Modern C++

bit.ly/ModernCppSpanish

Daniel Illescas Romero

17 de abril de 2018

Universidad de Granada [UGR]

Índice

Estilo y buenas prácticas

Nombrado de variables

- Mala idea

```
int main() {
   int a, b;
   double _c; // NUNCA guiones bajos delante de nombres
}
```

- Buena idea

```
int main() {
   unsigned int age = 20;
   int points = 0;
   float height = 1.7;
}
```

Nombrado de variables

- Mejor idea [C++11]

```
int main() {
   uint8_t age{10};
   int64_t points{};
   float height{};
}
```

- Friki! [C++11]

```
int main() {
   auto age = uint8_t{20};
   auto points = int64_t{0};
   auto height = float{5.67};
}
```

Estilos de programación

- Nombres variables [3]

```
// Palabras separadas por un delimitador
float daniel_height = 1.7;
// CamelCase (separadas por mayúsculas)
float johnHeight = 1.7;
```

- Llaves (Nota: ¡usadlas siempre!) [2]

```
// K&R (Kernighan and Ritchie)
if () {

}
// Allman
if ()
{

}
```

Macros y variables globales

- Evita las macros

```
1  // ¿Tipo?, ¿Scope?
2  // No es constante y puede des-definirse
3  #define PI 3.14159
4  #define SIZE 10
5
6  int main() {
7
8     // float PI = 10; -> Sería: float 3.14159 = 10; ¿?
9
10    int numbers[SIZE];
11
12    #undef SIZE
13
14    // Error: "SIZE" no está declarado ...
15    cout << SIZE << endl;
16 }
```

Macros y variables globales

- Procura NO usar variables globales

```cpp
#include <vector>
#include <array>

// Da lugar a problemas... MALA IDEA
// ERROR por ambigüedad con: std::__1::size
int size = 20; // ¿Podría ser negativo un tamaño también?

namespace utils { // Esto sería una alternativa segura
   constexpr size_t arraySize = 20;
}
// ¿Cómo paso un array entonces? Así NO sabes tamaño
void doSomething(int arr[]) {}
void doSomething(int arr[size]) {} // NO...

// Opción válida
void doSomething(int* arr, const size_t arrSize) {}
// Buena opción
void doSomething(int* begin, int* end) {}

// Si sabemos tamaño
void doSomething(std::array<int,20> arr) {}
// Si NO sabemos tamaño
void doSomething(std::vector<int> arr) {}
```

Dependencia de plataformas

- El uso de `system` puede llevar a errores

```
1 // No funciona en la línea de comandos de Windows
2 std::system("ls -l > test.txt");
3
4 // Solo funciona en Windows :/
5 std::system("pause");
```

- Algunas soluciones:
  - Para pausar la ejecución del programa: `cin.ignore();`
  - Para listar ficheros en cualquier plataforma podemos usar:

    `std::filesystem::recursive_directory_iterator`

Dependencia de plataformas

- Si se utlizan librerías antiguas y dependientes no tenemos código *portable* (ni bonito a veces...) [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    int iret1 = pthread_create(&thread1, NULL, print_message_function, (void*)
          message1);
11 }
```

Dependencia de plataformas

- Procura usar librerías propias del lenguaje y/o actualizadas

```cpp
#include <thread>

void doSomething(int number) { ... }
void doSomething2(int& number) { ... }

int main() {

   int number = 0;

   // paso por valor
   std::thread thread1(doSomething, number + 1);

   // paso por referencia
   std::thread thread2(doSomething2, std::ref(number));

   // thread3 está corriendo "doSomething2()", thread2 ya 'no es un thread'
   std::thread thread3(std::move(thread2));

   thread1.join();
   thread3.join();
}
```

Semantic Versioning [7] - Control de Versiones

Nomenclatura: MAJOR.MINOR.PATCH

Ejemplo: **1.3.4**

Incrementamos la versión cuando:

- **MAJOR**: se hacen cambios incompatibles de API.
- **MINOR**: al añadir funcionalidades retrocompatibles.
- **PATH**: correcciones de errores.

Semantic Versioning - Control de Versiones

Forma de trabajar:

- Empezar en la versión **0.1**.
- Ir incrementando la versión **0.x.y** mientras sea SW en desarrollo.
- Usar la **v1.0.0** para producción.
- No te pases, un número de versión muy grande no es legible.

Tamaño del Stack vs Heap

El tamaño del Stack (memoria estática) es muy limitado, por lo que debemos evitar usarlo al almacenar en vectores de grandes cantidades:

```
1  const size_t arrSize = 10'100'000;
2  // int numbers2[arrSize]; // SEGMENTATION FAULT
3  int* numbers2 = new int[arrSize];
4
5  for (size_t i = 0; i < arrSize; i++) {
6     numbers2[int(i)] = i;
7  }
8  cout << numbers2[50] << endl;
9
10 delete[] numbers2;
```

Tests de unidad (Unit tests)

A veces es conveniente el uso de tests, aunque por defecto C++ no tenga ninguna clase para ello. Estos nos permiten comprobar que nuestro código funciona correctamente:

```
1 evt::UnitTest initializationTest("Array - Initialization");
2
3 evt::Array<size_t> numbers {0,1,2,3,4};
4
5 for (size_t i = 0; i < numbers.size(); i++) {
6    initializationTest.assert(numbers.contains(i), "Array
         doesn't contain expected value");
7 }
```

Nuevo en C++

auto, inferencia de tipos - [1]

- Tipos básicos o simples

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

- Tipos más complejos

```
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, inferencia de tipos

- En funciones con tipos... un poco largos

```
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  }
```

- Otras formas de usarlo

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

Enteros de longitud fja [C++11] y *brace initializer*

- Tipos principales

```cpp
#include <cstdint>

int8_t, uint8_t // Entero 8 bits con signo, sin signo
int16_t, uint16_t // 16 bits
int32_t, uint32_t // 32 bits
int64_t, uint64_t // 64 bits
intmax_t, uintmax_t // Mayor capacidad (normalmente 64 bits)
size_t // Para representar capacidades [Entero 64 bits sin signo]
```

- Inicializar correctamente tipos

```cpp
uint64_t bigStuff {643456787654};
int32_t things = {750000000};
// uint8_t age {-20}; // Error al compilar
things += {2100000000};
// no se puede evitar el overflow...
// things ahora es: -1444967296
```

~~NULL~~, nullptr

- Asignar null

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

- ¡NULL da lugar a errores!

```
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); // ambiguo sin void doSomething(nullptr_t)
13    //doSomething(NULL); // ambiguo: todas las funciones son candidatas
14 }
```

Alternativa a *raw pointers*. `unique_ptr<_>`

- Liberación de memoria (automática)

```cpp
#include <iostream>
#include <memory>

using namespace std;

int main() {

   constexpr size_t arrSize {1000};
   unique_ptr<int[]> numbers(new int[arrSize]);
   //[C++14] auto numbers = make_unique<int[]>(arrSize);
   // numbers = otherNumbers; No se permite

   fill(&numbers[0], &numbers[arrSize], 10);

   for (size_t i = 0; i < arrSize; i++) {
      cout << numbers[i] << endl;
   }
}
```

Alternativa a *raw pointers*. `shared_ptr<_>`

- Instancias compartidas

```
1  class Dog {
2  public:
3     string name = "";
4     float height = 0;
5     Dog(const string& name, float height): name(name),
          height(height) {}
6  };
7
8  class Person {
9  public:
10    string name = "";
11    float height = 0;
12    shared_ptr<const Dog> dog;
13    Person(const string& name, float height): name(name),
          height(height) {}
14 };
```

Alternativa a *raw pointers*. `shared_ptr<_>`

- Instancias compartidas

```
1  int main() {
2
3     auto dog1 = make_shared<Dog>("Pepito", 0.4);
4
5     Person daniel("Daniel", 1.7), sister("SheIsABot", 1.6);
6
7     daniel.dog = dog1;
8     sister.dog = dog1;
9
10    dog1->height = 1;
11    //daniel.dog->height = 2; // ERROR, daniel.dog es const
12
13    cout << daniel.dog->height << endl;
14 }
```

Casting de tipos en C++

- Castings normales

```
int number = 100;
float height = (int)number; // C
height = int(number); // C++
number = static_cast<int>(3.14);
// const_cast, dynamic_cast, reinterpret_cast
```

- Conversiones implícitas o explícitas

```
struct Foo {
   // implicit conversion
   operator int() const { return 7; }
   // explicit conversion
   explicit operator int*() const { return nullptr; }
   explicit Foo(size_t elementsCount) { ... }
...
Foo x;
int* q = x; // Error
```

Range-based for loop. I/O manipulators

- Bucle for ”moderno”para colecciones [C++11]

```
vector<string> names{"daniel", "manuel"};

for (const auto& name: names) {
    cout << names << endl;
}
```

- Mostrar `true`/`false` con `bool` y más precisión `float`

```
#include <iomanip>

boolalpha(cout); // o: cout.flags(std::ios_base::boolalpha);
// "noboolalpha(cout)" para deshabilitarlo
bool isHidden = false;
cout << true << ' ' << isHidden << endl; // true false

const long double pi = std::acos(-1.L);
cout.precision(std::numeric_limits<long double>::digits10 + 1);
cout << pi << endl; // 3.141592653589793239
```

numeric_limits e initializer_list

- Límites de valores

```
// Antiguamente [<climits>]
INT_MIN // -2147483648
LONG_MAX // 9223372036854775807

// Hoy en día [<cstdint>] y [<limits>]
INT32_MIN // -2147483648
UINT8_MAX // 255
numeric_limits<uint16_t>::max() // 65535
numeric_limits<float>::lowest() // -3.40282e+38
```

- initializer_list

```
class vector {
   vector( ... ) {}
   vector(initializer_list ilist) {}
 ...

vector numbers = {1,2,3,4}; // vector
auto moreNumbers = {1,3,5,7}; // initializer_list
```

Valor aleatorio [C++11]

- Generar un número aleatorio entre un rango

```
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 }
```

Funciones de <algorithm>

- Ordena, desordena, rellena, copia...

```
1  vector<int> numbers {1,2,3,4,5,6};
2
3  random_device randomDevice;
4  mt19937 generator(randomDevice());
5
6  // Posible salida: 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 // Resultado numbers: {9,10,3,4,5,6}
14 copy(numbers2.begin(), numbers2.end(), numbers.begin());
15
16 // Resultado: 3 4 5 6 9 10
17 sort(numbers.begin(), numbers.end());
18
19 // ¿Y si quiero ordenarlo al revés...?
```

Programación Funcional

[C++11] Expresiones lambda λ

- Pasar funciones inline

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

- Sintaxis

```
[ capture-list ] ( params ) -> ret { body }

capture-list:
[a,&b] - Captura "a" por copia, "b" por referencia
[this] - Captura el valor del objeto actual
[&] - Captura variables por referencia
[=] - Captura variables por copia
[] - No captura nada
```

[C++11] Expresiones lambda λ

- ¿Cómo aceptar funciones?

```
1  #include <functional>
2
3  double operation(double lhs, double rhs,
4           std::function<double(double,double)> operationFunctor) {
5     return operationFunctor(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) [8]

- 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
```

Avanzado

Expresiones constantes (constexpr) [5]

- Funciones

```
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!");
```

Expresiones constantes (`constexpr`)

- Objetos

```
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 }
```

Expresiones constantes (`constexpr`)

- Comprobaciones en tiempo de ejecución

```
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 }
```

Valor opcional

- A veces no “debemos” devolver un valor concreto

```
#include <iostream>
#include <vector>
#include <experimental/optional>

using namespace std::experimental;

size_t indexOf(int number, std::vector<int> numbers) {
   for (size_t i = 0; i < numbers.size(); i++) {
      if (number == numbers[i]) {
         return i;
      }
   }
   return 0; // Confusión, ¿y si la Posición que devuelvo es 0?
}

std::optional<size_t> safeIndexOf(int number, std::vector<int> numbers) {
   for (size_t i = 0; i < numbers.size(); i++) {
      if (number == numbers[i]) {
         return i;
      }
   }
   return nullopt;
}
```

Valor opcional

- La forma más segura es con el optional

```
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 }
```

Valor opcional

- Valor que devuelve un vector al acceder a una posición

```
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 }
```

std::enable_if<>

- Restringir tipos en Templates

```cpp
#include <type_traits>

template <typename ArithmeticType,
    typename = typename std::enable_if<
        std::is_arithmetic<ArithmeticType>::value
    >::type>
ArithmeticType doSomething(ArithmeticType number) {
    return number * 100;
}

int main() {

    doSomething(800); // OK

    // candidate template ignored: disabled by 'enable_if'
    doSomething("Daniel"s); // ERROR
}
```

Buenas prácticas y consejos para clases [6]

- include/Human.hpp

```cpp
#pragma once // en vez de los clásicos guards (#ifndef ... )

#include <cstdint>
#include <string_view> // C++17
// using namespace std; // No usar en ficheros .h, .hpp

namespace evt { // MUY recomendable
   class Human {
      // Inicialización de variables directamente [c++11]
      // NUNCA usar guiones bajos al principio de nombres
      uint8_t age_{};
      std::string_view name_{};
   public:
      // constexpr Human(){} // Podría o no tener sentido
      constexpr Human(const uint8_t age, const std::string_view& name):
            age_(age), name_(name) {}

      constexpr uint8_t age() const {
         return this->age_;
      }
      constexpr std::string_view name() const {
         return this->name_;
      }
   };
}
```

Buenas prácticas y consejos para clases

- Main.cpp

```cpp
#include "include/Human.hpp"

// using namespace evt; (opcional)

// No es recomendable pasar datos complejos por copia...
// void giveMeTheObject(evt::Human human) { ... }

// ... ni tampoco solo por referencia (se podrían editar los originales)
// void giveMeTheObject(evt::Human& human) { ... }

// Mejor por referencia constante
void giveMeTheObject(const evt::Human& human) { ... }

int main() {
   constexpr evt::Human daniel(10, "Daniel");
   static_assert(daniel.name() == "Daniel", "incorrect!");
}
```

Compilación

Compilar con nuevas versiones

- Añadir la versión de C++ tras `-std=`

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

- *Flags* recomendados

```
-Wall ->   Habilita la mayoría de warnings
-Wextra -> Habilita más warnings
-O1 / -O2 / -Os / -O3 / -fast -> Optimizaciones
~~
g++ main.cpp -std=c++14 -Wall -Wextra -Os
```

Makefile de ejemplo

- Ejemplo completo

```
## FLAGS ##
Libraries = -L lib
Headers = -I include
Sources = main.cpp $(Headers) $(Libraries)
CompilerFlags = -std=c++14 -Os -Wall -Wextra
OutputName = test

## TARGETS ##
all:
   @g++ $(CompilerFlags) $(Sources) -o $(OutputName)

clean:
   @rm -i $(OutputName)*
```

Bibliografía

Bibliografía i

- 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).
- Posix thread (pthread) libraries.
  URL: http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html.

Bibliografía ii

- Alex Allain.
  Constexpr - generalized constant expressions in c++11.
  URL: https://www.cprogramming.com/c++11/c++11-compile-time-processing-with-constexpr.html.
- lefticus.
  C++ best practices.
  URL: https://www.gitbook.com/book/lefticus/cpp-best-practices/details.
- Tom Preston-Werner.
  Semantic versioning.
  URL: http://semver.org.

Bibliografía iii


Daniel Illescas Romero.
Array.hpp - fast & pretty container for c++.
URL:
https://github.com/illescasDaniel/Array.hpp.

Recomendaciones:

- http://devdocs.io
- https://zealdocs.org
- http://velocity.silverlakesoftware.com
- https://tex.stackexchange.com
- https://github.com/illescasDaniel/Modern-Cpp-Spanish