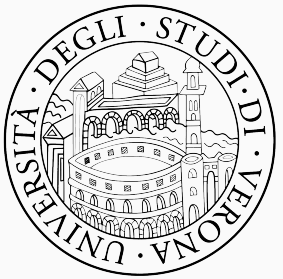
Modern C++ Programming

10. Code Conventions

Federico Busato

University of Verona, Dept. of Computer Science
2018, v1.0



Agenda

- **Basic Concepts**
  - Translation Unit
  - Linkage
  - Global and local scope
- **Variables Storage**
  - Storage class specifiers
  - Storage duration
- **Dealing with Multiple Files**
  - One definition rule
  - Limit template instantiations
- **Namespace**
  - One definition rule
  - Namespace alias
  - Inline namespace
  - Anonymous namespace
- **C++ Project Organization**
  - Project Files
  - Include and library
- **Coding Style and Conventions**
  - File names and spacing
  - `#include`
  - Namespace
  - Variables
  - Functions
  - Structs and Classes
  - C++11/C++14 features
  - Control Flow
  - Entity names
  - Issues

Coding Styles and Conventions

Most important rule:

BE CONSISTENT!!

“The best code explains itself”

GOOGLE

Coding Styles

Coding styles are common guidelines to improve the readability, prevent common errors, and make the code more uniform

Most popular coding styles:

- *LLVM Coding Standards*
  `llvm.org/docs/CodingStandards.html`
- *Google C++ Style Guide*
  `google.github.io/styleguide/cppguide.html`

File names and Spacing

File names:

- Lowercase Underscore (`my_file`) — Google
- Camel UpperCase (`MyFile`) — LLVM

Spacing:

※ Never use tab — LLVM, Google,
  - tab → 2 spaces — Google
  - tab → 4 spaces — LLVM

※ Separate commands, operators, etc., by a space (Google, LLVM)

```
if(a*b<10&&c)          // wrong!!
if (a * c < 10 && c)   // correct
```

※ Line length (width) should be at most **80 characters** long (help code view on a terminal) — LLVM, Google

Order of #include — LLVM, Google

(1) Class header (it is only one)

(2) Local project includes (in alphetical order)

(3) System includes (in alphetical order)

System includes are self-contained, local includes might not

Project includes — LLVM, Google

- should be indicated with "" syntax
- should be absolute paths from the project include root
  e.g. `#include "directory1/header.hpp"`

System includes — LLVM, Google

- should be indicated with <> syntax
  e.g. `#include <iostream>`

- **Use only necessary includes**
- **Include as less as possible, especially in header files**
- **Every includes must be self-contained** (the project must compile with every include order)
- **Report at least one function used for each include**

  `<iostream>  // std::cout, std::cin`
- **Use C++ headers instead of C headers:**

  `<cassert>` instead of `<assert.h>`

  `<cmath>` instead of `<math.h>`, etc.

Example:

```cpp
#include "MyClass.hpp"              // MyClass
#include "my_dir/my_headerA.hpp"    // npA::ClassA, npB::f2()
#include "my_dir/my_headerB.hpp"    // np::g()
#include <iostream>                 // std::cout
#include <cmath>                    // std::fabs()
#include <vector>                   // std::vector
```

Namespaces

Namespace guidelines:

- Avoid `using`-directives at global scope — LLVM, Google
- Limit `using`-directives at local scope and prefer explicit namespace specification — Google
- Always place code in a namespace — Google
- Avoid *anonymous* namespaces in headers

Style guidelines:

- The contents of namespaces are not indented — Google
- Close namespace declarations with `} // namespace <namespace_identifier>` — LLVM
- Close anonymous namespace declarations with `} // namespace anonymous`

Variables

- Avoid static and global variables — LLVM, Google
- Prefer variable/iterator preincrement — LLVM, Google
- Place a variables in the narrowest scope possible, and initialize variables in the declaration — Google, Isocpp
- Declaration of pointer variables or arguments may be placed with the asterisk *adjacent* to either the *type* or to the variable *name* for all in the same way

  `char* c;` `char *c;` — Google
- Use fixed-width integer type (e.g. `int64_t`) — Google
- Use brace initialization to convert arithmetic types (narrowing) e.g. `int64_t{x}` — Google

Functions

Code guidelines:

- *Do not return pointers to local initialized heap memory!*
- Prefer return values rather than output parameters GOOGLE
- Limit overloaded functions GOOGLE
- Default arguments are allowed only on *non*-virtual functions GOOGLE

Style guidelines:

- All parameters should be aligned if possible (especially in the declaration) GOOGLE

```
void f(int         a,
       const int* b);
```

- Parameter names should be the same for declaration and definition
- Do not use `inline` when declaring a function (only in the definition → `.i.hpp` files) LLVM

Code guidelines:

- Use a `struct` only for passive objects that carry data; everything else is a `class` LLVM, Google
- Objects that are fully initialized by constructor call Google
- Avoid multiple inheritance Google
- *Do not return pointers to local initialized heap memory!*

Minors:

- Use braced initializer lists for aggregate types `A{1, 2};` LLVM, Google
- Do not use braced initializer lists for constructors LLVM
- Do not define implicit conversions. Use the `explicit` keyword for conversion operators and single-argument constructors Google

Style guidelines:

- Class inheritance declarations order:
  `public`, `protected`, `private` GOOGLE
- First data members, then function members
- Declare class data members in special way*. Examples:
  - Trailing underscore (e.g. `member_var_`) GOOGLE
  - Leading underscore (e.g. `_member_var`) EDALAB, .NET
  - Public members (e.g. `m_member_var`)
- **Do not use '`this->`' keyword**

*
- It helps to keep track of class variables and local function variables
- The first character is helpful in filtering through the list of available variables

```
struct A {          // passive data structure
    int   x;
    float y;
};

class B {
public:
    B();
    void public_function();

protected:
    int  _a;                      // in general, it is not visible in
                                  // derived classes
    void _protected_function(); // "protected_function()" is not wrong
                                  // it may be public in derived classes
private:
    int   _x;
    float _y;

    void _private_function();
};
```

Use C++11/C++14 features where possible

- Use *constant expressions* instead *macros* GOOGLE
- `static_cast` `reiterpreter_cast` instead *old style cast* `(type)` GOOGLE
- Use *range-for* loops whatever possible LLVM
- Use `auto` type deduction to make the code more readable

```
auto array = new int[10];
auto var = static_cast<int>(var);
```

LLVM, GOOGLE

- `nullptr` instead `0` or `NULL` LLVM
- Use `[[deprecated]]` to indicate deprecated functions
- Use `using` instead `typedef`

Use C++11/C++14 features for classes:

- Use `explicit` constructors
- Use *defaulted* default constructor
- Use `override` function keyword
- Use `final` function keyword

Control Flow

- Multi-lines statements and complex conditions require curly braces GOOGLE
- Boolean expression longer than the standard line length requires to be consistent in how you break up the lines GOOGLE
- Curly braces are not required for single-line statements (but allowed) GOOGLE
- The `if` and `else` keywords belong on separate lines GOOGLE

```
if (c1) {  // not mandatory
    <statement>
}

if (c2) {  // required
    <statement1>
    <statement2>
}
```

```
if (complex_condition1 &&
    complex_condition2) { // required
    <statement1>
}

// error!!
if (c1) <statement1>; else <statement2>
```

Control Flow

- Do not use `else` after a `return` LLVM
- Use *early exits* (`continue` , `break` , `return`) to simplify the code LLVM
- Turn predicate loops into predicate functions LLVM
- Merge multiple conditional statements

```
void f() {
    if (c1) {
        <statement1>
        return/break/continue;
    }   // error!!
    else
        <statement2>
}
void f() {
    if (c1) {
        <statement1>
        return/break/continue;
    }   // correct
    <statement2>
}
```

```
for (<loop_condition1>) { // should be
    if (<condition2>) {   // an external
        var = ...         // function
        break;            //
    }                     //
}                         //

if (<condition1>) { // error!!
    if (<condition2>)
        <statement>
}
                  // correct
if (<condition1> && <condition2>)
    <statement>
```

General rule: *avoid abbreviation and very long names*

variable Variable names should be nouns
- Uppercase Camel style e.g. `MyVar` LLVM
- Lowercase separated by underscore e.g. `my_var` Google

constant
- `k` prefix, e.g. `kConstantVar` Google
- Upper case separated by underscore `CONSTANT_VAR`

function Should be verb phrases (as they represent actions)
- Lowercase camel style, e.g. `myFunc()` LLVM
- Uppercase camel style for standard functions
  e.g. `MyFunc()` Google
  Lowercase separated by underscore for cheap functions
  e.g. `my_func()` Google, Std

namespace Lowercase separated by underscore
e.g. `my_namespace` Google, LLVM

typename Uppercase camel style (including `classes`, `structs`, `enums`, `typedefs`, etc.)
e.g. `HelloWorldClass` LLVM, Google

enum name
- `k` prefix
  e.g. `enum MyEnum { kEnumVar1, kEnumVar2 }` Google
- Uppercase camel style
  e.g. `enum MyEnum { EnumVar1, EnumVar2 }` LLVM
  - prefer `enum class`

macro Uppercase separated by underscore
e.g. `MY_MACRO` Google
- do not use macro for enumerator, constant, and functions

Other Issues

Do not use *RTTI* (`dynamic_cast`) or *exceptions*

LLVM, Google

Code style

- Use common loop variable names
  - `i,j,k,l` used in order
  - `it` for iterators
- Use `true, false` for boolean variables instead numeric value `0, 1`
- Prefer consecutive alignment

```
int           var1 = ...
long long int var2 = ...
```

- Use the same line ending (e.g. `'\n'`) for all files
- Use `UTF-8` file encoding for portability
- Close files with a blank line

Code Documentation

- Each file should start with a license LLVM
- Each file should include
  `@author` (name, surname, affiliation, email),
  `@version`, `@date`
- Use always the same style
- Comment style
  - Multiple lines

```
/**
 * comment1
 * comment2
 */
```

  - single line

```
/// comment
```