C++ Guidelines

David Gmeindl

July 25, 2018

Contents

1 Style & Guidelines

A complete definition of TCA naming conventions and code-guidelines 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:

Scoping

1. Namespaces:
   - With few exceptions, place code in a namespace. Namespaces should have unique names based on the project name, and possibly its path.
   - Namespaces subdivide the global scope into distinct, named scopes, and so are useful for preventing name collisions in the global scope.
2. Local Variables:
   - Place a function's variables in the narrowest scope possible, and initialize variables in the declaration.

```
int i;
i = f();      // Bad -- initialization separate from declaration.
```

```
int j = g();  // Good -- declaration has initialization.
```

```
// bad
std::vector<int> v;
v.push_back(1);  // Prefer initializing using brace initialization.
v.push_back(2);
```

```
// good
std::vector<int> v = {1, 2};  // Good -- v starts initialized.
```

Classes

1. Implicit Conversions:
   - Do not define implicit conversions. Use the explicit keyword for conversion operators and single-argument constructors.

naming conventions

1.1 DO's

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

1.2 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

- avoid initialization that can fail if you can't signal an error:

  There is no easy way for constructors to signal errors, short of crashing the program (not always appropriate) or using exceptions.
- use the constructor of classes to initialize values if possible
- use the constructor initializer lists

```
Auto::Auto(double power, double speed, std::string name)
 : power_(power)
 , speed_(speed)
 , name_(name)
{}
```

- 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:

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

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

```cpp
#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

- use a struct only for passive objects that carry data
- everything else is a class
- `structs` should be used for passive objects that carry data, and may have associated constants, but lack any functionality other than access/setting the data members. The accessing/setting of fields is done by directly accessing the fields rather than through method invocations. Methods should not provide behavior but should only be used to set up the data members, e.g., constructor, destructor, initialize(), reset(), validate().
- If more functionality is required, a `class` is more appropriate.
- Note: naming conventions of members are different (_ after private member in class)
- Classes should always encapsulate as much data as needed to ease the handling with the class from the outside: clean interfaces

Examples

```cpp
struct Saturation
{
    bool active      = false;
    double threshold = 1.0e99;

    inline bool operator==(const Saturation &rhs) const
    {
        return ((active == rhs.active) && (threshold == rhs.threshold));
    }

    inline bool operator!=(const Saturation &rhs) const { return !operator==(rhs); }
};
```

Listing 1: Data container struct

```cpp
class ReceiverPath
{
  public:
    ReceiverPath();

    ReceiverPath(data::ReceiverFlag receiver_index,
                 std::vector<tcautils::EllipsoidState<tcautils::DateTime>> nodes);

    ReceiverPath(data::ReceiverFlag receiver_index,
                 std::vector<tcautils::EllipsoidState<double>> nodes,
                 tcautils::DateTime start_time);
    virtual ~ReceiverPath();

    const std::vector<tcautils::EllipsoidState<tcautils::DateTime>> &getNodes() const;
    void initialize();
    data::ReceiverState *getReceiverState(tcautils::DateTime epoch);

  private:
    data::ReceiverFlag receiver_index_;
    std::vector<tcautils::EllipsoidState<tcautils::DateTime>> nodes_;
    std::map<tcautils::DateTime, ReceiverPathData *> receiver_path_data_;
    tcautils::EarthRotation earth_rotation_;
    const ReceiverPathData *findNearestNode(const tcautils::DateTime &epoch) const;
};
```

Listing 2: Class

4 Pointers

Example 1 - Memory Leak

```cpp
#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
```

```cpp
#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 3: 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).

```cpp
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 }
```

```
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);

```cpp
#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
```

```cpp
#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 4: Example 2: MyClass.h & MyClass.cpp

```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 ?  ?  ?  ?
                                                         //                ^

  myvector.resize( std::distance(myvector.begin(),it) ); // 10 20 30 20 10

  // using predicate comparison:
  std::unique (myvector.begin(), myvector.end(), myfunction);   // (no changes)

  // print out content:
  std::cout << "myvector contains:";
  for (it=myvector.begin(); it!=myvector.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```