

# C++ Guidelines

David Gmeindl

July 25, 2018

## Contents

<b>1</b>	<b>Style</b>	<b>2</b>
1.1	Naming . . . . .	2
1.2	DO's . . . . .	2
1.3	DONT's . . . . .	2
<b>2</b>	<b>Constructors &amp; initializations</b>	<b>2</b>
2.1	Initialization . . . . .	2
2.2	Constructors & Destructors . . . . .	2
<b>3</b>	<b>Classes vs Data-Containers</b>	<b>3</b>
<b>4</b>	<b>Pointers</b>	<b>4</b>
<b>5</b>	<b>Smart Pointers</b>	<b>6</b>
<b>6</b>	<b>Programming idioms</b>	<b>6</b>
6.1	Resource acquisition is initialization (RAII) . . . . .	6
6.2	Rule of three (five, zero) . . . . .	6
<b>7</b>	<b>Std libraries</b>	<b>6</b>

# 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 & initializations

## 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:

---

```

1  class Auto {
2  private:
3      double power_ = 0.0;
4      double speed_ = 0.0;
5      std::string name_ = "Audi R8";
6  }

```

---

- 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:

---

```

1  #include <vector>
2  #include "ConstructorClass.h"
3
4  int main()
5  {
6      std::vector<ConstructorClass> vec;
7      vec.push_back(ConstructorClass());
8      return 0;
9  }

```

---



---

```

1  #ifndef CONSTRUCTORCLASS_H
2  #define CONSTRUCTORCLASS_H
3
4  class ConstructorClass
5  {
6
7  public:
8      ConstructorClass(int a, int b)
9          : a_(a)
10           , b_(b)
11      {}
12
13  private:
14      int a_;
15      int b_;
16  };
17
18  #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

<pre> 1  #ifndef MYCLASS_H 2  #define MYCLASS_H 3 4  #include &lt;string&gt; 5 6  class MyClass { 7  public: 8      MyClass(); 9 10     std::string getString() const; 11     const std::string *getStringPt() const; 12 13 private: 14     std::string *my_string_; 15 }; 16 17 #endif // MYCLASS_H </pre>	<pre> 1  #include "MyClass.h" 2 3  MyClass::MyClass() 4  { 5      my_string_ = new std::string("abc"); 6  } 7 8  std::string MyClass::getString() const 9  { 10     return *my_string_; 11 } 12 13 const std::string *MyClass::getStringPt() 14 ↪ const 15 { 16     return my_string_; 17 } </pre>
---	--

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).

```

1  #include <iostream>
2
3  #include "MyClass.h"
4
5  int main()
6  {
7      const std::string *p;
8      {
9          MyClass c;
10         std::cout << c.getString() << std::endl;
11         p = c.getStringPt();
12     }
13     std::cout << *p << std::endl;
14     return 0;
15 }

```

---

```

1  valgrind --tool=memcheck --leak-check=full ./main
2
3  ==1485== HEAP SUMMARY:
4  ==1485==    in use at exit: 32 bytes in 1 blocks
5  ==1485== total heap usage: 3 allocs, 2 frees, 73,760 bytes allocated
6  ==1485==
7  ==1485== 32 bytes in 1 blocks are definitely lost in loss record 1 of 1
8  ==1485==    at 0x4C3017F: operator new(unsigned long) (vg_replace_malloc.c:334)
9  ==1485==    by 0x108E9F: MyClass::MyClass() (in
10 ↪ /home/david/projects/cpp-intro/src/pointers/ex-1/main)
11 ==1485==    by 0x108D4D: main (in /home/david/projects/cpp-intro/src/pointers/ex-1/main)
12 ==1485== LEAK SUMMARY:
13 ==1485==    definitely lost: 32 bytes in 1 blocks
14 ==1485==    indirectly lost: 0 bytes in 0 blocks
15 ==1485==    possibly lost: 0 bytes in 0 blocks

```

```

16 ==1485==      still reachable: 0 bytes in 0 blocks
17 ==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);

<pre> 1  #ifndef MYCLASS_H 2  #define MYCLASS_H 3 4  #include &lt;string&gt; 5 6  class MyClass { 7  public: 8      MyClass(); 9      ~MyClass(); 10 11     std::string getString() const; 12     const std::string *getStringPt() const; 13 14 private: 15     std::string *my_string_; 16 17 }; 18 19 #endif // MYCLASS_H </pre>	<pre> 1  #include "MyClass.h" 2 3  MyClass::MyClass() 4  { 5      my_string_ = new std::string("abc"); 6  } 7 8 9  MyClass::~MyClass() 10 { 11     if(my_string_) 12         delete my_string_; 13 } 14 15 16 std::string MyClass::getString() const 17 { 18     return *my_string_; 19 } 20 21 const std::string *MyClass::getStringPt() 22 ↪ const 23 { 24     return my_string_; 25 } </pre>
--	---

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

```

1  #include <iostream>
2
3  #include "MyClass.h"
4
5  int main()
6  {
7
8      const std::string *p;
9      {
10         MyClass c;
11         std::cout << c.getString() << std::endl;
12         p = c.getStringPt();
13     }
14     std::cout << *p << std::endl;
15     return 0;
16 }

```

```

1 david@david-pc:~/projects/cpp-intro/src/pointers/ex-2 $ ./main
2 abc
3 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](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](https://en.cppreference.com/w/cpp/language/rule_of_three)

## 7 Std libraries

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

algorithm

i.e. find\_if

---

```

1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4
5  int main() {
6      std::vector<int> myvector = {10, 25, 40, 55};
7
8      std::vector<int>::iterator it =
9          std::find_if (myvector.begin(), myvector.end(),
10                      [](const int &it){
11                          return (it%2) == 1;
12                      });
13      std::cout << "The first odd value is " << *it << '\n';
14
15      return 0;
16  }
```

---

i.e. unique

---

```

1  // unique algorithm example
2  #include <iostream>      // std::cout
3  #include <algorithm>      // std::unique, std::distance
4  #include <vector>         // std::vector
5
6  bool myfunction (int i, int j) {
7      return (i==j);
8  }
9
10 int main () {
11     int myints[] = {10,20,20,20,30,30,20,20,10};           // 10 20 20 20 30 30 20 20 10
12     std::vector<int> myvector (myints,myints+9);
13
14     // using default comparison:
15     std::vector<int>::iterator it;
16     it = std::unique (myvector.begin(), myvector.end());    // 10 20 30 20 10 ? ? ? ?
17                                                             //                ^
```

---

```
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 }
```

---