CppCoreGuidelines, Part 1
Introducción y Filosofía

Juanmi Huertas
R&D Software Engineer - HP
Juanmi.Huertas@hp.com

using std::cpp 2017 30/11/2017

Introducción

¿Qué son las *CppCoreGuidelines*?

"This document is a set of guidelines for using C++ well.
The aim of this document is to help people to use modern C++ effectively.
By "modern C++" we mean C++11 and C++14 (and soon C++17).

In other words,
**what would you like your code to look like in 5 years' time,
given that you can start now?**
In 10 years' time?"

Cita del abstract de las *CppCoreGuidelines*

"Within C++, there is a much smaller and cleaner language struggling to get out"
"And no, that smaller and cleaner language is not Java or C#."
Bjarne Stroustrup, *The Design and Evolution of C++. pp 207 y una aclaración posterior de su FAQ.*

CppCoreGuidelines índice

P: Philosophy
I: Interfaces
F: Functions
C: Classes and class hierarchies
Enum: Enumerations
R: Resource management
ES: Expressions and statements
E: Error handling
Per: Performance

Con: Constants and immutability
T: Templates and generic programming
CP: Concurrency
SL: The Standard library
SF: Source files
CPL: C-style programming
Pro: Profiles
N: Non-Rules and myths
NL: Naming and layout

Disclaimer: Esta charla es sobre C++. Pero muchos de estos principios son aplicables en cualquier lenguaje..

1) Código autexplicado (P.1 and P.3)

Escribímos código para otras personas (y ordenadores también).

```
// Que significa s?
change_speed(double s);
// ...
change_speed(2.3);

VS

// mejor: el significado de s es claro
change_speed(Speed s);
// ...
// error: le faltan las unidades
change_speed(2.3);
change_speed(23m / 10s);
```

```
class Date {

    // ...

public:

    int month();          // NO

    Month month() const;  // SI

    // ...
};
```

P.1: Express ideas directly in code

1) Código autexplicado (P.1 and P.3)

Escribímos código para otras personas (y ordenadores también).

```
bool ?????? (vector<string>& v, string val)
{
    // ...
    int index = -1;
    for (int i = 0; i < v.size(); ++i)
        if (v[i] == val) {
            index = i;
            break;
        }
    // ...
}
```

```
bool ?????? (vector<string>& v, string val)
{
    // ...
    auto p = find(begin(v), end(v), val);
    // ...
}
```

P.1: Express ideas directly in code 5/16

1) Código autexplicado (P.1 and P.3)

Código para gente y ordenadores.
Deja al código hablar.

¿Código auto-comentado?
Los comentarios no se ejecutan.

No confies en los comentarios.
Cree en el código.

```
// Que digo cuando escribo ... ?

for(int i = 0; i < v.size(); i++ ) {
  // ... uso el indice i y v[i] ...
  // ... puede que cambie i o v[i] ...
}

for (const auto& x : v) {
  // ... usare el valor de x ...
}

for (auto& x : v) {
  // ... usare el valor de x y lo editare ...
}
```

P.3: Express intent 6/16

2) Código legible (P.11)

Código lioso es difícil de escribir y es difícil de leer.
Y por tanto de arreglar.

El código enrevesado es feo.
El código claro es bonito.

```
int sz = 100;
int* p = (int*) malloc(sizeof(int) * sz);
int count = 0;
// ...
for (;;) {
    // ... read an int into x,
    // ... exit if there are no more 'ints'
    // ... check that x is valid ...
    if (count == sz){
        p = (int*) realloc(p, sizeof(int) * sz * 2);
        sz *= 2;
    }
    p[count++] = x;
    // ...
}
```

```
vector<int> v;
v.reserve(100);
// ...
for (int x; cin >> x; ) {
    // ... check that x is valid ...
    v.push_back(x);
}
```

P.11: Encapsulate messy constructs, rather than spreading through the code 7/16

3) Código libre de errores (P.4, P.5, P.6 and P.7)

Conoce lo que estás usando.

Cuidado con los *union* (usa *variant*!).
Cuidado con el *array decay*.
Cuidado con los errores de rango.
Cuidado con los *narrowing conversions*.
Cuidado con los *casts*.

```
union U {
   int i;
   float f;
};
U u;
u.i = 42;
std::cout << u.f;
```

```
std::variant<int,float> u, v;
u = 42;
//auto f = std::get<float>(u); error
auto i1 = std::get<int>(u);
auto i2 = std::get<0>(u);
v = u;
assert(std::holds_alternative<int>(v));
```

```
extern void f(int* p);

void g(int n)
{
  // f no conoce n
  f(new int[n]);
}
```

```
extern void f(vector<int>& v);

void g(int n)
{
  vector<int> v(n);
  f(v); // f conoce n
}
```

P.4: Ideally, a program should be statically type safe 8/16

3) Código libre de errores (P.4, P.5, P.6 and P.7)

Conoce lo que estás usando.

```
std::string GetFormData();
std::string CleanData(const std::string& data);
void ExecuteQuery(const std::string& query);
```

```
template <typename T>
struct FormData{
  explicit FormData(const string& input):m_input(input) {}
  std::string m_input;
};

struct clean{};
struct dirty{};
```

```
FormData<dirty> GetFormData();
std::optional<FormData<clean>> CleanData(const FormData<dirty>& data);
void ExecuteQuery(const FormData<clean>& query);
```

P.4: Ideally, a program should be statically type safe 9/16

3) Código libre de errores (P.4, P.5, P.6 and P.7)

No dejes para mañana...
...lo que puedes hacer hoy!

No dejes para tiempo de ejecución...
...lo que puedes hacer en tiempo de compilación!

```
void read(int* p, int n); // lee n enteros en *p
int a[100];
read(a, 1000);    // MAL
```

```
void read(vector<int>& v); // lee enteros con la dim de v
vector<int> a(100);
read(a);          // UN POCO MEJOR
```

```
// Int es un alias para enteros
// NO HACER
int bits = 0;
for (Int i = 1; i; i <<= 1)
    ++bits;
if (bits < 32)
    cerr << "Int too small\n"
```

```
// Int es un alias para enteros
// COMPROBADO EN COMPILACIÓN
static_assert(sizeof(Int) >= 4);
```

P.5: Prefer compile-time checking to run-time checking
P.6: What cannot be checked at compile time should be checkable at run time
P.7: Catch run-time errors early 10/16

4) Código 'barato' (P.8, P.9 and P.10)

¿Sobrevivirá tu programa los errores de memoria?
Usa solo aquello que necesites. Nada más.

```
{
  Object* ptr = new Object();
  ptr->foo();
  delete ptr;
}
```

```
{
  auto ptr = make_unique<Object>();
  ptr->foo();
}
```

```
void f(char* name)
{
  FILE* input = fopen(name, "r");
  // ...
  if(something) return;
  // ...
  fclose(input);
}
```

```
void f(char* name)
{
  ifstream input {name};
  // ...
  if(something) return;
  // ...
}
```

P.8: Don't leak any resources 11/16

4) Código 'barato' (P.8, P.9 and P.10)

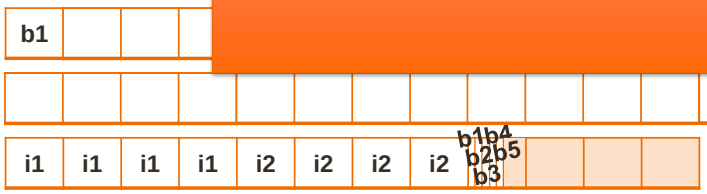
No desperdicies tiempo. No desperdicies espacio. Esto es C++.

```
struct waste{
   bool b1;
   int i1;
   bool b2;
   int i2;
   bool b3;
   bool b4;
   bool b5;
} //sizeof(waste)
```

```
struct good{
   int i1;
   int i2;
```

```
struct better{
   int i1;
   int i2;
   : 1;
   : 1;
   : 1;
   : 1;
   : 1;
   better) == 12
```

5 bool = 2^5 estados = ¿32 estados?
¿Cuántos estados quieres?
¿Por qué no usar un enum?



P.9: Don't waste time or space 12/16

4) Código 'barato' (P.8, P.9 and P.10)

Es más fácil razonar sobre objectos inmutables...
... tanto para las personas como el compilador.

Hay otros apartados que tratan esto...
...pero, por comentar algo:

```
template<int N>
struct Fib
{ enum { val = Fib<N-1>::val + Fib<N-2>::val }; };

template<>
struct Fib<1>
{ enum { val = 1 }; };

template<>
struct Fib<0>
{ enum { val = 0 }; };
```

Con.1: By default, make objects immutable

Con.2: By default, make member functions const

Con.3: By default, pass pointers and references to consts

Con.4: Use const to define objects with values that do not change after construction

Con.5: Use constexpr for values that can be computed at compile time

```
constexpr unsigned fibonacci(const unsigned x)
{
  return x <= 1 ?
    x :
    fibonacci(x - 1) + fibonacci(x - 2);
}
```

P.10: Prefer immutable data to mutable data 13/16

5) Código estándar (P.2, P.13 and P.12)

El estándar es... Bueno, estándar. Úsalo.

Usad C++ "actualizado"...
C++ Moderno (C++ 11, C++14... C++17!)

Si **de verdad** necesitas usar una extensión...
Encapsúlala, para que sea fácil de cambiar. Compresor.

Intentad used los algoritmos que ofrece la STL, y construye tus algoritmos siguiendo ese estilo.

```
std::sort(begin(v), end(v), std::greater<>());

auto it = std::partition(begin(v), end(v), [](int i){return i%2 == 0;});
```

Desde luego, el sentido común es mucho más importante.

```
// std::unordered_map<int, int> ??

// std::list<int> ??
```

P.2: Write in ISO Standard C++
P.13: Use support libraries as appropriate 14/16

5) Código estandar (P.2, P.13 and P.12)

Usad herramientas que puedan hacer “tareas aburridas”.

Clang, por ejemplo, nos ofrece una serie de herramientas útiles:

- **clang-check**: Puede ser usado para hacer un test de errores e imprimir el AST.
- **clang-format**: Puede ser usado para ayudar a formatear el código.
- **clang-tidy**: OMG! clang-tidy: puede detectar muchas cosas. ¡Moderniza tu código rápido!
- **clang-rename**: Para factorizar más facilmente el código.
- **scan-build**: Puede ser usado para hacer análisis estático del código.
- **y mucho más**: Por ejemplo, herramientas que detectan errores de memoria.

Resumiendo... (std::unordered_map< ItemTalk, std::vector< CoreCppGuideline > >)

1) Código autoexplicado contiene su propia explicación.
P.1: Express ideas directly in code, P.3: Express intent.

2) Código legible es fácil de leer y de entender.
P.11: Encapsulate messy constructs, rather than spreading through the code.

3) Código libre de errores no tiene errores.
P.4: Ideally, a program should be statically type safe, P.5: Prefer compile-time checking to run-time checking,
P.6: What cannot be checked at compile time should becheckable at run time,
P.7: Catch run-time errors early.

4) Código ‘barato’ es más rápido.
P.8: Don't leak any resources, P.9: Don't waste time or space, P.10: Prefer immutable data to mutable data.

5) Código estándar sigue los principios de Ingeniería de Software.
P.2: Write in ISO Standard C++, P.12: Use supporting tools as appropriate,
P.13: Use support libraries as appropriate.