing 9: `capacity.cc`

```
#include "strings.ih"

size_t Strings::capacity()
{
  return d_capacity;
}
```

Listing 10: `destructor.cc`

```
#include "strings.ih"

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

Listing 11: `reserve.cc`

```
#include "strings.ih"

// Only reserves if the requested size exceeds the capacity.
void Strings::reserve(size_t size)
{
  if (size > d_capacity)
  {
    size = calcCapacity(size);
    resize(size);

    d_capacity = size;
  }
}
```

Listing 12: `resize.cc`

```
#include "strings.ih"

// If the array shrinks then the strings that fall
// outside of the array will be ignored.
void Strings::resize(size_t size)
{
  if (size > d_capacity)
  {
    char **n_str = new char*[size];
    for (size_t index = 0; index != d_size; ++index)
      n_str[index] = d_str[index];

    delete[] d_str;
    d_str = n_str;
  }
}
```

Assignment 44, placement new

We were tasked with setting up the `Strings` class using placement new.

Code listings

Listing 13: `strings.h`

```
#ifndef STRINGS_H
#define STRINGS_H

#include<iostream>

class Strings
{
  char **d_str = static_cast<char **>(
          operator new(sizeof(char *)));
  size_t d_size = 0;
  size_t d_capacity = 1;

  void reserve(size_t size);

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

    char **str();
    size_t size();
    size_t capacity();

    void setStr(char **str);
    void setSize(size_t size);
    void resize(size_t size);

    char *at(size_t index);
    char const *at(size_t index) const;
    void addString(std::string newString);
    void addString(char *charArray);
    size_t calcCapacity(size_t size);
};

#endif
```

Listing 14: at1.cc

```
#include "strings.ih"

char *Strings::at(size_t index)
{
  return d_str[index];
}
```

Listing 15: at2.cc

```
#include "strings.ih"

char const *Strings::at(size_t index) const
{
  return d_str[index];
}
```

Listing 16: addstring1.cc

```
#include "strings.ih"

void Strings::addString(string newString)
{
  reserve(d_size + 1);
  d_str[d_size] = &newString[0];
  ++d_size;
}
```

Listing 17: addstring2.cc

```
#include "strings.ih"

void Strings::addString(char *charArray)
{
  reserve(d_size + 1);
  d_str[d_size] = charArray;
  ++d_size;
}
```

Listing 18: capacity.cc

```
#include "strings.ih"
```

```
size_t Strings::capacity()
{
  return d_capacity;
}
```

Listing 19: `destructor.cc`

```
#include "strings.ih"

Strings::~Strings()
{
  operator delete(d_str);
}
```

Listing 20: `reserve.cc`

```
#include "strings.ih"

// Only reserves if the requested size exceeds the capacity.
void Strings::reserve(size_t size)
{
  if (size > d_capacity)
  {
    size = calcCapacity(size);
    resize(size);

    d_capacity = size;
  }
}
```

Listing 21: `resize.cc`

```
#include "strings.ih"

// If the array shrinks then the strings that fall
// outside of the array will be ignored.
void Strings::resize(size_t size)
{
  if (size > d_capacity)
  {
    char **n_str = static_cast<char **>(
            operator new(size * sizeof(char *)));

    for (size_t index = 0; index != d_size; ++index)
      n_str[index] = d_str[index];

    operator delete(d_str);
    d_str = n_str;
  }
}
```