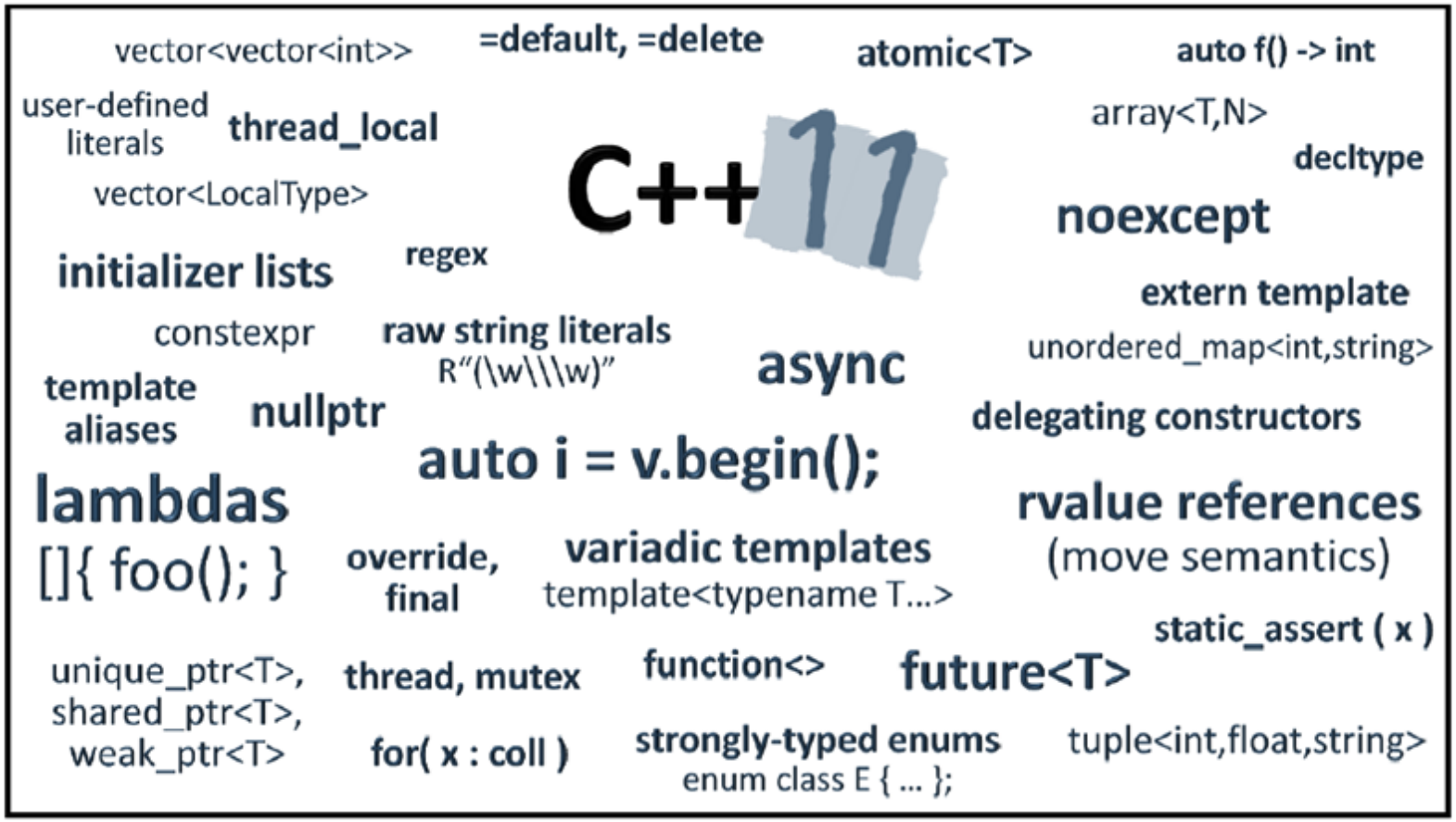
C++ Express

9 Août

Pascal Havé

<pascal.have@ifpen.fr>



C++11

- Résultat d'un long travail depuis la précédente norme C++03
  - Conserver la stabilité et la compatibilité avec C++98
  - Principe du zero-overhead : pas de surcoût si pas utilisé
  - Extension de la bibliothèques standard plus que du langage lui-même
  - Augmenter les performances et résoudre des problèmes actuels
  - « Rendre le C++ facile à apprendre et à enseigner sans enlever les fonctionnalités requises par les programmeurs experts »
- Déjà bien implémentée dans différents compilateurs[1]
  - Clang, GCC, Intel, Microsoft sont les plus avancés (ordre décroissant)
- Approuvée par l'ISO le 12 Août 2011
  - La norme fait désormais 1338 pages
  - Dernier draft gratuit: N3242

[1] http://wiki.apache.org/stdcxx/C++0xCompilerSupport

MSVC: http://blogs.msdn.com/b/vcblog/archive/2011/09/12/10209291.aspx

GCC: http://gcc.gnu.org/gcc-4.7/cxx0x_status.html

LLVM: http://clang.llvm.org/cxx_status.html#cxx0x

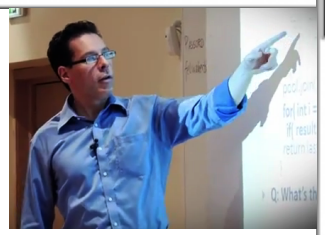
XLC: https://www.ibm.com/developerworks/mydeveloperworks/blogs/5894415f-be62-4bc0-81c5-3956e82276f3/entry/xlc_compiler_s_c_11_support50?lang=en

C++11 : simplification ou complexification?

"C++11 Feels Like a New Language"

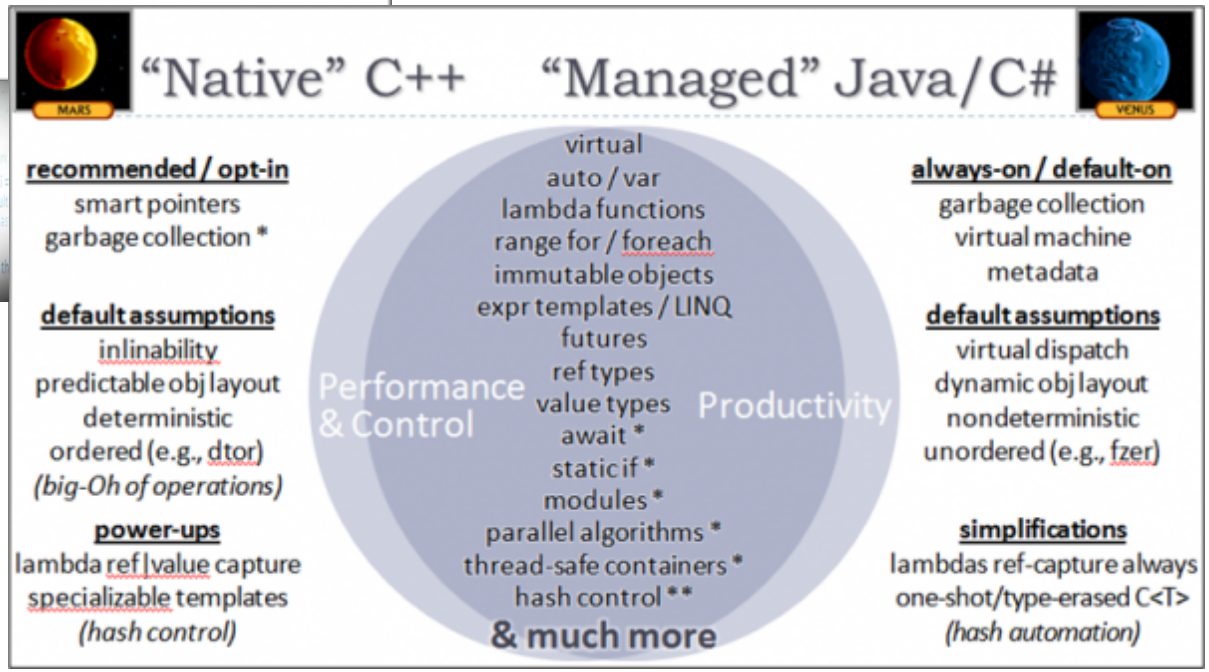
- Corollary: Lots of what people "know" about C++ is no longer true.
- Changes to coding **style/idioms/guidance.**
  - That's why it feels new. Style/idioms/guidance define a language.
  - Features that significantly change style/idioms/guidance include:

Core Language		Library
auto	range-for	smart pointers
lambdas	move semantics	async & future
uniform initialization		



Herb Sutter

- Plus de concepts
- Plus de finesse
- Plus de contrôle
- …
- Plus complexe
- …
- La tendance des langages modernes



C++11: Morceaux choisis

- Fonctionnalités déjà implémentées dans Clang (≥3.1), GCC (≥4.7), Intel (≥13), MSVC (≥11)
  - `auto`, `decltype`, new function declaration syntax
  - Lambda expressions and closures
  - Range-based for-loop
  - Extended STL
  - `override` and `final`
  - Defaulted And Deleted Functions (not in MSVC 11)
  - R-Value References and move operator
  - Initializer lists and uniform initialization (not in MSVC 11)

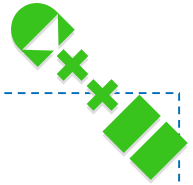
C++11 : auto et decltype

Comment simplifier la définition de variables issues de retour complexe

- « Le compilateur connaît les types, autant lui demander de le faire pour nous. »

⚠ En se détachant des typages, vous supposez que le compilateur peut correctement déduire les types. S'il y a méprise cela sera votre faute, pas la sienne.

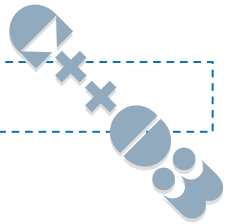
```
std::vector<std::set<int> > v(5);
for(std::vector<std::set<int> >::iterator i = v.begin(); i != v.end(); ++i) { /* ... */ }
std::set<int> & s1 = v[1];
s1.insert(5);

const std::set<int> & s2 = v[2];
std::set<int>::const_iterator finder2 = s2.find(5);
if (finder2 != s2.end()) { /* found ! */ }
```



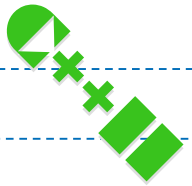
```
std::vector<std::set<int> > v(5);
for(auto i = v.begin(); i != v.end(); ++i) { /* ... */ }
auto & s1 = v[1]; /* inférence de type */
decltype(v[1]) & s1b = v[1]; /* type à partir d'une expression */
s1.insert(5);

const auto & s2 = v[2];
auto finder2 = s2.find(5); /* inférence de type */
const decltype(std::set<int>().find(int())) finder2b = s2.find(5); /* à partir d'une expression */
if (finder2 != s2.end()) { /* found ! */ }
```

C++11 : Nouvelle syntaxe des fonctions

- Nouvelle syntaxe d'un style plus « fonctionnelle »
  - Bien plus qu'une « coquetterie »
  - Massivement exploité dans d'autres mécanismes de C++11

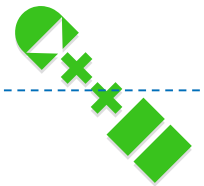


```
int f03(int i) { return i+1; }
```



```
auto f11(int i) -> int { return i+1; }
```

- Mixable avec les nouveaux `auto` et `decltype`



```
template<typename U, typename V>
auto f(U u, V v) -> decltype(u+v) { return u+v; }
```

Mélange de la phase de déclaration et d'implémentation

```
std::function<int(int)> localf = f11;
localf(1);
auto x = f(1,1.);
```

  - <functional> contient le type `std::function` créé pour décrire les fonctions
    - Bien mieux que des pointeurs de fonctions

C++11 : λ functions

- Avec la nouvelle syntaxe de fonctions, il est possible de dissocier le nommage d'une fonction et son corps.

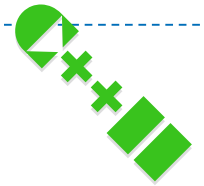
Une lambda fonction est une fonction anonyme définie par une signature

(intput)->output

et précédée d'une relation de fermeture vis-à-vis de son environnement.

Il existe deux types de relation à l'environnement: référence et copie:

[] - Capture nothing.
[=] - Capture everything by value (read-only)
[&] - Capture everything by reference.
[var] - Capture var by value;
nothing else, in either mode, is captured.
[&var] - Capture var by reference;
nothing else, in either mode, is captured.



```
#include <iostream>
#include <functional>
#include <vector>
#include <algorithm>

int globalf1(int & i) { i = 2; return i; }
auto globalf2(int&) -> int; // prédéclaration de globalf2

int main()
{
  std::vector<int> v(5); // Des données
  int local = 10; // variable locale

  // C++03 style
  std::for_each(v.begin(), v.end(), globalf1);

  // C++11 style
  struct F1 { int operator()(int & i) const { return i = 1; } } f1;
  std::for_each(v.begin(), v.end(), f1); // impossible en C++03 si F1 est un type local

  // Déclaration d'une fonctions int -> int
  std::function<int(int&)> f2;

  auto f3 = [](int & i)->int { return i+2; };
  f2 = f3; // car f3 est bien du type int&->int

  // si ->int est omis, alors le type de retour est implicitement void
  std::for_each(v.begin(), v.end(), [](int & i) { i=3; });

  // capture d'une données locale par référence
  auto f4 = [&local](int & i)->int { i += local; return i; };
  f2 = f4; // malgré la capture f4 est toujours du type (int&)->int

  std::for_each(v.begin(), v.end(), [](int & i)  { std::cout << i << std::endl; });

  std::function<void(void)> fend = []{ std::cout << "the end¥n"; }; // no arg and return
  fend();
}

auto globalf2(int& i) -> int { return i+=4; }
```

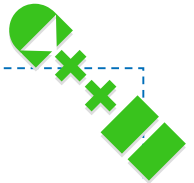
C++11 : range-based for-loop

Comment simplifier l'usage des itérateurs pour l'usage très standard qui est de boucler sur l'ensemble des éléments d'un conteneur.

- Nouvelle syntaxe de `for`:

  for (*for-range-declaration* : *expression*) *statement*

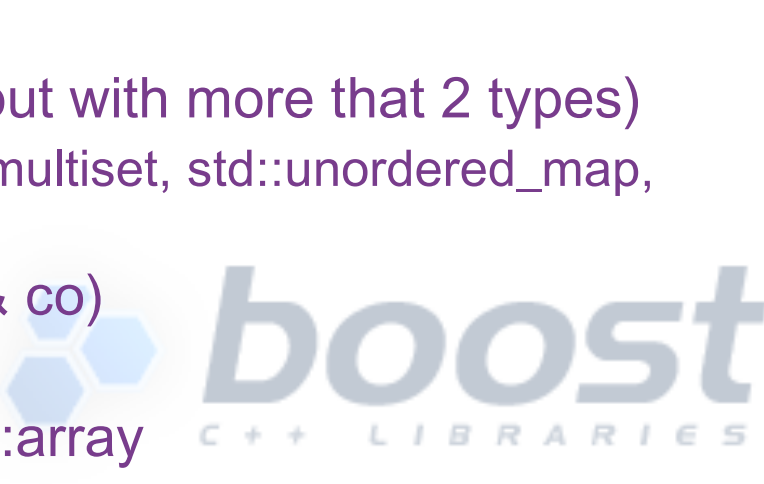
- Valable sur tous les conteneurs supportant des itérateurs (begin, end[
- Valable aussi sur les tableaux C de dimension statique et valarray.



```
#include <vector>
#include <iostream>
#include <valarray>

int main() {
  int array[5] = { 1, 2, 3, 4, 5 };
  for(int i : array) { std::cout << i << std::endl; }

  std::vector<int> v(5);
  for(int & i : v) i += 5;

  std::valarray<double> v2(1., 5);
  for(auto & i : v) i += 5.;

  int * array2 = new int[20];
  for(int & i : array2) i = 20; // error: not static size for array2

  return 0;
}
```

C++11 : Extended STL

- Pour répondre à de nouveaux besoins et suite à des initiatives variées, la STL a été étendue, et en particulier:
  - Threading facilities
  - Tuple (std::tuple is like std::pair but with more that 2 types)
  - std::unordered_set, std::unordered_multiset, std::unordered_map, std::unordered_multimap
  - Smart pointers (std::shared_ptr & co)
    - std::auto_ptr is deprecated
  - template<typename T,int N> std::array
  - <regex> : regular expressions
  - <random> : many random number generators
  - …



C++11 : final, override, default, delete on methods

How to improve the intention of the developper on some default C++ behaviour

- `final` means « no override »
- `override` means « override inherited method »
- `default` means « use default behaviour »
- `delete` means « no not use default behaviour »

```
#ifdef __GXX_EXPERIMENTAL_CXX0X__
#define CPP11
#else /* __GXX_EXPERIMENTAL_CXX0X__ */
#define final
#define override
#endif /* __GXX_EXPERIMENTAL_CXX0X__ */
```

C++03 / C++11 directive switch

```
struct Base final { int i; }; // Base class: a final class, no virtual dtor

struct DerivedBase : public Base { }; // illegal: cannot override final struct

class BaseClass {
public: virtual ~BaseClass() { }
public:virtual int f(int)       { return 0; }
public:virtual int g()    final { return 0; }
public:virtual int h()          { return 0; }
};

class DerivedClass final : public BaseClass {
public:
#ifdef CPP11
  DerivedClass() = default; // use default constructor
#endif /* CPP11 */
  virtual int f(double) override { return 1; } // does not override any method of Class
  int g() { return 1; } // illegal: cannot override final methode
  int h() override { return 0; } // ok override BaseClass::h
};
```

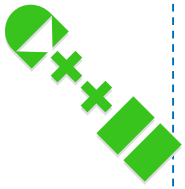
```
class NonCopyable {
public:
  NonCopyable() : m_data(new int[5]) { }
  virtual ~NonCopyable() { delete[] m_data; }
#ifdef CPP11
  NonCopyable(const NonCopyable &) = delete;
  void operator=(const NonCopyable &) = delete;
#else /* CPP11 */
private: // C++03 trick
  NonCopyable(const NonCopyable &);
  void operator=(const NonCopyable &);
#endif /* CPP11 */
private:
  int * m_data; // default copy is not safe with pointer
};

int main()
{
  NonCopyable a;
  NonCopyable b(a); // copy constructor
  a = b; // operator=
  return 0;
}
```

C++11 : r-value ref and move operator

How to improve the performance by removing some useless copy of temporary objects

- A r-value reference can be explicitly described as an argument of a function using `&&` qualifier
- Moving without any copy all temporary values
- `std::move` method for forcing a data to be a temporary

```
class MyVector {
public:
  MyVector(int val, const std::string name) : m_data(new int[N] { }
  virtual ~MyVector() { delete[] m_data, m_data = nullptr; }
  int size() const { return N; }
public:
  MyVector(const MyVector & v) : m_data(new int[N] {
    for(int i=0;i<N;++i) m_data[i] = v.m_data[i];
  }
  MyVector & operator=(const MyVector & v) {
    if (this != &v) { for(int i=0;i<N;++i) m_data[i] = v.m_data[i]; }
    return *this;
  }
  MyVector(MyVector && v) { m_data = v.m_data; v.m_data = nullptr; }
  MyVector & operator=(MyVector && v) {
    if (this != &v) { delete[] m_data; m_data = v.m_data; v.m_data = nullptr; }
    return *this;
  }
private:
  int * m_data;
};
```



rvalue and move semantics. `m_data` are stolen from temporary object `v`

C++11 : Initializer lists and uniform initialization

```
int array[] = { 1, 2, 3, 4 };
```

is a convenient style for setting a collection of data.
Is there any way to reuse this for complex object ?

… and more.

```
#include <iostream>
#include <vector>

struct TrivialObject
{
  float first;
  int second;
  int third; // ignored if two args initialization
};

class Collection
{
public:
  Collection(std::initializer_list<int> l)
     : m_n(l.size()), m_vector(l) { }
  Collection(int n)
     : m_n{n} { } /* uniform initialization */
  void info() const
     { std::cout << "Size=" << m_n << std::endl; }
private:
  int m_n;
  std::vector<int> m_vector;
};
```

```
int main() {
  int tab[] = { 1, 2, 3, 4 };
  int n = sizeof(tab) / sizeof(int);
  for(int i=0; i<n;++i)
    std::cout << "tab[" << i << "] = " << tab[i] << std::endl;

  //One TrivialObject, with first=0.43f and second=10 and third = ??
  TrivialObject scalar = {0.43f, 10};
  TrivialObject anArray[] = { {13.4f, 3, 1},
                              {43.28f, 29},
                              {5.934f, 17} }; //array of three TrivialObjects

  // Many styles of initialization (equivalent)
  Collection c1 = { 1, 2, 3, 4 };
  Collection c2({ 1, 2, 3, 4 });
  Collection c3 { 1, 2, 3, 4 };

  // OK also for scalar but...
  int a { 6 };

  // { } will prefer initializer_list rather than scalar initialization
  Collection c5 { 4 }; // initializer list ctor
  c5.info();
  Collection c6 ( 4 ); // int ctor
  c6.info();
}
```

C++11: … and more features

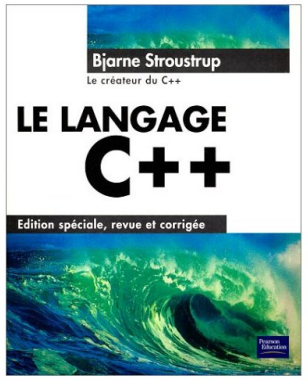
- Already available features in Clang (≥3.1), GCC (≥4.7), Intel (≥13), MSVC (≥11)
  - `static_assert`
  - `extern template`
  - `long long, nullptr`
  - Right Angle Brackets
  - Local and Unnamed Types as Template Arguments
  - Strongly-typed `enums`
  - Built-in Type Traits
- And more for the future
  - constexpr
  - Explicit conversion operators
  - Template aliases
  - Variadic Templates
  - New string and user-defined literals
  - Inherited constructors
  - Modification to the definition of plain old data

```
template< typename First, typename Second, int third>
class SomeType;

template< typename Second>
typedef SomeType<OtherType, Second, 5> TypedefName;

template< typename Second>
using TypedefName = SomeType<OtherType, Second, 5>;
```

Références

- Le langage C++

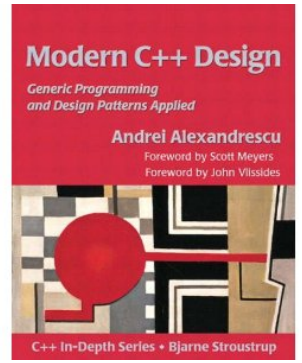
  Bjarne Stroustrup, 2004, Pearson Education
  Next edition for jan. 2013



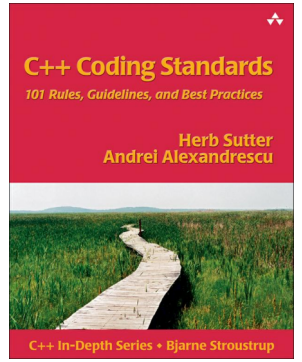
- Modern C++ Design :

  "Generic Programming and Design Patterns Applied"

  Andrei Alexandrescu, 2001, Addison-Wesley

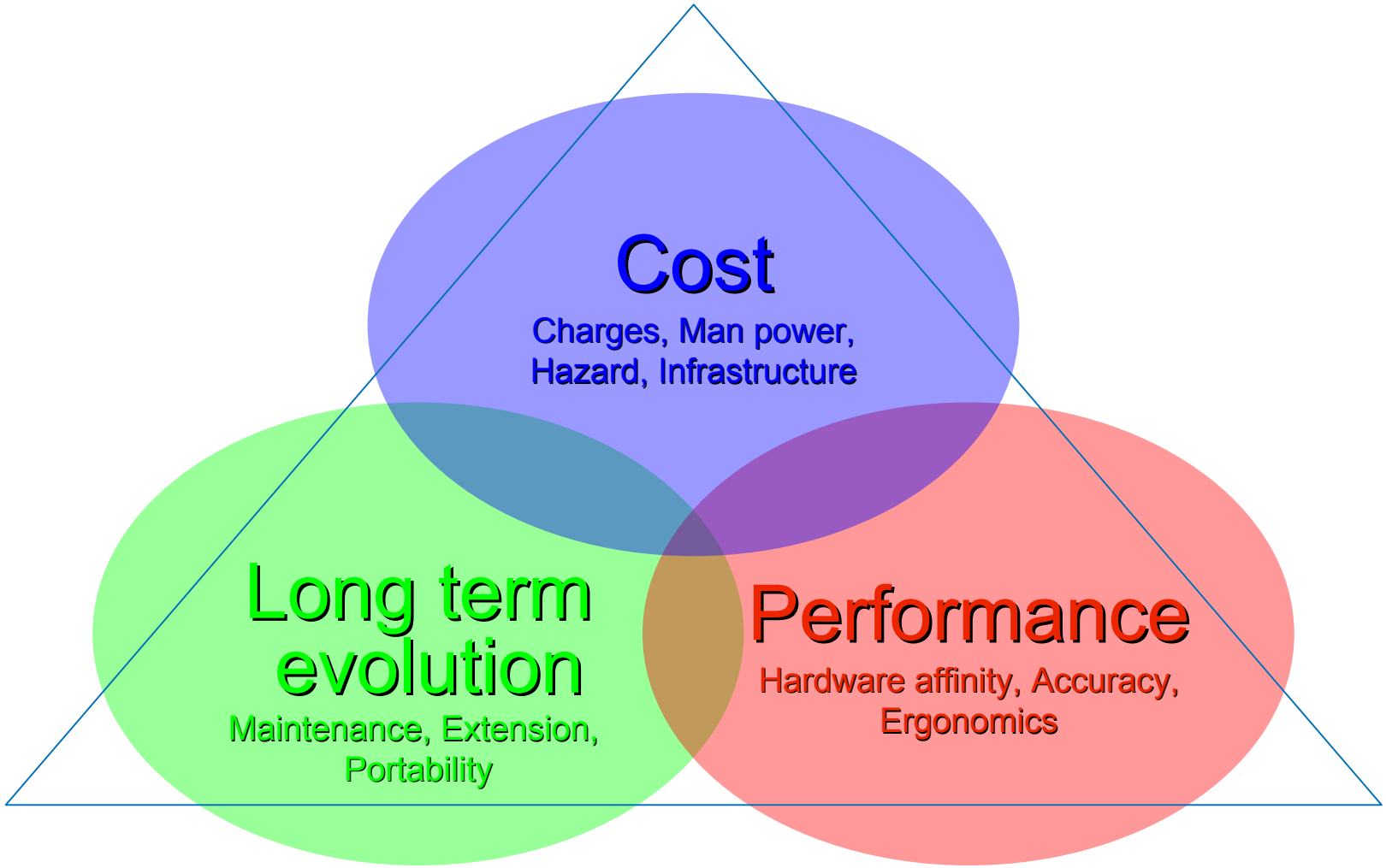


- C++ Coding Standards: 101 Rules, Guidelines, and Best Practices
  Herb Sutter, Andrei Alexandrescu, 2004, Addison-Wesley



Best practice : motivations

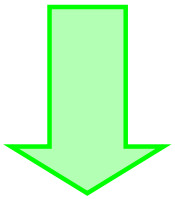
- good compromise ?



Best practice : your subject

You **don't** know what you want

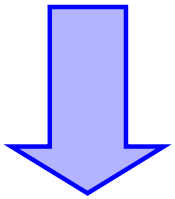
- Research project
- Your customer may change
- Prospective approach



Flexibility

You know what you want

- Industrial project
- One fixed target
- Reproduction



Stability

And performance ?

Best practice

Premature optimization is the root of all evil

The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming.

D. Knuth in *Computer Programming as an Art* (1974)

Best practice : Area of your project

- **Foresee** the area of your project

Iterative *specifications* for open areas		Full specifications for industrial project
Rome was not built in a day		

- The team : team skills *vs* specifications

Iterations driven by the team		A team driven by specifications
Product first, Agile methods		Documentation first, V cycle method

- Validate your production

Iteration done, NEXT*!*		Job Done
Intermediate validation by customer		Final validation by customer

Refactor, not because you *know* the abstraction, but because you want to *find it*

Best practice : Long term evolution

- The key of a long term evolution is your flexibility to integrate *unpredictable* concept

OOP ideas

Divide (and conquer) your problem into small and simple concepts

- Each concept must have a well defined scope of responsibility and its interaction and with the others
- Check the orthogonality of your set of concepts
- Define and check the validity domain of your implementation
- Think about usage before thinking about implementation
  End user / developer driven development.

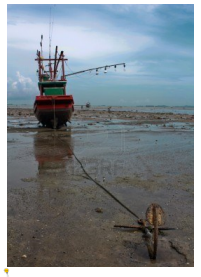
…

even for non-OO programming

Think **before** coding

and think after coding (peer reviewing)

Best practice : The good and the bad



- Anti-patterns
  - God object : Concentrating too many functions in a single part of the design
  - Boat anchor : Retaining a part of a system that no longer has any use
  - Lava flow : Retaining undesirable code because removing it is too expensive or has unpredictable consequences
  - Spaghetti code : Programs whose structure is barely comprehensible, especially because of misuse of code structures
  - Reinventing the square wheel : Failing to adopt an existing solution and instead adopting a custom solution which performs much worse than the existing one
  - And many more…

- Design patterns
  - Singleton : Ensure a class has only one instance, and provide a global point of access to it.
  - Flyweight : Use sharing to support large numbers of similar objects efficiently.
  - Factory method : Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
  - Iterator : Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
  - And more…

Best practice : make your code S.O.L.I.D.

S.O.L.I.D. principles

Following the rules on the *paint* can won't teach you how to *paint*.

- Single responsibility principle
  - an object should have only a single responsibility.
- Open/closed principle
  - "software entities … should be open for extension, but closed for modification".
- Liskov substitution principle
  - "objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program". See also design by contract.
- Interface segregation principle
  - "many client specific interfaces are better than one general purpose interface."
- Dependency inversion principle
  - one should "Depend upon Abstractions. Do not depend upon concretions."

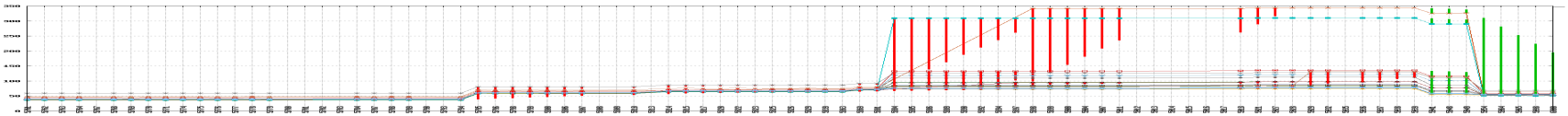
Best practice : optimization

When your concepts are stable, you can optimize from the algorithmic root level to the run time level.

- Optimization levels
  - Algorithmic level
    - complexity measure
  - Design level
    - OO principles
  - Source code level
    - implementation performance
  - Compile level
    - generated code and compiler
  - Run time level
    - system and hardware tuning

Best practice : optimization

- Optimize only if you can prove its benefits (and evaluate the new efficiency if possible)
  - Use profiler, counter, timer, checker… to detect bottleneck (don't optimize before a measure)
  - Avoid single point of failure (conception and management)
  - However, you can define a lower bound for the performance at each iteration
- No intrusive optimization before stable concepts (if "stable" is possible)
- Follow the progress of your devs using a test base
  - Non regression
  - Performance improvements



Leave well alone

Perfectionism can be counter-productive