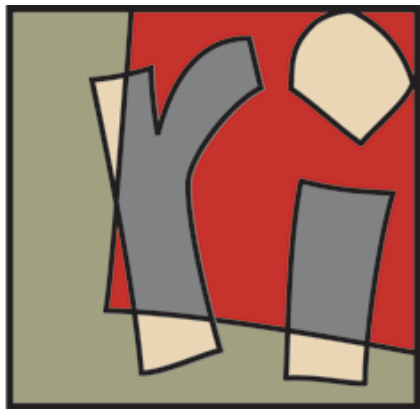


# C++ – Vers 2011 et au-delà



Joel Falcou

LRI UMR 8623 - Université Paris Sud

29,30,31 octobre 2014

# Un peu d'histoire ...

---

## C++ à travers les âges

- 1978 Bjarne Stroustrup travaille sur Simula67 qui s'avère peu efficace
- 1980 Démarrage du projet *C with Classes*, du C orienté objet compilé via CFront.
- 1983 *C with Classes* devient C++
- 1985 Parution de *The C++ Programming Language*
- 1998 Première standardisation de C++ : ISO/IEC 14882 :1998
- 2005 Parution du *C++ Technical Report 1* qui deviendra C++0x
- 2011 Ratification de C++0x sous le nom C++11
- 2014 Ratification de C++14
- 2017,2020,... Prochaines *milestones* du langage

# Pourquoi C++11/14/1y sont-ils importants ?

---

## *Make simple things simple - Stroustrup 2014*

- Les tâches simples doivent s'exprimer simplement
- Les tâches complexes ne doivent pas être outrageusement complexes
- Rien ne doit être impossible
- Ne pas sacrifier les performances !

## Pourquoi évoluer

- C++ est perçu comme un langage d'expert
- C++11/14 réduit ce gap d'apprentissage
- Moins de travail pour les *Language Lawyers*

# L'atelier C++ II / 14

---

## Objectifs

- Se familiariser avec les nouveautés du langage
- Connaître les nouvelles bonnes pratiques
- Pratiquer, Pratiquer, Pratiquer !

## Déroulement

- 3 jours d'alternance théorie/pratique
- Exercices sur machine
- Dernière demi-journée : Atelier modernisation
- Cours interactif, n'hésitez pas à m'interrompre

# Types et Variables

# Types et Variables

---

## Objectifs

- Simplifiez la manipulation des types
- Renforcer la sémantique dès que possible
- Limiter les erreurs silencieuses

## Éléments de réponses

- `nullptr`
- `enum class`
- Initialisation uniforme
- Tableau généraliste
- Inférence de type automatique
- Gestion des temporaires

# nullptr

---

## Problématiques

- Comment spécifier un pointeur nul du bon type ?
- L'ambiguïté de 0
- Les limites de NULL

## nullptr en action

```
#include <iostream>

int main()
{
    int* pi = nullptr;
    float* pf = nullptr;

    if(!pi) std::cout << "pi est nul." << std::endl;

    if(pf == nullptr) std::cout << "pf est nul." << std::endl;

    return 0;
}
```

# Enumérations typées

---

## Problématiques

- En C++03, les énumérations ne sont pas typées
- Sémantique faiblement renforcée
- Erreur logique possible

## Solution

- Notion de `enum class`
- Se comporte comme une `enum`
- Se comporte comme un vrai type à part entière
- C++14 : Choix du type sous-jacent



# Enumérations typées

---

```
#include <iostream>

int main()
{
    enum Color { RED , BLUE };
    enum Fruit { BANANA, APPLE };

    Color a = RED;
    Fruit b = BANANA;

    // Peut etre un warning ???
    if (a == b)
        std::cout << "a et b sont egaux" << std::endl;
    else
        std::cout << "a et b sont differents" << std::endl;

    return 0;
}
```

# Enumérations typées

---

```
#include <iostream>

int main()
{
    enum class Color { RED, BLUE };
    enum class Fruit { BANANA, APPLE };

    Color a = Color::RED;
    Fruit b = Fruit::BANANA;

    // ERREUR DE COMPILATION
    if (a == b)
        std::cout << "a et b sont egaux" << std::endl;
    else
        std::cout << "a et b sont differents" << std::endl;

    return 0;
}
```

# Enumérations typées

---

```
#include <iostream>

int main()
{
    enum class Color : char { RED, BLUE };
    enum class Fruit { BANANA, APPLE };

    Color a = Color::RED;
    Fruit b = Fruit::BANANA;

    // ERREUR DE COMPILATION
    /* if (a == b)
        std::cout << "a et b sont egaux" << std::endl;
    else
        std::cout << "a et b sont differents" << std::endl;
    */
    std::cout << sizeof(a) << "\n";
    std::cout << sizeof(b) << "\n";

    return 0;
}
```

# Initialisation Uniforme

---

## Le challenge

```
#include <vector>

struct employe
{
    int id;
    int age;
    float salaire;
};

int main()
{
    employe Joe = {1, 42, 60000.0f};
    int anArray[] = { 3, 2, 7, 5, 8 };

    // Ne compile pas
    //std::vector<int> a = {1,2,3,4,5};

    // Compile mais verbeux
    std::vector<int> a(5);

    for(std::size_t i=0;i<a.size();++i)
        a[i] = 1+i;

    return 0;
}
```

11 of 77

# Initialisation Uniforme

---

## La solution

```
#include <vector>

struct employe
{
    int id;
    int age;
    float salaire;
};

int main()
{
    employe Joe{1, 42, 60000.0f};
    int anArray[]={ 3, 2, 7, 5, 8 };

    // Construction par liste d'initialiseur
    std::vector<int> a{1,2,3,4,5};

    // Constrction via paire d'iterateur
    std::vector<int> b(&anArray[0],&anArray[0]+5);

    return 0;
}
```

# Initialisation Uniforme

---

## Interaction avec les fonctions

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

struct point2D
{
    int mx,my;
    point2D(int x, int y) : mx(x),my(y) {}
};

void display(point2D const& p)
{
    std::cout << "(" << p.mx << ", " << p.my << ")" << std::endl;
}

void display(std::vector<float> const& )
{
}

point2D some_position()
{
    return {2, 7};
}

int main()
{
    int if static_cast<int>(4.f) };
13 of 77
```

# Tableau Généraliste

---

## Pourquoi autre chose que $T[N]$ ?

- Les tableaux C ne sont pas des objets à part entière
- Difficile à passer en paramètres et à renvoyer d'une fonction
- Pas une vraie Séquence

```
// Fonction prenant un tableau de 5 int en parametre  
// et retournant un tableau de 5 int ...
```

```
int ( &f( int (&arr)[5] ) )[5];
```

# Tableau Généraliste

---

## std::array<T,N>

- Objet copiable et manipulable naturellement
- Se comporte comme une Séquence
- Fournit begin(), end(), [], size() ...

```
// Fonction prenant un tableau de 5 int en parametre  
// et retournant un tableau de 5 int ...
```

```
std::array<int,5> f( std::array<int,5> const & arr )  
{  
    std::array<int,5> that{arr};  
  
    for(int i=0;i<that.size();i++)  
        that[i] += i;  
  
    return that;  
}
```



# Inférence de type automatique

---

## Problématiques

- Déclarer une variable nécessite son type
- Dans le cas des initialisations, le type est déjà disponible
- Dans le cas des retour de fonctions, il y a répétition
- Dans certains contextes, le type est complexe à exprimer

```
float i = 3.f
```

```
std::vector<std::string> v;  
std::vector<std::string>::size_type sv = v.size();
```

```
template<typename Container>  
typename Container::const_iterator begin(Container const& c)  
{  
    return c.begin();  
}
```

# Inférence de type automatique

---

## Le mot-clé auto

- auto remplace le type d'une variable lors d'une initialisation
- Son type est déduit du type de son initialiseur
- Incompatibilité avec C89

```
int main()
{
    auto x = 0;
    auto c = 'a';
    auto d = 0.5;
    auto usa_national_debt = 1440000000000LL;

    auto cpx = c + x;
    auto dd = 1/d;
}
```

# Inférence de type automatique

---

## Le mot-clé `decltype`

- Détermine le type d'une expression à la compilation
- Contrairement à `auto`, il déduit le type exact

```
#include <vector>

int main()
{
    std::vector<int> v{1,2,3,4,5,6};

    // a est 'int'
    auto a = v[0];

    // b est 'int&'
    decltype(v[0]) b = v[0];
}
```

# Inférence de type automatique

---

## auto et les qualificateurs de type

- auto est complétable avec des qualificateurs
- Comportement identique à la résolution des types templates

```
#include <vector>

int main()
{
    std::vector<int> v{1,2,3,4,5,6};

    // a est 'int'
    auto a = v[0];

    // b est 'int&'
    auto& b = v[0];

    // c est 'int const&'
    auto const& c = v[0];

    // p est 'int*'
    auto* p = &v[0];
}
```

# Inférence de type automatique

---

## Impact sur le retour des fonctions

- `auto` et `decltype` simplifient l'écriture du prototype des fonctions
- Notion de *trailing return type*

```
template<typename T1, typename T2>  
/* ????? */  
add(T1 const& a, T2 const& b)  
{  
    return a+b;  
}
```

# Inférence de type automatique

---

## Impact sur le retour des fonctions

- `auto` et `decltype` simplifient l'écriture du prototype des fonctions
- Notion de *trailing return type*

```
// C++ 11
template<typename T1, typename T2>
auto add(T1 const& a, T2 const& b) -> decltype(a+b)
{
    return a+b;
}
```

# Inférence de type automatique

---

## Impact sur le retour des fonctions

- `auto` et `decltype` simplifient l'écriture du prototype des fonctions
- Notion de *trailing return type*

```
// C++ 14
template<typename T1, typename T2>
auto add(T1 const& a, T2 const& b)
{
    return a+b;
}
```

# Gestion des Temporaires

---

## *lvalue vs rvalue*

- lvalue : objet avec une identité, un nom
- rvalue : objet sans identité
- La durée de vie d'une rvalue est en général bornée au statement
- Une rvalue peut survivre dans une référence vers une lvalue constante

```
int a = 42; // lvalue
int b = 43; // lvalue
```

```
a = b; // ok
b = a; // ok
a = a * b; // ok
int c = a * b; // ok
a * b = 42; // erreur
```

```
int getx() { return 17;}
```

```
const int& lcr = getx(); // ok
int& lr = getx(); // erreur
```



# Référence vers rvalue

---

## Objectifs

- Discriminez via un qualificateur lvalue et rvalue
- Explicitiez les opportunités d'optimisation
- Simplifier la définition d'interface

## Notation

- T& : référence vers lvalue
- T const& : référence vers lvalue constante
- T&& : référence vers rvalue

# Référence vers rvalue

---

## Exemple

```
#include <iostream>
#include <typeinfo>

void foo(int const&) { std::cout << "lvalue\n"; }
void foo(int&& x) { std::cout << "rvalue\n"; }

int bar() { return 1337; }

int main()
{
    int x = 3;
    int& y = x;
    int const& z = bar();

    foo(x);
    foo(y);
    foo(z);

    foo(4);
    foo(bar());
}
```

# Référence vers rvalue

---

## Le problème du forwarding

```
#include <iostream>
```

```
void foo(int const&) { std::cout << "lvalue\n"; }
```

```
void foo(int&&) { std::cout << "rvalue\n"; }
```

```
void chu(int&& x) { foo(x); }
```

```
int bar() { return 1337; }
```

```
int main()
```

```
{  
    foo(bar());
```

```
    chu(bar());  
}
```

# Référence vers rvalue

---

## Le problème du forwarding

```
#include <iostream>

void foo(int const&) { std::cout << "lvalue\n"; }
void foo(int&&) { std::cout << "rvalue\n"; }

void chu(int&& x) { foo( std::forward<int>(x) ); }

int bar() { return 1337; }

int main()
{
    foo(bar());

    chu(bar());
}
```

# Sémantique de transfert

---

## Problématique

- Copier un objet contenant des ressources est couteux
- Copier depuis un temporaire est doublement couteux (allocation+deallocation)
- Limite l'expressivité de certaine interface
- Pourquoi ne pas recycler le temporaire ?

## Solution

- Utiliser les rvalue-reference pour détecter un temporaire
- Extraire son contenu et le **transférer** dans un objet pérenne
- Stratégie généralisé à tout le langage et à la bibliothèque standard

# Sémantique de transfert

---

## Code original

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

// Copie multiples
std::vector<int> sort(std::vector<int> const& v)
{
    std::vector<int> that{v};

    std::sort( that.begin(), that.end() );

    return that;
}

// Interface détournée
void sort(std::vector<int> const& v, std::vector<int>& that)
{
    that = v;
    std::sort( that.begin(), that.end() );
}
```

# Sémantique de transfert

---

## Version à base de transfert

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

// Cas général
std::vector<int> sort(std::vector<int> const& v)
{
    std::vector<int> that{v};

    std::sort( that.begin(), that.end() );

    return that;
}

// Cas du temporaire
std::vector<int> sort(std::vector<int>&& v)
{
    std::vector<int> that{std::move(v)};

    std::sort( that.begin(), that.end() );

    return that;
}
```

De la boucle à l'algorithme



# De la boucle à l'algorithme

---

## Objectifs

- Augmentez l'expressivité du code
- S'assurer des performance/correction d'un algorithme
- Augmentez la localité du code

## Éléments de réponses

- Boucle for par interval
- Algorithmes standards
- Fonctions anonymes

# De la boucle à l'algorithme

---

## Code original

```
bool match_pattern(MemoryBuffer const& mem)
{
    return mem.size() > 2 && mem[0] == 'E' && mem[1] == 'Z';
}

bool process_buffer(std::vector<MemoryBuffer> const& mems)
{
    std::vector<MemoryBuffer>::const_iterator cit = mems.cbegin();

    for( ; cit != v.cend(); ++cit)
    {
        if (match_pattern(*cit))
            return true;
    }

    return false;
}
```

# De la boucle à l'algorithme

---

## auto à la rescousse !

```
bool match_pattern(MemoryBuffer const& mem)
{
    return mem.size() > 2 && mem[0] == 'E' && mem[1] == 'Z';
}
```

```
bool process_buffer(std::vector<MemoryBuffer> const& mems)
{
    for(auto cit = mems.cbegin(); cit != v.cend(); ++cit)
    {
        if (match_pattern(*cit))
            return true;
    }

    return false;
}
```

# Boucle for par interval

---

## Objectifs

- Simplifie l'écriture de boucle complète
- Gestion automatique du parcours
- Compatible avec auto et qualificateurs

## Syntaxe

```
for( range_element : range_expression ) loop_body
```

# Boucle for par interval

---

## Objectifs

- Simplifie l'écriture de boucle complète
- Gestion automatique du parcours
- Compatible avec auto et qualificateurs

## Code équivalent

```
{  
    auto && __range = range_expression ;  
  
    for( auto __begin = std::begin(__range), __end = std::end(__range);  
        __begin != __end; ++__begin  
    )  
    {  
        range_element = *__begin;  
        loop_body  
    }  
}
```

# Boucle for par interval

---

## Parcours compact

```
bool match_pattern(MemoryBuffer const& mem)
{
    return mem.size() > 2 && mem[0] == 'E' && mem[1] == 'Z';
}

bool process_buffer(std::vector<MemoryBuffer> const& mems)
{
    for(auto const& mem : mems)
    {
        if (match_pattern(mem))
            return true;
    }

    return false;
}
```

# Algorithmes standards

---

## Objectifs

- Fournir une implantation de référence des traitements classiques sur des conteneurs de données
- S'abstraire du type effectif de conteneur
- Maximiser la généricité de l'interface

## Mise en pratique

- Repose sur la notion d'itérateur
- Implantation optimale déduite à la compilation
- large gamme d'algorithme et de variante

## Algorithms standards

---

- `all_of`, `any_of`, `none_of`
- `for_each`
- `count`, `count_if`
- `mismatch`, `equal`
- `find`, `find_if`
- `find_end`, `find_first_of`
- `search`, `search_n`
- `nth_element`
- `max_element`, `min_element`
- `transform`
- `copy`, `copy_if`
- `remove`, `remove_if`
- `replace`, `replace_if`
- `reverse`, `rotate`, `shuffle`
- `sort`, `stable_sort`
- `fill`, `iota`
- `accumulate`
- `inner_product`



# Algorithms standards

---

## Version Algorithm

```
bool match_pattern(MemoryBuffer const& mem)
{
    return mem.size() > 2 && mem[0] == 'E' && mem[1] == 'Z';
}

bool process_buffer(std::vector<MemoryBuffer> const& mems)
{
    return find_if(std::cbegin(mems), std::cend(mems), match_pattern) != std::cend(mems);
}
```

# Fonction anonyme

---

## Objectifs

- Augmenter la localité du code
- Simplifier le design de fonction utilitaire
- Notion de *closure* et de fonction d'ordre supérieur

## Principes

- Bloc de code fonctionnel sans identité
- Syntaxe :

```
auto f = [ capture ] (parametres ... ) -> retour
{
    corps de fonction;
};
```

# Fonction anonyme

---

## Type de retour

- C++11 : Automatique si la fonction n'est qu'un return
- C++11 : Autre cas, à spécifier via `->`
- C++14 : Déduction automatique

# Fonction anonyme

---

## Paramètres

- C++11 : types concrets, pas de variadiques

```
auto somme = [](int a, int b) { return a+b }
```

- C++14 : type générique et variadique

```
auto somme = [](auto a, auto b) { return a+b; }  
auto as_tuple = [](auto... args) { return std::make_tuple(args...); }
```

# Fonction anonyme

---

## Capture de l'environnement

- [a] : capture a par copie
- [&a] : capture a par référence
- [=] : tout par copie
- [&] : tout par référence

```
int x,n;
```

```
auto f = [x](int a, int b) { return a*x+b; }  
auto g = [&x]() -> void { x++; }  
auto h = [&x,n]() -> void { x *=n; }  
auto s = [&]() { x = n; n = 0; return x; }
```

# Fonction anonyme

---

## Renforcement de la localité

```
bool process_buffer(std::vector<MemoryBuffer> const& mems)
{
    return find_if( cbegin(mems), cend(mems)
        , [](MemoryBuffer const& mem)
        {
            return mem.size() > 2 && mem[0] == 'E' && mem[1] == 'Z';
        }
        ) != cend(mems) ;
}
```

# Programmation Générique

# Pourquoi les templates?

---

## Programmation Générique

- Ecrire des structures et fonctions indépendantes des types manipulés

```
template<typename T> T min(T a, T b)
{
    return a < b ? a : b
}
```

## Programmation Générative

- Générer du code paramétré par des types
- *Design Patterns Policy, Strategy ou State*



# Quelques rappels

---

## Qu'est-ce qu'un template

- Entité à part en C++ différente des types et des valeurs
- Une "recette de cuisine" pour construire un type ou une fonction

## Notion d'instanciation

- "Construire" un template : **instancier**
- Une instance de template est un vrai type ou une vrai fonction
- Les recettes peuvent varier en fonctions des "ingrédients"

## Notion de spécialisation

- Spécification des recettes en fonction des paramètres
- Gestion de la variabilité et des branchements
- Par "accident", les template sont **Turing complet**

# Que faire avec des templates

---

## template de classe

- `template<typename T> struct foo ;` est un template
- `foo<int>` est un type

## template de fonction

- `template<typename U> int foo(U u) ;` est un template
- `foo<double>` est une fonction
- `foo(0.)` déduit U automatiquement

# Paramétrages des templates

---

## Paramètre de type

- `template<typename T> struct foo ;`
- `foo<int>`

## Paramètre de constante entière

- `template<int N> struct bar ;`
- `bar<42>`

## Paramètre template

- `template < template<int> T > struct chu ;`
- `chu < bar >`

# Les templates en C++11/14

---

## Nouvelles fonctionnalités

- La fin du cauchemar des »
- Les template alias
- Les *Traits*
- Assertion à la compilation
- Notion d'expression constante
- Les template variadiques
- La notion de tuple
- Maîtrise de la SFINAE

# Les Template Alias

---

## Objectifs

- Améliorer le design de bibliothèque
- Améliorer la pratique de la programmation générique
- Simplifier l'utilisation des templates en général

## Exemples

```
#include <array>
#include <iostream>

template<typename T>
using point3D = std::array<t,3>;

int main()
{
    point3D<int> x{1,2,3};

    std::cout << x.size() << "\n";
}
```

# Les Template Alias

---

## Objectifs

- Améliorer le design de bibliothèque
- Améliorer la pratique de la programmation générique
- Simplifier l'utilisation des templates en général

## Exemples

```
template<typename T> class allocator
{
    //...

    template<typename U> struct rebind { typedef allocator<U> other; };
};

// utilisation
allocator<T>::rebind<U>::other x;
```

# Les Template Alias

---

## Objectifs

- Améliorer le design de bibliothèque
- Améliorer la pratique de la programmation générique
- Simplifier l'utilisation des templates en général

## Exemples

```
template<typename T> class allocator
{
    //...

    template<typename U> using rebind = allocator<U>;
};

// utilisation
allocator<T>::rebind<U> x;
```

# Les Template Alias

---

## Objectifs

- Améliorer le design de bibliothèque
- Améliorer la pratique de la programmation générique
- Simplifier l'utilisation des templates en général

## Exemples

```
// utilisation "à la typedef"

template<typename T> struct as_vector
{
    using type = std::vector<T>;
};
```



# Les Traits

---

## Objectifs

- Introspection limitée sur les propriétés des types
- Génération de nouveau types
- Outils pour la spécialisation avancée
- Notion de **Méta-fonction** : fonction manipulant et retournant un type

# Les Traits

---

## Traits d'introspection

- Classification des types selon leur sémantique
- Vérification d'existence d'une interface donnée
- Récupération d'informations structurelles

## Exemple

```
#include <type_traits>
```

```
int main()
{
    std::cout << std::is_same<float,int>::value << "\n";
    std::cout << std::is_convertible<float,int>::value << "\n";
    std::cout << std::is_base_of<istream,ifstream>::value << "\n";
    std::cout << std::is_class<vector<int>>::value << "\n";
    std::cout << std::is_constructible<string,char*>::value << "\n";
    std::cout << std::is_polymorphic<istream>::value << "\n";
    std::cout << std::is_pointer<void*>::value << "\n";
}
```

# Les Traits

---

## Générateur de type

- Manipulation sûre des qualificateurs
- Création de type vérifiant certaines propriétés

## Exemple

```
#include <type_traits>

int main()
{
    int i;
    std::add_pointer<int>::type pi = &i;

    std::add_rvalue_reference<int>::type rri = std::forward<int>(i);
}
```

# Les Traits - Application

---

```
#include <type_traits>

template<bool B> using bool_ = std::integral_constant<bool,B>;

template<typename T> inline void copy(T* b, T* e, T const* src, bool_<true> const&)
{
    std::memcpy(b,src,(e-b)*sizeof(T));
}

template<typename T> inline void copy(T* b, T* e, T const* src, bool_<false> const&)
{
    while(b != e) *b++ = *src++;
}

template<typename T>
using is_trivially_copiable_t = typename std::is_trivially_copiable<T>::type;

template<typename T> void copy(T* b, T* e, T const* src)
{
    is_trivially_copiable_t<T> select;
    copy(b,e,src,select);
}
```

# Les static\_assert

---

## Objectifs

- assert : vérifie l'état logique d'un programme au runtime
- Comment vérifier l'état logique à la compilation ?
- Emission de message d'erreur customisé
- Interaction avec les *Traits*

## Exemple

```
#include <type_traits>

template<typename T> T factorial(T n)
{
    static_assert( std::is_integral<T>::value, "factorial requires integral parameter");
    return n<2 ? 1 : n*factorial(n-1);
}

int main()
{
    // error: static assertion failed: factorial requires integral parameter
    factorial(3.);
    factorial(5);
}
```

# Les Expressions Constantes

---

## Objectifs

- Simplifier le développement de méta-fonctions numériques
- Syntaxe homogène aux fonctions classiques
- Utilisable dans les contextes requérant une constante

## Syntaxe et Exemples

```
#include <iostream>

int factorial(int n)
{
    return n<2 ? 1 : n*factorial(n-1);
}

int main()
{
    std::cout << factorial(8) << "\n";
}
```

# Les Expressions Constantes

---

## Objectifs

- Simplifier le développement de méta-fonctions numériques
- Syntaxe homogène aux fonctions classiques
- Utilisable dans les contextes requérant une constante

## Syntaxe et Exemples

```
#include <iostream>

constexpr int factorial(int n) { return n<2 ? 1 : n*factorial(n-1); }

template<int N> void display() { std::cout << N << "\n"; }

int main()
{
    display<factorial(5)>();

    int x;
    std::cin >> x;
    std::cout << factorial(x) << "\n";
} 51 of 77
```

# Les Expressions Constantes

---

## Objectifs

- Simplifier le développement de méta-fonctions numériques
- Syntaxe homogène aux fonctions classiques
- Utilisable dans les contextes requérant une constante

## Syntaxe et Exemples

```
#include <iostream>

constexpr int factorial(int n)
{
    return n>=0 ? (n<2 ? 1 : n*factorial(n-1)) : throw std::out_of_range("");
}

int main()
{
    static constexpr int f = factorial(-1);
}
```



# Les templates variadiques

---

## Objectifs

- Pouvoir spécifier sans préprocessing une fonction ou classe template possédant une liste de paramètres variable
- Renforcer le typage de certaines constructions
- Simplifier le design de fonction utilitaires

## Syntaxe et Exemples

```
template<typename... Types> struct tuple {};
```

```
tuple<> no_args;  
tuple<int> arg1;  
tuple<float,int,void> arg3;
```

# Les templates variadiques

---

## Objectifs

- Pouvoir spécifier sans préprocessing une fonction ou classe template possédant une liste de paramètres variable
- Renforcer le typage de certaines constructions
- Simplifier le design de fonction utilitaires

## Syntaxe et Exemples

```
template<class ... Types> void f(Types ... args);
```

```
f(); // 0 argument  
f(1); // 1 argument  
f(1,"lol",5.); // 3 argument
```

# Les templates variadiques

---

## Objectifs

- Pouvoir spécifier sans preprocessing une fonction ou classe template possédant une liste de paramètres variable
- Renforcer le typage de certaines constructions
- Simplifier le design de fonction utilitaires

## Syntaxe et Exemples

```
#include <iostream>
```

```
template<class ... Types> void f(Types ... args) { std::cout << sizeof...(args) << " arguments\n"; }
```

```
int main()
{
    f();
    f(1);
    f(1, "lol", 5.);
}
```

# Les templates variadiques - En pratique

---

## Définition d'un printf moderne et typesafe :

```
void printf(const char *s)
{
    while (*s)
    {
        if (*s == '%')
        {
            if (*(s + 1) == '%')
            {
                ++s;
            }
            else
            {
                throw std::runtime_error("invalid format string: missing arguments");
            }
        }

        std::cout << *s++;
    }
}
```

# Les templates variadiques - En pratique

---

## Définition d'un printf moderne et typesafe :

```
template<typename T, typename... Args>
void printf(const char *s, T value, Args... args)
{
    while (*s)
    {
        if (*s == '%')
        {
            std::cout << value;
            s += 2;

            printf(s, args...);
            return;
        }
        std::cout << *s++;
    }
}
```

# std::tuple

---

## Objectifs

- Généralisation de pair
- S'appuie sur les templates variadiques pour agréger N types
- Fournit les fonctions nécessaire à l'accès de ces éléments

## Exemples

```
#include <tuple>

int main()
{
    std::tuple<char,int,double> v{'Z',3,4.2};

    auto c = std::get<0>(v);

    std::get<1>(v) *= 2;
}
```

# std::tuple

---

## Objectifs

- Généralisation de pair
- S'appuie sur les templates variadiques pour agréger N types
- Fournit les fonctions nécessaire à l'accès de ces éléments

## Exemples

```
#include <tuple>

int main()
{
    std::tuple<char,int,double> v{'Z',3,4.2};

    std::tuple_element<0,decltype(v)>::type c = std::get<0>(v);

    std::tuple_element<0,decltype(v)>::type& i std::get<1>(v);
    i *= 2;
}
```

# std::tuple

---

## Objectifs

- Généralisation de pair
- S'appuie sur les templates variadiques pour agréger N types
- Fournit les fonctions nécessaire à l'accès de ces éléments

## Exemples

```
#include <tuple>

int main()
{
    std::tuple<char,int,double> v{'Z',3,4.2};

    std::cout << std::tuple_size<decltype(v)>::value << "\n";
}
```



# Maitrise de la SFINAE

---

## Qu'est-ce que la SFINAE ?

- Lors de la résolution de la surcharge de fonction, il se peut qu'une surcharge instancie une fonction template
- Cette instanciation peut échouer
- Au lieu d'émettre une erreur, la surcharge est ignorée
- SFINAE = Substitution Failure Is Not An Error

```
#include <vector>

template<typename T>
typename T::size_type size(T const& t) { return t.size(); }

int main()
{
    size(std::vector<int>(8)); // OK

    // no matching function for call to 'size(int)'
    size(8);
}
```

# Maitrise de la SFINAE

---

## Qu'est-ce que la SFINAE?

- Lors de la résolution de la surcharge de fonction, il se peut qu'une surcharge instancie une fonction template
- Cette instanciation peut échouer
- Au lieu d'émettre une erreur, la surcharge est ignorée
- SFINAE = Substitution Failure Is Not An Error

## Intérêts

- Contrôler ces erreurs permet d'éliminer des fonctions sur des critères arbitraires
- Customisation de la surcharge de fonction ou de classe template
- Interaction avec les *Traits*
- `std::enable_if`

# Maitrise de la SFINAE

---

## Mise en œuvre

```
#include <tuple>
#include <iostream>

template <size_t n, typename... T>
typename std::enable_if<(n >= sizeof...(T))>::type
print_tuple(std::ostream&, const std::tuple<T...>&) {}

template <size_t n, typename... T>
typename std::enable_if<(n < sizeof...(T))>::type
print_tuple(std::ostream& os, const std::tuple<T...>& tup)
{
    if (n != 0) os << ", ";
    os << std::get<n>(tup);
    print_tuple<n+1>(os, tup);
}

template <typename... T>
std::ostream& operator<<(std::ostream& os, const std::tuple<T...>& tup)
{
    os << "[";
    print_tuple<0>(os, tup);
    return os << "]";
}
```

# Exercices I

# Exercice I

---

- Ecrire une structure `weighted_3dpoint` : point 3D pondéré
- Lire une série de tels points depuis un fichier texte
- Afficher la liste des points ainsi chargés
- Calculer le barycentre de ces points
- Rechercher dans le point le plus proche du barycentre

## Exercice 2

---

Ecrire une classe permettant de manipuler des chaînes de caractères constantes comme des expressions constantes.

- Indice 1 : `constexpr` s'applique au fonction membre
- Indice 2 : Gestion des erreurs = exceptions
- Pensez au test de validation

## Exercice 3

---

Ecrire en C++11 et C++14 une fonction `compose` qui prend deux fonctions quelconques en paramètres et renvoie une fonction effectuant la composition des deux fonctions passées en paramètres.

```
auto fg = compose(f, g);
```

# Nouveau Paradigme Objet et RAII



# Evolutions de la POO en C++

---

## Changements syntaxiques

- Construction par rvalue
- Construction par liste d'initialisation
- Assouplissement de la définition de constructeur
- Contrôle fin sur les fonctions membres

## Changements philosophiques

- Notion centrale de **Type Régulier**
- Disparition de la gestion manuelle des ressources
- Simplification de l'interaction avec la mémoire

# Construction/Affectation par rvalue

---

## Objectifs

- Optimiser la création d'objet en provenance d'un temporaire
- Optimiser les retour par valeurs
- Application à toute la STL

# Construction/Affectation par rvalue

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

struct A
{
    std::vector<int> data_;

    A() : data_(100) { std::cout << "A()\n"; }

    A(A const& s) : data_(s.data_) { std::cout << "A(A const&)\n"; }
    A(A&& s) : data_(std::move(s.data_)) { std::cout << "A(A&&)\n"; }

    ~A() {}

    A& operator=(A const& s)
    {
        std::cout << "A:=(A const&)\n";
        data_ = s.data_;
        return *this;
    }

    A& operator=(A&& s)
    {
        std::cout << "A:=(A&&)\n";
        data_.clear();
        data_.swap(s.data_);
        return *this;
    }
}

} 63 of 77
```

# Construction/Affectation par rvalue

---

```
A make() { return A{}; }
```

```
int main()
```

```
{  
    A a;  
    A b(a);  
    A c = a;  
    A d = make();  
    A e = std::move(a);
```

```
    a = b;  
    b = make();  
    c = std::move(d);  
}
```

# Construction/Affectation par rvalue

---

```
A make() { return A{}; }

int main()
{
    A a; // A()
    A b(a); // A(A const&)
    A c = a; // A(A const&)
    A d = make(); // A()
    A e = std::move(a); // A(A&&)

    a = b; // A:=(A const&)
    b = make(); // A(); A:=(A&&)
    c = std::move(d); // A:=(A&&)
}
```

# std::initializer\_list<T>

---

## Objectifs

- Simplifier la construction d'objet type conteneur
- Fournir un cadre *type safe* pour ce type d'opération

```
#include <initializer_list>

template<typename T> struct S
{
    std::vector<T> v;
    S(std::initializer_list<T> l) : v(l) { std::cout << l.size() << "\n"; }
    void append(std::initializer_list<T> l) { v.insert(v.end(), l.begin(), l.end()); }
};

int main()
{
    S<int> s = {1, 2, 3, 4, 5};
    s.append({6, 7, 8});

    auto a1 = {10, 11, 12};
    for (int x : a1) std::cout << x << '\n';
}
```

# Construction par délégation ou Héritage

---

## Objectifs

- Factoriser le code entre constructeur
- Réutiliser les constructeurs des classes de bases

```
class A
{
    public:
        A() : A(0) {}

        A(int i) : A(i, 0) {}

        A(int i, int j) : num1(i), num2(j), average(i+j/2)
        {}

    private:
        int num1, num2, average;
};
```

# Construction par délégation ou Héritage

---

## Objectifs

- Factoriser le code entre constructeur
- Réutiliser les constructeurs des classes de bases

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

template<typename T>
struct block : private std::vector<T>
{
    using parent = std::vector<T>;

    using parent::vector;
    using parent::size;
    using parent::operator[];
};

int main()
{
    block<float> f{1,2,3,4};
    std::cout << f.size() << "\n";
}
```



# Fonction membre par défaut et effaçable

---

## Règles d'apparitions des membres spéciaux

- Si un constructeur non-trivial est déclaré, le constructeur par défaut n'est pas généré
- Si un destructeur virtuel est déclaré, le destructeur par défaut n'est pas déclaré
- Si un constructeur/assignment par rvalue est déclaré alors :
  - Pas de constructeur par copie par défaut
  - Pas d'assignment par défaut
- Si un constructeur par copie, par rvalue, un destructeur ou un assignment est défini
  - Pas de constructeur par rvalue par défaut
  - Pas d'assignment par rvalue par défaut

## =delete,=default

- =default force la génération du membre spécial
- =delete force l'élimination du membre

# Fonction membre par défaut et effaçable

---

```
struct noncopyable
{
    noncopyable() =default;
    noncopyable(const noncopyable&) =delete;
    noncopyable& operator=(const noncopyable&) =delete;
};
```

```
struct A : noncopyable {};
```

```
int main()
{
    A a,b;
    A x{a};

    b = a;
}
```

# Fonction membre par défaut et effaçable

---

```
#include <cstddef>

struct static_only
{
    void* operator new(std::size_t) =delete;
};

struct widget : static_only
{};

int main()
{
    widget w;
    widget* pw = new widget;
}
```

# Fonction membre par défaut et effaçable

---

```
void call_with_true_double_only(float) =delete;  
void call_with_true_double_only(double param) {}
```

```
int main()  
{  
    call_with_true_double_only(3.);  
    call_with_true_double_only(3.f);  
}
```

# Qualification de \*this

---

## Problématique

- Comment spécifier la rvalue-ness du retour d'une fonction membre en fonction de la rvalue-ness de l'objet ?
- Extension de la notation A : :f() const

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

struct foo
{
    std::vector<int> data_;
    foo() : data(1000) {}

    std::vector<int> data() { return data_; }
    std::vector<int> data() const { return data_; }
};
```

# Qualification de \*this

---

## Problématique

- Comment spécifier la rvalue-ness du retour d'une fonction membre en fonction de la rvalue-ness de l'objet ?
- Extension de la notation A : :f() const

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

struct foo
{
    std::vector<int> data_;
    foo() : data(1000) {}

    std::vector<int> data() && { return std::move(data_); }
    std::vector<int> data() const& { return data_; }
};
```

# Contrôle des méthodes virtuelles

---

## Problématique

- Erreurs de prototype entre membres d'une classe et de ses filles
- Erreur silencieuse visible au runtime
- Comment s'assurer du bon sous-classage ?

```
class Base
{
    virtual void A(float=0.0);
    virtual void B() const;
};

class Derived: public Base
{
    virtual void A(int=0);
    virtual void B();
};
```

# Contrôle des méthodes virtuelles

---

## Problématique

- Erreurs de prototype entre membres d'une classe et de ses filles
- Erreur silencieuse visible au runtime
- Comment s'assurer du bon sous-classage ?

```
class Base
{
    virtual void A(float=0.0);
    virtual void B() const;
    virtual void C();
    void D();
};

class Derived: public Base
{
    virtual void A(int=0) override;
    virtual void B() override;
    virtual void C() override;
    void D() override;
};
```



# Contrôle des méthodes virtuelles

---

## Problématique

- Erreurs de prototype entre membres d'une classe et de ses filles
- Erreur silencieuse visible au runtime
- Comment s'assurer du bon sous-classage ?

```
class Base
{
    virtual void A() final;
};
```

```
class Derived: public Base
{
    virtual void A();
};
```

# Contrôle des méthodes virtuelles

---

## Problématique

- Erreurs de prototype entre membres d'une classe et de ses filles
- Erreur silencieuse visible au runtime
- Comment s'assurer du bon sous-classage ?

```
class Base final
{
};

class Derived: public Base
{
    // ERREUR
};
```

# Sémantique de Valeur

---

## Définition

Une classe possède une sémantique de valeur ssi deux instances de cette classe situées à des adresses différentes, mais au contenu identique, sont considérées égales.

## Structure classique

- Peut redéfinir des opérateurs (+, -, \*, ...)
- Possède un opérateur d'affectation
- Est comparable via < et ==
- Ne peut être utilisé comme classe de base

# Sémantique d'Entité

---

## Définition

Une classe a une sémantique d'entité si toutes les instances de cette classe sont nécessairement deux à deux distinctes, même si tous les champs de ces instances sont égaux. Elle modélise un concept d'identité : chaque objet représente un individu unique.

## Structure classique

- Ne redéfinit pas d'opérateur
- Ne possède pas d'opérateur d'affectation
- N'est pas comparable via  $<$  et  $==$
- Les copies sont explicites via un fonction adéquat (clone)

# Que faire ?

---

## Utilisez préférentiellement la sémantique de valeur

- Code clair et concis
- Bonne performance car pas d'allocation dynamique supplémentaire
- NRVO et élision de copie sont de la partie

## Ne délaissez pas les entités

- Extrêmement utile pour le data-driven code
- Bonne base pour des bibliothèques d'infrastructure
- Se marie élégamment avec les pointeurs à sémantique riche

# Principe de RAI

---

## Objectifs

- Assurer la sûreté de la gestion des ressources
- Minimiser la gestion manuelle de la mémoire
- Simplifier la gestion des exceptions

Resource Acquisition Is Initialisation

# Principe de RAII

---

## Mise en œuvre

- Constructeurs = prise de ressource
- Destructeur = libération de ressource
- Gestion de la ressource au niveau du bloc

# Principe de RAII

---

## Exemple

```
#include <string>
#include <mutex>
#include <iostream>
#include <fstream>
#include <stdexcept>

void write_to_file (const std::string & message)
{
    static std::mutex mutex;

    std::lock_guard<std::mutex> lock(mutex);

    std::ofstream file("example.txt");
    if (!file.is_open())
        throw std::runtime_error("unable to open file");

    file << message << std::endl;
}
```



# shared\_ptr

---

## Principes

- Pointeur à compteur de référence
- Libère la mémoire lorsque aucun référence ne pointe sur lui
- Cycles gérés par weak\_ptr

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

int main()
{
    std::shared_ptr<int> p1 = std::make_shared<int>(5);
    std::shared_ptr<int> p2 = p1;

    std::cout << *p1 << "\n";
    *p2 = 42;
    std::cout << *p1 << "\n";
    p1.reset();
    std::cout << *p2 << "\n";
    p2.reset();
}
```

# shared\_ptr

---

## Principes

- Pointeur à compteur de référence
- Libère la mémoire lorsque aucun référence ne pointe sur lui
- Cycles gérés par weak\_ptr

```
int main()
{
    std::shared_ptr<int> p1 = std::make_shared<int>(5);
    std::weak_ptr<int> wp1 = p1;

    {
        std::shared_ptr<int> p2 = wp1.lock();
        if(p2) std::cout << *p2 << "\n";
    }

    p1.reset();
    std::shared_ptr<int> p3 = wp1.lock();

    if(p3) std::cout << "nope :(\n";
}
```

# unique\_ptr

---

## Principes

- Pointeur à propriétaire unique
- Ne peut être copier, uniquement transférer
- Transfert = Transfert de propriété

## Mise en œuvre

```
#include <memory>

int main()
{
    std::unique_ptr<int> p1 = std::make_unique<int>(5);
    std::unique_ptr<int> p2 = p1;
    std::unique_ptr<int> p3 = std::move(p1);

    p3.reset();
    p1.reset();
}
```