

Modern C++

Daniel Illescas Romero

November 17, 2017

University Of Granada [UGR]

1. Style & Good practices
2. New in C++
3. Functional Programming
4. Advanced
5. Compilation
6. Reference

Style & Good practices

Naming convention

- Bad idea

```
1 int main() {  
2     int a, b;  
3     double c;  
4 }
```

- Good idea

```
1 int main() {  
2     unsigned int age = 20;  
3     int points = 0;  
4     float height = 1.7;  
5 }
```

Naming convention

- Best idea [C++11]

```
1 int main() {  
2     uint8_t age{10};  
3     int64_t points{};  
4     float height{};  
5 }
```

- Freak! [C++11]

```
1 int main() {  
2     auto age = uint8_t{20};  
3     auto points = int64_t{0};  
4     auto height = float{5.67};  
5 }
```

Programming styles

- Variable names [3]

```
1 // Delimiter separated words
2 float daniel_height = 1.7;
3 // CamelCase (letter-case separated words)
4 float johnHeight = 1.7;
```

- Braces (use it always!) [2]

```
1 // K&R (Kernighan and Ritchie)
2 if () {
3
4 }
5 // Allman
6 if ()
7 {
8
9 }
```

Macros & global variables

- Avoid the use of macros

```
1 // Type?, Scope?
2 // Is not constant and can be undefined
3 #define PI 3.14159
4 #define SIZE 10
5
6 int main() {
7
8     // float PI = 10; → Would be: float 3.14159 = 10 ¿?
9
10    int numbers[SIZE];
11
12    #undef SIZE
13
14    // Error: "SIZE" is not declared...
15    cout << SIZE << endl;
16 }
```

Macros & global variables

- Try NOT to use global variables

```
1  #include <vector>
2  #include <array>
3
4  // It leads to many problems... BAD IDEA
5  // Ambiguity ERROR with: std::__1::size
6  int size = 20; // Can a size be negative?
7
8  namespace utils { // This would be a safe alternative
9      constexpr size_t arraySize = 20;
10 }
11
12 // How do I pass an array? You don't know the size here
13 void doSomething(int arr[]) {}
14
15 void doSomething(int arr[size]) {} // NO...
16
17 // Valid option
18 void doSomething(int* arr, const size_t arrSize) {}
19
20 // If you know it's size
21 void doSomething(std::array<int,20> arr) {}
22
23 // If you DON'T know it's size
24 void doSomething(std::vector<int> arr) {}
```

Platform dependency

- The use of `system` can lead to many errors

```
1 // It doesn't work in the Windows command line (CMD)
2 std::system("ls -l > test.txt");
```

- If you use old and dependent libraries you can't get **portable** code [4]

```
1 #include <pthread.h>
2
3 void* print_message_function(void* ptr) {
4     char *message;
5     message = (char *) ptr;
6     printf("%s \n", message);
7 }
8
9 int main() {
10
11     int iret1 = pthread_create(&thread1, NULL, print_message_function, (void*)
        message1);
12 }
```

Platform dependency

- Try to use the own language libraries, or at least updated ones

```
1  #include <thread>
2
3  void doSomething(int number) { ... }
4  void doSomething2(int& number) { ... }
5
6  int main() {
7
8      int number = 0;
9
10     // Pass by value
11     std::thread thread1(doSomething, number + 1);
12
13     // Pass by reference
14     std::thread thread2(doSomething2, std::ref(number));
15
16     // thread3 is running "doSomething2()", thread2 is no longer a thread
17     std::thread thread3(std::move(thread2));
18
19     thread1.join();
20     thread3.join();
21 }
```

Naming: MAJOR.MINOR.PATCH

Example: **1.3.4**

Increment the version when you:

- **MAJOR:** make incompatible API changes
- **MINOR:** add functionality in a backwards-compatible manner
- **PATH:** make backwards-compatible bug fixes

Semantic Versioning

How you should do it:

- Start with the **0.1** version
- While the SW is in development increase the **0.x.y** version
- Use **v1.0.0** for production
- Try not to use really big numbers

New in C++

auto, type inference - [1]

- Basic types

```
1 auto number = 2; // int
2 auto width = 0.5; // double
3 auto name = "daniel"; // const char*
4 auto name = "daniel"s; // string
```

- More complex ones

```
1 vector<int> points{1,2,3};
2
3 for (vector<int>::iterator it = points.begin(); it != points.end(); ++it) {
4     cout << *it << endl;
5 }
6 for (auto it = points.begin(); it != points.end(); ++it) {
7     cout << *it << endl;
8 }
```

auto, type inference

- Can be used also in functions with large type names

```
1 std::chrono::time_point<std::chrono::high_resolution_clock> now() {  
2     return std::chrono::high_resolution_clock::now();  
3 }  
4  
5 auto now() {  
6     return std::chrono::high_resolution_clock::now();  
7 }
```

- Other ways

```
1 auto add(int value, double value2) → decltype(value + value2) {  
2     return value + value2;  
3 }  
4 auto calculateWeight( ... ) → double { ... }
```

Fixed width integers types [C++11] & *brace initializer*

- Main types

```
1 #include <stdint>
2
3 int8_t, uint8_t // Signed/ Unsigned type with 8 bits
4 int16_t, uint16_t // 16 bits
5 int32_t, uint32_t // 32 bits
6 int64_t, uint64_t // 64 bits
7 intmax_t, uintmax_t // Largest capacity (usually 64 bits)
8 size_t // To represent array capacities [Unsigned 64 bits integer]
```

- Correctly initialising types

```
1 uint64_t bigStuff {643456787654};
2 int32_t things = {3500000000};
3 // uint8_t age {-20}; // Compilation error
4 things += {2000000000};
5 // things = 1205032704 :(
6 // you can't avoid overflow...
```

NULL, nullptr

- NULL can lead to errors!

```
1 void doSomething(int* ptrI) { ... }
2 void doSomething(double* ptrD) { ... }
3 void doSomething(std::nullptr_t nullPointer) { ... }
4
5 int main() {
6
7     int* ptrI;
8     double* ptrD;
9
10    doSomething(ptrI);
11    doSomething(ptrD);
12    doSomething(nullptr); // ambiguous without void f(nullptr_t)
13    //doSomething(NULL); // ambiguous: all functions are candidates
14 }
```

- Assign null

```
1 int* something = NULL; // before C++11: NULL = 0
2 double* foo = nullptr; // C++11: std::nullptr_t
```

Alternative to *raw pointers*. `unique_ptr<int>`

- Automatic Memory Management

```
1  #include <memory>
2
3  using namespace std;
4
5  int* giveMeAnInt() {
6      return new int{100};
7  }
8
9  unique_ptr<int> giveMeACoolInt() {
10     return unique_ptr<int>(new int{100});
11 }
12
13 auto giveMeTheBestInt() {
14     return make_unique<int>(100);
15 }
16
17 int main() {
18     // unique_ptr automatically releases the memory, int* don't
19     auto number = giveMeTheBestInt();
20     cout << *number << endl;
21
22     int* otherNumber = giveMeAnInt();
23     delete otherNumber;
24 }
```

Alternative to *raw pointers*. `unique_ptr<int>`

- Automatic Memory Management

```
1  #include <iostream>
2  #include <memory>
3
4  using namespace std;
5
6  int main() {
7
8      constexpr size_t arrSize { 1000 };
9
10     unique_ptr<int[]> numbers(new int[arrSize]);
11     //[C++14] auto numbers = make_unique<int[]>(arrSize);
12
13     fill(&numbers[0], &numbers[arrSize], 10);
14
15     for (size_t i = 0; i < arrSize; i++) {
16         cout << numbers[i] << endl;
17     }
18 }
```

C++ Castings

- Normal Castings

```
1 int number = 100;
2 float height = (int)number; // C style
3 height = int(number); // C++ style
4 number = static_cast<int>(3.14);
5 // const_cast, dynamic_cast, reinterpret_cast
```

- Implicit and explicit conversions

```
1 struct Foo {
2     // implicit conversion
3     operator int() const { return 7; }
4     // explicit conversion
5     explicit operator int*() const { return nullptr; }
6     explicit Foo(size_t elementsCount) { ... }
7     ...
8     Foo x;
9     int* q = x; // Error
```

Range-based for loop. I/O manipulators

- Modern for-loop for collections [C++11]

```
1 vector<string> names{"daniel", "manuel"};
2
3 for (const auto& name: names) {
4     cout << name << endl;
5 }
```

- Display `true/false` with `bool` and use more `float` precision

```
1 #include <iomanip>
2
3 boolalpha(cout); // or: cout.flags(std::ios_base::boolalpha);
4 // "noboolalpha(cout)" to disable it
5 bool isHidden = false;
6 cout << true << ' ' << isHidden << endl; // true false
7
8 const long double pi = std::acos(-1.L);
9 cout.precision(std::numeric_limits<long double>::digits10 + 1);
10 cout << pi << endl; // 3.141592653589793239
```

numeric_limits and initializer_list

- Numeric limits

```
1 // Formerly [<climits>]
2 INT_MIN // -2147483648
3 LONG_MAX // 9223372036854775807
4
5 // Nowadays [<cstdint>] y [<limits>]
6 INT32_MIN // -2147483648
7 UINT8_MAX // 255
8 numeric_limits<uint16_t>::max() // 65535
9 numeric_limits<float>::lowest() // -3.40282e+38
```

- initializer_list

```
1 class vector {
2     vector( ... ) {}
3     vector(initializer_list ilist) {}
4     ...
5
6     vector numbers = {1,2,3,4}; // vector
7     auto moreNumbers = {1,3,5,7}; // initializer_list
```

Random number [C++11]

- Generate a random number in a range

```
1  #include <vector>
2  #include <random>
3
4  using namespace std;
5
6  int main() {
7
8      vector<int> numbers {1,2,3,4,5,6};
9
10     random_device randomDevice;
11     mt19937 generator(randomDevice());
12
13     uniform_int_distribution<int> randomValue(1, 90);
14     cout << randomValue(generator) << endl;
15 }
```

<algorithm> functions

- Sort, shuffle, fill, copy...

```
1  vector<int> numbers {1,2,3,4,5,6};
2
3  random_device randomDevice;
4  mt19937 generator(randomDevice());
5
6  // Possible output: 6 4 1 3 2 5
7  std::shuffle(numbers.begin(), numbers.end(), generator);
8
9  numbers = {1,2,3,4,5,6};
10
11 vector<int> numbers2 {9,10};
12
13 // numbers result: {9,10,3,4,5,6}
14 copy(numbers2.begin(), numbers2.end(), numbers.begin());
15
16 // Result: 3 4 5 6 9 10
17 sort(numbers.begin(), numbers.end());
18
19 // What if I want to change the order ... ?
```

Functional Programming

[C++11] Lambda expressions

- Inline functions as parameters

```
1 // Result: 10 9 6 5 4 3
2 sort(numbers.begin(), numbers.end(), [](int lhs, int rhs) {
3     return lhs ≥ rhs;
4 });
```

- Syntax

```
1 [ capture-list ] ( params ) → ret { body }
2
3 capture-list:
4 [a,&b] - Captures "a" by copy, "b" by reference
5 [this] - Captures the current object value
6 [&] - Captures variables by reference
7 [=] - Captures variables by copy
8 [] - Captures nothing
```

[C++11] Lambda expressions

- How to pass functions

```
1  #include <functional>
2
3  double operation(double lhs, double rhs,
4                  std::function<double(double,double)> operationFunc) {
5      return operationFunc(lhs, rhs);
6  }
7
8  int main() {
9
10     double multiply = operation(10, 30.1, [](double lhs, double rhs) {
11         return lhs * rhs;
12     });
13
14     double add = operation(20.1, 70, [](double lhs, double rhs) {
15         return lhs + rhs;
16     });
17
18     double number = 1000;
19     double custom = operation(20.1, 70, [&](double lhs, double rhs){
20         return (number * lhs) + rhs;
21     });
22 }
```

Filter, Map & Reduce (evt::Array) [7]

- Filter

```
1 Array<string> names {"Daniel", "John", "Peter"};
2 auto filtered = names.filter([](const string& str) {
3     return str.size() > 4;
4 }); // ["Daniel", "Peter"]
```

- Map & Reduce

```
1 size_t totalSize = Array<string>({"names", "john"})
2     .map<size_t>([](auto str) {
3         return str.size();
4     }) // [5, 4]
5     .reduce<size_t>([](auto total, auto strSize) {
6         return total + strSize;
7     }); // 9
```

Advanced

Constant expressions (constexpr) [5]

- Functions

```
1 constexpr uint64_t factorial(int n) {  
2     return n ≤ 1 ? 1 : (n * factorial(n - 1));  
3 }  
4  
5 int main {  
6     // error: static_assert failed "wrong!"  
7     static_assert(factorial(2) == 3, "wrong!");  
8 }
```

- Variables

```
1 constexpr uint64_t factorialResult = factorial(2);  
2 // static_assert(factorialResult == 3, "wrong!");  
3  
4 constexpr uint64_t other = factorialResult * 2;  
5 static_assert(other == 4, "wrong!");
```

Constant expressions (constexpr)

- Objects

```
1  class Circle {
2      int x_;
3      int y_;
4      int radius_;
5  public:
6      constexpr Circle (int x, int y, int radius):
7          x_(x), y_(y), radius_(radius) {}
8      constexpr double area() const {
9          return radius_ * radius_ * 3.1415926;
10     }
11     // ...
12 };
13
14 int main() {
15     constexpr auto myCircle = Circle(10,20,30);
16     static_assert(myCircle.area() < 3000, "wrong!");
17 }
```

Constant expressions (constexpr)

- Compile-time check

```
1  #include <type_traits> // is_unsigned, is_object, etc
2
3  template <typename Type>
4  void doSomething(Type something) {
5      if constexpr (std::is_same<Type, int>()) {
6          Type number = something * 10;
7          cout << "int!!: " << number << endl;
8      }
9      else if constexpr (std::is_same<Type, string>()) {
10         Type str = something;
11         cout << "string!!: " << str.length() << endl;
12     }
13 }
14
15 int main() {
16     doSomething(800);
17     doSomething("Daniel"s);
18 }
```

Optional value

- Sometimes we should not (or we can't) return a specific value

```
1  #include <iostream>
2  #include <vector>
3  #include <experimental/optional>
4
5  using namespace std::experimental;
6
7  size_t indexOf(int number, std::vector<int> numbers) {
8      for (size_t i = 0; i < numbers.size(); i++) {
9          if (number == numbers[i]) {
10             return i;
11         }
12     }
13     return 0; // It's confusing... what if the valid returned position is 0
14 }
15
16 std::optional<size_t> safeIndexOf(int number, std::vector<int> numbers) {
17     for (size_t i = 0; i < numbers.size(); i++) {
18         if (number == numbers[i]) {
19             return i;
20         }
21     }
22     return nullopt;
23 }
```

Optional value

- The most secure (& elegant) way is by using an optional value

```
1  int main() {
2
3      vector<int> otherNumbers{};
4
5      size_t index = indexOf(100, otherNumbers);
6      // Segmentation fault
7      // cout << otherNumbers[index] << endl;
8
9      vector<int> numbers{1,2,3,4,10};
10     if (auto index2 = safeIndexOf(100, numbers)) {
11         cout << numbers[*index2] << endl;
12         // index2.value_or(0)
13     }
14 }
```

Optional value

- Returned value by an array when you access a position

```
1 inline optional<Type> at(const size_t index) const {  
2     if (index ≥ count_) {  
3         return nullopt;  
4     } else {  
5         return this→ values[index];  
6     }  
7 }
```

- Template type constraints

```
1 #include <type_traits>
2
3 template <typename ArithmeticType,
4         typename = typename std::enable_if<
5             std::is_arithmetic<ArithmeticType>::value
6         >::type>
7     ArithmeticType doSomething(ArithmeticType number) {
8         return number * 100;
9     }
10
11 int main() {
12
13     doSomething(800); // OK
14
15     // candidate template ignored: disabled by 'enable_if'
16     doSomething("Daniel"s); // ERROR
17 }
```

Good practices and tips for classes

- include/Human.hpp

```
1  #pragma once // instead of the classic guards (#ifndef ...)
2
3  #include <cstdint>
4  #include <string_view> // C++17
5  // using namespace std; // Don't use in .h, .hpp files
6
7  namespace evt { // HIGHLY recommended
8      class Human {
9          // Direct variable initialization [c++11]
10         // NEVER use underscores at the beginning of a name
11         uint8_t age_{};
12         std::string_view name_{};
13     public:
14         // constexpr Human(){} // Might or might not make sense
15         constexpr Human(const uint8_t age, const std::string_view& name):
16             age_(age), name_(name) {}
17
18         constexpr uint8_t age() const {
19             return this->age_;
20         }
21         constexpr std::string_view name() const {
22             return this->name_;
23         }
24     };
25 }
```

Buenas prácticas y consejos para clases

- Main.cpp

```
1  #include "include/Human.hpp"
2
3  // using namespace evt; (opcional)
4
5  // using namespace evt; (opcional)
6
7  // Is not advisable to pass complex types by copy ...
8  // void giveMeTheObject(evt::Human human) { ... }
9
10 // ... or by reference (the object could be modified)
11 // void giveMeTheObject(evt::Human& human) { ... }
12
13 // The best way is by constant reference
14 void giveMeTheObject(const evt::Human& human) { ... }
15
16 int main() {
17     constexpr evt::Human daniel(10, "Daniel");
18     static_assert(daniel.name() == "Daniel", "incorrect!");
19 }
```

Compilation

Compilation with new C++ versions

- Add the C++ version after `-std=`

```
1 g++ main.cpp -std=c++11
2 g++ main.cpp -std=c++14
3 g++ main.cpp -std=c++17 // o: 1z
```

- *Flags* recomendados

```
1 -Wall → Enables important warnings
2 -Wextra → Enables more warnings
3 -O1 / -O2 / -Os / -O3 / -fast → Optimizations
4 ~
5 g++ main.cpp -std=c++14 -Wall -Wextra -Os
```

Makefile example

- Complete Example

```
1  ## FLAGS ##
2  Libraries = -L lib
3  Headers = -I include
4  Sources = main.cpp $(Headers) $(Libraries)
5  CompilerFlags = -std=c++14 -Os -Wall -Wextra
6  OutputName = test
7
8  ## TARGETS ##
9  all:
10     @g++ $(CompilerFlags) $(Sources) -o $(OutputName)
11
12  clean:
13     @rm -i $(OutputName)*
```

Reference



C++ reference.

URL: <http://en.cppreference.com/>.



Indentation style.

URL: https://en.wikipedia.org/wiki/Indentation_style.



Naming convention.

URL: [https://en.wikipedia.org/wiki/Naming_convention_\(programming\)](https://en.wikipedia.org/wiki/Naming_convention_(programming)).



Posix thread (pthread) libraries.

URL: <http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html>.



Alex Allain.

Constexpr - generalized constant expressions in c++11.

URL: <https://www.cprogramming.com/c++11/c++11-compile-time-processing-with-constexpr.html>.



Tom Preston-Werner.

Semantic versioning.

URL: <http://semver.org>.



Daniel Illescas Romero.

Array.hpp - fast & pretty container for c++.

URL:

<https://github.com/illescasDaniel/Array.hpp>.

Suggestions:

- <http://devdocs.io>
- <https://tex.stackexchange.com>