ArtWorks

Advanced C++

Nouveautés du langage

Microsoft Certified Professional Solution Developer Trainer

CompTIA. Certified Technical Trainer+

Michel André
michel@artworks.fr

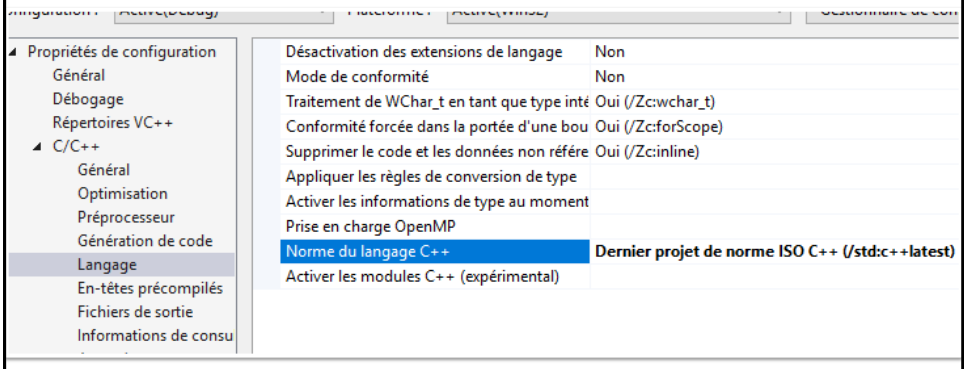
Objectifs de C++11

- Bjarne Stroustrup:
  - "C++11 feels like a new language: The pieces just fit together better than they used to and I find a higher-level style of programming more natural than before and as efficient as ever. If you timidly approach C++ as just a better C or as an object-oriented language, you are going to miss the point. The abstractions are simply more flexible and affordable than before"
  - "My ideal is to use programming language facilities to help programmers think differently about system design and implementation"

STANDARD C++

2

Utiliser les nouveautés sous Visual Studio



3

Retraits du langage

C++ 17

- le mot-clé register est retiré
  - mais demeure réservé
- l'opérateur déprécié ++(bool) est retiré
- la spécification d'exception dépréciée est retirée

```
void f () throw (int){
  bool b;
  ++b;
  if (…)
     throw 123;
}
```

échec

échec

4

Les nouveaux types numériques

```
#include <cstdint>
int8_t, int16_t,… , uint64_t, …
```

types normalisés et variantes

séparateurs

```
long long ll = 1'000'000'000'000;
```

entier d'au moins 64 bits

```
long double ld;
```

réel d'au moins 64 bits

Literal	Type
2	int
2u	unsigned
2l	long
2ul	unsigned long
2.0	double
2.0f	float
2.0l	long double

```
#include <cstddef>

std::byte b = 12;
```

C++ 17

5

Énumérations fortement typées

```
enum class Color {RED, GREEN, BLUE};
```

```
Color color {Color::GREEN};
if (Color::RED == color){// …}

int n {Color::RED};
```

échec

```
int n = static_cast<int> (color);
```

OK

```
enum class Colors : int8_t {
  RED = 1, GREEN = 2 BLUE = 3
};
```


6

nullptr_t et le littéral nullptr

- Tous les types pointeurs bruts ont une conversion naturelle vers le type nullptr_t et vice-versa

```
void func(long){}
void func(bool n){}
void func(int*){}

func (NULL);
func (nullptr);
```

ambiguïté

ambiguïté levée

```
cout << typeid(nullptr).name() << endl;
```

std::nullptr_t

7

Initialisation uniforme

déclarations

affectations

```
int i {};
int* p {};

i = {3.6};
i = 3.6;

long g{1'234'567'890'123};
```

initialisé à 0

initialisé à nullptr

échec, narrowing conversion

avertissement

échec, narrowing conversion

```
struct A {
  A(int n, int p);
};

A tab[] = ((1, 2),(2, 3));
A tab[] {{1, 2}, {2, 3}};

int* tab{new int[]{12, 78}};
```

échec

OK

OK

8

Initialisation d'objets

```
struct A {
  int n;
  float f;
  bool b;
  void* ptr;
  void print() const {
    cout << n << " " << f << " "
      << boolalpha << b << " " << ptr << endl;
  }
};
```

membres sont initialisés par défaut

```
A{}.print();

A a;
a.print();
```

pas d'initialisation

```
0 0 false 00000000
-858993460 -1.07374e+08 true CCCCCCCC
```


9

Ambigüité "Most Vexing Parse"

```
Widget w();

w.draw();
```

```
Widget w{};

w.draw();
```


10

Alias (synonyme) et using

- using remplace avantageusement typedef pour créer des grandeurs
  - création d'un nouveau nom pour un type existant (alias)
    - ce n'est pas un nouveau type

avant

```
typedef double Salaire;
typedef int (func *)(int, int);
```

après

```
using Salaire = double;
using func = int (*)(int,int);

int add(int a, int b){ return a + b; }
```

```
Salaire s{ 12.6 };
cout << s << endl;

func f1{ add };

cout << f1(2, 3) << endl;
```


Initialisation de tableaux / collections

- une collection s'initialise comme un tableau :

avant

```
vector<int> nombres;
nombres.push_back(1);
nombres.push_back(2);
```

```
vector<int> nombres {1, 2};
map<int, string> nombres {{1,"un"},{2,"deux"}};
```

- tableaux et collections se parcourent de façon unifiée:

```
int tab[] {89, 123, -12, 90};
sort (begin(tab), end(tab));

vector<int> v {89, -45, 999};
sort (begin(v), end(v));
```

-  std::array<T, SIZE> remplace avantageusement un tableau

La boucle "Range based" for

- syntaxe « à la Java »
  - fonctionne avec les tableaux et tous les conteneurs

avant

```
int tab[] {78, 123, -56};

const size_t taille (sizeof tab / sizeof tab[0]);
for (size_t index (0); index < taille; ++index)
  cout << tab[index] << "\n";

for (int i : tab)
  cout << i << "\n";
```

read only

```
string str{ "hello" };
for (char& c : str)
  c = 'X';
cout << str << endl;
```

read / write

XXXXX

Listes d'initialisation

```
#include <initializer_list>

void afficherListe(const initializer_list<int>& liste){

  initializer_list<int>::iterator i {cbegin(liste)};
  for(; i != liste.end(); ++i)
    cout << *i << "\n";

  for (int n : liste)
    cout << n << "\n";
}

afficherListe ({1, 2, 3, 4, 5});
```

nombre de paramètres variable

- 3 méthodes constantes:
  - const T *begin() const
  - const T *end() const
  - size_t size() const

accesseurs

- constructeur
  - initializer_list(const T *_First_arg, const T *_Last_arg)

Surcharge et liste d'initialisation

- Une méthode avec une liste d'initialisation prime si elle est invoquée avec l'initialisation uniforme

```
struct A {
  A(int a){ cout << "int\n"; }
  A(const initializer_list<int>& l){cout << "liste\n";}
};
```

```
A a{1};
A a2(1);
```

liste
int

```
cout << vector<int>(10, 5).size() << endl;
cout << vector<int>{10, 5}.size() << endl;

cout << string ( 60, 61 ) << endl;
cout << string { 60, 61 } << endl;
```

?

15

Littéraux binaires

```
const int a{ 42 };     // base décimale
const int b{ 052 };    // base octale
const int c{ 0x2A };   // base hexa
const int d{ 0b00101010 }; // base binaire
```

C++11

C++14

- les spécificateurs oct, dec, hex permettent l'affichage selon une base sélectionnée
  - showbase précise la base courante

```
cout << showbase << hex << 17 << endl;
```

0x11

- rien n'est prévu pour l'affichage de nombres en binaire
  - il faut recourir au bitset<NB_BITS>

```
cout << "0b" << bitset<6>{123} << endl;
```

0b001001

17

Les littéraux personnalisés

```
using Distance = long double;

Distance operator"" _km(long double val) {
  return Distance{ val };
}
Distance operator"" _mi(long double val) {
  return Distance{ val*1.6 };
}
```

clean code

```
cout << Distance{ 36.0_mi + 42.0_km } << "km\n";
```

- pour une entrée de type entier, le type doit être **unsigned long long**
- préfixer les littéraux par le caractère _
  - éviter d'éventuels conflits avec les littéraux de la librairie standard
- Tip: créer un nouveau type avec constructeur privé et opérateurs friend pour forcer l'usage des littéraux

18

Init-statement

C++ 20

```
if (init; condition) // …
switch (init; condition) // …
```

C++ 17

```
string myString{ " Hello John" };
if (const auto it{myString.find("Hello")}; it != string::npos)
  cout << "found on position " << it << "\n";
```

C++ 17

```
vector<int> v{/**/ };
for (size_t nbTimes{ 3 }; int n : v) {
  cout << nbTimes * n << endl;
}
```

C++ 20

19

Inférence de types avec auto

```
map<int,string> myMap;

map<int,string>::const_iterator itor {myMap.cbegin()};
Singleton& s {Singleton::instance()};
```

avant

```
auto itor {myMap.cbegin()};
auto& s {Singleton::instance()};
```

après

inférence de signatures

```
int add (int a, int b){return a+b;}
int sub (int a, int b){return a-b;}
```

```
auto f {add};
cout << typeid(f).name() << "\n";
cout << f(23, 67) << "\n";
f = sub;
cout << f(23, 67) << "\n";

auto a {f(1, 4)};
cout << typeid(a).name() << "\n";
```

```
int (__cdecl*)(int,int)
90
-44
int
```

Quelques pièges

```
auto a {123};
auto a {0, 23};
auto a = {0, 23};
auto v1 = vector < int > { 1, 2, 3 };
```

entier

échec

initializer_list<int>

```
auto a{ "hello" };
cout << typeid (a).name() << endl;
cout << typeid ("hello").name() << endl;
```

char const *
char const [6]

```
template <typename T>
void f(T t){}

f ({23,67});

template <typename T>
void f(initializer_list<T> t){}
```

échec

ok

22

auto et retour

OK

```
auto foo (){return 3;}
```

```
auto foo (){return {3, 123, 789};}
```

échec

```
template <class ONE, typename TWO>
auto add(ONE one, TWO two){
  return one + two;
}
```

OK

C++ 14

```
auto Correct(int i) {
  if (i == 1)
    return i;
  return Correct(i-1)+i;
}
```

ok

```
auto Wrong(int i) {
  if (i != 1)
    return Wrong(i-1)+i;
  return i;
}
```

échec

23

Structured Bindings

C++ 17

- décomposition d'un type structuré tel qu'une struct, un tableau, une paire ou un tuple

```
int tab[]{56, 67, -123};
auto [ a, b, _ ] = tab;
```

```
set<string> myset;
if (auto [iter, success] {myset.insert("Hello")}; success)
  cout << "Insert is successful." <<
    The value is " << std::quoted(*iter) << '\n';
```

```
Insert is successful.
The value is "Hello"
```


Structured Bindings et références

C++ 17

```
map<int, double> myMap;
// …
for (const auto& [key, value] : myMap) {
  // …
}
```

pair

```
int a[2] { 1,2 };
auto& [xr, yr] = a;
xr = yr = 0;
cout << a[0] << " " << a[1] << "\n";
```

```
0 0
```

Les spécificateurs default et delete

- delete inhibe la génération par défaut
- default force la génération par défaut

```
struct A {
  A(const A&) = delete;
  A& operator= (const A&) = delete;
  A(int n);

  A() = default;
};
```

constructeur paramétré …

…rétablissement du constructeur par défaut

Application: classe non copiable

```
class noncopyable {
protected:
  constexpr noncopyable () = default;

  noncopyable(const noncopyable &) = delete;
  noncopyable& operator=(const noncopyable &) = delete;
};
```

classe abstraite …

… et non copiable

```
class MyClass : private noncopyable {
// …
};
```

MyClass devient non copiable

Cas particulier de l'opérateur new

- l'allocation dynamique peut être interdite

```
class A {
public:
  // ...
  void* operator new (size_t size) = delete;
};
```

```
A a;

A* pA{ new A };
```

ok

échec

Les spécificateurs override et final

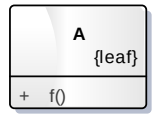
```
struct A {
  virtual void f() const {}
};

struct B : A {
  void f() override {}
};
```

échec

```
struct A final {
  void f(){}
};
struct B : A {};
```

échec



utile lorsque l'on veut éviter un destructeur virtuel

```
struct A  {virtual void f(){}};
struct B : A {void f() final {}};
struct C : B {void f(){}};
```

échec

Le spécificateur noexcept

- noexcept spécifie que la fonction ne lèvera pas d'exception, ou alors que le programme n'est pas récupérable
  - si une exception survient, le process termine via un appel à std::abort()
    - objets statiques non détruits, hooks de sortie non exécutés
- Optimisation possible par le compilateur
  - noexcept peut appeler std::unexpected et la pile n'est pas déroulée
- A moins que les données membres ne soient pas noexcept, les fonctions suivantes sont implicitement de type noexcept:
  - Default constructor and destructor
  - Move and copy constructor
  - Move and copy assignment operator

```
void func1() noexcept;
void func2() noexcept(true);

void func4() noexcept(false);
```

does not throw

may throw

L'opérateur noexcept

```
void f() {/****/}
void g() {/****/}


void func() noexcept (noexcept(f()) && noexcept (g()))
{
  // …
  f();
  // …
  g();
}
```

prédicat
exécuté par le compilateur

Constructeur délégué

- délègue à un autre son initialisation
  - utile pour maitriser les options de construction
    - idéalement avec un constructeur privé

```
class A {
  int option1;
  double option2;
  A(int option1, double option2) :
    option1{option1},
    option2{option2}
  {
  }
 public:
  A() : A{3, 10.5}{}
  A(int option1) : A{option1, 10.5}{}
  A(double option2) : A{3, option2}{}
};
```

privé

initialisation uniforme

par exemple

32

Constructeur hérité

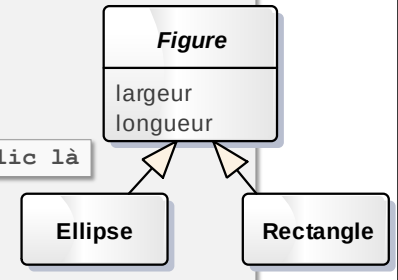
```
class Figure {
  unsigned largeur, longueur;
public:
  Figure (unsigned largeur, unsigned longueur) :
    largeur{largeur}, longueur{longueur}{}
  virtual ~Figure() = default;
  virtual void draw() const = 0;
};

class Rectangle : public Figure {
public:
  using Figure::Figure;
  void draw() const override;
};

class Ellipse : public Figure{
public:
  using Figure::Figure;
  void draw() const override;
};
```

public ici …

… car public là




```
Rectangle{10, 50}.draw();
Ellipse{20, 90}.draw();
```


33

Initialisation de membres

```
using Valeur = unsigned;

class Dé {

  Valeur valeur {getValeurAléatoire()};
  int x {1};
  int x2 (0);

public:

  Valeur getValeurAléatoire () const {
    // …
  }
};
```

plus besoin de constructeur

ne compile pas

Les données inline

C++ 17

- la déclaration et l'implémentation d'une variable globale sont conjointes dans un header

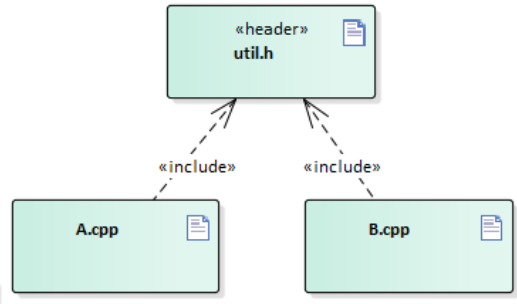
util.h

```
inline int n{};
```

A.cpp …

```
cout << n++ << endl;
```

… ou B.cpp



Donnée membre statique

C++ 17

```
class Voiture {
  static unsigned vitesse_max;

  // …
};
```

voiture.h

```
unsigned Voiture::vitesse_max {130};
```

voiture.cpp

C++ 17

```
class Voiture {
  inline static unsigned vitesse_max {130};

  // …
};
```


L'opérateur sizeof (Class::member)

```
struct A {
  string message;
  int valeur;
  bool ok;
};

cout << sizeof A::valeur << endl;


A a;
cout << sizeof a.valeur << endl;
```

classe

instance

Alignement mémoire

- alignas attribue une spécification d'alignement à un type / instance
  - puissance de 2
- alignof permet d'obtenir l'alignement d'un type

```
struct alignas (16) A {
  int f2;
  float f1;
  char c;
};

alignas (long long) char c {'Z'};
alignas (32) int n {123};

cout << alignof (long long) << endl;
cout << alignof (A) << endl;
```


38

Expressions constantes

- permet l'évaluation d'expressions lors de la compilation
  - mais également lors du runtime
- utile pour la méta programmation
  - programmation du compilateur

```
constexpr inline int square(int n){return n * n;}

array <int, square(2)> tab;
```

évalué à la compilation

```
constexpr unsigned constexpr_pow(int base, unsigned exp){
  return (exp == 0) ?
    1 : base * constexpr_pow(base, exp - 1);
}
```

```
cout << constexpr_pow(5, 3) << endl;
```


39

Levée des restrictions sur constexpr

C++ 14

- C++11 limite les fonctions constexpr à une seule expression
- Avec C++14, il est possible :
  - de déclarer des variables,
  - d'utiliser des instructions de contrôle (if, switch, for, while)
  - de créer et de modifier des objets
- Le compilateur C++ dispose ainsi d'un véritable interprète C++

Un exemple complet de classe

```
class Rectangle {
  unsigned width, height;
public:
  constexpr Rectangle(unsigned width, unsigned height) :
    width{ width },
    height{ height }
  {}
  constexpr unsigned getArea() const {
    return width* height;
  }
  constexpr Rectangle& reset(unsigned width,
    unsigned height)
  {
    this->width = width; this->height = height;
    return *this;
  }
};
```

instanciation

```
int main() {
  std::array<int, Rectangle{ 10, 20 }.
    reset (23, 67).getArea() > myArray;
  cout << myArray.size() << endl;
}
```

La classe std::ratio <num, denom>

- Compile time rational arithmetic

```
template <intmax_t Num, intmax_t Den = 1> class ratio;
```

- 2 membres statiques
  - static constexpr intmax_t num
  - static constexpr intmax_t den

évaluation à la compilation intégrant le calcul du PGCD

```
using sum = ratio_add<ratio<2, 3>, ratio<1, 6>>;

cout << "2/3 + 1/6 = " << sum::num << '/'
  << sum::den << '\n';
```

micro	std::ratio<1, 1000000>
milli	std::ratio<1, 1000>
centi	std::ratio<1, 100>

de nombreuses fractions sont disponibles

Les attributs

- Syntaxe unifiée pour les mécanismes d'extensions spécifiques aux implémentations.
  - __declspec (Microsoft), __attribute__ (IBM)
  - s'appliquent à un type, une fonction, enum, …

exemple

```
[[abc]] void foo() {}
```

- C++ propose des attributs standards:
  - noreturn, deprecated

```
struct A {
  [[deprecated("deprecated because of …")]] void f() {}
};
'A::f': deprecated because of …
A{}.f()

[[noreturn]] void terminate() {
  // …
  throw std::exception{"oupps"};
}
```

C++ 14

Nouveaux attributs

C++ 17

- [[fallthrough]]

```
switch (a) {
case 0:
  cout << "zero\n";
case 1:
  cout << "un\n";
  [[fallthrough]];
default:
  cout << "default\n";
}
```

No warning

- [[maybe_unused]]

```
[[maybe_unused]] int n;

int n2;
```

No warning

Warning

44

Raw string literal

- il n'est pas toujours souhaitable de gérer des caractères d'échappement

```
"\\\\p"
```

produit \\p

```
R"(\\p)";
```

produit la même chose
\ n'est pas interprété

```
cout << R"(
one
  two
    three)" << "\n";
```

WYSIWYG

un retour
à la ligne

```
one
  two
    three
```


45