CppCoreGuidelines, part 1
Introduction and Philosophy

Juanmi Huertas
R&D Software Engineer @ HP

Introduction

What are the 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?”

Quoted from CppCoreGuidelines Abstract

“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 and later clarification from his own FAQ.*

CppCoreGuidelines index

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: This is a C++ talk. We will talk C++ here. But the following principles are applicable to EVERY language.

1) Meaningful code (P.1 and P.3)

We write code for people (and computers).

```
change_speed(double s);   // bad: what does s signify?
// ...
change_speed(2.3);

VS

// better: the meaning of s is specified
change_speed(Speed s);
// ...
change_speed(2.3);        // error: no unit
change_speed(23m / 10s);  // meters per second
```

```
class Date {

    // ...

public:

    int month();            // don’t

    Month month() const;    // do

    // ...
};
```

P.1: Express ideas directly in code

1) Meaningful code (P.1 and P.3)

We write code for people (and computers).

```
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

1) Meaningful code (P.1 and P.3)

We write code for people (and computers).
Let the code do the talking.

Self-documented code?
Comments are not executed.

Don’t trust the comments.
Believe in the code.

```
// What am I saying when I write ... ?

for(int i = 0; i < v.size(); i++ ) {
  // ... do something with v[i] ...
  // ... i may change, v[i] may change ...
}


for (const auto& x : v) {
  // ... do something with the value of x ...
}


for (auto& x : v) {
  // ... modify x ...
}
```

P.3: Express intent

2) Legible code (P.11)

Messy code is difficult to write and to read.
Messy code is difficult to fix.

Messy code is messy.
Pretty code is pretty.

```
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

3) Error-free code (P.4, P.5, P.6 and P.7)

Know what you are using.

Be careful with union (use variant!).
Be careful with array decay.
Be careful with range errors.
Be careful with narrowing conversions.
Be careful with 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 doesn’t know n
  f(new int[n]);
}
```

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

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

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

3) Error-free code (P.4, P.5, P.6 and P.7)

Know what you are using.

```
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

3) Error-free code (P.4, P.5, P.6 and P.7)

Don’t leave for tomorrow…
…what can be done today

Don’t leave for run-time…
…what can be checked compiling!

```
void read(int* p, int n); // read n integers into *p
int a[100];
read(a, 1000);     // bad
```

```
void read(vector<int> v); // read ints until range of v
vector<int> a(100);
read(a);           // better
```

```
// Int is an alias used for integers
// don't: avoidable code
int bits = 0;
for (Int i = 1; i; i <<= 1)
    ++bits;
if (bits < 32)
    cerr << "Int too small\n"
```

```
// Int is an alias used for integers
// do: compile-time check
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

4) 'Cheap' code (P.8, P.9 and P.10)

Will your program survive an Out Of Memory error?
Just use what you need to use. Nothing more.

```
{
  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

4) ‘Cheap’ code (P.8, P.9 and P.10)

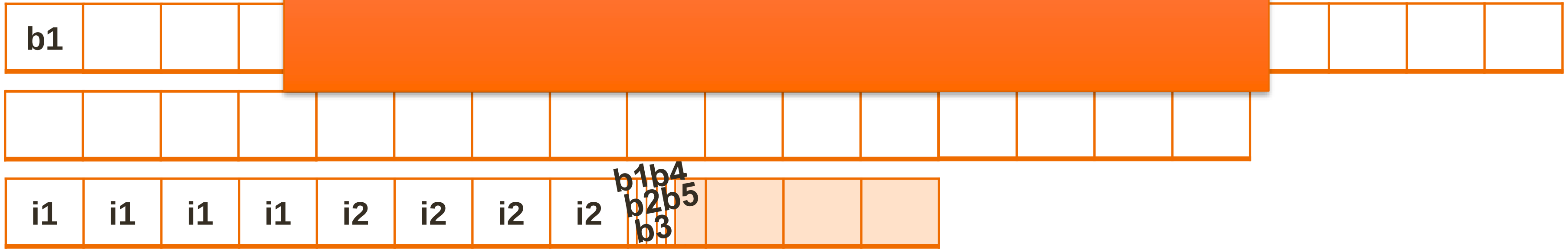
Don’t waste time. Don’t waste space. This is C++.

```
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;
          etter) == 12
```

5 bool = 2^5 states = 32 states?
Why not an enum?



P.9: Don’t waste time or space

4) 'Cheap' code (P.8, P.9 and P.10)

It is easier to reason about constants…
…both for people and for compilers.

We will cover this later…
…but, just as an appetizer:

```
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

5) Standard code (P.2, P.13 and P.12)

The standard is... well, standard. So use it.

If you _really_ need to use an extension...
Encapsulate it, so it can turned off if needed.

Try to use up_to_date C++...
Modern C++ (C++ 11, C++14... C++17)

Try and use the Algorithms library offered in the STL, and build algorithms based on that.

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

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

Obviously, common sense is more important.

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

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

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

5) Standard code (P.2, P.13 and P.12)

Let the computer do boring tasks for you.

Clang offers tons of interesting tools:

- **clang-check**: Can be used to do error basic checking and AST dumping.
- **clang-format**: Can be used to help doing some formatting on your code.
- **clang-tidy**: OMG! clang-tidy: can diagnose a variety of things. Modernize your code!
- **clang-rename**: Is a refactoring tool.
- **scan-build**: Can be used to do use the static analyzer on the code.
- **and more**: Example: tools that help debugging memory issues.

P.12: Use supporting tools as appropriate

Summing up (unordered_map<CppGuidelines::value_type, this_talk:: ::value_type>)

1) Meaningful code contains it's own meaning.

P.1: Express ideas directly in code, P.3: Express intent.

2) Legible code is easy to read an to understand.

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

3) Error-free code has no errors.

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) 'Cheap' code perform faster.

P.8: Don't leak any resources, P.9: Don't waste time or space, P.10: Prefer immutable data to mutable data.

5) Standard code follows Software Engineer principles.

P.2: Write in ISO Standard C++, P.12: Use supporting tools as appropriate,
P.13: Use support libraries as appropriate.

Next topic?

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