Programming report
Week 5 Assignments C++

Jaime Betancor Valado
Christiaan Steenkist
Remco Bos
Diego Ribas Gomes

October 19, 2016

Assignment 40, `strcpy`

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

How pointer implementation works

To the function a C-string is passed as a pointer`src` pointing to a const char. The `src` pointer assigns the character at the current incrementation to the same incrementation of `dst` (i.e. first element where `src` is pointing at is assigned to the first element where `dst` is pointing at, the next incrementation copies the second element, etc). The loop terminates after the end of the C-string is reached, i.e. when `'\0'`.

How index-using implementation works

An empty char array `dst` must be provided where the `src` char array can copy its elements to. In reality the arrays are pointers to a memory location where the characters are stored(`src`) or gets copied to(`dst`). The incrementation variable is instantiated as 0 to start at the first element. In the while-loop the first element of `dst` is assigned the value of the first element of `src`. Then the loop increments the index variable to copy the other elements to `dst`. Just like the pointer version, the while-loop terminates when the `'\0'` character is reached.

0.1 Why pointers are preferred

It takes less code to do the pointer version, so it is also easier to read the code. It is faster and more efficient in memory, because only a memory address has to be provided instead of a whole array. This also gives pointers the ability to allocate memory dynamically, because it uses the elements at the address one by one. If an array is used, than all its elements have to be provided, then one by one the elements are copied(`src`) to the other array(`dst`) which is empty at default before copying. So the array is copied and memory must be allocated for it. The pointer version points at the element(`src`) in memory and assigns it to the memory position of the other pointer(`dst`). So using an array and iterating over its elements is slower and less efficient than using a pointer that take the element directly from memory.

Code listings

Listing 1: `example.cc`

```
void strcpy(char dst[], char const src[])
{
    int index = 0;
    while((dst[index] = src[index]))
    {
        index++;
    }
}
```

Assignment 41, new and delete

Here are the variants of new/delete.

1 new/delete

To allocate primitive types or objects; eg:

```
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
```

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

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

2 new[]/delete[]

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

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

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

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

3 operator new(sizeInBytes)/operator delete

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

```
// the raw memory returned by new is a void *, here assigned to a
  string * variable
string *stPtr = static_cast<string *>(operator new(10 * sizeof(
 std::string)));
```

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

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

4 placement new

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

```
//installs a string into the block of memory created in the
 example from section 3.1
string *sp = new(chPtr) std::string("example");
```

The destructor has to be called explicitly to destroy the object, followed by operator delete to deallocate the memory as in section 3.2, eg:

```
sp->~string();
```

Assignment 42, `Strings` destructor

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

Code listings

Listing 2: `strings.h`

```
#ifndef STRINGS_H
#define STRINGS_H

#include <cstddef>
#include <string>

class Strings
{
  std::string *d_str = new std::string[1];
  size_t d_size = 0;

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

    size_t size();

    void addString(char *charArray);

    std::string at(size_t index);
    std::string const at(size_t index) const;
```

```
    std::string *rawStrings(size_t amount);
};

#endif
```

Listing 3: `destructor.cc`

```
#include "strings.ih"

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

Assignment 43, double pointers

We were tasked with changing the `Strings` from single pointer to double pointer. The class now uses a double pointer to store pointers to the strings which is doubled each time the amount of strings exceeds the capacity.

Code listings

Listing 4: `strings.h`

```
#ifndef STRINGS_H
#define STRINGS_H

#include <cstddef>
#include <string>

class Strings
{
  std::string **d_str = new std::string*[1];
  size_t d_size = 0;
  size_t d_capacity = 1;

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

    size_t size();
    size_t capacity();

    void resize(size_t size);
    void reserve(size_t size);

    void addString(char *charArray);
```

```
    std::string at(size_t index);
    std::string const at(size_t index) const;

    std::string **rawPointers(size_t amount);
};

#endif
```

Listing 5: `at1.cc`

```
#include "strings.ih"

string Strings::at(size_t index)
{
  if (index > d_size)
    return " ";

  return *d_str[index];
}
```

Listing 6: `at2.cc`

```
#include "strings.ih"

string const Strings::at(size_t index) const
{
  if (index > d_size)
    return " ";

  return *d_str[index];
}
```

Listing 7: `addstring.cc`

```
#include "strings.ih"

void Strings::addString(char *charArray)
{
  reserve(d_size + 1);

  d_str[d_size] = new string;
  *d_str[d_size] = string(charArray);

  ++d_size;
}
```

Listing 8: `capacity.cc`

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

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

Listing 9: `destructor.cc`

```
#include "strings.ih"

Strings::~Strings()
{
  for (size_t index = 0; index != d_size; ++index)
    delete d_str[index];
  delete[] d_str;
}
```

Listing 10: `reserve.cc`

```
#include "strings.ih"

// Only reserves if the requested size exceeds the capacity.
void Strings::reserve(size_t size)
{
  if (size > d_capacity)
  {
    // Get the next power of two for the capacity.
    size = ceil(log2(size));
    size = 1 << size;

    string **raw = rawPointers(size);
    for (size_t index = 0; index != d_size; ++index)
      raw[index] = d_str[index];

    delete[] d_str;

    d_str = raw;
    d_capacity = size;
  }
}
```

Listing 11: `resize.cc`

```
#include "strings.ih"

// If the array shrinks then the strings that fall
// outside of the array will be removed
void Strings::resize(size_t size)
{
```

```
  if (size > d_capacity)
  {
    reserve(size);
    for (size_t index = d_size; index != size; ++size)
      d_str[index] = new string;
  }
  else
  {
    for (size_t index = size; index != d_size; ++size)
    {
      delete d_str[index];
      d_str[index] = new string;
    }
    d_size = size;
  }
}
```

Listing 12: `rawpointers.cc`

```
#include "strings.ih"

string **Strings::rawPointers(size_t amount)
{
  string **ptr = new string*[amount];
  return ptr;
}
```

Assignment 44, placement new

We were tasked with setting up the `Strings` class using placement new. Just as in assignment 43 the array is doubled each time the amount of strings exceeds the capacity. This implementation uses a single pointer variable and the same classes as the previous exercise but altered for placement new and single pointers.

Code listings

Listing 13: `strings.h`

```
#ifndef STRINGS_H
#define STRINGS_H

#include <cstddef>
#include <string>

class Strings
{
  std::string *d_str = static_cast<std::string *>(
          operator new(1 * sizeof(std::string)));
```

```
  size_t d_size = 0;
  size_t d_capacity = 1;

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

    size_t size();
    size_t capacity();

    void resize(size_t size);
    void reserve(size_t size);

    void addString(char *charArray);

    std::string at(size_t index);
    std::string const at(size_t index) const;

    std::string *rawStrings(size_t amount);
};

#endif
```

Listing 14: `at1.cc`

```
#include "strings.ih"

string Strings::at(size_t index)
{
  if (index > d_size)
    return " ";

  return d_str[index];
}
```

Listing 15: `at2.cc`

```
#include "strings.ih"

string const Strings::at(size_t index) const
{
  if (index > d_size)
    return " ";

  return d_str[index];
}
```

Listing 16: addstring.cc

```
#include "strings.ih"

void Strings::addString(char *charArray)
{
  reserve(d_size + 1);

  new(d_str + d_size) string(charArray);

  ++d_size;
}
```

Listing 17: capacity.cc

```
#include "strings.ih"

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

Listing 18: destructor.cc

```
#include "strings.ih"

Strings::~Strings()
{
  for (size_t index = 0; index != d_size; ++index)
  {
    d_str[index].~string();
  }
  operator delete(d_str);
}
```

Listing 19: reserve.cc

```
#include "strings.ih"

// Only reserves if the requested size exceeds the capacity.
void Strings::reserve(size_t size)
{
  if (size > d_capacity)
  {
    // Get the next power of two for the capacity.
    size = ceil(log2(size));
    size = 1 << size;

    string *raw = rawStrings(size);
    for (size_t index = 0; index != d_size; ++index)
```

```
    {
      new(raw + index) string(d_str[index]);
      d_str[index].~string();
    }

    operator delete(d_str);

    d_str = raw;
    d_capacity = size;
  }
}
```

Listing 20: `resize.cc`

```
#include "strings.ih"

// If the array shrinks then the strings that fall
// outside of the array will be removed
void Strings::resize(size_t size)
{
  if (size > d_capacity)
  {
    reserve(size);
    for (size_t index = d_size; index != size; ++size)
      new(d_str + index) string();
  }
  else
  {
    for (size_t index = size; index != d_size; ++size)
    {
      string *ptr = d_str + index;
      ptr->~string();
    }
    d_size = size;
  }
}
```

Listing 21: `rawpointers.cc`

```
#include "strings.ih"

string *Strings::rawStrings(size_t amount)
{
  string *ptr = static_cast<string *>(
          operator new(amount * sizeof(std::string)));
  return ptr;
}
```