C++ Introduction

David Gmeindl

July 25, 2018

Contents

1 Style

1.1 Naming

A complete definition of TCA naming conventions is available at `https://192.168.1.221/references/tca-code-guidelines`. It is a subset of the C++-Google Code-Guidelines and differs slightly.

Here the most important should be listed:

naming conventions

1.2 DO's

- use std libraries where-ever possible (known and TESTED behavior, programmed to be efficient)
-

1.3 DONT's

-

2 Constructors & intitializations

2.1 Initialization

- initialize everything with reasonable values
- some std classes do not need to be initialized specifically to have default values

  (i.e. `std::string s; // initializes empty string automatically`)

2.2 Constructors & Destructors

Constructors

- use the constructor of classes to initialize values if possible
- use the constructor initializer lists

```
1  Auto::Auto(double power, double speed, std::string name)
2   : power_(power)
3   , speed_(speed)
4   , name_(name)
5  {}
```

- in initializer lists the order is important
- in C++ 11 initialization of standard types can be done in the header file:

```
class Auto {
private:
    double power_ = 0.0;
    double speed_ = 0.0;
    std::string name_ = "Audi R8";
}
```

- if no constructor is defined, the compiler provides a standard constructor
- for container types (`http://www.cplusplus.com/reference/stl/`), a standard constructor has to be defined. The following would not work:

```
#include <vector>
#include "ConstructorClass.h"

int main()
{
    std::vector<ConstructorClass> vec;
    vec.push_back(ConstructorClass());
    return 0;
}
```

```
#ifndef CONSTRUCTORCLASS_H
#define CONSTRUCTORCLASS_H

class ConstructorClass
{

public:
    ConstructorClass(int a, int b)
        : a_(a)
        , b_(b)
    {}

private:
    int a_;
    int b_;
};

#endif
```

- as soon as dynamic types (pointers) are part of a class, a self-written constructor, copy-constructor & destructor is advised to handle the pointer data correctly
- special types of constructors:
  - copy constructor
  - move constructor
- see rule of three below (section 6.2)

Destructors

- **NOTE:** A destructor should be defined as soon as dynamic types are part of the class.

3 Classes vs Data-Containers

4 Pointers

Example 1 - Memory Leak

```
#ifndef MYCLASS_H
#define MYCLASS_H

#include <string>

class MyClass {
public:
    MyClass();

    std::string getString() const;
    const std::string *getStringPt() const;

private:
    std::string *my_string_;

};

#endif // MYCLASS_H
```

```
#include "MyClass.h"

MyClass::MyClass()
{
    my_string_ = new std::string("abc");
}

std::string MyClass::getString() const
{
    return *my_string_;
}

const std::string *MyClass::getStringPt()
↪ const
{
    return my_string_;
}
```

Listing 1: Example 1: MyClass.h & MyClass.cpp

A pointer in `MyClass` is created in the constructor (`MyClass::MyClass()`), but never deleted; as soon as the destructor of `MyClass` is called (see scope in main), the pointer has to be deleted, but this never happens. On the contrary, if the std::string pointer is deleted in the destructor, p outside the limited scope (line 13) cannot be accessed anymore (see next example).

```
#include <iostream>

#include "MyClass.h"

int main()
{
    const std::string *p;
    {
        MyClass c;
        std::cout << c.getString() << std::endl;
        p = c.getStringPt();
    }
    std::cout << *p << std::endl;
    return 0;
}
```

```
valgrind --tool=memcheck --leak-check=full ./main

==1485== HEAP SUMMARY:
==1485==     in use at exit: 32 bytes in 1 blocks
==1485==   total heap usage: 3 allocs, 2 frees, 73,760 bytes allocated
==1485==
==1485== 32 bytes in 1 blocks are definitely lost in loss record 1 of 1
==1485==    at 0x4C3017F: operator new(unsigned long) (vg_replace_malloc.c:334)
==1485==    by 0x108E9F: MyClass::MyClass() (in
↪ /home/david/projects/cpp-intro/src/pointers/ex-1/main)
==1485==    by 0x108D4D: main (in /home/david/projects/cpp-intro/src/pointers/ex-1/main)
==1485==
==1485== LEAK SUMMARY:
==1485==    definitely lost: 32 bytes in 1 blocks
==1485==    indirectly lost: 0 bytes in 0 blocks
==1485==      possibly lost: 0 bytes in 0 blocks
```

```
==1485==    still reachable: 0 bytes in 0 blocks
==1485==         suppressed: 0 bytes in 0 blocks
```

Example 2 - Segmentation Fault

To produce no memory leaks, the allocated memory for `std::string` is cleaned in the destructor of `MyClass`. However, a segfault is created, as soon as the pointer points to memory, where no data is assigned anymore (see example below);

```
#ifndef MYCLASS_H
#define MYCLASS_H

#include <string>

class MyClass {
public:
    MyClass();
    ~MyClass();

    std::string getString() const;
    const std::string *getStringPt() const;

private:
    std::string *my_string_;

};

#endif // MYCLASS_H
```

```
#include "MyClass.h"

MyClass::MyClass()
{
    my_string_ = new std::string("abc");
}


MyClass::~MyClass()
{
    if(my_string_)
        delete my_string_;
}


std::string MyClass::getString() const
{
    return *my_string_;
}

const std::string *MyClass::getStringPt()
↪ const
{
    return my_string_;
}
```

Listing 2: Example 2: MyClass.h & MyClass.cpp

```
#include <iostream>

#include "MyClass.h"

int main()
{

    const std::string *p;
    {
        MyClass c;
        std::cout << c.getString() << std::endl;
        p = c.getStringPt();
    }
    std::cout << *p << std::endl;
    return 0;
}
```

```
david@david-pc:~/projects/cpp-intro/src/pointers/ex-2 $ ./main
abc
Segmentation fault (core dumped)
```

5 Smart Pointers

6 Programming idioms

6.1 Resource acquisition is initialization (RAII)

https://en.wikipedia.org/wiki/Resource_acquisition_is_initialization

6.2 Rule of three (five, zero)

https://en.cppreference.com/w/cpp/language/rule_of_three

7 Std libraries

http://www.cplusplus.com/reference/std/

algorithm

i.e. find_if

```cpp
#include <iostream>
#include <algorithm>
#include <vector>

int main() {
    std::vector<int> myvector = {10, 25, 40, 55};

    std::vector<int>::iterator it =
        std::find_if (myvector.begin(), myvector.end(),
                      [](const int &it){
                        return (it%2) == 1;
        });
    std::cout << "The first odd value is " << *it << '\n';

    return 0;
}
```

i.e. unique

```cpp
// unique algorithm example
#include <iostream>     // std::cout
#include <algorithm>    // std::unique, std::distance
#include <vector>       // std::vector

bool myfunction (int i, int j) {
  return (i==j);
}

int main () {
  int myints[] = {10,20,20,20,30,30,20,20,10};           // 10 20 20 20 30 30 20 20 10
  std::vector<int> myvector (myints,myints+9);

  // using default comparison:
  std::vector<int>::iterator it;
  it = std::unique (myvector.begin(), myvector.end());   // 10 20 30 20 10 ?  ?  ?  ?
                                                         //                ^
```

```
18
19    myvector.resize( std::distance(myvector.begin(),it) ); // 10 20 30 20 10
20
21    // using predicate comparison:
22    std::unique (myvector.begin(), myvector.end(), myfunction);   // (no changes)
23
24    // print out content:
25    std::cout << "myvector contains:";
26    for (it=myvector.begin(); it!=myvector.end(); ++it)
27      std::cout << ' ' << *it;
28    std::cout << '\n';
29
30    return 0;
31  }
```