Sémantique par valeur en C++

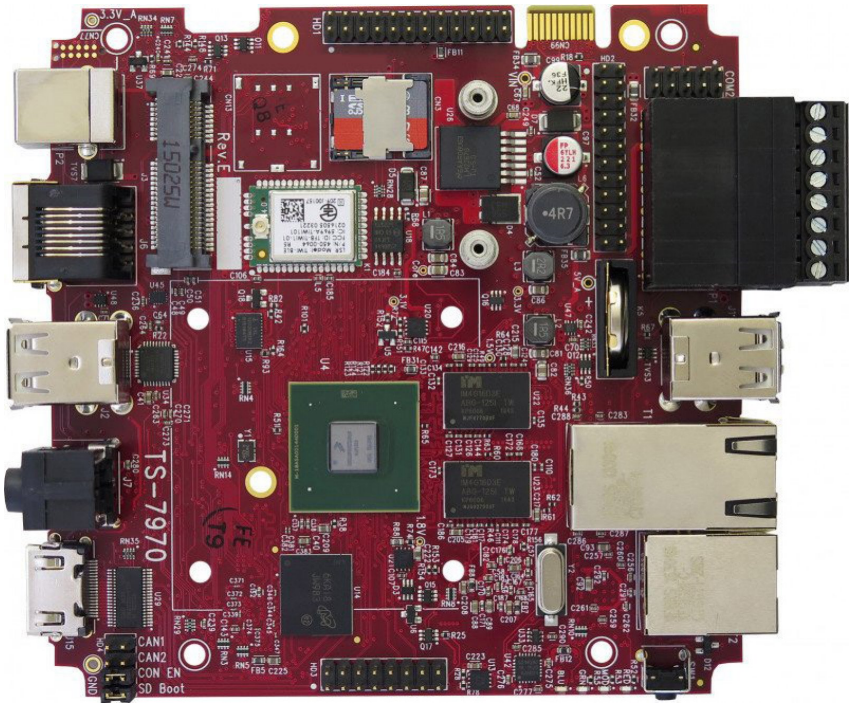
Une approche moderne

Charles Perry

Savoir-Faire Linux

January 26, 2023

À propos du présentateur



Équipe ingénierie produit:

- Intégration des composant logiciels dans une plateforme embarquée
- Intégration du noyau Linux et développement de pilotes
- Développement d'applications
- Et bien plus!

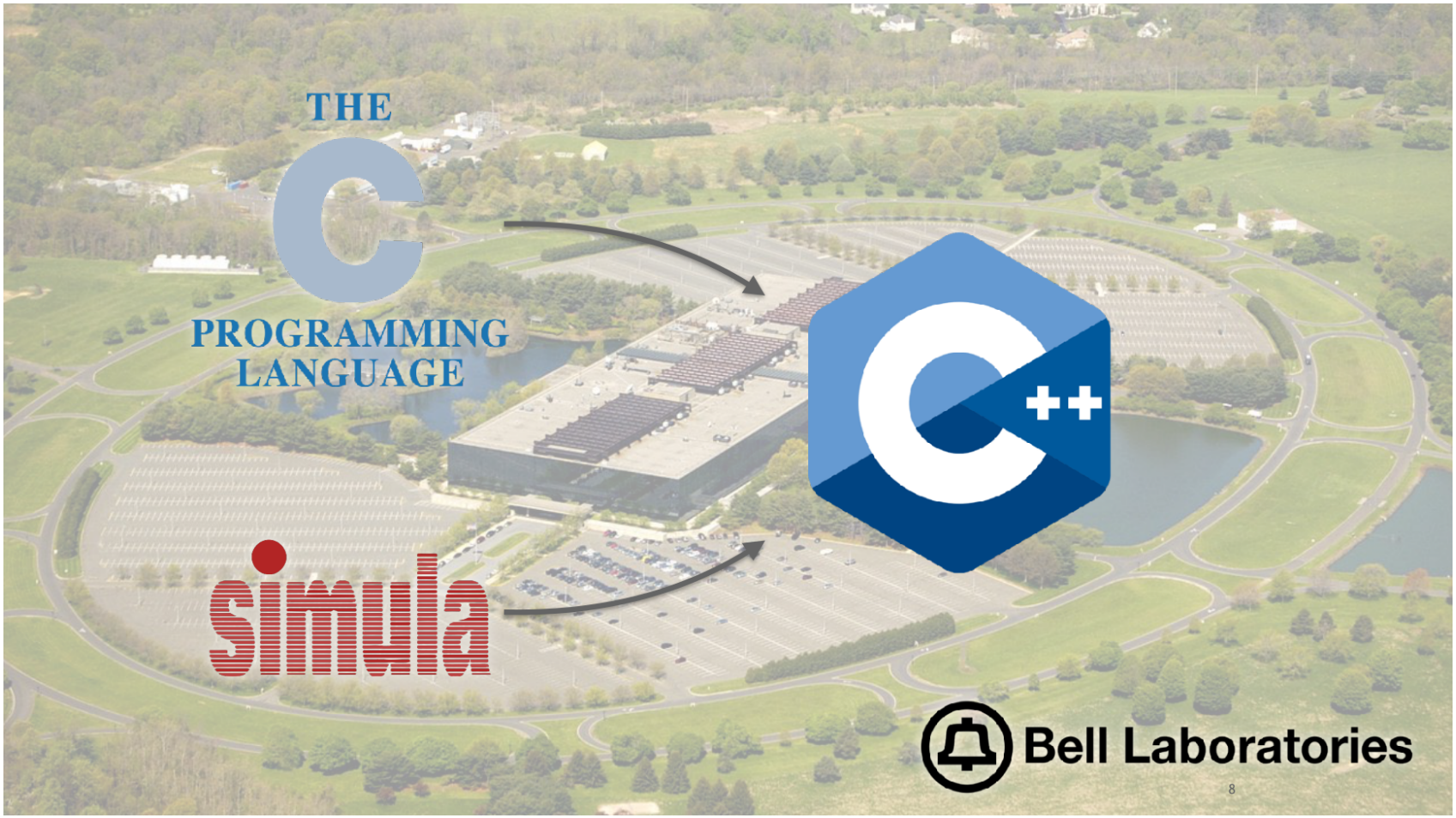
À propos de la présentation

- Matériel original: "Back To Basics: Value Semantics" par Klaus Iglberger (Cppcon 2022)
- Disponnible sur github

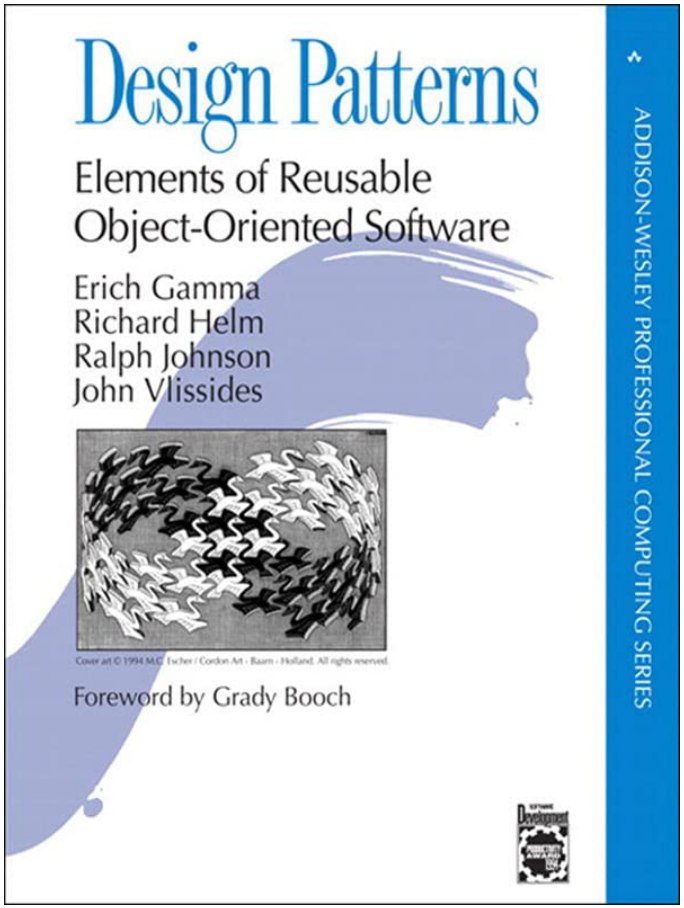
Un peu d'histoire

- 1979: Utilise simula dans sa thèse de doctorat intitulée "Communication and control in distributed computer systems"
- 1979: Rejoint le laboratoire Bell au New Jersey ("Bell Labs")
- 1985: C++ est disponible au grand public

Un peu d'histoire

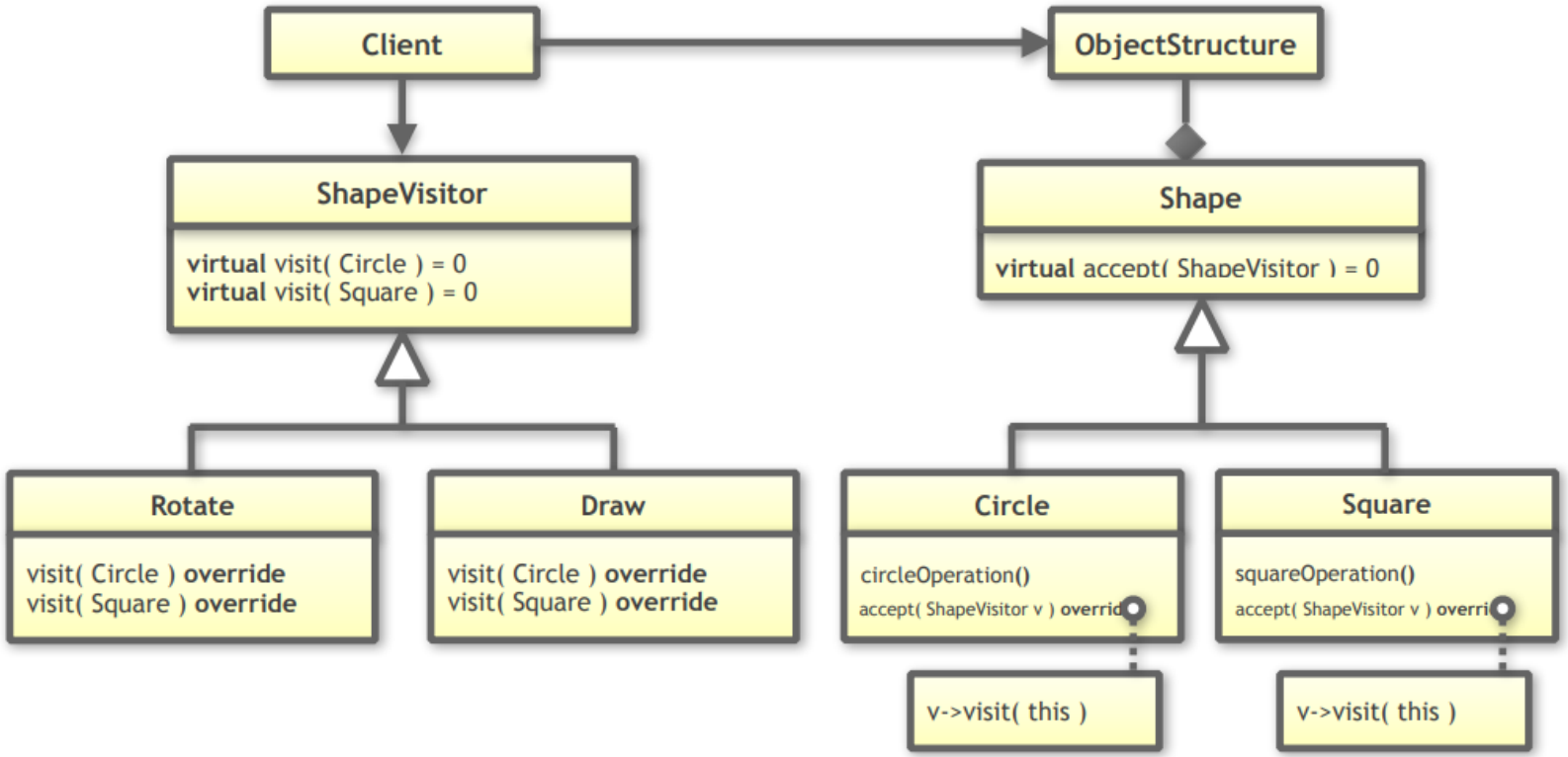


Design Patterns

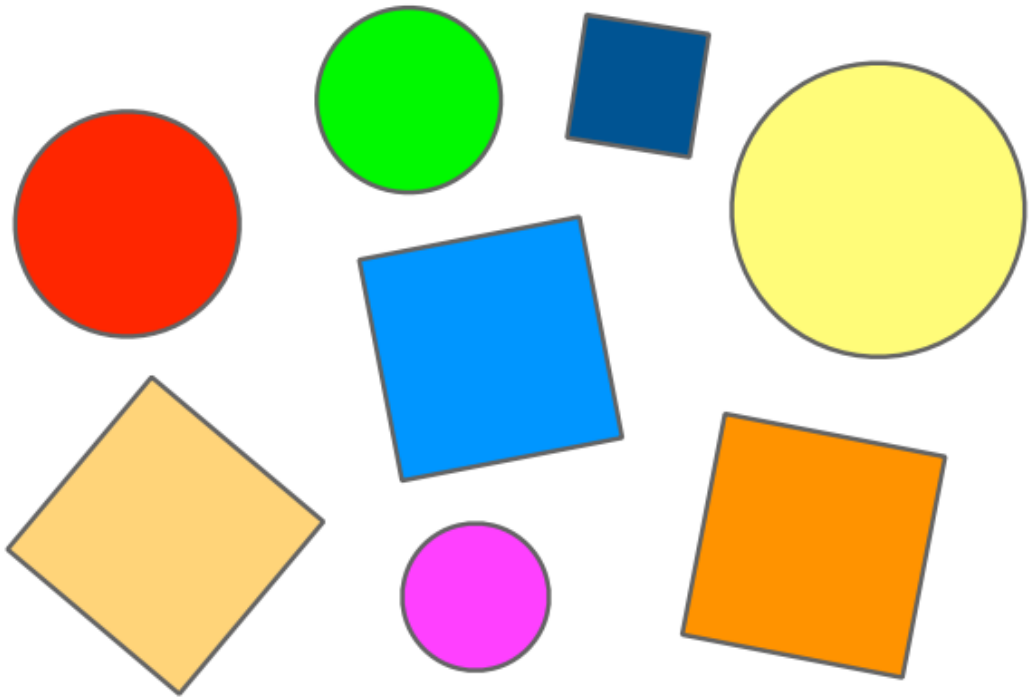


- Livre du "Gang of four"
- Publié en 1994
- Jette les bases de 23 "design patterns" utilisant le principe de programmation orientée objet (POO).
- Tous les "design patterns" reposent sur le principe d'héritage.

Visiteur


Client
ObjectStructure
ShapeVisitor
virtual visit(Circle) = 0
virtual visit(Square) = 0
Shape
virtual accept(ShapeVisitor) = 0
Rotate
visit(Circle) override
visit(Square) override
Draw
visit(Circle) override
visit(Square) override
Circle
circleOperation()
accept(ShapeVisitor v) overri
Square
squareOperation()
accept(ShapeVisitor v) overri
v->visit(this)
v->visit(this)

Exemple: dessiner des formes



Dessiner des formes: POO classique

```
class Circle;
class Square;

class ShapeVisitor
{
public:
  virtual ~ShapeVisitor() = default;
  virtual void visit( const Circle& ) const = 0;
  virtual void visit( const Square& ) const = 0;
};
```

Dessiner des formes: POO classique

```
class Shape
{
public:
  Shape() = default;
  virtual ~Shape() = default;

  virtual void accept( const ShapeVisitor& ) = 0;
};
```

Dessiner des formes: POO classique

```
class Circle : public Shape
{
public:
  explicit Circle( double rad, /* ... */ )
    : radius_{ rad }, /* ... */
    {}

  double getRadius() const;
  // ... getCenter(), getRotation(), ...

  void accept( const ShapeVisitor& ) override;

private:
  double radius_;
  /* ... */
};
```

Dessiner des formes: POO classique

```
class Square : public Shape
{
public:
  explicit Square( double s, /* ... */ )
    : side_{ s }, /* ... */
    {}

  double getSide() const;
  // ... getCenter(), getRotation(), ...

  void accept( const ShapeVisitor& ) override;

private:
  double side_;
  /* ... */
};
```

Dessiner des formes: POO classique

```
class Draw : public ShapeVisitor
{
public:
  void visit( const Circle& ) const override;
  void visit( const Square& ) const override;
};
```

Dessiner des formes: POO classique

```
void drawAllShapes(
  const std::vector<std::unique_ptr<Shape>>& shapes)
{
  for(const auto& s : shapes)
  {
    s->accept(Draw{})
  }
}
```

Dessiner des formes: POO classique

```
int main()
{
  using Shapes = std::vector<std::unique_ptr<Shape>>;

  Shapes shapes;
  shapes.emplace_back(std::make_unique<Circle>( 2.0 ));
  shapes.emplace_back(std::make_unique<Square>( 1.5 ));
  shapes.emplace_back(std::make_unique<Circle>( 4.2 ));

  drawAllShapes(shapes);
}
```

Désavantages de l'approche POO classique

Le design POO classique a plusieurs désavantages:

- Deux hiérarchies de classes (intrusif).
- Deux appels de fonction virtuelle par opération (performance).
- Nécessite d'accéder aux objets via un pointeur (complexité accidentelle).
- Plusieurs allocations dynamiques petites (performance).
- Gestion manuelle de la mémoire (prompt aux erreurs).

std::variant contre-attaque


```
using Shapes = std::variant<Circle, Square>;
```

std::variant?

- Introduit avec C++17
- Représente une valeur d'une liste de type (one of).
- Type-safe union
- Pas d'allocation dynamique.

```
class std::variant<int, float, double>
{
    union {
        int a;
        float b;
        double c;
    } value_;
    int active_;
public:
    /* ... */
}
```

Dessiner des formes: Sémantique par valeur

```
class Circle
{
public:
  explicit Circle( double rad, /* ... */ )
    : radius_{ rad }, /* ... */
    {}

  double getRadius() const;
  // ... getCenter(), getRotation(), ...

private:
  double radius_;
  /* ... */
};
```

Dessiner des formes: Sémantique par valeur

```
class Square
{
public:
  explicit Square( double s, /* ... */ )
    : side_{ s }, /* ... */
    {}

  double getSide() const;
  // ... getCenter(), getRotation(), ...

private:
  double side_;
  /* ... */
};
```

Dessiner des formes: Sémantique par valeur

```
class Draw
{
public:
  void operator()( const Circle& ) const;
  void operator()( const Square& ) const;
};
```

Dessiner des formes: Sémantique par valeur

```
using Shape = std::variant<Circle, Square>;

void drawAllShapes(
  const std::vector<Shape>& shapes)
{
  for(const auto& s : shapes)
  {
    std::visit(Draw{}, s);
  }
}
```

Dessiner des formes: Sémantique par valeur

```
int main()
{
  using Shapes = std::vector<Shape>;

  Shapes shapes;
  shapes.emplace_back( Circle( 2.0 ) );
  shapes.emplace_back( Square( 1.5 ) );
  shapes.emplace_back( Circle( 4.2 ) );

  drawAllShapes(shapes);
}
```

Avantages de l'approche par valeur

Le design par valeur a plusieurs avantages:

- Aucune hiérarchie de classe (non-intrusif).
- Aucun appel de fonction virtuelle (performance).
- Plus besoin d'accéder aux objets via un pointeur (simplicité).
- Aucune allocation dynamique (performance).
- Aucune gestion manuelle de la mémoire (simplicité).

std::visit ???

std::visit

Defined in header <variant>

```
template <class Visitor, class... Variants>
constexpr /*see below*/ visit( Visitor&& vis, Variants&&... vars );
```
(1) (since C++17)

```
template <class R, class Visitor, class... Variants>
constexpr R visit( Visitor&& vis, Variants&&... vars );
```
(2) (since C++20)

Applies the visitor `vis` (*Callable* that can be called with any combination of types from variants) to the variants `vars`.

These overloads participate in overload resolution only if every type in `std::remove_reference_t<Variants>...` is a (possibly const-qualified) specialization of `std::variant`, or a (possibly const-qualified) class `C` such that there is exactly one `std::variant` specialization that is a base class of `C` and it is a public and unambiguous base class.

Effectively returns

```
std::invoke(std::forward<Visitor>(vis),
            std::get<is>(std::forward<VariantBases>(vars))...)
```

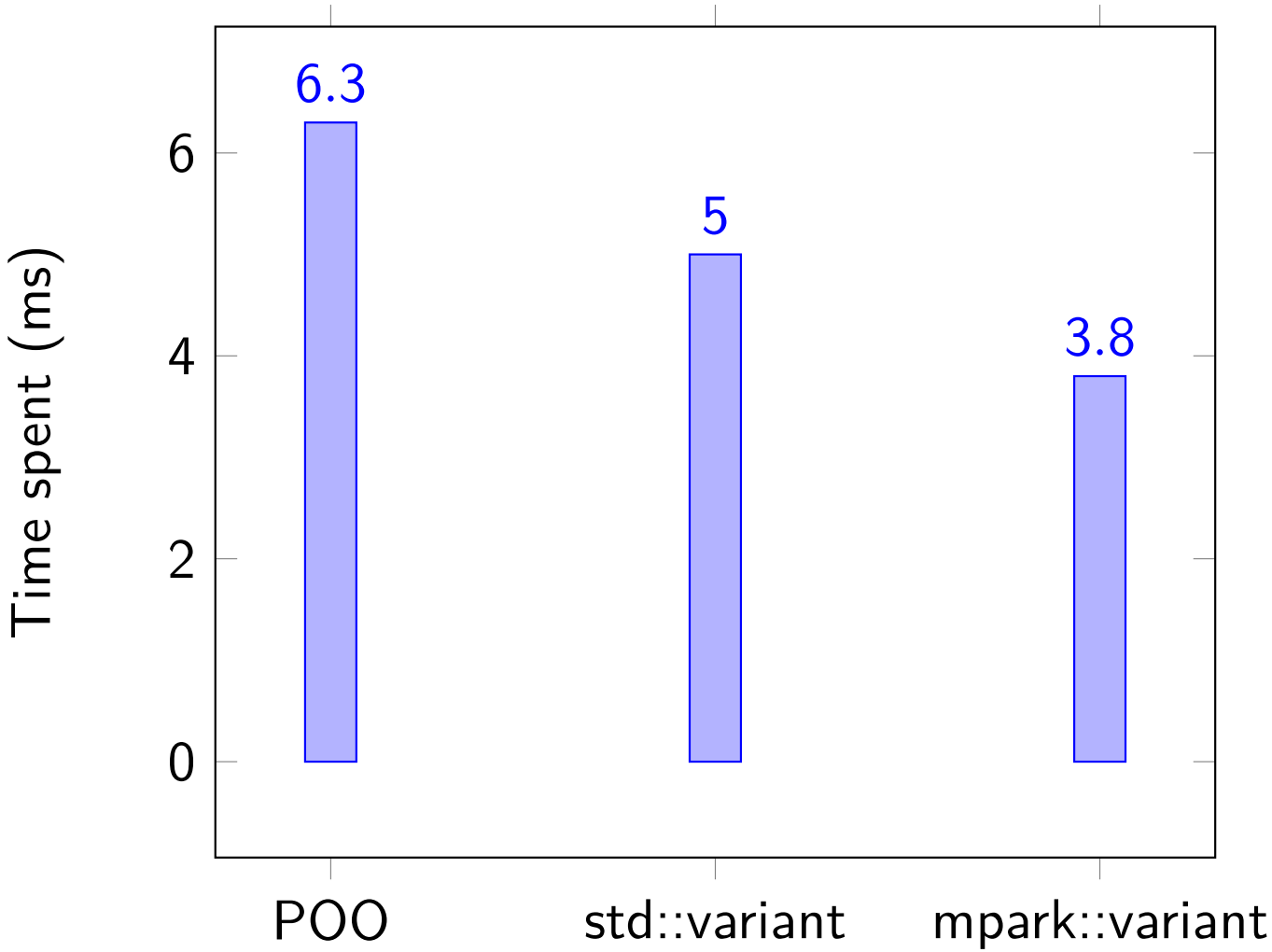
Performance

- À titre qualitatif seulement, les résultats peuvent varier.
- Compiler avec −O1 minimalement.

Performance

- 1 millions de formes.
- GCC 11.3
- Utilisation de google/benchmark pour mesurer le temps CPU et assurer la convergence.
- Intel Core i7-4770 CPU @ 3.40GHz
- Cache:
  - 32 KiB L1 Data
  - 32 KiB L1 Instruction
  - 256 KiB L2
  - 8 MiB L3

Performance



Exemple 2: exactitude du mot clé const

```
void print(const std::vector<int>& vec);

int main()
{
  std::vector<int> v{1,2,3,4};

  const std::vector<int> w{ v };
  const std::span<int> s{ v };

  w[2] = 99; // Compilation error!
  s[2] = 99; // Ok!

  print( v );
}
```

Exemple 2: exactitude du mot clé const

```
void print(const std::vector<int>& vec);

int main()
{
  std::vector<int> v{1,2,3,4};

  const std::vector<int> w{ v };
  const std::span<const int> s{ v };

  w[2] = 99; // Compilation error!
  s[2] = 99; // Compilation error!

  print( v );
}
```

Exactitude du mot clé const

Le mot clé `const` peut prendre différents sens avec une sémantique de référence, ce qui n'est pas le cas avec la sématique par valeur.

- Un pointeur: `const int* const`
- Un span: `const` std::span<`const int`>
- Pour un type à sémantique par valeur, `const` veut dire `const`! Par exemple: `const` std::vector<std::vector<std::string>> ne permet pas de modifier le moindre caractère.

Exemple 3: problèmes d'invalidation

```
void print(std::span<int> s);

int main()
{
  std::vector<int> v{1,2,3,4};

  const std::span<int> s{ v };

  print( s ); // Ok

  v = {5,6,7,8,9}; // s is invalidated here

  print( s ); // Undefined behavior
}
```

Exemple 4: problèmes d'invalidation

```
void print(std::span<const int> s);

int main()
{
  std::vector<int> v{1, -3, 42, 4, -8, 22, 42, 37, 4,9};

  print(v); // 1, -3, 42, 4, -8, 22, 42, 37, 4, 9

  const auto pos = std::max_element(begin(v), end(v));

  v.erase(std::remove(begin(v), end(v), *pos), end(v));

  print(v); // 1, -3, 4, -8, 22, 42, 37, 9
}
```

Exemple 4: problèmes d'invalidation

std::remove, std::remove_if

Defined in header <algorithm>

```
template< class ForwardIt, class T >
ForwardIt remove( ForwardIt first, ForwardIt last, const T& value );          (1)   (until C++20)
template< class ForwardIt, class T >
constexpr ForwardIt remove( ForwardIt first, ForwardIt last, const T& value );     (since C++20)
```

Problèmes d'invalidation

- La plupart des problèmes d'invalidation surviennent lorsqu'une référence est utilisé trop longtemps.
- En règle général, passer un argument par référence ne pose pas problème. Toutefois, retourner d'une fonction par référence peut poser problème.

Exemples de types à sémantique...

... par valeur

- std :: vector et tous les conteneurs de la STL (C++98)
- std :: function (C++11)
- std :: variant (C++17)
- std :: optional (C++17)
- std :: expected (C++23)

... par référence

- T&
- T*
- std :: span
- std :: string_view
- std :: unique_ptr
- std :: shared_ptr
- std :: iterator et autres itérateurs

Caractéristiques de la sémantique...

... par valeur

- Copie profonde (deep copy)
- `const` se propage

... par référence

- Copie superficielle (shallow copy)
- `const` ne se propage pas

Conclusion

Lorsque possible, préférer la sémantique par valeur à la sémantique par référence. Votre code en sera:

- plus facile à comprendre
- plus facile à écrire
- plus facile à maintenir
- potentiellement plus rapide