

# Le C++ 11 au C++ 17



Facile



Normal



Difficile



Professionnel



Expert

[https://wiki.waze.com/wiki/Your\\_Rank\\_and\\_Points](https://wiki.waze.com/wiki/Your_Rank_and_Points)

1. Introduction
2. Évolutions
3. Extensions de la bibliothèque standard
4. Les éléments dépréciés/supprimés C++17
5. Conclusion
6. Bibliographie



- Apport de C++11 à C++17
- Environnement de travail
- Options de compilation



- 1973 : Création du langage C par B. Kernighan et D. Ritchie : C K&R
- 1989 : Normalisation par l'ANSI : ANSI C ou C89
- 1990 : Normalisation par l'ISO : C90 (= C89)
- 1998 : Mise à jour de la norme : C++98
- 1999 : Mise à jour de la norme : C99
- 2003 : Mise à jour de la norme C++98 et C++03
- 2011 : Mise à jour de la norme C++11 : *approuvée le 12 août 2011 et publiée sous le nom de ISO/CEI 14882:2011 en septembre 2011*
- 2014 : Mise à jour mineure de la norme : C++14 - août 2014<sup>1</sup> *ISO/CEI 14882:2014*
- 2017 : Mise à jour de la norme C++17 : *ISO/CEI 14882:2017*



C++11 introduit plusieurs nouveautés au langage initial, ainsi que de nouvelles fonctionnalités à la bibliothèque standard du C++

C ++ 17 introduit un certain nombre de nouvelles fonctionnalités de langages.

- nouveau type *std::byte*,
- ajout du support des caractères littéraux au format Unicode 8 bits,
- les variables inline,
- les expressions de réductions (fold expressions),
- les liaisons structurées (Structured Bindings),
- une interface pour les systèmes de fichiers,
- la parallélisation des algorithmes de la STL,
- les lambdas *constexpr*, etc.
- Il supprime aussi l'opérateur ++ pour les booléens,
- les trigraphs,
- le mot clé *register*, *std::auto\_ptr* et *std::random\_shuffle*.
- Etc...

# 1 – Introduction -> Apport de C++11 au C++17



Le comité a appliqué les directives suivantes :

- **Garder la stabilité** et la compatibilité avec le C++98 et, si possible, avec le C.
- **Préférer l'introduction de nouvelles fonctionnalités par la bibliothèque standard**, plutôt que par le langage lui-même.
- **Améliorer le C++ pour faciliter** la mise en place de systèmes et de bibliothèques, plutôt qu'introduire de nouvelles fonctionnalités seulement utiles pour des applications spécifiques.
- **Augmenter la protection des types** en fournissant des alternatives plus sécurisées que les actuelles, plutôt non sécurisées.
- **Augmenter les performances** et les **capacités à travailler** directement avec le matériel.
- **Implémenter le principe du « zero-overhead »** (on ne paye le coût d'une fonctionnalité que si l'on s'en sert).
- **Rendre le C++ facile à apprendre et à enseigner** sans enlever les fonctionnalités requises par les programmeurs experts.
- **Proposer des solutions propres aux problèmes actuels.**
- **Préférer les changements qui peuvent faire évoluer les techniques de programmation.**

[https://wikimonde.com/article/C%2B%2B11#Changements\\_pr.C3.A9vus\\_pour\\_la\\_mise\\_.C3.A0\\_jour\\_de\\_la\\_norme](https://wikimonde.com/article/C%2B%2B11#Changements_pr.C3.A9vus_pour_la_mise_.C3.A0_jour_de_la_norme)



1. Visual Studio Code
2. Eclipse
3. NetBeans
4. Sublime Text
5. Atom
6. Code::Blocks
7. Qt Creator
8. CodeLite
9. CodeWarrior
10. Dev C++
11. MinGw
12. GNAT Programming Studio
13. C++ Builder
14. Anjuta
15. Clion
16. MonoDevelop

# 1 – Introduction -> Environnement de travail

## -> Option de compilation : g++



C++11 : g++ version à partir de 4.8









- g++ -std=c++11 example11.cpp -o main

C++ 14 : g++ version à partir de 5.0

- g++ -std=c++14 example14.cpp -o main

C++ 17 : g++ version à partir de 7.0

- g++ -std=c++17 example14.cpp -o main

- GCC (Updated 2019-09)
  - C++11 core language support status  (complete as of 4.8.1, except for n2670, which no compiler implements)
  - C++14 core language support status  (complete as of 5.1)
  - C++17 core language support status  (complete as of 7.1)
  - C++20 core language support status 
  - C++11 library support status  (complete as of 5.1)
  - C++14 library support status  (complete as of 5.1)
  - C++17 library support status 
  - C++20 library support status 

[http://en.cppreference.com/w/cpp/compiler\\_support](http://en.cppreference.com/w/cpp/compiler_support)



# 1 – Introduction

## -> Option de compilation : clang



C++11 : clang version à partir de 3.3

- clang -std=c++11 example11.cpp -o main

C++ 14 : clang version à partir de 3.4

- clang -std=c++14 example14.cpp -o main

C++ 17 : clang version à partir de 4.0

- g++ -std=c++17 example14.cpp -o main

[https://clang.llvm.org/cxx\\_status.html](https://clang.llvm.org/cxx_status.html)

- Clang++ (Updated 2017-09)

- C++11 core language support status [🔗](#) (complete as of 3.3)
- C++11 library support status (complete as of 2012-07-29 [🔒](#))
- C++14 core language support status [🔗](#) (complete as of 3.4)
- C++14 library support status [🔗](#) (complete as of 3.5)
- Technical Specifications support status [🔗](#)
- C++17 core language support status [🔗](#) (complete as of 5.0)
- C++17 library support status [🔗](#)
- C++20 core language support status [🔗](#)
- C++20 library support status [🔗](#)
- Core language defect report status [🔗](#)

[http://en.cppreference.com/w/cpp/compiler\\_support](http://en.cppreference.com/w/cpp/compiler_support)

# 1 – Introduction

## -> Option de compilation : icc



C++11 : Intel® C++ Compiler version à partir de 15.0







- `icc /Qstd=c++11 example11.cpp -o main`
- `icc -std=c++11 example11.cpp -o main`

C++14 : Intel® C++ Compiler Intel® C++ Compiler version à partir de 17.0

- `icc /Qstd=c++14 example11.cpp -o main`
- `icc -std=c++14 example11.cpp -o main`

C++17 : Intel® C++ Compiler de la version à partir de 18.0

- `icc /Qstd=c++17 example11.cpp -o main`
- `icc -std=c++17 example11.cpp -o main`
















- Intel C++ (Updated 2018-11)
  - [C++11 core language support status](#)  (complete as of 15.0)
  - [C++14 core language support status](#)  (functionally complete as of 17.0 - N3664 is an optimization)
  - [C++17 core language support status](#)  (incomplete)
  - [C++17 features of Intel 19.0 beta](#) 
  - Intel does not ship an implementation of the C++ standard library, except for
    - [Parallel STL](#)  (an implementation of the C++17 standard library algorithms with support for execution policies)
  - [Intel's compatibility with versions of libstdc++ from GCC](#) 

[http://en.cppreference.com/w/cpp/compiler\\_support](http://en.cppreference.com/w/cpp/compiler_support)

# 1 – Introduction

## -> Option de compilation : Visual Studio



- Microsoft Visual Studio (updated 2019-08)
  - C++17/20 Features and Fixes in Visual Studio 2019 
  - STL Features and Fixes in VS 2017 15.8 
  - C++17 Announcing: MSVC Conforms to the C++ Standard  (complete as of 15.7)
  - C++17 Features And STL Fixes In VS 2017 15.5 
  - C++17 Features And STL Fixes In VS 2017 15.3 
  - C++11/C++14/C++17 core language and library status in VS2017.3 
  - C++11/C++14/C++17 core language support status
    - C++11/14/17 core language support status in VS2010, VS2012, VS2013, and VS2015 
    - VS2013 vs. VS2015 CTP0 
    - VS2013 vs. VS2015 CTP1 
    - VS2013 vs. VS2015 CTP3  (includes the roadmap table)
    - VS2015 ("VS14") preview 
    - VS2015 ("VS14") release candidate  (C++11 remains incomplete, but C++17 support appears)
    - VS2019 
  - C++11 and C++14 library support status 
  - C++11/14/17 Features In VS 2015 RTM  including core language and standard library (including technical specifications)
  - C++14/17 features in VS 2015 Update 2 standard library  library is feature complete up to current C++17 with few minor issues (some defect reports, some constexprs, etc)
  - C++14/17 Features and STL Fixes in VS "15" Preview 5  including a detailed C++17 status table

[http://en.cppreference.com/w/cpp/compiler\\_support](http://en.cppreference.com/w/cpp/compiler_support)



## 2 - Évolution



- Mots-clés du C++
- Types prédéfinis
- Type de littéral
- Boucle Pour ... faire ... => for ( ) { } (C++11)
- Synonyme de type : using
- Le mot clé : nullptr
- Le mot clé : volatile
- Copies, déplacements et indirections
- La déduction de type
- Le mot clé : auto
- Le mot clé : decltype
- Tableaux
- Énumérations
- Union
- Amélioration des initialisations
- Expressions lambda
- std::function
- Les Objets avec C++11
- static\_assert
- std::bind

## 2 – Évolution -> Mots-clés du C++



alignas (depuis C++11)	enum	return
alignof (depuis C++11)	explicit	short
and	export (1)	signed
and_eq	extern (1)	sizeof (1)
asm	false	static
auto (1)	float	static_assert (depuis C++11)
bitand	for	static_cast
bitor	friend	struct
bool	goto	switch
break	if	template
case	inline (1)	this
catch	int	thread_local (depuis C++11)
char	long	throw
char16_t (depuis C++11)	mutable (1)	true
char32_t (depuis C++11)	namespace	try
class	new	typedef
compl	noexcept (depuis C++11)	typeid
const	not	typename
constexpr (depuis C++11)	not_eq	union
const_cast	nullptr (depuis C++11)	unsigned
continue	operator	using (1)
decltype (depuis C++11)	or	virtual
default (1)	or_eq	void
delete (1)	private	volatile
do	protected	wchar_t
double	public	while
dynamic_cast	register	xor
else	reinterpret_cast	xor_eq

override (C++11)  
final (C++11)

## Préprocesseur #

\_Pragma (depuis C++11)

if	ifdef	include
elif	ifndef	line
<u>else</u>	define	error
endif	undef	pragma
defined		

<http://fr.cppreference.com/w/cpp/keyword>

## 2 – Évolution -> Types prédéfinis



Déterminant du type	Type équivalent	Largeur (bits) en mémoire					
		C++ standard	LP32 / Win16	ILP32 / Win32 & Unix	LLP64 / Win64	LP64 / Unix	
short	short int	au moins 16	16	16	16	16	
short int							
signed short							
signed short int							
unsigned short	unsigned short int						
unsigned short int							
int	int	au moins 16	16	32	32	32	
signed							
signed int							
unsigned	unsigned int						
unsigned int							
long	long int						au moins 32
long int							
signed long							
signed long int							
unsigned long		unsigned long int					
unsigned long int							
long long	long long int (C++11)	au moins 64	64	64	64	64	
long long int							
signed long long							
signed long long int							
unsigned long long	unsigned long long int (C++11)						
unsigned long long int							

## 2 – Évolution -> Types prédéfinis



void	type vide
wchar_t	type représentant caractère large. (ex. UTF-8)
char16_t	type représentant un caractère UTF-16. (depuis C++11)
char32_t	type représentant un caractère UTF-32. (depuis C++11)
bool	true/false

```
std::cout << " char      : " << sizeof(char)      << " octets" << std::endl;
std::cout << " char16_t : " << sizeof(char16_t) << " octets" << std::endl;
std::cout << " char32_t : " << sizeof(char32_t) << " octets" << std::endl;
std::cout << " wchar_t  : " << sizeof(wchar_t)  << " octets" << std::endl;
std::cout << " bool      : " << sizeof(bool)    << " octets" << std::endl;
std::cout << " int       : " << sizeof(int)     << " octets" << std::endl;
std::cout << " short int: " << sizeof(short int) << " octets" << std::endl;
std::cout << " long int  : " << sizeof(long int)  << " octets" << std::endl;
std::cout << " long long int : " << sizeof(long long int) << " octets" << std::endl;
std::cout << " long int  : " << sizeof(long int)  << " octets" << std::endl;
```

```
char      : 1 octets
char16_t  : 2 octets
char32_t  : 4 octets
wchar_t   : 2 octets
bool      : 1 octets
int       : 4 octets
short int : 2 octets
long int  : 4 octets
long long int : 8 octets
long int  : 4 octets
```



## 2 – Évolution -> Types prédéfinis : Les caractères UTF-8 (C++17)



Le C++17 apporte aussi la possibilité d'utiliser le préfixe **u8** sur les caractères avec, celui-ci étant déjà disponible pour les chaînes de caractère depuis C++11.

```
using namespace std::string_literals; // enables s-suffix for std::string literals

void exempleType() {
    // https://docs.microsoft.com/fr-fr/cpp/cpp/string-and-character-literals-cpp?view=vs-2019
    // Character literals
    auto c0 = 'A'; // char
    auto c1 = u8'A'; // char
    auto c2 = L'A'; // wchar_t
    auto c3 = u'A'; // char16_t
    auto c4 = U'A'; // char32_t

    // Multicharacter literals
    auto m0 = 'abcd'; // int, value 0x61626364

    // String literals
    auto s0 = "hello"; // const char*
    auto s1 = u8"hello"; // const char*, encoded as UTF-8
    auto s2 = L"hello"; // const wchar_t*
    auto s3 = u"hello"; // const char16_t*, encoded as UTF-16
    auto s4 = U"hello"; // const char32_t*, encoded as UTF-32

    // Raw string literals containing unescaped \ and "
    auto R0 = R("Hello \ world"); // const char*
    auto R1 = u8R("Hello \ world"); // const char*, encoded as UTF-8
    auto R2 = LR("Hello \ world"); // const wchar_t*
    auto R3 = uR("Hello \ world"); // const char16_t*, encoded as UTF-16
    auto R4 = UR("Hello \ world"); // const char32_t*, encoded as UTF-32

    // Combining string literals with standard s-suffix
    auto S0 = "hello"s; // std::string
    auto S1 = u8"hello"s; // std::string
    auto S2 = L"hello"s; // std::wstring
    auto S3 = u"hello"s; // std::u16string
    auto S4 = U"hello"s; // std::u32string

    // Combining raw string literals with standard s-suffix
    auto S5 = R("Hello \ world")s; // std::string from a raw const char*
    auto S6 = u8R("Hello \ world")s; // std::string from a raw const char*, encoded as UTF-8
    auto S7 = LR("Hello \ world")s; // std::wstring from a raw const wchar_t*
    auto S8 = uR("Hello \ world")s; // std::u16string from a raw const char16_t*, encoded as UTF-16
    auto S9 = UR("Hello \ world")s; // std::u32string from a raw const char32_t*, encoded as UTF-32

    //affiche tablme ascii : http://www.asciitable.com/
    std::wcout << L'\x82' << std::endl;
    wprintf(L"%c\n", 130);
}
```



### template< class T > class optional

**std::optional** gère une valeur contenue optionnelle , c'est-à-dire une valeur qui peut être ou ne pas être présente.

```
#include <optional>
#include <iostream>
using namespace std;
optional<int> division_entiere(int num, int denom) {
    if (denom == 0) return {}; // optional vide
    return { num / denom }; // optional non vide
}
int main() {
    if (int num, denom; cin >> num >> denom)
        if (auto quotient = division_entiere(num, denom); quotient) // si on a un résultat...
            cout << quotient.value() << endl; // ... alors on l'affiche
}
```

<https://h-deb.clg.qc.ca/Sujets/Divers--cplusplus/optional.html>



## 2 – Évolution -> Types prédéfinis ( C++ 17 )



```
#include <string>
#include <functional>
#include <iostream>
#include <optional>

// optional can be used as the return type of a factory that may fail
std::optional<std::string> create(bool b) {
    if (b)
        return "Godzilla";
    return {};
}

// std::nullopt can be used to create any (empty) std::optional
auto create2(bool b) {
    return b ? std::optional<std::string>{"Godzilla"} : std::nullopt;
}

// std::reference_wrapper may be used to return a reference
auto create_ref(bool b) {
    static std::string value = "Godzilla";
    return b ? std::optional<std::reference_wrapper<std::string>>{value}
            : std::nullopt;
}

int exempleType_optional()
{
    std::cout << "create(false) returned "
              << create(false).value_or("empty") << '\n';

    // optional-returning factory functions are usable as conditions of while and if
    if (auto str = create2(true)) {
        std::cout << "create2(true) returned " << *str << '\n';
    }

    if (auto str = create_ref(true)) {
        // using get() to access the reference_wrapper's value
        std::cout << "create_ref(true) returned " << str->get() << '\n';
        str->get() = "Mothra";
        std::cout << "modifying it changed it to " << str->get() << '\n';
    }
}
```

```
create(false) returned empty
create2(true) returned Godzilla
create_ref(true) returned Godzilla
modifying it changed it to Mothra
```



## 2 – Évolution -> Types prédéfinis ( C++ 17 )



**enum class byte : unsigned char {}**

**std::byte** est un type distinct qui implémente le concept d'octet tel que spécifié dans la définition du langage C++.

Comme char et unsigned char , il peut être utilisé pour accéder à la mémoire brute occupée par d'autres objets, mais contrairement à ces types, **ce n'est pas un type de caractère ni un type arithmétique**.

*Un octet n'est qu'un ensemble de bits et les seuls opérateurs définis pour ce dernier sont ceux au niveau des bits.*

- Les opérateurs de décalages :
  - std::byte operator<<(std::byte, IntegerType shift)
  - std::byte operator>>(std::byte, IntegerType shift)
- Les opérateurs d'affectation par décalage :
  - std::byte& operator<=(std::byte&, IntegerType shift)
  - std::byte& operator>=(std::byte&, IntegerType shift)
- Les opérateurs logiques bit à bit :
  - std::byte operator|(std::byte, std::byte)
  - std::byte operator&(std::byte, std::byte)
  - std::byte operator^(std::byte, std::byte)
  - std::byte operator~(std::byte, std::byte)
- Les opérateurs d'affectation :
  - std::byte& operator|(std::byte&, std::byte)
  - std::byte& operator&(std::byte&, std::byte)
  - std::byte& operator^(std::byte&, std::byte)
- IntegerType to\_integer(std::byte b)



### class any

**std::any** décrit un conteneur de type sécurisé pour des valeurs uniques de tout type.

- Un objet de la classe any stocke une instance de tout type qui répond aux exigences du constructeur ou est vide, ce qui est appelé l' état de la classe any objet.
- Les fonctions any\_cast non membres fournissent un accès de type sécurisé à l'objet contenu.

## 2 – Évolution -> Types prédéfinis ( C++ 17 )



```
#include <any>

int exempleType_any()
{
    std::cout << std::boolalpha;
    // any type
    std::any a = 1;
    std::cout << a.type().name() << ": " << std::any_cast<int>(a) << '\n';
    a = 3.14;
    std::cout << a.type().name() << ": " << std::any_cast<double>(a) << '\n';
    a = true;
    std::cout << a.type().name() << ": " << std::any_cast<bool>(a) << '\n';
    // bad cast
    try
    {
        a = 1;
        std::cout << std::any_cast<float>(a) << '\n';
    }
    catch (const std::bad_any_cast& e)
    {
        std::cout << e.what() << '\n';
    }

    // has value
    a = 1;
    if (a.has_value())
    {
        std::cout << a.type().name() << '\n';
    }

    // reset
    a.reset();
    if (!a.has_value())
    {
        std::cout << "no value\n";
    }

    // pointer to contained data
    a = 1;
    int* i = std::any_cast<int>(&a);
    std::cout << *i << "\n";
}
```

```
i: 1
d: 3.14
b: true
bad any_cast
i
no value
1
```

## 2 – Évolution -> Type de littéral



Un type de littéral est un type dont la disposition peut être déterminée au moment de la compilation.

- void
- Types scalaires
- Références
- Tableaux de types void, de types scalaires ou de références
- Classe contenant :
  - un destructeur trivial,
  - un ou plusieurs constructeurs **constexpr** autres que des constructeurs de copie ou de déplacement.
  - tous les membres de données non statiques et classes de base doivent également être des types de littéral et être non volatiles.

## 2 – Évolution

-> Boucle Pour ... faire ... => for ( ) { }



Pour (déclaration: expression) faire { bloc d 'instructions }

for (déclaration: expression) { bloc d 'instructions }

```
#include <iostream>
#include <string>

int main()
{
    int tab[] = { 1,2,3,4,5 };

    for ( int i : tab ) std::cout << i << std::endl;
    std::cout << std::endl;

    for ( int &i : tab ) i*=10;

    for ( int i : tab ) std::cout << i << std::endl;
    std::cout << std::endl;

    return 0;
}
```



```
1
2
3
4
5

10
20
30
40
50
```





**if( initialisation; condition)**

instruction1;

**else**

instruction2;

## 2 – Évolution

### -> Les initialiseurs dans les instructions de sélections



```
int exempleIf()
{
    const std::string myString = "Hello World";

    auto it = myString.find("Hello");
    if (it != std::string::npos)
        std::cout << it << " Hello\n";

    auto it2 = myString.find("World");
    if (it2 != std::string::npos)
        std::cout << it2 << " World\n";

    // additional enclosing scope so 'it' doesn't 'leak'
    {
        auto it = myString.find("Hello");
        if (it != std::string::npos)
            std::cout << "Hello\n";
    }

    {
        auto it = myString.find("World");
        if (it != std::string::npos)
            std::cout << "World\n";
    }

    // C++17 with init if:
    if (const auto it = myString.find("Hello"); it != std::string::npos)
        std::cout << it << " Hello\n";

    if (const auto it = myString.find("World"); it != std::string::npos)
        std::cout << it << " World\n";
}
```

```
0 Hello
6 World
Hello
World
0 Hello
6 World
```



```
switch ( initialisation; condition) {  
    case choix1 : instruction1;  
    case choix2 : instruction1;  
    default: instruction1;  
}
```

## 2 – Évolution

### -> Les initialiseurs dans les instructions de sélections



```
int exempleIf_Switch() {  
  
    // Set up rand function to be used  
    // later in program  
    srand(time(NULL));  
  
    // Before C++17  
    int i = 2;  
    if ( i % 2 == 0 )  
        cout << i << " is even number" << endl;  
  
    // After C++17  
    // if(init-statement; condition)  
    if (int i = 4; i % 2 == 0 )  
        cout << i << " is even number" << endl;  
  
    // Switch statement  
    // switch(init;variable)  
    switch (int i = rand() % 100; i) {  
    default:  
        cout << "i = " << i << endl; break;  
    }  
}
```

```
2 is even number  
4 is even number  
i = 57
```

## 2 – Évolution

### -> Synonyme de type : using



- `typedef std::vector < std::vector < double > > MatriceDouble;`
- `typedef void( *func )(int);`

- **using** `MatriceDouble_C11 = std::vector < std::vector < double > > ;`
- **using** `func_C11 = void(*) (int);`
- `template<typename T> using ptr = T*;`

## 2 – Évolution

### -> Le mots clé : nullptr



Le nouveau mot-clé **nullptr** est une constante du type **nullptr\_t**, non convertible en entier du langage avec le caractère particulier d'être assignable à tous les types de pointeurs.

```
#define NULLPTR 0
```

```
void f(int *) {}
```

```
int * p1 = 0;  
int * p2 = NULLPTR;  
int * p3 = nullptr;
```

```
f(0);  
f(NULLPTR);  
f(nullptr);
```

```
std::cout << " 0          : " << sizeof (0) << " " << typeid(0).name() << std::endl;  
std::cout << " NULLPTR : " << sizeof (NULLPTR) << " " << typeid(NULLPTR).name() << std::endl;  
std::cout << " nullptr : " << sizeof (nullptr) << " " << typeid(nullptr).name() << std::endl;
```

```
0          : 4 i  
NULLPTR    : 4 i  
nullptr    : 4 Dn
```



une variable **volatile** est une variable sur laquelle aucune optimisation de compilation n'est appliquée.

```
int p = 5;  
f(p);
```

Après optimisation par le compilateur

```
f(5) ; // p a été remplacé
```

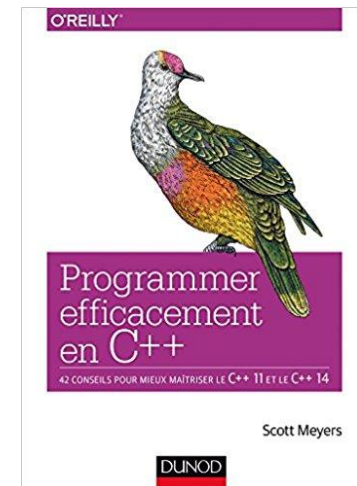
```
volatile int p = 5;  
f(p);
```

Après optimisation par le compilateur

```
int p = 5;  
f(p) ; // p a été remplacé
```



- Conseil n° 8. Préférer nullptr à 0 et à NULL
- Conseil n° 9. Préférer les déclarations d'alias aux typedef







- ***lvalues*** (*left-value*) : variables constantes ou non
- ***rvalues*** (*right-value*) : valeurs temporaires littérales ou expressions
- ***lvalue-reference*** : n'acceptent que des *lvalues*
- ***rvalue-reference*** : n'acceptent que des *rvalues*. Mise en place avec l'opérateur **&&**, les variables ne sont jamais constantes
- **universal reference** : variable qui sera définie lors de l'instanciation du template. La substitution du type définira si c'est une ***lvalue-reference*** ou une ***rvalue-reference*** (opérateur **&&**)

## 2 – Évolution

### -> La déduction de type



- Déclaration d'une fonction template

```
template <typename T> void fonction (ParamType param);
```

- Utilisation

```
fonction(expression)
```

- Exemple

```
template <typename T> void fonction(const T& val ) {  
    std::cout << "nom de la fonction : " << __func__ << "(const T&)\tT est de type  
: " << typeid(val).name() <<std::endl;  
}
```

C++11

```
nom de la fonction : fonction(const T&) T est de type : i
```

```
int main() {  
    int x = 32;  
    fonction (x);  
    return 0;  
}
```

## 2 – Évolution

-> La déduction de type :

**cas 1 - référence non universelle ou pointeur**



```
void calcul( float) {}
```

```
template <typename T> void fonction_1(T& val ) {  
    std::cout << "nom de la fonction : " << __func__ << "(T&)\tT est de type : " <<  
    typeid(val).name() << std::endl;  
}
```

```
template <typename T> void fonction_1(const T& val ) {  
    std::cout << "nom de la fonction : " << __func__ << "(const T&)\tT est de type : " <<  
    typeid(val).name() << std::endl;  
}
```

```
template <typename T> void fonction_1(T* val ) {  
    std::cout << "nom de la fonction : " << __func__ << "(T*)\tT est de type : " <<  
    typeid(val).name() << std::endl;  
}
```

```
template <typename T> void fonction_1(const T* val ) {  
    std::cout << "nom de la fonction : " << __func__ << "(const T*)\tT est de type : " <<  
    typeid(val).name() << std::endl;  
}
```

## 2 – Évolution

-> La déduction de type :

cas 1 - référence non universelle ou pointeur



```
float f = 2.2f;  
const float cf = f;  
float& rf = f;  
float tf[100];
```

```
fonction_1(f);  
fonction_1(cf);  
fonction_1(&rf);  
fonction_1(&cf);  
fonction_1(&tf);  
fonction_1(2.2f);  
  
fonction_1(calcul);
```

```
nom de la fonction : fonction_1(T&)      T est de type : f  
nom de la fonction : fonction_1(const T&)  T est de type : f  
nom de la fonction : fonction_1(T*)       T est de type : Pf  
nom de la fonction : fonction_1(const T*)  T est de type : PKf  
nom de la fonction : fonction_1(T*)       T est de type : PA100_f  
nom de la fonction : fonction_1(const T&)  T est de type : f  
nom de la fonction : fonction_1(T*)       T est de type : PFvfE
```

## 2 – Évolution

### -> La déduction de type : cas 2 - référence universelle



C++11

```
float f = 2.2f;  
const float cf = f;  
float& rf = f;  
float tf[100];
```

```
void calcul( float ) {}
```

```
template <typename T> void fonction_2(T&& val ) {  
    std::cout << "nom de la fonction : " << __func__ << "(T&&)\tT est de type : " << typeid(val).name() << std::endl;  
}
```

```
fonction_2(f);  
fonction_2(cf);  
fonction_2(&rf);  
fonction_2(&cf);  
fonction_2(&tf);  
fonction_2(2.2f);  
  
fonction_2(calcul);
```

```
nom de la fonction : fonction_2(T&&)    T est de type : f  
nom de la fonction : fonction_2(T&&)    T est de type : f  
nom de la fonction : fonction_2(T&&)    T est de type : PKf  
nom de la fonction : fonction_2(T&&)    T est de type : Pf  
nom de la fonction : fonction_2(T&&)    T est de type : PA100_f  
nom de la fonction : fonction_2(T&&)    T est de type : f  
nom de la fonction : fonction_2(T&&)    T est de type : FvfE
```

## 2 – Évolution

-> La déduction de type :

cas 3 – ni référence, ni pointeur



```
float f = 2.2f;  
const float cf = f;  
float& rf = f;  
float tf[100];
```

```
void calcul( float) {}
```

```
template <typename T> void fonction_3(T val ) {  
    std::cout << "nom de la fonction : " << __func__ << "(T)\tT est de type : " <<  
    typeid(val).name() << std::endl;  
}
```

```
fonction_3(f);  
fonction_3(cf);  
fonction_3(&rf);  
fonction_3(&cf);  
fonction_3(&tf);  
fonction_3(2.2f);  
fonction_3(calcul);
```

```
nom de la fonction : fonction_3(T)      T est de type : f  
nom de la fonction : fonction_3(T)      T est de type : f  
nom de la fonction : fonction_3(T)      T est de type : Pf  
nom de la fonction : fonction_3(T)      T est de type : PKf  
nom de la fonction : fonction_3(T)      T est de type : Pf  
nom de la fonction : fonction_3(T)      T est de type : f  
nom de la fonction : fonction_3(T)      T est de type : PFvfE
```

## 2 – Évolution

-> Le mot clé : auto



- Définition
- Avantage
- Inconvénient
- La déduction de Type

## 2 – Évolution

-> Le mot clé : auto - définition



```
auto i = 27;  
auto d = 3.4;  
auto f = 3.4f;  
  
cout << "i : " << typeid(i).name() << endl;  
cout << "d : " << typeid(d).name() << endl;  
cout << "f : " << typeid(f).name() << endl;
```

```
i : i  
d : d  
f : f
```

**auto** : permet de déduire le type à la compilation et indique au compilateur d'utiliser l'expression d'initialisation d'une variable déclarée, ou d'un paramètre d'expression lambda, pour déduire son type.

- Le mot clé auto est un moyen simple de déclarer une variable qui a un type complexe
- La déduction du type auto est identique à ceux des template



## 2 – Évolution

-> Le mot clé : auto - avantage



- **Robustesse** : si le type de l'expression est modifié, y compris quand un type de retour de fonction est modifié, cela fonctionne. Les variables auto doivent être initialisées.
- **Performances** : vous êtes certain qu'il n'y aura aucune conversion implicites silencieuses.
- **Usage** : vous n'avez pas à vous soucier des fautes de frappe ni des problèmes liés à l'orthographe du nom de type.
- **Efficacité** : votre codage peut être plus efficace.

## 2 – Évolution

-> Le mots clé : auto - inconvenient



Les types proxy ( invisible ) peuvent conduire auto à une mauvaise déduction du type.

```
class ImageReel {
public:
    ImageReel() {
        std::cout << "nom de la fonction : " << __func__ << "()\t" << typeid (*this).name() << std::endl;
    }
    ImageReel(const ImageReel&) {
        std::cout << "nom de la fonction : " << __func__ << "(const ImageReel&)\t" << typeid (*this).name() << std::endl;
    }
};

class ImageProxy {
public:
    ImageProxy(ImageReel&) {
        std::cout << "nom de la fonction : " << __func__ << "(ImageReel&)\t" << typeid (*this).name() << std::endl;
    }
};
```

```
nom de la fonction : ImageReel()          9ImageReel
nom de la fonction : ImageProxy(ImageReel&) 10ImageProxy

nom de la fonction : ImageReel(const ImageReel&) 9ImageReel
nom de la fonction : ImageProxy(ImageReel&) 10ImageProxy
```

```
ImageReel img;
ImageProxy imgproxy = img;
sautligne();
auto imgautoerr = img;
sautligne();
auto imgauto = static_cast<ImageProxy> (img);
```

## 2 – Évolution -> Le mot clé : auto

### - La déduction de Type auto:

#### cas 1 - référence non universelle ou pointeur



```
auto f = 2.2f;  
const auto cf = f;  
float tf[100];  
auto af = tf;  
auto pf = calcul;
```

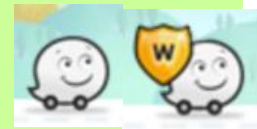
```
cout << "f : " << typeid (f).name() << endl;  
cout << "cf : " << typeid (cf).name() << endl;  
cout << "tf : " << typeid (tf).name() << endl;  
cout << "af : " << typeid (af).name() << endl;  
cout << "pf : " << typeid (pf).name() << endl;
```

```
f : f  
cf : f  
tf : A100_f  
af : Pf  
pf : PFvfe
```

## 2 – Évolution -> Le mot clé : auto

### - La déduction de Type auto:

#### cas 2 - référence universelle



```
auto&& f = 2.2f;  
auto&& cf = f;  
float tf[100];  
auto&& af = tf;  
auto&& pf = calcul;
```

```
cout << "f : " << typeid (f).name() << endl;  
cout << "cf : " << typeid (cf).name() << endl;  
cout << "tf : " << typeid (tf).name() << endl;  
cout << "af : " << typeid (af).name() << endl;  
cout << "pf : " << typeid (pf).name() << endl;
```

```
f      : f  
cf     : f  
tf     : A100_f  
af     : A100_f  
pf     : Fvfe
```



```
const auto& f = 2.2f;
```

```
const auto& rf = f;
```

```
float tf[100];
```

```
auto& af = tf;
```

```
auto& pf = calcul;
```

```
cout << "f : " << typeid (f).name() << endl;
```

```
cout << "rf : " << typeid (rf).name() << endl;
```

```
cout << "tf : " << typeid (tf).name() << endl;
```

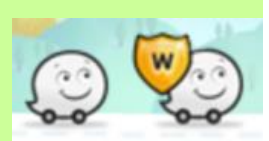
```
cout << "af : " << typeid (af).name() << endl;
```

```
cout << "pf : " << typeid (pf).name() << endl;
```

```
f      : f
rf     : f
tf     : A100_f
af     : A100_f
pf     : FvfE
```

## 2 – Évolution -> Le mot clé : auto

### - La déduction de Type auto avec `std::initializer_list<> ↔ {}`



```
auto d1 = 5.5;  
auto d1a(5.5);
```

```
auto d2 = {5.5};  
auto d2a {5.5};  
auto d2b = std::initializer_list<double>{5.5};
```

```
cout << "d1 : " << typeid (d1).name() << endl;  
cout << "d1a : " << typeid (d1a).name() << endl;  
cout << "d2 : " << typeid (d2).name() << endl;  
cout << "d2a : " << typeid (d2a).name() << endl;  
cout << "d2b : " << typeid (d2b).name() << endl;
```

```
f : f  
rf : f  
tf : A100_f  
af : A100_f  
pf : FvFE  
  
d1 : d  
d1a : d  
d2 : St16initializer_listIdE  
d2a : d  
d2b : St16initializer_listIdE
```

## 2 – Évolution -> Le mot clé : decltype



- `decltype` permet de typer une variable à partir du type d'une autre variable

```
auto d = 5.5;  
decltype(d) dd;  
  
cout << "d : " << typeid(d).name() << endl;  
cout << "dd : " << typeid(dd).name() << endl;
```

```
d : d  
dd : d
```

## 2 – Évolution -> Le mot clé : decltype



```
//C++14
template<typename T, typename U>
decltype(auto) add1_14(T& t, U& u) {
    return t + u;
}

template<typename T, typename U>
decltype(auto) add2_14(T&& t, U&& u) {
    return std::forward<T>(t) + std::forward<U>(u);
}

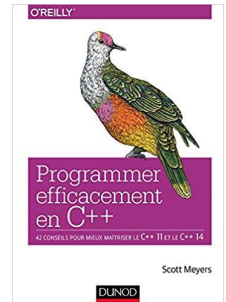
//C++11
template<typename T, typename U>
auto add1_11(T& t, U& u) -> decltype (t + u){
    return t + u;
}

template<typename T, typename U>
auto add2_11(T&& t, U&& u) -> decltype (std::forward<T>(t) + std::forward<U>(u)) {
    return std::forward<T>(t) + std::forward<U>(u);
}
```





- Conseil n° 1. Comprendre la déduction de type de template
- Conseil n° 2. Comprendre la déduction de type auto
- Conseil n° 3. Comprendre decltype
- Conseil n° 4. Afficher les types déduits
- Conseil n° 5. Préférer auto aux déclarations de types explicites
- Conseil n° 6. Opter pour un initialiseur au type explicite lorsque auto déduit
- Conseil n° 24. Distinguer les références universelles et les références rvalue
- Conseil n° 26. Éviter la surcharge sur les références universelles



## 2 – Évolution -> Tableaux



C++11 : `std::initializer_list` : `perme`

```
int tabValue0[5];
int tabValue1[] = {1, 2, 3, 4, 5};
int tabValue2[10] = {1, 2, 3, 4, 5};
```

```
std::array<int, 5> myarray0;
std::array<int, 5> myarray1 = {10, 20, 30, 40, 50};
std::array<int, 10> myarray2 ( {10, 20, 30, 40, 50} );
std::array<int, 10> myarray3 = myarray2;
myarray0 = myarray1;
```

```
cout << "tabValue0 : \t" << typeid (tabValue0).name() << " : \t" << sizeof (tabValue0) << endl;
cout << "tabValue1 : \t" << typeid (tabValue1).name() << " : \t" << sizeof (tabValue1) << endl;
cout << "tabValue2 : \t" << typeid (tabValue2).name() << " : \t" << sizeof (tabValue2) << endl;
cout << endl;
cout << "myarray0 : \t" << typeid (myarray0).name() << " : \t" << sizeof (myarray0) << endl;
cout << "myarray1 : \t" << typeid (myarray1).name() << " : \t" << sizeof (myarray1) << endl;
cout << "myarray2 : \t" << typeid (myarray2).name() << " : \t" << sizeof (myarray2) << endl;
cout << "myarray3 : \t" << typeid (myarray3).name() << " : \t" << sizeof (myarray3) << endl;
```

```
tabValue0 : A5_i : 20
tabValue1 : A5_i : 20
tabValue2 : A10_i : 40

myarray0 : St5arrayIiLj5EE : 20
myarray1 : St5arrayIiLj5EE : 20
myarray2 : St5arrayIiLj10EE : 40
myarray3 : St5arrayIiLj10EE : 40
```

## 2 – Évolution -> Tableaux : <array> => std::array



### array

array::array	C++11
<b>member functions:</b>	
array::at	C++11
array::back	C++11
array::begin	C++11
array::cbegin	C++11
array::cend	C++11
array::crbegin	C++11
array::crend	C++11
array::data	C++11
array::empty	C++11
array::end	C++11
array::fill	C++11
array::front	C++11
array::max_size	C++11
array::operator[]	C++11
array::rbegin	C++11
array::rend	C++11
array::size	C++11
array::swap	C++11
<b>non-member overloads:</b>	
get (array)	C++11
relational operators (array)	C++11
<b>non-member specializations:</b>	
tuple_element<array>	C++11
tuple_size<array>	C++11

```
std::array<int, 5> myarray1 = {10, 20, 30, 40, 50};
std::array<int, 5> myarray2 = myarray1;
```

```
for ( int i : myarray1) std::cout << i << endl;
for ( int& i : myarray1) i*=10;
cout << endl;
for ( const int& i : myarray1) std::cout << i << endl;
cout << endl;
```

```
for ( auto i : myarray2) std::cout << i << endl;
for ( auto& i : myarray2) i*=10;
cout << endl;
for ( const auto& i : myarray2) std::cout << i << endl;
cout << endl;
```

```
10
20
30
40
50

100
200
300
400
500

10
20
30
40
50

100
200
300
400
500
```

## 2 – Évolution -> Enumération



### • enum non délimités C++98

```
enum Color {RED, GREEN, BLUE};
Color c = BLUE;
cout << "enum Color : \t" << std::boolalpha << std::is_enum<Color>::value << " : \t BLUE " << c << " : \t" << sizeof (c) << endl;
cout << endl;
```

```
enum Color : true : BLUE 2: 4
```

### • enum délimités

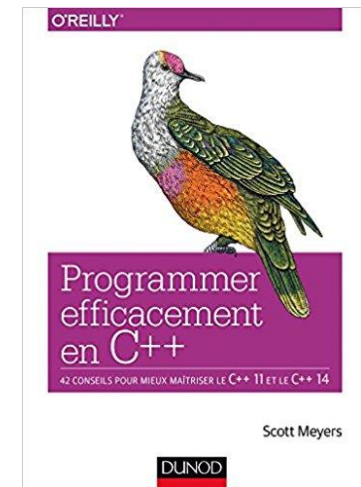
```
enum class Color_11 : true : BLUE 2: 4
enum class Color_int16_11 : true : BLUE 3: 2
enum class Color_char_11 : true : BLUE b: 1
```

```
enum class Color_11 {RED, GREEN, BLUE};
enum class Color_int16_11 : std::uint16_t { RED = 1, GREEN = 2, BLUE = 3 };
enum class Color_char_11 : char { RED = 'r', GREEN = 'g', BLUE = 'b'};

Color_11 c1 = Color_11::BLUE;
cout << "enum class Color_11 : \t" << std::boolalpha << std::is_enum<Color_11>::value
<< " : \t BLUE " << static_cast<int>(c1) << " : \t" << sizeof (c1) << endl;
Color_int16_11 c2 = Color_int16_11::BLUE;
cout << "enum class Color_int16_11 : \t" << std::boolalpha << std::is_enum<Color_int16_11>::value
<< " : \t BLUE " << static_cast<std::uint16_t>(c2) << " : \t" << sizeof (c2) << endl;
Color_char_11 c3 = Color_char_11::BLUE;
cout << "enum class Color_char_11 : \t" << std::boolalpha << std::is_enum<Color_char_11>::value
<< " : \t BLUE " << static_cast<char>(c3) << " : \t" << sizeof (c3) << endl;
```



- Conseil n° 10. Préférer les enum délimités aux enum non délimités



## 2 – Évolution -> Union ( structure polytype) -> Union simple



```
union ChampsType {  
    char ch;  
    int i;  
    double d;  
    int *int_ptr;  
};
```

```
ChampsType b;  
b.d=5.1;  
std::cout << b.d << " " << b.ch << std::endl;
```

5.1 f

## 2 – Évolution -> Union ( structure polytype) -> Unions illimitées (C++11)



```
union ChampsTypeClass {  
  
    char ch;  
    int i;  
    double d;  
    std::shared_ptr<int> int_ptr;  
    std::array<int, 5> tab;  
    std::string s;  
  
    ChampsTypeClass () {}  
    ~ChampsTypeClass () {}  
};
```

```
ChampsTypeClass e;  
new (&e.s) std::string("David");  
cout << e.s << endl;  
e.s.~string();
```

David

## 2 – Évolution -> Union ( structure polytype) -> Unions illimitées (C++11)



```
class Exemple {
public:
    enum class Type { STRING = 0, NUMERO = 1 } a_type;

    Exemple(const Exemple::Type& type) : a_type(type) {
        if (a_type == Type::STRING) {
            new (&nom) std::string("David");
        } else {
            numero = 100005607;
        }
    }

    void afficher() const {
        if (a_type == Type::STRING) {
            std::cout << nom;
        } else {
            std::cout << numero;
        }
        std::cout << "(" << sizeof (*this) << ")" << std::endl;
    }

    ~Exemple() {
        if (a_type == Type::STRING) nom.~string();
    }

private:
    union {
        std::string nom;
        int numero;
    };
};
```

```
Exemple Enum(Exemple::Type::NUMERO);
Enum.afficher();
Exemple Enom(Exemple::Type::STRING);
Enom.afficher();
```

```
100005607(28)
David(28)
```



## 2 – Évolution -> Union ( structure polytype) -> std::variant ( C++ 17 )



### template < classe ... Types > class variant

**std::variant** représente une  
union de type sécurisée.

Une instance de **std::variant** à  
un moment donné contient une  
valeur de l'un de ses types  
alternatifs ou, en cas d'erreur,  
aucune valeur,

```
int exempleVariant()
{
    std::variant<int, float> v, w;
    v = 12; // v contains int
    int i = std::get<int>(v);
    w = std::get<int>(v);
    w = std::get<0>(v); // same effect as the previous line
    w = v; // same effect as the previous line

    // std::get<double>(v); // error: no double in [int, float]
    // std::get<3>(v);      // error: valid index values are 0 and 1

    try {
        std::get<float>(w); // w contains int, not float: will throw
    }
    catch (const std::bad_variant_access&) {}

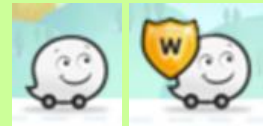
    using namespace std::literals;

    std::variant<std::string> x("abc");
    // converting constructors work when unambiguous
    x = "def"; // converting assignment also works when unambiguous

    std::variant<std::string, void const*> y("abc");
    // casts to void const * when passed a char const *
    assert(std::holds_alternative<void const*>(y)); // succeeds
    y = "xyz"s;
    assert(std::holds_alternative<std::string>(y)); // succeeds
}
```

## 2 – Évolution

### -> Amélioration des initialisations



- Généralisation de la notation {}
- liste d'initialisation : fonction
- liste d'initialisation : classe

## 2 – Évolution -> Amélioration des initialisations -> Généralisation de la notation {}



```
class A {  
public:  
    A(int i):a_i(i) {}  
private:  
    int a_i;  
};  
  
struct B {  
    int a_i;  
    double a_d;  
};  
  
class Paire{  
public:  
    Paire(int x,int y):a_p{x,y}  
{}  
private:  
    int a_p[2];  
};
```

- **Initialisation de variable ou d'objet**  
`int i = 5; // équivalent à int i(5);`  
`A (5) ; // équivalent à A a(5);`
- **Initialisation agrégés**  
`B s = { 1,1.5};`  
`int t[]={ 1,2,3,4};`
- **initialisation de variable ou objet ( C++ 11 )**  
`int j = {20}; // équivalent à int j {20};`  
`int t1[]={ 1,2,3,4}; // équivalent à int t1[] = { 1,2,3,4};`  
`int tab2d[2][2] { 1,2,3,4}; // équivalent à int tab2d[2][2] = {{1,2},{3,4}};`  
`Paire p{1,2}; // équivalent à Exemple::Paire p(1,2); <=> Paire p={1,2};`  
`int * tab3 = new int[5]{1,2,3}; // équivalent à int * tab3 = new int[5]{1,2,3,0,0}`

## 2 – Évolution -> Amélioration des initialisations -> liste d'initialisation : fonction



```
void g(double t[4]) {std::cout << "g(double [4])" << std::endl;}

void g(std::initializer_list<double> l) {std::cout << "g(std::initializer_list<double>)" << std::endl;}

void g(double d) { std::cout << "g(double)" << std::endl;}

void g()          { std::cout << "g()" << std::endl;}
```

```
g({1.1,2.2,3.3});

g({1,2,3});

g({1});

g({});

g();

g(1);
```

```
g(std::initializer_list<double>)
g(std::initializer_list<double>)
g(std::initializer_list<double>)
g(std::initializer_list<double>)
g()
g(double)
```

## 2 – Évolution -> Amélioration des initialisations -> liste d'initialisation : classe



```
class MultiElement{
public:
    MultiElement( std::initializer_list<int> li ){
        std::cout << "MultiElement( std::initializer_list<int> )" << std::endl;
    }
    MultiElement( int i,double j){
        std::cout << "MultiElement( int ,double )" << std::endl;
    }
    MultiElement( double j){
        std::cout << "MultiElement( double )" << std::endl;
    }
    MultiElement( const std::string& str) {
        std::cout << "MultiElement( const std::string& )" << std::endl;
    }
};
```

```
MultiElement m1{};
```

```
MultiElement m2{1,2,3};
```

```
MultiElement m3(1,2.);
```

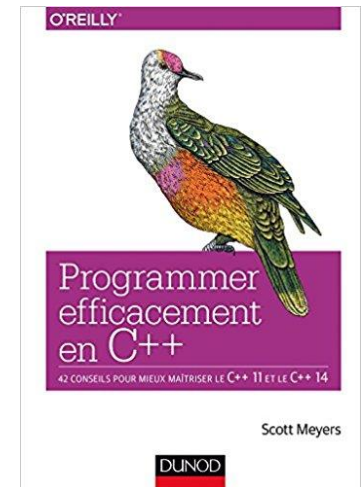
```
MultiElement m4(2.);
```

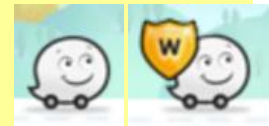
```
MultiElement m5{"test« };
```

```
MultiElement( std::initializer_list<int> )
MultiElement( std::initializer_list<int> )
MultiElement( int ,double )
MultiElement( double )
MultiElement( const std::string& )
```



- Conseil n° 7. Différencier () et {} lors de la création des objets



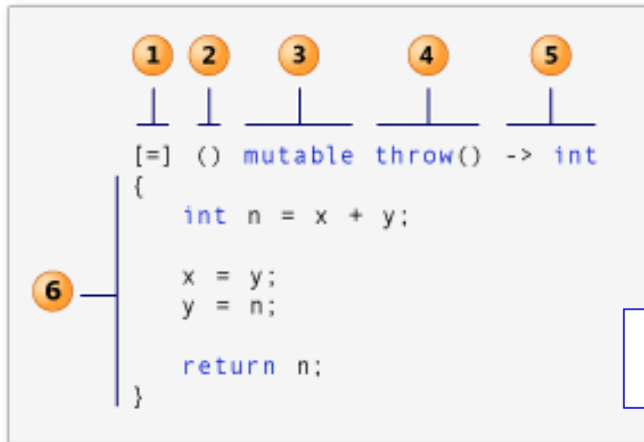


La fonction lambda est une fonction **inline** créée directement dans votre code et éventuellement passée directement dans les paramètres d'une de vos méthodes.

- **Avantages**

- La fonction est inline : **plus de performance.**
- La fonction est privée à votre code
- Pour des fonctions courtes : code plus concis (elle évite la création de fonctions)

## 2 – Évolution -> Expressions lambda

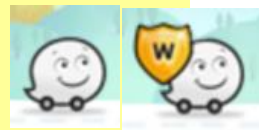


```
auto p = [] () { cout << "Je suis une fonction lambda" << endl; };
p();
```

1. Partie capture qui spécifie les symboles visibles dans le champ où la fonction est déclarée sera visible à l'intérieur du corps de la fonction. Une liste de symboles peut être adoptée comme suit:
  - `[a,&b]` où `a` est capturé par valeur et `b` est capturé par référence.
  - `[this]` capte le pointeur `this`
  - `[&]` captures all symbols by reference
  - `[=]` captures all by value
  - `[]` captures nothing
2. liste des paramètres, comme dans fonctions nommées
3. **mutable** : permet à body de modifier les paramètres capturés par copie, et d'appeler leurs fonctions de membre non-const
4. **throw** : fournit la spécification d'exception ou la noexcept clause pour l'opérateur `()` du type de fermeture
5. type de retour : s'il n'est pas présent il est implicite dans les énoncés de retour de fonction (ou void si elle ne retourne aucune valeur)
6. corps de la fonction lambda



## 2 – Évolution -> Expressions lambda : Exemple



```
vector<int> v = {30,-20, 40, -40,10,20};

std::sort(v.begin(), v.end(), myfunction); // avec bool myfunction(int a, int b) { return a < b; }
for (auto i: v) std::cout << i << " ";
std::cout << std::endl;

std::sort(v.begin(), v.end(), [](int a, int b) { return a > b; });

for (auto i: v) std::cout << i << " ";
std::cout << std::endl;
std::cout << std::endl;
```

```
-40 -20 10 20 30 40
40 30 20 10 -20 -40
```

```
int a = 2;
auto p1 = [&a] () { return a = a*a; }; // Retour implicite
auto p2 = [&a] () -> int { return a = a*a; }; // Retour explicite
auto p3 = [&a] () -> decltype(a) { return a = a*a; }; // Retour explicite

cout << "p1 = " << p1() << " a = " << a << endl;
cout << "p2 = " << p2() << " a = " << a << endl;
cout << "p3 = " << p3() << " a = " << a << endl;
```

```
p1 = 4 a = 2
p2 = 16 a = 4
p3 = 256 a = 16
```

## 2 – Evolution -> Expressions lambda : Exemple



```
int plus(const int& lhs, const int& rhs) { return lhs + rhs; }
```

```
auto plus2 = std::bind(&plus, std::placeholders::_1, 2);  
std::cout << plus2(123) << std::endl;
```

```
auto plus_lambda2 = std::bind([](const int& lhs, const int& rhs)->int{ return lhs + rhs; }, std::placeholders::_1, 2);  
std::cout << plus_lambda2(123) << std::endl;
```

125  
125

## 2 – Évolution -> Expressions lambda : Exemple



```
int i = 10;
int j = 200;
int res = 0;

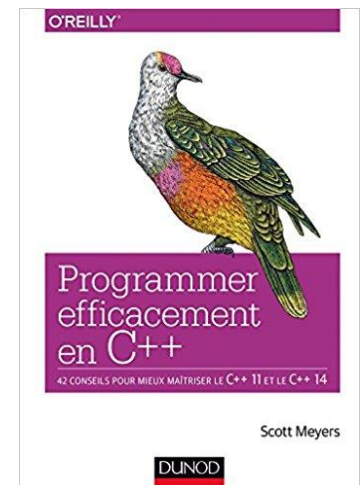
auto divparam = [] (const int& ip, const int& jp) throw (std::exception) { if ( ip == 0) throw std::exception(); return jp/ip;};
auto divvalue = [=] () throw (std::exception) { if ( i == 0) throw std::exception(); return j/i;};
auto divref = [&] () throw ( std::exception ) { if ( i == 0) throw std::exception(); res = j/i ; return res;};
auto divmutable = [i,j,res] () mutable throw( std::exception ) -> int { if ( i == 0) throw std::exception(); res = j/i ; return res;};
auto divrefmutable = [i,j,&res] () mutable throw( std::exception ) -> int& { if ( i == 0) throw std::exception(); res = j/i ; return res;};

res=0;
std::cout << res << " " ;
std::cout << "divparam => " << divparam(i,j) << " ";
std::cout << res << std::endl;
res=0;
std::cout << res << " " ;
std::cout << " divvalue => " << divvalue() << " ";
std::cout << res << std::endl;
res=0;
std::cout << res << " " ;
std::cout << " divref => " << divref() << " ";
std::cout << res << std::endl;
res=0;
std::cout << res << " " ;
std::cout << " divmutable => " << divmutable() << " ";
std::cout << res << std::endl;
res=0;
std::cout << res << " " ;
std::cout << " divrefmutable => " << divrefmutable() << " ";
std::cout << res << std::endl;
res=0;
std::cout << res << " " ;
std::cout << "[&] () mutable throw( std::exception ) -> int& => " <<
    ( [&] () mutable throw( std::exception ) -> int& { if ( i == 0) throw std::exception(); res = j/i ; return res;}) () << " ";
std::cout << res << std::endl;
```

```
0 divparam => 20 0
0 divvalue => 20 0
0 divref => 20 20
0 divmutable => 20 0
0 divrefmutable => 20 20
```



- Conseil n° 31. Éviter les modes de capture par défaut
- Conseil n° 32. Utiliser des captures généralisées pour déplacer des objets dans des fermetures
- Conseil n° 34. Préférer les expressions lambda à `std::bind`



## 2 – Évolution -> std::function



La classe std::function est un adaptateur générique de fonction.

Une instance de std::function permet de manipuler une cible callable, pour laquelle operator() est défini

```
class Cint {
public:
    Cint(const int& num) : a_num(num) {}
    void plus_int(const int& i) const { std::cout << this->a_num+i << std::endl; }
private:
    int a_num;
};

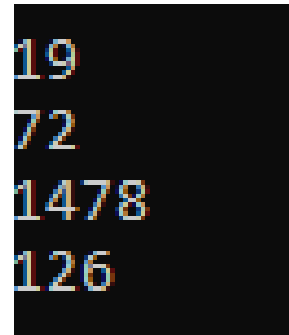
void afficher(int i)
{
    std::cout << i << std::endl;
}
```

```
std::function<void(const int&)> f_afficher = afficher;
f_afficher(19);
```

```
std::function<void()> f_afficher_N = [] () { afficher(72); };
f_afficher_N();
```

```
f_afficher_N = std::bind(afficher, 1478);
f_afficher_N();
```

```
std::function<void(const Cint&, const int&)> f_afficher_plus_int = &Cint::plus_int;
Cint c(125);
f_afficher_plus_int(c, 1);
```



```
19
72
1478
126
```

## 2 – Évolution -> Les Objets avec C++11



- Constructeurs
- Interdiction de surcharge : final
- Obligation de surcharge : override
- Le mot clé : constexpr
- Littérales définies par l'utilisateur : operator " "
- Espaces de noms : inline namespace
- Le mot clé : noexcept
- Déclaration étendu friend
- Bibliothèque <chrono>
- Paramètres template étendus pour les templates template variadic
- extern template class

## 2 –Évolution -> Les Objets avec C++11 -> Constructeur



- Initialisation des données membres non-statiques
- Constructeurs délégués
- Constructeurs d'héritage
- Constructeurs par déplacement
- *Opérateur d'affectation et déplacement*
- *Méthode et déplacement*
- Constructeurs avec « ***Initializer-list*** »
- Constructeur « ***explicit*** »
- Constructeur et fonctions par défaut supprimées



```
class A {
```

```
public:
```

```
    static const int m1 = 7;           // ok
```

```
    int m2 = 7;                       // ok
```

```
    const int m3 = 7;                 //ok
```

```
    static int m4 = 7;                // erreur : non constant
```

```
    static const int m5 = var;         // erreur : n'est pas une  
    expression constante
```

```
    static const std::string m6 = "odd"; // erreur : n'est pas un  
    type intégral
```

```
};
```



## 2 – Évolution -> Les Objets avec C++11

### -> Constructeur : Constructeurs délégués



C++11 introduit la possibilité de déléguer des constructeurs => les constructeurs de classe peuvent être invoqués au sein d'autres constructeurs de la même classe.

```
class B {  
    B(int newA) : a(newA) { ...  
    }  
  
    public:  
        B() : B(15) { ... }  
  
    public:  
        int a;  
};
```

## 2 – Évolution -> Les Objets avec C++11

### -> Constructeur : Constructeurs délégués



```
class Personne {
public:
    Personne()
        :Personne("sans nom") {
            std::cout << __func__ << "()" << std::endl;
        }

    Personne(const std::string& nom)
        :Personne(nom,"sans adresse") {std::cout << __func__ << "(const std::string&)" << std::endl;}

    Personne(const std::string& nom,const std::string& adresse)
        :a_nom(nom),a_adresse(adresse) {std::cout << __func__ << "const std::string& ,std::string& adresse" << std::endl;}

private:
    std::string a_nom;
    std::string a_adresse;
};
```

Personne p;

```
Personneconst std::string& ,std::string& adresse
Personne(const std::string&)
Personne()
```

## 2 – Évolution -> Les Objets avec C++11

### -> Constructeur : Constructeurs d'héritage



Permet d'initialiser une classe dérivée avec le même ensemble de constructeur que la classe de base

```
struct A {  
    A(int );  
};  
struct B: A {  
    B(int,int );  
};
```

```
struct C: A {  
    using A::A;  
};  
struct D: B {  
    using B::B;  
};
```

```
struct C: A {  
    C();  
    C(int);  
    C(const C&);  
};  
struct D: B {  
    D();  
    D(int);  
    D(int,int)  
    D(const B&);  
};
```

## 2 – Évolution -> Les Objets avec C++11

### -> Constructeur : Constructeurs d'héritage



```
class Personne {
public:
    Personne()
    :Personne("sans nom") {
        std::cout << __func__ << "()" << std::endl;
    }

    Personne(const std::string& nom)
    :Personne(nom, "sans adresse") {std::cout << __func__ << "(const std::string&)" << std::endl;}

    Personne(const std::string& nom, const std::string& adresse)
    :a_nom(nom), a_adresse(adresse) {std::cout << __func__ << "const std::string& ,std::string& adresse" << std::endl;}

private:
    std::string a_nom;
    std::string a_adresse;
};
```

```
class Personnel : public Personne {
public:
    using Personne::Personne;
};
```

```
Personnel p1;
std::cout << std::endl;
Personnel p2("David");
std::cout << std::endl;
Personnel p3("David", "aix en provence");
std::cout << std::endl;
```

```
Personneconst std::string& ,std::string& adresse
Personne(const std::string&)
Personne()

Personneconst std::string& ,std::string& adresse
Personne(const std::string&)

Personneconst std::string& ,std::string& adresse
```



Un **constructeur de déplacement** vous permet d'implémenter **une sémantique de déplacement**, qui peut améliorer considérablement les performances de vos applications.

L'objectif est de voler autant de ressources que possible à l'objet original, aussi rapidement que possible, **attention** car l'original n'auras plus de valeur.

<http://akrzemi1.developpez.com/tutoriels/c++/constructeur-par-deplacement/#LIII>



```
class Vecteur {
private:
    unsigned a_taille;
    std::unique_ptr<double[]> a_tab = nullptr;
public:
    Vecteur(const unsigned n)
    : a_taille(n), a_tab(new double[n], std::default_delete<double[]>()) {
        std::cout << __func__ << "(const unsigned) this(" << this <<
            " ) tab(" << a_tab.get() << " ) taille(" << &a_taille << " )" << std::endl;
    }

    Vecteur(const Vecteur& v)
    : a_taille(v.a_taille), a_tab(new double[v.a_taille]) {
        for (unsigned i = 0; i < a_taille; ++i) a_tab[i] = v.a_tab[i];
        std::cout << __func__ << "(const Vecteur& ) this(" << this <<
            " ) tab(" << a_tab.get() << " ) taille(" << &a_taille << " )" << std::endl;
    }

    Vecteur(Vecteur&& v)
    : a_taille(std::move(v.a_taille)), a_tab(std::move(v.a_tab)) {
        std::cout << __func__ << "(Vecteur&& ) this(" << this <<
            " ) tab(" << a_tab.get() << " ) taille(" << &a_taille << " )" << std::endl;
    }

    ~Vecteur() {
        std::cout << __func__ << " this(" << this <<
            " ) tab(" << a_tab.get() << " ) taille(" << &a_taille << " )" << std::endl;
    }
};
```

```
{
    Vecteur v1(5);
    Vecteur v2(v1);
    Vecteur v3(std::move(v1));
    std::cout << std::endl;
}
```

```
Vecteur(const unsigned) this(0x65f8a0) tab(0x2997eb0) taille(0x65f8a0)
Vecteur(const Vecteur& ) this(0x65f898) tab(0x2997f58) taille(0x65f898)
Vecteur(Vecteur&& ) this(0x65f890) tab(0x2997eb0) taille(0x65f890)
~Vecteur this(0x65f890) tab(0x2997eb0) taille(0x65f890)
~Vecteur this(0x65f898) tab(0x2997f58) taille(0x65f898)
~Vecteur this(0x65f8a0) tab(0) taille(0x65f8a0)
```

## 2 – Évolution -> Les Objets avec C++11

-> Constructeur :

Opérateur d'affectation et déplacement



```
class Vecteur {
private:
    unsigned a_taille;
    std::unique_ptr<double[] > a_tab = nullptr;
public:
    Vecteur(const unsigned n)
        : a_taille(n), a_tab(new double[n] , std::default_delete<double[]>()) {

    Vecteur(const Vecteur& v)
        : a_taille(v.a_taille), a_tab(new double[v.a_taille]) {...5 lines }

    Vecteur(Vecteur&& v)
        : a_taille(std::move(v.a_taille)), a_tab(std::move(v.a_tab)) {...4 lines }

    ~Vecteur() {...4 lines }

    Vecteur& operator=(const Vecteur& v) {
        if (&v == this) return *this;
        a_tab.reset(new double[v.a_taille]);
        a_taille = v.a_taille;
        std::cout << __func__ << "(const Vecteur& v) this(" << this <<
            ") tab(" << a_tab.get() << ") taille(" << &a_taille << ")" << std::endl;

        return *this;
    }

    Vecteur& operator=(Vecteur&& v) {
        if (&v == this) return *this;
        a_tab = std::move(v.a_tab);
        a_taille = std::move(v.a_taille);
        std::cout << __func__ << "(Vecteur&& v) this(" << this <<
            ") tab(" << a_tab.get() << ") taille(" << &a_taille << ")" << std::endl;
        return *this;
    }
};
```

```
{
    Vecteur v1(5);
    Vecteur v2(v1);
    Vecteur v3(1);
    std::cout << std::endl;

    v3 = std::move(v1);
    v1 = v2;
    std::cout << std::endl;
}
```

```
Vecteur(const unsigned) this(0x65f888) tab(0x2997eb0) taille(0x65f888)
Vecteur(const Vecteur& ) this(0x65f880) tab(0x2997f58) taille(0x65f880)
Vecteur(const unsigned) this(0x65f878) tab(0x2997ee0) taille(0x65f878)

operator=(Vecteur&& v) this(0x65f878) tab(0x2997eb0) taille(0x65f878)
operator=(const Vecteur& v) this(0x65f888) tab(0x2997f88) taille(0x65f888)

~Vecteur this(0x65f878) tab(0x2997eb0) taille(0x65f878)
~Vecteur this(0x65f880) tab(0x2997f58) taille(0x65f880)
~Vecteur this(0x65f888) tab(0x2997f88) taille(0x65f888)
```

## 2 – Évolution -> Les Objets avec C++11

### -> Constructeur : Méthode et déplacement



```
class Vecteur {
private:
    unsigned a_taille;
    std::unique_ptr<double[] > a_tab = nullptr;
public:
    Vecteur(const unsigned n)
        : a_taille(n), a_tab(new double[n] , std::default_delete<double[]>

    Vecteur(const Vecteur& v)
        : a_taille(v.a_taille), a_tab(new double[v.a_taille]) {...5 lines

    Vecteur(Vecteur&& v)
        : a_taille(std::move(v.a_taille)), a_tab(std::move(v.a_tab)) {...4 lines }

    ~Vecteur() {...4 lines }

    Vecteur& operator=(const Vecteur& v) {...9 lines }

    Vecteur& operator=(Vecteur&& v) {...8 lines }

    void typeref() & { std::cout << __func__ << "()" & " << std

    void typeref() && {std::cout << __func__ << "()" && " << st

    void typeref() const& {std::cout << __func__ << "()" const & " << std::endl;}

    void typeref() const&& {std::cout << __func__ << "()" const && " << std::endl;}

};
```

```
{
    Vecteur v1(5);
    Vecteur v2(2);
    const Vecteur v3(5);
    std::cout << std::endl;
    v2 = std::move(v1);
    v2.typeref();
    std::move(v2).typeref();
    Vecteur(l).typeref();

    v3.typeref();
    std::move(v3).typeref();
    std::cout << std::endl;
}
```

```
Vecteur(const unsigned) this(0x65f870) tab(0x2997eb0) taille(0x65f870)
Vecteur(const unsigned) this(0x65f868) tab(0x2997ee0) taille(0x65f868)
Vecteur(const unsigned) this(0x65f860) tab(0x2997f58) taille(0x65f860)

operator=(Vecteur&& v) this(0x65f868) tab(0x2997eb0) taille(0x65f868)
typeref() &
typeref() &&
Vecteur(const unsigned) this(0x65f8a8) tab(0x2997ee0) taille(0x65f8a8)
typeref() &&
~Vecteur this(0x65f8a8) tab(0x2997ee0) taille(0x65f8a8)
typeref() const &
typeref() const &&

~Vecteur this(0x65f860) tab(0x2997f58) taille(0x65f860)
~Vecteur this(0x65f868) tab(0x2997eb0) taille(0x65f868)
~Vecteur this(0x65f870) tab(0) taille(0x65f870)
```





C++11 introduit le patron de classe **`std::initializer_list`** qui permet d'initialiser les conteneurs avec la même syntaxe que celle permettant en C d'initialiser les tableaux, donc à l'aide d'une **suite de valeurs entre accolades**.

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

template<class T> void maFunction(initializer_list<T> values)
{
    cout << « Nombre élément : " << values.size() << endl;
    for (auto i = begin(values); i < end(values); ++i)
    {
        cout << *i << " ";
    }
    cout << endl;
}

maFunction<int>({ 1, 2, 3 });

vector<string> a = { "Ignoring", "The", "Voices" };
```



```
class Vecteur {
private:
    unsigned a_taille;
    std::unique_ptr<double[] > a_tab = nullptr;
public:

    Vecteur (const std::initializer_list<double>& v )    : a_taille(v.size()), a_tab( new double[v.size()] ) {
        int j=0;
        for (auto d: v) { a_tab[j]=d;++j;}
        std::cout << __func__ << "(std::initializer_list<double> v ) this(" << this <<
            " ) tab(" << a_tab.get() << " ) taille(" << &a_taille << " )" << std::endl;

        for (unsigned i=0; i < a_taille;++i) { std::cout << a_tab[i] << std::endl; }
    }
}
```

```
std::initializer_list<double> ilist1{ 5, 6, 7 };
std::initializer_list<double> ilist2( ilist1 );

for (auto i : ilist1)
    std::cout << i << std::endl;

std::cout << std::endl;
Vecteur v1 (ilist1);
std::cout << std::endl;
Vecteur v2 {1,2,3};
std::cout << std::endl;
```

```
5
6
7
Vecteur(std::initializer_list<double> v ) this(0x65f828) tab(0x2f77e70) taille(0x65f828)
5
6
7
Vecteur(std::initializer_list<double> v ) this(0x65f820) tab(0x2f77eb0) taille(0x65f820)
1
2
3
~Vecteur this(0x65f820) tab(0x2f77eb0) taille(0x65f820)
~Vecteur this(0x65f828) tab(0x2f77e70) taille(0x65f828)
```

## 2 – Évolution -> Les Objets avec C++11

### -> Constructeur : Constructeur « explicit »



**explicit** empêche le compilateur de convertir une donnée en une autre (constructeur ou opérateur de conversion)

```
class Age
{
    private:
        int theage;
    public :
        Age(int): theage(a){}
        int& operator int() { return this->
theage; }
};
```

void fonction(Age b);

fonction(67) //OK implicite

Age a(32) ;

Int i=a; // OK

```
class Age
{
    private:
        int theage;
    public :
        explicit Age(int): theage(a){}
        explicit int& operator int() { return
this-> theage; }
};
```

void fonction(Age b);

**fonction(67) //KO**

fonction(Age(67));

Age a(32) ;

**Int i=a; // KO**

Int i = (int)a;



```
ClassName() = default;  
const ClassName& operator=(const ClassName&) =  
default;
```

l'utilisateur peut toujours forcer la génération du constructeur implicitement déclarée avec le mot-clé default

```
ClassName() = delete;  
const ClassName& operator=(const ClassName&) =  
delete;
```

l'utilisateur peut empêcher la génération du constructeur déclarée avec le mot-clé delete



```
class Ex{
public:
    Ex() = default;
    Ex(const Ex&) = delete;
    Ex(Ex&&) = delete;

    Ex& operator=(const Ex&) = default;
};

class SpecialiseEx : public Ex{
public:
    SpecialiseEx() = delete;
    SpecialiseEx(const SpecialiseEx&) = default;
    SpecialiseEx(SpecialiseEx&&)= default;

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

## 2 – Évolution -> Les Objets avec C++11

### -> Interdiction de surcharge : final



C + 11 ajoute également la capacité d'empêcher d'hériter de classes ou simplement prévenir la surcharge de méthodes dans les classes dérivées. Cela se fait avec l'identificateur spécial **final** .

```
struct Base1 final { };
```

```
struct Derived1 : Base1 { }; // Erreur car la classe Base1 est final
```

```
struct Base2 {  
    virtual void f() final;  
};
```

```
struct Derived2 : Base2 {  
    void f(); // Erreur car la fonction virtual Base2::f est final  
};
```

## 2 – Évolution -> Les Objets avec C++11

### -> Obligation de surcharge : **override**



L'identificateur spécial **override** signifie que le compilateur vérifie la classe de base (es) pour voir s'il y a une fonction virtuelle avec cette signature exacte. Et s'il n'y en a pas, le compilateur indiquera une erreur.

```
struct Base {  
    virtual void some_func(float);  
};
```

```
struct Derived : Base {  
    virtual void some_func(int) override; // Erreur car la fonction  
some_func(int) n'est pas déclarée dans Base,  
};
```



Les expressions constantes permettent de spécifier avec le mot clé **constexpr** qu'une expression, une fonction ou un constructeur pourra être évalué lors de la compilation.

**Constexpr**, comme le mot clé **const**, peut être appliqué aux variables pour spécifier que leur valeur n'est pas modifiable.



## 2 – Évolution -> Les Objets avec C++11

### -> Le mot clé : constexpr : règles générales



Une **fonction** ou un **constructeur constexpr** est implicitement **inline**..

Une fonction **constexpr** :

- Peut être récursive, mais pas virtuelle.
- Sa valeur de retour et ses paramètres doivent tous être des types de littéral.
- Le corps de la fonction peut être défini avec la valeur = default ou = delete.
- La fonction ne doit pas contenir d'instructions :
  - goto,
  - blocs try,
  - de variables non initialisées,
  - de définitions de variables qui ne sont pas des types de littéral, ou qui sont statiques ou de thread local.
  - constructeur ne peut pas être défini en tant que constexpr si la classe englobante contient des classes de base virtuelles.

Une variable peut être déclarée avec constexpr si elle a un type de littéral et est initialisée.

Toutes les déclarations d'une variable ou d'une fonction constexpr doivent comporter le spécificateur constexpr.

Si l'initialisation est effectuée par un constructeur, celui-ci doit également être déclaré avec constexpr.

Une référence peut être déclarée avec constexpr si l'objet référencé a été initialisé par une expression constante et que toutes les conversions implicites appelées pendant l'initialisation sont aussi des expressions constantes.

Une spécialisation explicite d'un modèle :

- non constexpr peut être déclarée avec constexpr:
- Une spécialisation explicite d'un modèle constexpr ne doit pas obligatoirement être déclarée avec constexpr:

## 2 – Évolution -> Les Objets avec C++11 -> Le mot clé : constexpr ( C++ 17)



Applicable avec les lambada fonction,

```
constexpr int Increment(int n)
{
    return [n] { return n + 1; }();
}

void test_lambad_constexpr() {
    int y = 32;
    auto answer = [y]() constexpr
    {
        int x = 10;
        return y + x;
    };

    std::cout << answer() << std::endl;
    std::cout << Increment(y) << std::endl;
}
```

42  
33

## 2 – Évolution -> Les Objets avec C++11

### -> Le mot clé : constexpr : Exemple



```
constexpr int fac(int n)
{
    return n == 1 ? 1 : n*fac(n - 1);
}
```

```
class Carre {
public:
    constexpr Carre(double d):a_cote(d) {}

    void afficher() const {
        std::cout << "Carre de cote : " << this->a_cote << std::endl;
        std::cout << "perimetre      : " << this->perimetre() << std::endl;
    }

    constexpr double perimetre() const { return this->a_cote * 2;}
private:
    double a_cote = 0.0;
};
```

```
constexpr Carre a2(5.5);
a2.afficher();

auto start = std::chrono::steady_clock::now();
constexpr auto x = fac(12);
auto stop = std::chrono::steady_clock::now();
auto dur = std::chrono::duration_cast<std::chrono::milliseconds>(stop - start);
std::cout << "fac(12)=" << x << " a durer " << static_cast<double>(dur.count()) << " milli seconds!" << std::endl;
```

```
Carre de cote : 5.5
perimetre      : 11
fac(12)=479001600 a durer 0 milli seconds!
```

## 2 – Évolution -> Les Objets avec C++11

### -> Littérales définies par l'utilisateur : operator " "



```
ReturnType operator "" _a(unsigned long long int); // Literal operator for user-defined INTEGRAL literal
ReturnType operator "" _b(long double);           // Literal operator for user-defined FLOATING literal
ReturnType operator "" _c(char);                   // Literal operator for user-defined CHARACTER literal
ReturnType operator "" _d(wchar_t);                // Literal operator for user-defined CHARACTER literal
ReturnType operator "" _e(char16_t);               // Literal operator for user-defined CHARACTER literal
ReturnType operator "" _f(char32_t);               // Literal operator for user-defined CHARACTER literal
ReturnType operator "" _g(const char*, size_t);    // Literal operator for user-defined STRING literal
ReturnType operator "" _h(const wchar_t*, size_t); // Literal operator for user-defined STRING literal
ReturnType operator "" _i(const char16_t*, size_t); // Literal operator for user-defined STRING literal
ReturnType operator "" _g(const char32_t*, size_t); // Literal operator for user-defined STRING literal
ReturnType operator "" _r(const char*);            // Raw literal operator
template<char...> ReturnType operator "" _t();      // Literal operator template
```

<https://docs.microsoft.com/fr-fr/cpp/cpp/user-defined-literals-cpp>

```
e operator"" _km( long double val)
{
    return val/1000.;
}

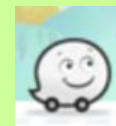
e operator"" _mm( long double val)
{
    return val*1000.;
}

e operator"" _m( long double val)
{
    return val;
}
```

```
std::cout << 1._m << std::endl;
std::cout << 1._km << std::endl;
std::cout << 1._mm << std::endl;
```

```
1
0.001
1000
```

## 2 – Évolution -> Les Objets avec C++11 à C++17 -> les attributs



Les attributs, introduits dans le langage avec la norme **C++11**, constituent un moyen d'annoter certains éléments du langage.

Les attributs peuvent recevoir des paramètres,  
**C++11** a ajouté 2 attributs :

**[[noreturn]]** Spécifie qu'une fonction ne retourne jamais de valeur

**[[carries\_dependency]]** Spécifie que la fonction propage les dépendances de données classement en ce qui concerne la synchronisation de thread, Dans un programme utilisant des variables atomiques avec modèle `memory_order_consume`, où les transformations permises par les optimiseurs reposent sur sa compréhension des dépendances entre les données (*Data Dependency*), passer une adresse en paramètre à une fonction peut en faire perdre la trace par l'optimiseur, réduisant la qualité du code généré. L'annotation d'un tel paramètre par **[[carries\_dependency]]** indique au compilateur qu'il vaut la peine de suivre la trace de ce paramètre de plus près.

**C++14** a ajouté 1 attribut

**[[deprecated]]** : permet de signaler un élément déprécié depuis et autorise l'ajout d'un message en argument

**C++17** a ajouté 3 attributs

**[[fallthrough]]** : permet aux instructions switch. L'objectif est de préciser au compilateur qu'une absence de saut du flot de contrôle est volontaire (que ce soit un `break` ou un `return`).

**[[maybe\_unused]]** : permet de signaler au compilateur qu'une variable peut être inutilisée et qu'il n'y a pas lieu de s'inquiéter.

**[[nodiscard]]** : permet de refuser le droit du programmeur d'ignorer le retour d'une fonction

## 2 – Évolution -> Les Objets avec C++11 à C++17 -> les attributs : Exemple



```
[[deprecated]]
void test_deprecated(int ) {}

void test_unused() {
    [[maybe_unused]]
    int i;
}

[[nodiscard]]
int test_nodiscard(int i) { return i * i; }

//[[carries_dependency]]

int test_fallthrough(int i) {
    int b = 0;
    switch (i) {
        case 0: case 1:
            [[fallthrough]]; // fallthrough explicite, pas de warning.
        case 2:
            return 0;
        case 3:
            b = 45;
            // warning possible du compilateur.
        default:
            std::cout << "default"<< std::endl;
    };
    return 0;
}

int exemple_attribut()
{
    test_deprecated(42); // warning: 'void test_deprecated(int)' is deprecated [-Wdeprecated-declarations]
    test_nodiscard(42); // warning: ignoring return value of 'int test_nodiscard(int)', declared with attribute nodiscard [-Wunused-result]
    test_unused();

    return 0;
}
```

## 2 – Évolution -> Les Objets avec C++11

### -> Espaces de noms : inline namespace



**inline namespace ns\_nom { déclarations }**

Les définitions apparaîtront à la fois à l'intérieur de **ns\_nom** et dans l'espace de noms l'incluant.

```
namespace N1 {  
    inline namespace N1_1 {  
        class A {};  
    }  
}
```

```
N1::A a1;  
N1::N1_1::A a2;
```

## 2 – Évolution -> Les Objets avec C++17 -> espaces de noms imbriqués



```
namespace <nom1>::<nom2>::... :: <nomN> {  
    <déclaration>;  
}
```

```
namespace A {  
    namespace B {  
        namespace C {  
            int f();  
        }  
    }  
}
```

```
namespace A::B::C {  
    int f();  
}
```



## 2 – Évolution -> Les Objets avec C++11

### -> Le mot clé : noexcept



**noexcept(expression)**  
**noexcept**

**noexcept** et son synonyme **noexcept(true)** spécifient que la fonction ne lèvera jamais d'exception

**noexcept** est une version améliorée de **throw ()** , qui est « deprecated » en C++11

```
template < class T > void foo ( ) noexcept ( noexcept ( T ( ) ) ) { }  
void bar ( ) noexcept ( true ) { }  
void baz ( ) noexcept { throw 42 ; } // noexcept equivalent à noexcept(true)  
  
int main ( )  
{  
    foo < int > ( ) ; // noexcept(noexcept(int())) => noexcept(true), so this is fine  
  
    bar ( ) ; // fine  
    baz ( ) ; // compiles, but at runtime this calls std::terminate  
}
```

## 2 – Évolution -> Les Objets avec C++11

### -> Le mot clé : noexcept



**noexcept** : Spécifie si une fonction peut lever des exceptions.

```
int division ( int i, int j) {
    if ( j == 0) throw std::runtime_error("division par 0 : division");
    return i/j;
}

int division_noexcept_false ( int i, int j) noexcept(false) {
    if ( j == 0) throw std::runtime_error("division par 0 : division_noexcept_false");
    return i/j;
}

int division_noexcept_test_false ( int i, int j) noexcept(noexcept(division(i,j))) {
    return division(i,j);
}
```

```
division par 0 : division
division par 0 : division_noexcept_false
division par 0 : division
```

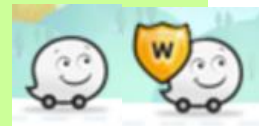
```
try {
    division (0,0);
} catch ( std::exception &e ){
    std::cerr << e.what() << std::endl;
};

try {
    division_noexcept_false (0,0);
} catch ( std::exception &e ){
    std::cerr << e.what() << std::endl;
};

try {
    division_noexcept_test_false (0,0);
} catch ( std::exception &e ){
    std::cerr << e.what() << std::endl;
};
```

## 2 – Évolution -> Les Objets avec C++11

### -> Le mot clé : noexcept



```
int division_noexcept ( int i, int j) noexcept {
    if ( j == 0) throw std::runtime_error("division par 0 : division_noexcept");
    return i/j;
}

int division_noexcept_true ( int i, int j) noexcept(true) {
    if ( j == 0) throw std::runtime_error("division par 0 : division_noexcept_true");
    return i/j;
}

int division_noexcept_test_true ( int i, int j) noexcept( ! noexcept(division(i,j))) {
    return division(i,j);
}
```

```
try {
    division_noexcept(0,0);
} catch ( std::exception &e){
    std::cerr << e.what() << std::endl;
};
```

```
terminate called after throwing an instance of 'std::runtime_error'
what():  division par 0 : division_noexcept
```

```
try {
    division_noexcept_true(0,0);
} catch ( std::exception &e){
    std::cerr << e.what() << std::endl;
};
```

```
terminate called after throwing an instance of 'std::runtime_error'
what():  division par 0 : division_noexcept_true
```

```
try {
    division_noexcept_test_true (0,0);
} catch ( std::exception &e ){
    std::cerr << e.what() << std::endl;
};
```

```
terminate called after throwing an instance of 'std::runtime_error'
what():  division par 0 : division
```

## 2 – Évolution -> Les Objets avec C++11 -> Déclaration étendue friend



```
class Banque {};  
  
class Client {  
    friend Banque; // OK : la classe Banque est amie de Client  
};  
  
using Employer = Client;  
  
class Societe {  
    friend Banque; // OK : la classe Banque est amie de Societe  
    friend class Impots; // OK : on déclare une nouvelle classe qui sera amie  
};  
  
template <typename T>  
class All {  
    friend T;  
};  
  
All<Client> ac; // OK : la classe Banque est amie de la classe All<Client>  
All<int> ai; // OK : l'amitié est ignorée pour les types de base
```



```
static_assert( constant-expression, string-literal );  
static_assert( constant-expression );
```

La déclaration **static\_assert** teste une assertion logicielle au moment de la compilation.

Si l'expression constante spécifiée a la valeur `FALSE`, le compilateur affiche le message spécifié, s'il est fourni, et la compilation échoue

## 2 – Évolution -> Les Objets avec C++ 17

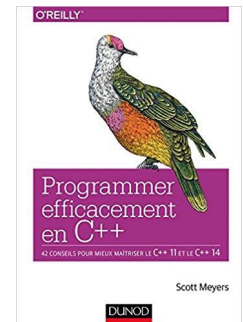
### -> static\_assert



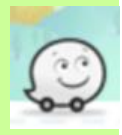
```
namespace Exemples {  
  
template<typename T>  
class A {  
    static_assert(std::is_default_constructible_v<T>);  
};  
  
class X {  
public:  
    X() = delete;  
    X(int) {}  
};  
  
int exemple_static_assert()  
{  
    Exemples::A<Exemples::X> a; // ERROR: static_assert failed  
    return 0;  
}
```



- Conseil n° 11. Préférer les fonctions supprimées aux fonctions indéfinies privées
- Conseil n° 12. Déclarer les fonctions de substitution avec override
- Conseil n° 14. Déclarer noexcept les fonctions qui ne lancent pas d'exceptions
- Conseil n° 15. Utiliser constexpr dès que possible
- Conseil n° 17. Comprendre la génération d'une fonction membre spéciale
- Conseil n° 33. Utiliser decltype sur des paramètres auto&& pour les passer à std::forward



## 2 – Évolution -> Les Objets avec C++11 -> Bibliothèque <chrono>



<chrono>	
<b>classes:</b>	
common_type (duration)	C++11
common_type (time_point)	C++11
duration	C++11
duration_values	C++11
high_resolution_clock	C++11
steady_clock	C++11
system_clock	C++11
time_point	C++11
treat_as_floating_point	C++11
<b>functions:</b>	
duration_cast	C++11
time_point_cast	C++11
<b>class typedefs:</b>	
hours	C++11
microseconds	C++11
milliseconds	C++11
minutes	C++11
nanoseconds	C++11
seconds	C++11

La bibliothèque chrono définit trois types principaux (durées, horloges et points temporels) ainsi que des fonctions utilitaires et des typedefs communs



## 2 – Évolution -> Les Objets avec C++11 -> Bibliothèque <chrono>



```
using seconds_type = std::chrono::duration<int> ;
using milliseconds_type = std::chrono::duration<int,std::milli>
using hours_type = std::chrono::duration<int,std::ratio<60*60>>;

hours_type h_unjour (24);                // 24h
seconds_type s_unjour (60*60*24);        // 86400s
milliseconds_type ms_unjour (s_unjour);  // 86400000ms

std::cout << ms_unjour.count() << "ms\t dans 1 jour" << std::endl;
std::cout << h_unjour.count() << "h \t dans 1 jour" << std::endl;
std::cout << s_unjour.count() << "s \t dans 1 jour" << std::endl;

std::cout << std::endl;

seconds_type s_uneheure (60*60);         // 3600s
hours_type h_uneheure (std::chrono::duration_cast<hours_type>(s_uneheure));
milliseconds_type ms_uneheure (s_uneheure); // 3600000ms

std::cout << ms_uneheure.count() << "ms\t dans 1 heure" << std::endl;
std::cout << h_uneheure.count() << "h \t dans 1 heure" << std::endl;
std::cout << s_uneheure.count() << "s \t dans 1 heure" << std::endl;

std::cout << std::endl;
```

```
86400000ms    dans 1 jour
24h           dans 1 jour
86400s        dans 1 jour

3600000ms     dans 1 heure
1h            dans 1 heure
3600s         dans 1 heure
```

## 2 – Évolution -> Les Objets avec C++11 -> Bibliothèque <chrono>



```
std::cout << "system_clock durer possible :\n";  
std::cout << "min: " << std::chrono::system_clock::duration::min().count() << "\n";  
std::cout << "max: " << std::chrono::system_clock::duration::max().count() << "\n";  
std::cout << "zero: " << std::chrono::system_clock::duration::zero().count() << "\n";
```

```
system_clock durer possible :  
min: -9223372036854775808  
max: 9223372036854775807  
zero: 0
```

## 2 – Évolution -> Les Objets avec C++11



```
*****  
*****  
*****  
*****  
*****  
*****  
*****  
*****  
*****  
*****  
Horloge interns 46873000ns 0s 46873000ns 46ms .  
*****
```

```
using std::chrono::steady_clock;
steady_clock::time_point debut = steady_clock::now();

for (int i=0; i<1000; ++i) std::cout << "*";
std::cout << std::endl;

steady_clock::time_point fin = steady_clock::now();

steady_clock::duration d = fin - debut;
steady_clock::duration d0 = std::chrono::duration_cast<std::chrono::seconds>(d);
steady_clock::duration d1 = std::chrono::duration_cast<std::chrono::nanoseconds>(d);
milliseconds_type d2 = std::chrono::duration_cast<std::chrono::milliseconds>(fin - debut);

std::cout << "Horloge interns " << d.count() << "ns " << d0.count() << "s " << d1.count() << "ns " << d2.count() << "ms .\n";
```

## 2 – Évolution -> Les Objets avec C++11 -> Bibliothèque <chrono>



```
using std::chrono::system_clock;

std::chrono::duration<int, std::ratio<60*60*24> > one_day (1);

system_clock::time_point today= system_clock::now();
system_clock::time_point tomorrow = today + one_day;

std::time_t tt;
tt = system_clock::to_time_t ( today );
std::cout << "La date est " << ctime(&tt)  ;
tt = system_clock::to_time_t ( tomorrow );
std::cout << "La date de demain est " << ctime(&tt) << std::endl ;
```

```
La date est Sat Jan 27 10:26:59 2018
La date de demain est  Sun Jan 28 10:26:59 2018
```



```
template<typename... Arguments> class VariadicTemplate;
```

```
VariadicTemplate<double, float> instance;
```

```
VariadicTemplate<bool, unsigned short int, long> instance;
```

```
VariadicTemplate<char, std::vector<int>, std::string, std::string,  
std::vector<long long>> instance;
```

```
template<typename... Arguments> void  
SampleFunction(Arguments... parameters)
```

<https://www.supinfo.com/articles/single/3017-introduction-aux-variadic-templates-c>

## 2 – Évolution -> Les Objets avec C++11

### -> Paramètres template étendus pour les template variadic



```
void ft()
{
    std::cout << "-----" << std::endl;
}
```

```
template<typename T, typename... Args>
void ft(T value, Args... args){
    std::cout << "valeur : " << value << std::endl;
    std::cout << " -> nb argument restant: " << sizeof...(Args) << std::endl;
    ft(args...);
}
```

```
valeur : 1
-> nb argument restant: 5
valeur : 2
-> nb argument restant: 4
valeur : 3
-> nb argument restant: 3
valeur : 4
-> nb argument restant: 2
valeur : 5
-> nb argument restant: 1
valeur : 6
-> nb argument restant: 0
-----
```

```
ft(1, 2, 3, 4, 5, 6);
```

## 2 – Évolution -> Les Objets avec C++11

### -> Paramètres template étendus pour les template variadic



```
template<class... Args> class ClassT {
public:
    ClassT() {std::cout << __func__ << "()" << std::endl; }
};

template<class T, class... Args> class ClassT <T , Args...> : ClassT <Args...> {
public:
    ClassT()
        :ClassT < Args...>(),t_valeur() {
            std::cout << __func__ << "()" -> nb argument restant: " << sizeof...(Args) << std::endl;
        }
    ClassT(T t, Args... ts)
        :ClassT < Args...>(ts...),t_valeur(t) {
            std::cout << __func__ << "(...) -> nb argument restant: " << sizeof...(Args) << std::endl;
        }
protected:
    T t_valeur;
};
```

```
ClassT()
ClassT() -> nb argument restant: 0
ClassT()
ClassT(...) -> nb argument restant: 0
ClassT(...) -> nb argument restant: 1
ClassT(...) -> nb argument restant: 2
```

```
ClassT<char> ct1;
ClassT<int,float,std::string> ct2(1,2.0f,std::string("valeur"));
```

## 2 – Évolution -> Les Objets avec C++11 -> extern template class



```
extern template MyStack<int, 6>::MyStack( void );
```

Les templates ne sont actuellement pas pris en compte par l'éditeur de liens avant C++11 : il est nécessaire d'incorporer leur définition dans tous les fichiers sources les utilisant en programmation modulaire. => compilation longue et gourmande puisque la classe était recompilée dans chaque fichier source, pour chaque type utilisé.

C++11 permettra l'utilisation du mot-clé **extern** pour rendre les templates globaux. Les fichiers désirant utiliser le template n'ont qu'à le déclarer.

le mot clé **extern** empêche le compilateur de générer le même code d'instanciation dans plus d'un module objet.

Il faut instancier la fonction de modèle en utilisant les paramètres de modèle explicites spécifiés dans au moins un module lié.





`static_assert` prend en 1er argument une condition, et en second le message d'erreur à afficher si la condition n'est pas respectée.

```
template <typename T, unsigned int taille>
class Tableau{
    public: Tableau(T value = T())
    {
        static_assert(taille >= 0, "Erreur ; taille du tableau inferieur à 0");
    }
};
```

## 2 – Évolution -> std::bind



std::bind permet de créer une fonction à partir d'une autre fonction et d'une liste d'arguments (valeurs, placeholder).

```
#include <iostream>
#include <functional>

int mult(int lhs, int rhs) { return lhs * rhs; }

int main() {
    using namespace std::placeholders; // pour _1
    auto mult_par_2 = std::bind(&mult, _1, 2);
    std::cout << mult_par_2(321) << std::endl;
    return pause();
}
```

642







- Pointeurs intelligents
- Sémantique du déplacement
- Gestion de la concurrence : thread
- Librairie C
- Conteneurs
- Divers

## 3 – Extensions de la bibliothèque standard -> Pointeurs intelligents



- `std::shared_ptr<>` : pointeur intelligent avec sémantique de propriétaires partagés
- `std::weak_ptr<>` : référence faible à un objet géré par `std::shared_ptr`
- `std::unique_ptr<>` : pointeur intelligent avec sémantique de propriétaire unique



`std::shared_ptr` est un pointeur intelligent qui permet le partage d'un objet via un pointeur.

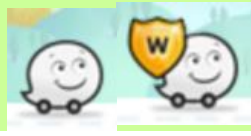
Plusieurs instances de `shared_ptr` peuvent posséder le même objet, et cet objet est détruit dans l'un des cas suivant :

- le dernier `shared_ptr` possédant l'objet est détruit
- le dernier `shared_ptr` possédant l'objet est assigné à un autre pointeur via `operator=()` or `reset()`.

<b>shared_ptr</b>	
<code>shared_ptr::shared_ptr</code>	C++11
<code>shared_ptr::~~shared_ptr</code>	C++11
<b>member functions:</b>	
<code>shared_ptr::get</code>	C++11
<code>shared_ptr::operator bool</code>	C++11
<code>shared_ptr::operator*</code>	C++11
<code>shared_ptr::operator-&gt;</code>	C++11
<code>shared_ptr::operator=</code>	C++11
<code>shared_ptr::owner_before</code>	C++11
<code>shared_ptr::reset</code>	C++11
<code>shared_ptr::swap</code>	C++11
<code>shared_ptr::unique</code>	C++11
<code>shared_ptr::use_count</code>	C++11
<b>non-member overloads:</b>	
<code>operator&lt;&lt; (shared_ptr)</code>	C++11
<code>relational operators (shared_ptr)</code>	C++11
<code>swap (shared_ptr)</code>	C++11

# 3 – Extensions de la bibliothèque standard

## -> Pointeurs intelligents : `std::shared_ptr<>`



```

1 // shared_ptr constructor example
2 #include <iostream>
3 #include <memory>
4
5 struct C {int* data;};
6
7 int main () {
8     std::shared_ptr<int> p1;
9     std::shared_ptr<int> p2 (nullptr);
10    std::shared_ptr<int> p3 (new int);
11    std::shared_ptr<int> p4 (new int, std::default_delete<int>());
12    std::shared_ptr<int> p5 (new int, [](int* p){delete p;}, std::allocator<int>());
13    std::shared_ptr<int> p6 (p5);
14    std::shared_ptr<int> p7 (std::move(p6));
15    std::shared_ptr<int> p8 (std::unique_ptr<int>(new int));
16    std::shared_ptr<C> obj (new C);
17    std::shared_ptr<int> p9 (obj, obj->data);
18
19    std::cout << "use_count:\n";
20    std::cout << "p1: " << p1.use_count() << '\n';
21    std::cout << "p2: " << p2.use_count() << '\n';
22    std::cout << "p3: " << p3.use_count() << '\n';
23    std::cout << "p4: " << p4.use_count() << '\n';
24    std::cout << "p5: " << p5.use_count() << '\n';
25    std::cout << "p6: " << p6.use_count() << '\n';
26    std::cout << "p7: " << p7.use_count() << '\n';
27    std::cout << "p8: " << p8.use_count() << '\n';
28    std::cout << "p9: " << p9.use_count() << '\n';
29    return 0;
30 }

```

Output:

```

use_count:
p1: 0
p2: 0
p3: 1
p4: 1
p5: 2
p6: 0
p7: 2
p8: 1
p9: 2

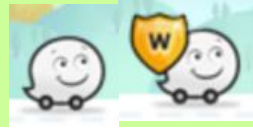
```

Réf : <http://www.cplusplus.com/reference/memory>



# 3 – Extensions de la bibliothèque standard

## -> Pointeurs intelligents : `std::shared_ptr<>`



```
std::shared_ptr<int> sp0(new int(5));
std::shared_ptr<int>::element_type val = *sp0;
std::cout << "*sp0 == " << val << std::endl;

std::shared_ptr<int> sp1(new int(5));

sp0.reset();
std::cout << "sp0.get() == 0 == " << std::boolalpha << (sp0.get() == 0) << std::endl;
std::cout << "*sp1.get() == " << *sp1.get() << std::endl;
std::cout << "(bool)sp0 == " << std::boolalpha << (bool)sp0 << std::endl;
std::cout << "(bool)sp1 == " << std::boolalpha << (bool)sp1 << std::endl;

std::shared_ptr<int> sp2(sp1);
std::shared_ptr<int> sp3;
sp3 = sp2;
std::cout << "sp1.use_count() == " << sp1.use_count() << std::endl;
sp2.reset(new int(5));
std::cout << "sp1.use_count() == " << sp1.use_count() << std::endl;

std::shared_ptr<T1> t1(new T2());
std::shared_ptr<T2> t2 = std::static_pointer_cast<T2>(t1);
std::cout << "t1.use_count() == " << t1.use_count() << std::endl;
```

```
*sp0 == 5
sp0.get() == 0 == true
*sp1.get() == 5
(bool)sp0 == false
(bool)sp1 == true
sp1.use_count() == 3
sp1.use_count() == 2
t1.use_count() == 2
```

```
class T1 {};
class T2: public T1 {};
```

## 3 – Extensions de la bibliothèque standard

### -> Pointeurs intelligents : `std::weak_ptr<>`



`weak_ptr` fournit l'accès à un objet appartenant à une ou plusieurs instances de `shared_ptr` (travailler en collaboration), mais ne participe pas au décompte de références (aucune responsabilité associée).

weak_ptr	
<code>weak_ptr::weak_ptr</code>	<code>C++11</code>
<code>weak_ptr::~weak_ptr</code>	<code>C++11</code>
[-] <b>member functions:</b>	
... <code>weak_ptr::expired</code>	<code>C++11</code>
... <code>weak_ptr::lock</code>	<code>C++11</code>
... <code>weak_ptr::operator =</code>	<code>C++11</code>
... <code>weak_ptr::owner_before</code>	<code>C++11</code>
... <code>weak_ptr::reset</code>	<code>C++11</code>
... <code>weak_ptr::swap</code>	<code>C++11</code>
... <code>weak_ptr::use_count</code>	<code>C++11</code>
[-] <b>non-member overloads:</b>	
... <code>swap (weak_ptr)</code>	<code>C++11</code>

## 3 – Extensions de la bibliothèque standard

### -> Pointeurs intelligents : `std::weak_ptr<>`



```
1 // weak_ptr constructor example
2 #include <iostream>
3 #include <memory>
4
5 struct C {int* data;};
6
7 int main () {
8     std::shared_ptr<int> sp (new int);
9
10    std::weak_ptr<int> wp1;
11    std::weak_ptr<int> wp2 (wp1);
12    std::weak_ptr<int> wp3 (sp);
13
14    std::cout << "use_count:\n";
15    std::cout << "wp1: " << wp1.use_count() << '\n';
16    std::cout << "wp2: " << wp2.use_count() << '\n';
17    std::cout << "wp3: " << wp3.use_count() << '\n';
18    return 0;
19 }
```

Output:

```
use_count:
wp1: 0
wp2: 0
wp3: 1
```

Réf : <http://www.cplusplus.com/reference/memory>

# 3 – Extensions de la bibliothèque standard

## -> Pointeurs intelligents : `std::weak_ptr<>`



```
std::shared_ptr<int> sp0(new int(5));
std::weak_ptr<int> wp0(sp0);
std::weak_ptr<int>::element_type val = *wp0.lock();
std::cout << "wp0.lock() == " << val << std::endl;
```

```
std::cout << std::endl;
```

```
std::weak_ptr<int> wp;
std::shared_ptr<int> sp(new int(10));
wp = sp;
```

```
std::cout << "wp.expired() == " << std::boolalpha << wp.expired() << std::endl;
```

```
std::cout << "wp.lock() == " << *wp.lock() << std::endl;
```

```
std::cout << "(bool)wp.lock() == " << std::boolalpha << (bool)wp.lock() << std::endl;
```

```
sp.reset();
```

```
std::cout << "wp.expired() == " << std::boolalpha << wp.expired() << std::endl;
```

```
std::cout << "(bool)wp.lock() == " << std::boolalpha << (bool)wp.lock() << std::endl;
```

```
std::cout << std::endl;
```

```
sp.reset(new int(5));
```

```
std::cout << "wp.expired() == " << std::boolalpha << wp.expired() << std::endl;
```

```
wp=sp;
```

```
std::cout << "wp.expired() == " << std::boolalpha << wp.expired() << std::endl;
```

```
std::shared_ptr<int> sp2 = sp;
```

```
sp.reset();
```

```
std::cout << "wp.expired() == " << std::boolalpha << wp.expired() << std::endl;
```

```
*wp0.lock() == 5
```

```
wp.expired() == false
```

```
*wp.lock() == 10
```

```
(bool)wp.lock() == true
```

```
wp.expired() == true
```

```
(bool)wp.lock() == false
```

```
wp.expired() == true
```

```
wp.expired() == false
```

```
wp.expired() == false
```

## 3 – Extensions de la bibliothèque standard

### -> Pointeurs intelligents : `std::unique_ptr<>`



`std::unique_ptr` est un pointeur intelligent qui :

- garantie la propriété unique d'un objet à travers un pointeur,
- détruit l'objet pointé lorsque le `unique_ptr` devient inaccessible.

Remarque : Remplace `auto_ptr`, qui est déconseillée en C++11 et supprimé en C++17

unique_ptr	
<code>unique_ptr::unique_ptr</code>	C++11
<code>unique_ptr::~~unique_ptr</code>	C++11
[-] <b>member functions:</b>	
... <code>unique_ptr::get</code>	C++11
... <code>unique_ptr::get_deleter</code>	C++11
... <code>unique_ptr::operator bool</code>	C++11
... <code>unique_ptr::operator*</code>	C++11
... <code>unique_ptr::operator-&gt;</code>	C++11
... <code>unique_ptr::operator=</code>	C++11
... <code>unique_ptr::operator[]</code>	C++11
... <code>unique_ptr::release</code>	C++11
... <code>unique_ptr::reset</code>	C++11
... <code>unique_ptr::swap</code>	C++11
[-] <b>non-member overloads:</b>	
... relational operators ( <code>unique_ptr</code> )	C++11
... swap ( <code>unique_ptr</code> )	C++11

## 3 – Extensions de la bibliothèque standard

### -> Pointeurs intelligents : `std::unique_ptr<>`



```
1 // unique_ptr constructor example
2 #include <iostream>
3 #include <memory>
4
5 int main () {
6     std::default_delete<int> d;
7     std::unique_ptr<int> u1;
8     std::unique_ptr<int> u2 (nullptr);
9     std::unique_ptr<int> u3 (new int);
10    std::unique_ptr<int> u4 (new int, d);
11    std::unique_ptr<int> u5 (new int, std::default_delete<int>());
12    std::unique_ptr<int> u6 (std::move(u5));
13    std::unique_ptr<void> u7 (std::move(u6));
14    std::unique_ptr<int> u8 (std::auto_ptr<int>(new int));
15
16    std::cout << "u1: " << (u1?"not null":"null") << '\n';
17    std::cout << "u2: " << (u2?"not null":"null") << '\n';
18    std::cout << "u3: " << (u3?"not null":"null") << '\n';
19    std::cout << "u4: " << (u4?"not null":"null") << '\n';
20    std::cout << "u5: " << (u5?"not null":"null") << '\n';
21    std::cout << "u6: " << (u6?"not null":"null") << '\n';
22    std::cout << "u7: " << (u7?"not null":"null") << '\n';
23    std::cout << "u8: " << (u8?"not null":"null") << '\n';
24
25    return 0;
26 }
```

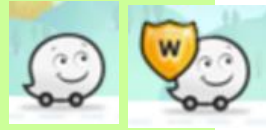
Output:

```
u1: null
u2: null
u3: not null
u4: not null
u5: null
u6: null
u7: not null
u8: not null
```

Réf : <http://www.cplusplus.com/reference/memory>

### 3 – Extensions de la bibliothèque standard

-> Pointeurs intelligents : `std::make_shared<>`

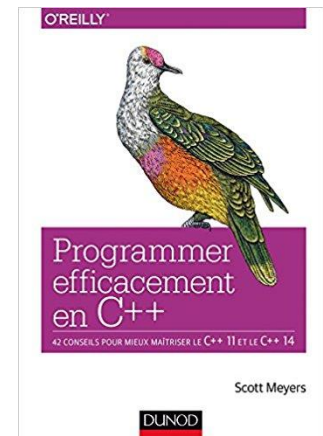


**`std::make_shared`** permet de créer et rendre un pointeur **`shared_ptr`** qui pointe vers les objets alloués qui sont construits de zéro ou de plusieurs arguments à l'aide de l'allocateur par défaut.

```
auto sp = std::shared_ptr<Example>(new Example(argument));  
auto msp = std::make_shared<Example>(argument);
```



- Conseil n° 18. Utiliser `std::unique_ptr` pour la gestion d'une ressource à propriété exclusive
- Conseil n° 19. Utiliser `std::shared_ptr` pour la gestion d'une ressource à propriété partagée
- Conseil n° 20. Utiliser `std::weak_ptr` pour des pointeurs de type `std::shared_ptr` qui peuvent pendouiller
- Conseil n° 21. Préférer `std::make_unique` et `std::make_shared` à une utilisation directe de `new`





## 3 – Extensions de la bibliothèque standard

### -> Sémantique déplacement : `std::move`



`std::move` permet d'obtenir une référence rvalue à son argument.  
la ressource de l'argument est exécutée plus rapidement, laissant l'argument avec une valeur vide

```
// move example
#include <utility>      // std::move
#include <iostream>     // std::cout
#include <vector>       // std::vector
#include <string>       // std::string

int main () {
    std::string la = "La";
    std::string valeur = "valeur";
    std::vector<std::string> myvector;

    myvector.push_back (la);                // copies
    myvector.push_back (std::move(valeur)); // moves

    std::cout << "myvector contient :";
    for (std::string& x:myvector) std::cout << ' ' << x;
    std::cout << std::endl;
    std::cout << "la -> " << la << std::endl;
    std::cout << "valeur -> " << valeur << std::endl;

    return 0;
}
```

```
myvector contient : La valeur
la -> La
valeur ->
```

## 3 – Extensions de la bibliothèque standard

### -> Sémantique déplacement : `std::forward`



**`std::forward`** convertit de manière **conditionnelle** son argument en une référence **rvalue**, si l'argument est une **rvalue** ou **lvalue**.

```
#include <utility>      // std::forward
#include <iostream>      // std::cout

// fonction avec lvalue et une rvalue reference overloads:
void lire (const int& x) {std::cout << "[lvalue]";}
void lire (int&& x) {std::cout << "[rvalue]";}

// function template applique une rvalue reference to deduced type:
template <class T> void fn (T&& x) {
    lire (x);                // toujours une lvalue
    lire (std::forward<T>(x)); // rvalue
}

int main () {
    int a;
    std::cout << "appelle de la fonction : fn<> avec une lvalue: ";
    fn (a);
    std::cout << std::endl;

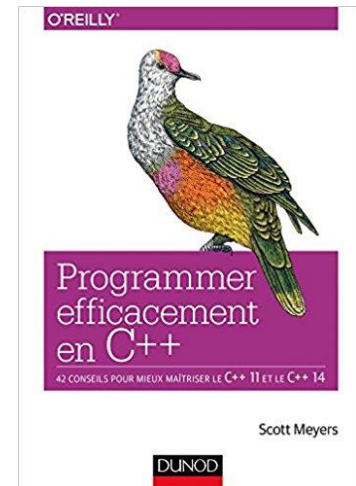
    std::cout << "appelle de la fonction : fn<> avec une rvalue: ";
    fn (0);
    std::cout << std::endl;

    return 0;
}
```

appelle de la fonction : fn<> avec une lvalue: [lvalue][lvalue]  
appelle de la fonction : fn<> avec une rvalue: [lvalue][rvalue]



- Conseil n° 23. Comprendre `std::move` et `std::forward`
- Conseil n° 25. Utiliser `std::move` sur des références rvalue, `std::forward` sur des références universelles

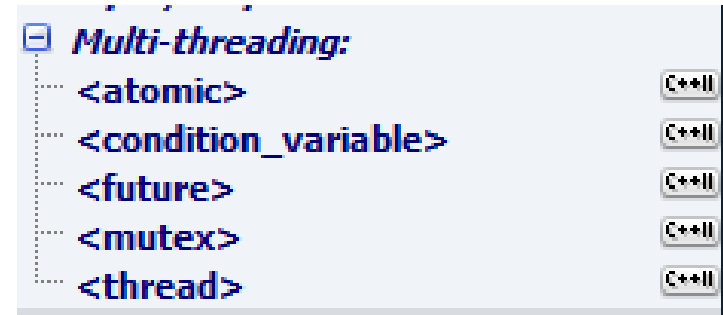


## 3 – Extensions de la bibliothèque standard

### -> Gestion de la concurrence : thread



- Généralités
- Standard C++11 : `std::thread`
- Transmission d'argument
- Mise en sommeil
- Fonction membre : `std::thread`
- Partage de données : `std::mutex`
- Thread asynchrone : `std::async`
- Variable atomique : `std::atomic`
- Capacités d'exécution d'une plateforme - `hardware_concurrency()`





Un **processus** est un programme en cours d'exécution par un ordinateur.

- Un ensemble de ressources personnalisées comme l'espace mémoire, le temps, CPU, ports réseau, etc.
- un espace d'adressage en mémoire vive pour stocker la pile, les données de travail, etc.

Le **multitâche** c'est la possibilité que le système d'exploitation a pour faire fonctionner plusieurs programmes en même temps et cela, sans conflit.

- Un système **monoprocasseur** (et non pas monoprocesseur), simule le multitâche en attribuant un temps de traitement pour chaque processus.
- Un système **multiprocasseur** peut exécuter les différents processus sur les différents processeurs du système.

Un programme est dit **multitâche** s'il est capable de lancer plusieurs parties de son code en même temps, à chaque partie du code sera associé un processus (sous-processus) pour permettre l'exécution en parallèle.

**Remarque** : chaque processus coûte cher au lancement, sans oublier qu'il va avoir son propre espace mémoire, la commutation ainsi que la communication interprocessus seront lourdes à gérer.

### 3 – Extensions de la bibliothèque standard

-> Gestion de la concurrence : thread

-> Généralités -> Thread (processus légers )



Un thread est un sous-ensemble d'un processus, partageant son espace mémoire et ses variables.

#### Avantages :

- les coûts associés suite à son lancement sont réduits, donc plus rapide.
- à chaque thread est associé des unités propres à lui : comme sa pile, le masque de signaux, des données privées, etc.
- les threads peuvent être exécutés en parallèle par un système multitâche.

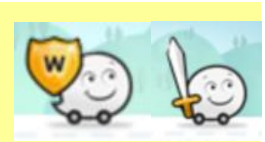
#### Inconvénient :

- Programmation plus difficile : obligation de mettre en place des mécanismes de synchronisation , risques élevés d'interblocage, de famine, d'endormissement

### 3 – Extensions de la bibliothèque standard

-> Gestion de la concurrence : thread

-> Standard C++11 : `std::thread`



Le standard « C++11 » permet une gestion simplifiée des threads à travers un ensemble d'éléments définis dans « `std::thread` ».

```
#include <iostream>
#include <thread>
void fct() {
    static int i= 0;
    i++;
    std::cout << "thread fct[" << i << "]" <<std::endl;
}
```

```
std::thread t1{ fct };
std::thread t2{ fct };

t1.join(); // attente de la fin de la tâche
t2.join();
```

thread	
thread::thread	C++11
thread::~~thread	C++11
member functions:	
thread::detach	C++11
thread::get_id	C++11
thread::join	C++11
thread::joinable	C++11
thread::native_handle	C++11
thread::operator =	C++11
thread::swap	C++11
member types:	
thread::id	C++11
thread::native_handle_type	C++11
static member functions:	
thread::hardware_concurrency	C++11
non-member overloads:	
swap (thread)	C++11

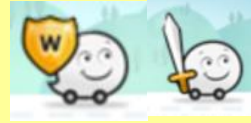
Un résultat possible

thread fct[thread fct[2]1]

### 3 – Extensions de la bibliothèque standard

-> Gestion de la concurrence : thread

-> Transmission d'argument



```
void fct2(std::vector<int>& v) {
    static int i = 0;
    i++;
    std::cout << "thread (" << this_thread::get_id() << ")" << __func__ << "[" << i << "]" << std::endl;
}
```

```
std::vector<int> v(10);
std::thread t3{ std::bind(fct2, v) };
std::thread t4{ std::bind(fct2, v) };
std::thread t5{ fct2, v};
std::thread t6{fct2, v};
t3.join();
t4.join();
t5.join();
t6.join();
```

#### this\_thread

##### **functions within namespace:**

get\_id

C++11

sleep\_for

C++11

sleep\_until

C++11

yield

C++11

Un résultat possible

```
thread (thread (12388thread (6588)fct2[9960)fct2[1]4
]
)fct2[2]
thread (2052)fct2[3]
```



### 3 – Extensions de la bibliothèque standard

-> Gestion de la concurrence :

thread -> Mise en sommeil



```
#include <iostream>
#include <thread>
#include <chrono>

int main()
{
    std::cout << "Chrono de 10 seconde:" << std::endl;;
    for (int i = 10; i>0; --i) {
        std::cout << i << std::endl;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    }
    std::cout << "Chrono finit!" << std::endl;

    return 0;
}
```

```
Chrono de 10 seconde:
10
9
8
7
6
5
4
3
2
1
Chrono finit!
```

### 3 – Extensions de la bibliothèque standard

-> Gestion de la concurrence :

thread -> fonction membre : `std::thread`



```
class A {  
public:  
    void fct(unsigned int val) {  
        static int i = 0;  
        i++;  
        std::cout << "thread (" << this_thread::get_id() << ") fctboucle[" << i << "]" << std::endl;  
    }  
};
```

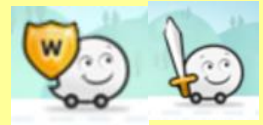
```
A a;  
int p = 2;  
thread t(&A::fct, &a, p);  
t.join();  
thread t1(&A::fct, &a, p);  
t1.join();
```

```
thread (6756) fctboucle[1]  
thread (708) fctboucle[2]
```

### 3 – Extensions de la bibliothèque standard

-> Gestion de la concurrence : thread ->

Partage de données : std::mutex



```
#include <iostream>
#include <thread>
#include <vector>
std::mutex m1;
void fctmutex() {
    static int i = 0;
    i++;
    m1.lock();
    std::cout << "thread (" << this_thread::get_id() << ") fctmutex[" << i << "]" << std::endl;
    m1.unlock();
}
```

```
std::thread t1{ fctmutex };
std::thread t2{ fctmutex };

t1.join(); // attente de la fin de la tâche
t2.join();
```

```
thread (14440) fctmutex[1]
thread (13716) fctmutex[2]
```



`std::async` permet d'exécuter une fonction (une méthode) en arrière-plan à l'aide d'un thread, si possible.

```
#include <future>
#include <iostream>
#include <thread>

void fct_async() {
    static int i = 0;
    ++i;
    std::cout << "thread (" << std::this_thread::get_id() << ") "<< __func__ << " [" << i << "]" << std::endl;
}
```

```
std::future<void> result(std::async(fct_async));
std::future<void> result1(std::async(fct_async));
std::future<void> result2(std::async(fct_async));
std::cout << "debut main" << std::endl;
result.get();
result1.get();
result2.get();
```

```
debut mainthread (thread (8720)fct_async [2]
8728)fct_async [1]
thread (3900)fct_async [3]
```

### 3 – Extensions de la bibliothèque standard

-> Gestion de la concurrence :

thread -> Variable atomic : `std::atomic`



Un mutex permet de verrouiller l'accès à une section critique du code, partagée entre plusieurs threads.

Cette opération peut-être couteuse en ressources et en latence.

Pour éviter le surcoût nécessaire pour cette gestion, le « C++ » offre la possibilité de réaliser **des opérations atomiques**.

Une opération atomique va s'exécuter jusqu'à son terme et ne sera pas interrompue par une autre tâche pendant son exécution.

```
std::atomic<bool> OK(false);  
std::atomic_flag gagnant = ATOMIC_FLAG_INIT;  
std::vector<std::thread> threads;  
int taille = 1000000;
```

lancement de 10 threads qui compte 1000000 element.  
le thread #3 a gagné!

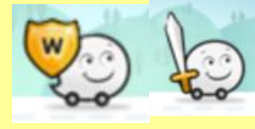
```
auto compteur = [&](int id) {  
    while (!OK) std::this_thread::yield();  
    for (volatile int i = 0; i < taille; ++i) {}  
    if (!gagnant.test_and_set()) std::cout << "le thread #" << id << " a gagn\x82!" << std::endl;  
};
```

```
std::cout << "lancement de 10 threads qui compte " << taille << " element."<< endl;  
for (int i = 1; i <= 10; ++i) threads.push_back(std::thread(compteur, i));  
OK = true;  
for (auto& th : threads) th.join();
```

### 3 – Extensions de la bibliothèque standard

-> Gestion de la concurrence : thread

-> Variable atomic : std::atomic



atomic	
atomic::atomic	C++11
<b>member functions:</b>	
atomic::compare_exchange_strong	C++11
atomic::compare_exchange_weak	C++11
atomic::exchange	C++11
atomic::is_lock_free	C++11
atomic::load	C++11
atomic::operator T	C++11
atomic::operator =	C++11
atomic::store	C++11
<b>member functions (spec.):</b>	
atomic::fetch_add	C++11
atomic::fetch_and	C++11
atomic::fetch_or	C++11
atomic::fetch_sub	C++11
atomic::fetch_xor	C++11
atomic::operator--	C++11
atomic::operator (comp. assign.)	C++11
atomic::operator++	C++11

<atomic>	
<b>classes:</b>	
atomic	C++11
atomic_flag	C++11
<b>enum types:</b>	
memory_order	C++11
<b>functions:</b>	
atomic_signal_fence	C++11
atomic_thread_fence	C++11
kill_dependency	C++11
<b>initialization macros:</b>	
ATOMIC_FLAG_INIT	C++11
ATOMIC_VAR_INIT	C++11
<b>functions (C-style atomics):</b>	
atomic_compare_exchange_strong	C++11
atomic_compare_exchange_strong_explicit	C++11
atomic_compare_exchange_weak	C++11
atomic_compare_exchange_weak_explicit	C++11
atomic_exchange	C++11
atomic_exchange_explicit	C++11
atomic_fetch_add	C++11
atomic_fetch_add_explicit	C++11
atomic_fetch_and	C++11
atomic_fetch_and_explicit	C++11
atomic_fetch_or	C++11
atomic_fetch_or_explicit	C++11
atomic_fetch_sub	C++11
atomic_fetch_sub_explicit	C++11
atomic_fetch_xor	C++11
atomic_fetch_xor_explicit	C++11
atomic_flag_clear	C++11
atomic_flag_clear_explicit	C++11
atomic_flag_test_and_set	C++11
atomic_flag_test_and_set_explicit	C++11
atomic_init	C++11
atomic_is_lock_free	C++11
atomic_load	C++11
atomic_load_explicit	C++11
atomic_store	C++11
atomic_store_explicit	C++11

### 3 – Extensions de la bibliothèque standard

#### -> Gestion de la concurrence :

#### thread -> Variable atomic : std::atomic



```
#include <iostream>      // std::cout
#include <atomic>         // std::atomic
#include <thread>         // std::thread
#include <vector>         // std::vector

std::atomic<int> global_counter(0);

void increase_global(int n) { for (int i = 0; i<n; ++i) ++global_counter; }

void increase_reference(std::atomic<int>& variable, int n) { for (int i = 0; i<n; ++i) ++variable; }
```

```
class C : public std::atomic<int> {
public:
    C() : std::atomic<int>(0) {}
    void increase_member(int n) { for (int i = 0; i<n; ++i) fetch_add(1); }
};
```

```
int main()
{
    std::vector<std::thread> threads;

    std::cout << "increase global counter with 10 threads...\n";
    for (int i = 1; i <= 10; ++i) threads.push_back(std::thread(increase_global, 1000));

    std::cout << "increase counter (foo) with 10 threads using reference...\n";
    std::atomic<int> foo(0);
    for (int i = 1; i <= 10; ++i) threads.push_back(std::thread(increase_reference, std::ref(foo), 1000));

    std::cout << "increase counter (bar) with 10 threads using member...\n";
    C bar;
    for (int i = 1; i <= 10; ++i) threads.push_back(std::thread(&C::increase_member, std::ref(bar), 1000));

    std::cout << "synchronizing all threads...\n";
    for (auto& th : threads) th.join();

    std::cout << "global_counter: " << global_counter << '\n';
    std::cout << "foo: " << foo << '\n';
    std::cout << "bar: " << bar << '\n';
    char c;
    std::cin >> c;
    return 0;
}
```

```
increase global counter with 10 threads...
increase counter (foo) with 10 threads using reference...
increase counter (bar) with 10 threads using member...
synchronizing all threads...
global_counter: 10000
foo: 10000
bar: 10000
```



hardware\_concurrency() : retourne le nombre de threads simultanés pris en charge par la mise en œuvre.

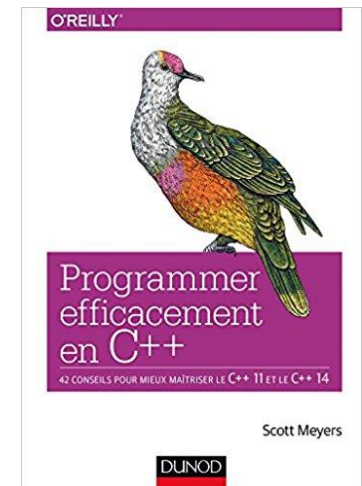
```
#include <iostream>
#include <thread>

int main() {
    unsigned int n = std::thread::hardware_concurrency();
    std::cout << n << " concurrent threads are supported.\n";
    return 0;
}
```





- Conseil n° 16. Rendre les fonctions membres const sûres vis-à-vis des threads
- Conseil n° 35. Préférer la programmation multitâche plutôt que multithread
- Conseil n° 36. Spécifier `std::launch::async` si l'asynchronisme est primordial
- Conseil n° 37. Rendre les `std::thread` non joignables par tous les chemins
- Conseil n° 38. Être conscient du comportement variable du destructeur du descripteur de thread.
- Conseil n° 39. Envisager les futurs `void` pour communiquer ponctuellement un événement
- Conseil n° 40. Utiliser `std::atomic` pour la concurrence, `volatile` pour la mémoire spéciale



## 3 – Extensions de la bibliothèque standard -> Librairie C

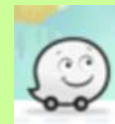


- **`<inttypes>`**
- **`<stdbool>`**
- **`<stdint>`**
- **`<tgmath>`**
- **`<fenv>`**
- **`<cuchar>`**

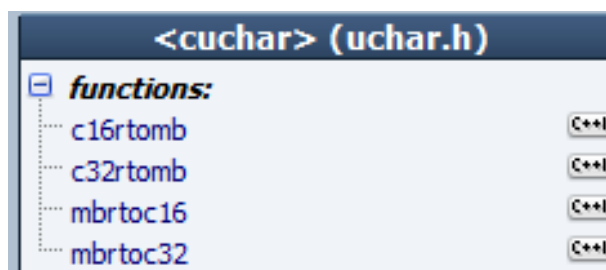
C library:	
<code>&lt;cassert&gt;</code> (assert.h)	
<code>&lt;cctype&gt;</code> (ctype.h)	
<code>&lt;cerrno&gt;</code> (errno.h)	
<code>&lt;cfenv&gt;</code> (fenv.h)	C++11
<code>&lt;cfloat&gt;</code> (float.h)	
<code>&lt;inttypes&gt;</code> (inttypes.h)	C++11
<code>&lt;ciso646&gt;</code> (iso646.h)	
<code>&lt;climits&gt;</code> (limits.h)	
<code>&lt;locale&gt;</code> (locale.h)	
<code>&lt;cmath&gt;</code> (math.h)	
<code>&lt;setjmp&gt;</code> (setjmp.h)	
<code>&lt;csignal&gt;</code> (signal.h)	
<code>&lt;stdarg&gt;</code> (stdarg.h)	
<code>&lt;stdbool&gt;</code> (stdbool.h)	C++11
<code>&lt;stddef&gt;</code> (stddef.h)	
<code>&lt;stdint&gt;</code> (stdint.h)	C++11
<code>&lt;stdio&gt;</code> (stdio.h)	
<code>&lt;stdlib&gt;</code> (stdlib.h)	
<code>&lt;string&gt;</code> (string.h)	
<code>&lt;tgmath&gt;</code> (tgmath.h)	C++11
<code>&lt;ctime&gt;</code> (time.h)	
<code>&lt;cuchar&gt;</code> (uchar.h)	C++11
<code>&lt;wchar&gt;</code> (wchar.h)	
<code>&lt;wctype&gt;</code> (wctype.h)	



- **<stdint.h>** déclare un ensemble de fonctions et de macros pour des conversions précises entre types entiers
- **<stdint.h>** définit divers types d'entiers, ainsi que des macros spécifiant leurs limites , c'est un sous-ensemble de <stdint.h> (introduit par C99).
- **<stdbool.h>** ajoute un type bool et les valeurs true et false en tant que définitions de macro.
- **<cmath>** déclare des opérations mathématiques sur des types génériques. En C++ inclue <cmath> and <complex>.



- **<cuchar>** permet la prise en charge des caractères 16 bits et 32 bits, qui conviennent pour être codés à l'aide d'UTF-16 et d'UTF-32.



- **<cfenv>** déclare un ensemble de fonctions et de macros pour accéder et contrôler à l'environnement en virgule flottante (floating-point), avec des types spécifiques





- **`<array>`**
- **`<forward_list>`**
- **`<unordered_map>`**
- **`<unordered_set>`**

	<b>Containers:</b>	
.....	<code>&lt;array&gt;</code>	
.....	<code>&lt;deque&gt;</code>	
.....	<code>&lt;forward_list&gt;</code>	
.....	<code>&lt;list&gt;</code>	
.....	<code>&lt;map&gt;</code>	
.....	<code>&lt;queue&gt;</code>	
.....	<code>&lt;set&gt;</code>	
.....	<code>&lt;stack&gt;</code>	
.....	<code>&lt;unordered_map&gt;</code>	
.....	<code>&lt;unordered_set&gt;</code>	
.....	<code>&lt;vector&gt;</code>	

## 3 – Extensions de la bibliothèque standard -> <array>



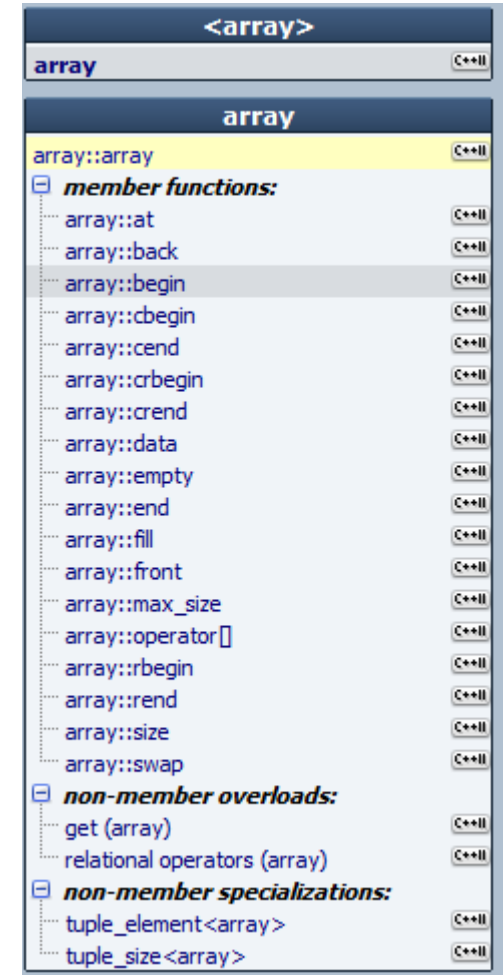
Fournit des méthodes pour la création, la manipulation, la recherche ainsi que le tri des tableaux

### Example

```
1 // array::operator[]
2 #include <iostream>
3 #include <array>
4
5 int main ()
6 {
7     std::array<int,10> myarray;
8     unsigned int i;
9
10    // assign some values:
11    for (i=0; i<10; i++) myarray[i]=i;
12
13    // print content
14    std::cout << "myarray contains:";
15    for (i=0; i<10; i++)
16        std::cout << ' ' << myarray[i];
17    std::cout << '\n';
18
19    return 0;
20 }
```

Output:

```
myarray contains: 0 1 2 3 4 5 6 7 8 9
```



<http://www.cplusplus.com/reference/array/array/>



<forward\_list> fournit une classe patron *std::forward\_list* représentant une liste chaînée.

forward_list	
forward_list::forward_list	C++11
forward_list::~~forward_list	C++11
<b>member functions:</b>	
forward_list::assign	C++11
forward_list::before_begin	C++11
forward_list::begin	C++11
forward_list::cbefore_begin	C++11
forward_list::cbegin	C++11
forward_list::cend	C++11
forward_list::clear	C++11
forward_list::emplace_after	C++11
forward_list::emplace_front	C++11
forward_list::empty	C++11
forward_list::end	C++11
forward_list::erase_after	C++11
forward_list::front	C++11
forward_list::get_allocator	C++11
forward_list::insert_after	C++11
forward_list::max_size	C++11
forward_list::merge	C++11
forward_list::operator=	C++11
forward_list::pop_front	C++11
forward_list::push_front	C++11
forward_list::remove	C++11
forward_list::remove_if	C++11
forward_list::resize	C++11
forward_list::reverse	C++11
forward_list::sort	C++11
forward_list::splice_after	C++11
forward_list::swap	C++11
forward_list::unique	C++11
<b>non-member overloads:</b>	
relational operators (forward_list)	C++11
swap (forward_list)	C++11

### 3 – Extensions de la bibliothèque standard -> <unordered\_map>



<unordered\_map>  
fournit la classe patron  
*std::unordered\_map* et  
*std::unordered\_multimap*  
représentant un tableau  
associatif et une  
multimap.

unordered_map	
unordered_map::unordered_map	C++11
unordered_map::~~unordered_map	C++11
<b>member functions:</b>	
unordered_map::at	C++11
unordered_map::begin	C++11
unordered_map::bucket	C++11
unordered_map::bucket_count	C++11
unordered_map::bucket_size	C++11
unordered_map::cbegin	C++11
unordered_map::cend	C++11
unordered_map::clear	C++11
unordered_map::count	C++11
unordered_map::emplace	C++11
unordered_map::emplace_hint	C++11
unordered_map::empty	C++11
unordered_map::end	C++11
unordered_map::equal_range	C++11
unordered_map::erase	C++11
unordered_map::find	C++11
unordered_map::get_allocator	C++11
unordered_map::hash_function	C++11
unordered_map::insert	C++11
unordered_map::key_eq	C++11
unordered_map::load_factor	C++11
unordered_map::max_bucket_count	C++11
unordered_map::max_load_factor	C++11
unordered_map::max_size	C++11
unordered_map::operator=	C++11
unordered_map::operator[]	C++11
unordered_map::rehash	C++11
unordered_map::reserve	C++11
unordered_map::size	C++11
unordered_map::swap	C++11
<b>non-member overloads:</b>	
operators (unordered_map)	C++11
swap (unordered_map)	C++11

unordered_multimap	
unordered_multimap::unordered_multimap	C++11
unordered_multimap::~~unordered_multimap	C++11
<b>member functions:</b>	
unordered_multimap::begin	C++11
unordered_multimap::bucket	C++11
unordered_multimap::bucket_count	C++11
unordered_multimap::bucket_size	C++11
unordered_multimap::cbegin	C++11
unordered_multimap::cend	C++11
unordered_multimap::clear	C++11
unordered_multimap::count	C++11
unordered_multimap::emplace	C++11
unordered_multimap::emplace_hint	C++11
unordered_multimap::empty	C++11
unordered_multimap::end	C++11
unordered_multimap::equal_range	C++11
unordered_multimap::erase	C++11
unordered_multimap::find	C++11
unordered_multimap::get_allocator	C++11
unordered_multimap::hash_function	C++11
unordered_multimap::insert	C++11
unordered_multimap::key_eq	C++11
unordered_multimap::load_factor	C++11
unordered_multimap::max_bucket_count	C++11
unordered_multimap::max_load_factor	C++11
unordered_multimap::max_size	C++11
unordered_multimap::operator=	C++11
unordered_multimap::rehash	C++11
unordered_multimap::reserve	C++11
unordered_multimap::size	C++11
unordered_multimap::swap	C++11
<b>non-member overloads:</b>	
operators (unordered_multimap)	C++11
swap (unordered_multimap)	C++11



### 3 – Extensions de la bibliothèque standard -> <unordered\_set>

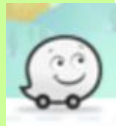


<unordered\_set> fournit  
la classe patron  
*std::unordered\_set* et  
*std::unordered\_multiset*.

unordered_set	
unordered_set::unordered_set	C++11
unordered_set::~unordered_set	C++11
<b>member functions:</b>	
unordered_set::begin	C++11
unordered_set::bucket	C++11
unordered_set::bucket_count	C++11
unordered_set::bucket_size	C++11
unordered_set::cbegin	C++11
unordered_set::cend	C++11
unordered_set::clear	C++11
unordered_set::count	C++11
unordered_set::emplace	C++11
unordered_set::emplace_hint	C++11
unordered_set::empty	C++11
unordered_set::end	C++11
unordered_set::equal_range	C++11
unordered_set::erase	C++11
unordered_set::find	C++11
unordered_set::get_allocator	C++11
unordered_set::hash_function	C++11
unordered_set::insert	C++11
unordered_set::key_eq	C++11
unordered_set::load_factor	C++11
unordered_set::max_bucket_count	C++11
unordered_set::max_load_factor	C++11
unordered_set::max_size	C++11
unordered_set::operator=	C++11
unordered_set::rehash	C++11
unordered_set::reserve	C++11
unordered_set::size	C++11
unordered_set::swap	C++11
<b>non-member overloads:</b>	
operators (unordered_set)	C++11
swap (unordered_set)	C++11

unordered_multiset	
unordered_multiset::unordered_multiset	C++11
unordered_multiset::~unordered_multiset	C++11
<b>member functions:</b>	
unordered_multiset::begin	C++11
unordered_multiset::bucket	C++11
unordered_multiset::bucket_count	C++11
unordered_multiset::bucket_size	C++11
unordered_multiset::cbegin	C++11
unordered_multiset::cend	C++11
unordered_multiset::clear	C++11
unordered_multiset::count	C++11
unordered_multiset::emplace	C++11
unordered_multiset::emplace_hint	C++11
unordered_multiset::empty	C++11
unordered_multiset::end	C++11
unordered_multiset::equal_range	C++11
unordered_multiset::erase	C++11
unordered_multiset::find	C++11
unordered_multiset::get_allocator	C++11
unordered_multiset::hash_function	C++11
unordered_multiset::insert	C++11
unordered_multiset::key_eq	C++11
unordered_multiset::load_factor	C++11
unordered_multiset::max_bucket_count	C++11
unordered_multiset::max_load_factor	C++11
unordered_multiset::max_size	C++11
unordered_multiset::operator=	C++11
unordered_multiset::rehash	C++11
unordered_multiset::reserve	C++11
unordered_multiset::size	C++11
unordered_multiset::swap	C++11
<b>non-member overloads:</b>	
operators (unordered_multiset)	C++11
swap (unordered_multiset)	C++11

## 3 – Extensions de la bibliothèque standard -> Divers



- **`<tuple>`**
- **`<random>`**
- **`<functional>`**
- **`<regex>`**
- **`<type_traits>`**
- **`<ratio>`**
- **`<system_error>`**
- **`<typeindex>`**
- **`<codecvt>`**

Other:	
<code>&lt;algorithm&gt;</code>	
<code>&lt;bitset&gt;</code>	
<code>&lt;chrono&gt;</code>	C++11
<code>&lt;codecvt&gt;</code>	C++11
<code>&lt;complex&gt;</code>	
<code>&lt;exception&gt;</code>	
<code>&lt;functional&gt;</code>	
<code>&lt;initializer_list&gt;</code>	C++11
<code>&lt;iterator&gt;</code>	
<code>&lt;limits&gt;</code>	
<code>&lt;locale&gt;</code>	
<code>&lt;memory&gt;</code>	
<code>&lt;new&gt;</code>	
<code>&lt;numeric&gt;</code>	
<code>&lt;random&gt;</code>	C++11
<code>&lt;ratio&gt;</code>	C++11
<code>&lt;regex&gt;</code>	C++11
<code>&lt;stdexcept&gt;</code>	
<code>&lt;string&gt;</code>	
<code>&lt;system_error&gt;</code>	C++11
<code>&lt;tuple&gt;</code>	C++11
<code>&lt;typeindex&gt;</code>	C++11
<code>&lt;typeinfo&gt;</code>	
<code>&lt;type_traits&gt;</code>	C++11
<code>&lt;utility&gt;</code>	
<code>&lt;valarray&gt;</code>	

## 3 – Extensions de la bibliothèque standard -> <tuple>



Fournit des méthodes statiques pour la création d'objets multiples.

### Example

```
1 // tuple example
2 #include <iostream>      // std::cout
3 #include <tuple>         // std::tuple, std::get, std::tie, std::ignore
4
5 int main ()
6 {
7     std::tuple<int,char> foo (10,'x');
8     auto bar = std::make_tuple ("test", 3.1, 14, 'y');
9
10    std::get<2>(bar) = 100;                // access element
11
12    int myint; char mychar;
13
14    std::tie (myint, mychar) = foo;          // unpack elements
15    std::tie (std::ignore, std::ignore, myint, mychar) = bar; // unpack (with ignore)
16
17    mychar = std::get<3>(bar);
18
19    std::get<0>(foo) = std::get<2>(bar);
20    std::get<1>(foo) = mychar;
21
22    std::cout << "foo contains: ";
23    std::cout << std::get<0>(foo) << ' ';
24    std::cout << std::get<1>(foo) << '\n';
25
26    return 0;
27 }
```

Output:

```
foo contains: 100 y
```

<tuple>	
<b>classes:</b>	
tuple	C++11
tuple_element	C++11
tuple_size	C++11
<b>functions:</b>	
forward_as_tuple	C++11
get	C++11
make_tuple	C++11
tie	C++11
tuple_cat	C++11
<b>objects:</b>	
ignore	C++11

tuple	
tuple::tuple	C++11
<b>member functions:</b>	
tuple::operator=	C++11
tuple::swap	C++11
<b>non-member overloads:</b>	
relational operators (tuple)	C++11
swap (tuple)	C++11
<b>non-member specializations:</b>	
uses_allocator<tuple>	C++11

<http://www.cplusplus.com/reference/tuple/tuple/?kw=tuple>



les **structured bindings** peuvent être vu comme une syntaxe simplifiée pour `std::tie` par rapport au tuple

Les **structured bindings** fonctionnent non seulement sur `std::tuple`, mais également sur les structures, `std::pair` et les tableaux (`std::array` ou tableau C-style).

# 3 – Extensions de la bibliothèque standard -> <random>



Représente un générateur de nombres pseudo-aléatoires. Il s'agit d'un périphérique qui produit une séquence de nombres conformes à certains prérequis statistiques liés à l'aspect aléatoire.

## Example

```
1 // cauchy_distribution
2 #include <iostream>
3 #include <random>
4
5 int main()
6 {
7     const int nrolls=10000; // number of experiments
8     const int nstars=100;   // maximum number of stars to distribute
9
10    std::default_random_engine generator;
11    std::cauchy_distribution<double> distribution(5.0,1.0);
12
13    int p[10]={};
14
15    for (int i=0; i<nrolls; ++i) {
16        double number = distribution(generator);
17        if ((number>=0.0)&&(number<10.0)) ++p[int(number)];
18    }
19
20    std::cout << "cauchy_distribution (5.0,1.0):" << std::endl;
21
22    for (int i=0; i<10; ++i) {
23        std::cout << i << "-" << (i+1) << ": ";
24        std::cout << std::string(p[i]*nstars/nrolls, '*') << std::endl;
25    }
26
27    return 0;
28 }
```

Possible output:

```
cauchy_distribution (5.0,1.0):
0-1: *
1-2: **
2-3: ****
3-4: *****
4-5: *****
5-6: *****
6-7: *****
7-8: ****
8-9: **
9-10: *
```

<http://www.cplusplus.com/reference/random/?kw=random>

Le C++ 11 et le ++ 17

Version 1.1 - 09/2019

<random>	
<b>distributions:</b>	
bernoulli_distribution	Call
binomial_distribution	Call
cauchy_distribution	Call
chi_squared_distribution	Call
discrete_distribution	Call
exponential_distribution	Call
extreme_value_distribution	Call
fisher_f_distribution	Call
gamma_distribution	Call
geometric_distribution	Call
lognormal_distribution	Call
negative_binomial_distribution	Call
normal_distribution	Call
piecewise_constant_distribution	Call
piecewise_linear_distribution	Call
poisson_distribution	Call
student_t_distribution	Call
uniform_int_distribution	Call
uniform_real_distribution	Call
weibull_distribution	Call
<b>generators:</b>	
default_random_engine	Call
discard_block_engine	Call
independent_bits_engine	Call
knuth_b	Call
linear_congruential_engine	Call
mersenne_twister_engine	Call
minstd_rand	Call
minstd_rand0	Call
mt19937	Call
mt19937_64	Call
random_device	Call
ranlux24	Call
ranlux24_base	Call
ranlux48	Call
ranlux48_base	Call
shuffle_order_engine	Call
subtract_with_carry_engine	Call
<b>other:</b>	
generate_canonical	Call
seed_seq	Call



## 3 – Extensions de la bibliothèque standard -> <functional>

<functional> est une bibliothèque standard C++ qui fournit un ensemble de modèles de classes prédéfinis pour les objets fonction, y compris les opérations arithmétiques, comparaisons et logiques..

```
#include <iostream>
#include <functional>

template <typename T> class CAnyData {
public:
    T m_value;
    CAnyData(T value) : m_value { value } {}
    void print(void) { std::cout << m_value << std::endl; }
    void printAfterAdd(T value) { std::cout << (m_value + value) << std::endl; }
};

int main(void)
{
    /* A function wrapper to a member variable of a class */
    CAnyData<int> dataA { 2016 };
    std::function<int (CAnyData<int> &)> funcA = &CAnyData<int>::m_value;
    std::cout << funcA(dataA) << std::endl;

    /* A function wrapper to member function without parameter passing */
    CAnyData<float> dataB { 2016.1 };
    std::function<void (CAnyData<float> &)> funcB = &CAnyData<float>::print;
    funcB(dataB);

    /* A function wrapper to member function with passing a parameter */
    std::function<void (CAnyData<float> &, float)> funcC = &CAnyData<float>::printAfterAdd;
    funcC(dataB, 0.1);

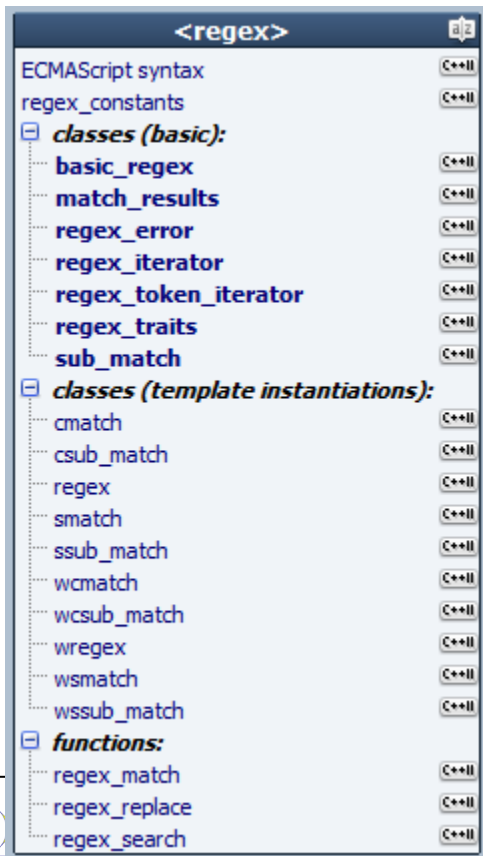
    /* A function wrapper to member function generated by std::bind */
    std::function<void (float)> funcD = std::bind(&CAnyData<float>::printAfterAdd, &dataB, std::placeholders::_1);
    funcD(0.2);

    return 0;
}
```

<functional>	
<b>functions:</b>	
bind	++
cref	++
mem_fn	++
not1	
not2	
ref	++
<b>wrapper classes:</b>	
<b>operator classes:</b>	
bit_and	++
bit_or	++
bit_xor	++
divides	
equal_to	
greater	
greater_equal	
less	
less_equal	
logical_and	
logical_not	
logical_or	
minus	
modulus	
multiplies	
negate	
not_equal_to	
plus	
<b>other classes:</b>	
<b>namespaces:</b>	
placeholders	++
<b>deprecated:</b>	
binary_function	
bind1st	
bind2nd	
binder1st	
binder2nd	
const_mem_fun1_ref_t	
const_mem_fun1_t	
const_mem_fun_ref_t	
const_mem_fun_t	
mem_fun	
mem_fun1_ref_t	
mem_fun1_t	
mem_fun_ref	
mem_fun_ref_t	
mem_fun_t	
pointer_to_binary_function	
pointer_to_unary_function	
ptr_fun	
unary_function	

### 3 – Extensions de la bibliothèque standard -> <regex>

Bibliothèque qui définit des classes permettant d'analyser des expressions régulières (C++) et des modèles et des fonctions pour rechercher le texte correspondant correspond à un objet d'expression régulière.



```
#include <iostream>
#include <string>
#include <regex>

void test_regex_search(const std::string& input)
{
    std::regex rgx("((1[0-2])|(0?[1-9])):([0-5][0-9])((am)|(pm))");
    std::smatch match;

    if (std::regex_search(input.begin(), input.end(), match, rgx))
    {
        std::cout << "Match\n";

        //for (auto m : match)
        // std::cout << " submatch " << m << '\n';

        std::cout << "match[1] = " << match[1] << '\n';
        std::cout << "match[4] = " << match[4] << '\n';
        std::cout << "match[5] = " << match[5] << '\n';
    }
    else
        std::cout << "No match\n";
}

int main()
{
    const std::string time1 = "9:45pm";
    const std::string time2 = "11:53am";

    test_regex_search(time1);
    test_regex_search(time2);
}
```

Output from the program:

```
Match
match[1] = 9
match[4] = 45
match[5] = pm
Match
match[1] = 11
match[4] = 53
match[5] = am
```



# 3 – Extensions de la bibliothèque standard -> <type\_traits>



La bibliothèque <type\_traits> définit une série de classes pour obtenir des informations de type sur la compilation.

- Classes standard pour aider à créer des constantes de compilation.
- Caractères de type : classes permettant d'obtenir les caractéristiques des types sous la forme de valeurs constantes à la compilation.
- Transformations de type : classes permettant d'obtenir de nouveaux types en appliquant des transformations spécifiques aux types existants.

```

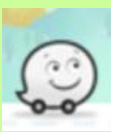
<type_traits>
- helper classes:
  - false_type
  - integral_constant
  - true_type
- type traits:
  - alignment_of
  - extent
  - has_virtual_destructor
  - is_abstract
  - is_arithmetic
  - is_array
  - is_assignable
  - is_base_of
  - is_class
  - is_compound
  - is_const
  - is_constructible
  - is_convertible
  - is_copy_assignable
  - is_copy_constructible
  - is_default_constructible
  - is_destructible
  - is_empty
  - is_enum
  - is_floating_point
  - is_function
  - is_fundamental
  - is_integral
  - is_literal_type
  - is_lvalue_reference
  
```

```

<type_traits>
- helper classes:
  - false_type
  - integral_constant
  - true_type
+ type traits:
- type transformations:
  - add_const
  - add_cv
  - add_lvalue_reference
  - add_pointer
  - add_rvalue_reference
  - add_volatile
  - aligned_storage
  - aligned_union
  - common_type
  - conditional
  - decay
  - enable_if
  - make_signed
  - make_unsigned
  - remove_all_extents
  - remove_const
  - remove_cv
  - remove_extent
  - remove_pointer
  - remove_reference
  - remove_volatile
  - result_of
  - underlying_type
  
```



## 3 – Extensions de la bibliothèque standard -> <ratio>



La bibliothèque <ratio> déclare des classe de ratio et plusieurs types auxiliaires pour fonctionner avec eux.

Un ratio exprime une proportion comme un ensemble de deux constantes de temps de compilation (son numérateur et son dénominateur).

The following presents standard instantiations of ratio

type	definition	description
yocto	ratio<1,1000000000000000000000>	10 <sup>-24</sup> *
zepto	ratio<1,100000000000000000000>	10 <sup>-21</sup> *
atto	ratio<1,10000000000000000000>	10 <sup>-18</sup>
femto	ratio<1,100000000000000000>	10 <sup>-15</sup>
pico	ratio<1,100000000000000>	10 <sup>-12</sup>
nano	ratio<1,1000000000>	10 <sup>-9</sup>
micro	ratio<1,1000000>	10 <sup>-6</sup>
milli	ratio<1,1000>	10 <sup>-3</sup>
centi	ratio<1,100>	10 <sup>-2</sup>
deci	ratio<1,10>	10 <sup>-1</sup>
deca	ratio<10,1>	10 <sup>1</sup>
hecto	ratio<100,1>	10 <sup>2</sup>
kilo	ratio<1000,1>	10 <sup>3</sup>
mega	ratio<1000000,1>	10 <sup>6</sup>
giga	ratio<1000000000,1>	10 <sup>9</sup>
tera	ratio<1000000000000,1>	10 <sup>12</sup>
peta	ratio<1000000000000000,1>	10 <sup>15</sup>
exa	ratio<1000000000000000000,1>	10 <sup>18</sup>
zetta	ratio<100000000000000000000,1>	10 <sup>21</sup> *
yotta	ratio<10000000000000000000000,1>	10 <sup>24</sup> *

```

1 // ratio example
2 #include <iostream>
3 #include <ratio>
4
5 int main ()
6 {
7     typedef std::ratio<1,3> one_third;
8     typedef std::ratio<2,4> two_fourths;
9
10    std::cout << "one_third= " << one_third::num << "/" << one_third::den << std::endl;
11    std::cout << "two_fourths= " << two_fourths::num << "/" << two_fourths::den << std::endl;
12
13    typedef std::ratio_add<one_third,two_fourths> sum;
14
15    std::cout << "sum= " << sum::num << "/" << sum::den;
16    std::cout << " (which is: " << ( double(sum::num) / sum::den ) << ")" << std::endl;
17
18    std::cout << "1 kilogram has " << ( std::kilo::num / std::kilo::den ) << " grams";
19    std::cout << std::endl;
20
21    return 0;
22 }

```

Output:

```

one_third= 1/3
two_fourths= 1/2
sum= 5/6 (which is 0.833333)
1 kilogram has 1000 grams

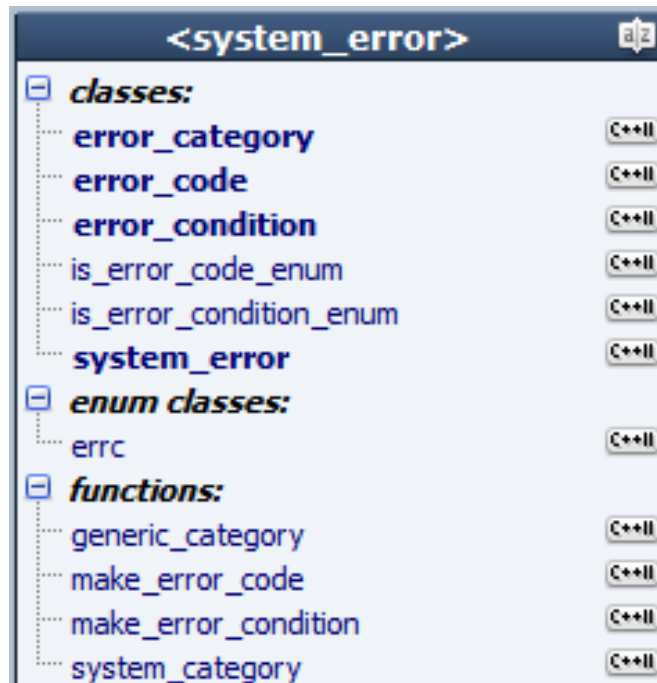
```

<ratio>	
ratio	Call
ratio_add	Call
ratio_divide	Call
ratio_equal	Call
ratio_greater	Call
ratio_greater_equal	Call
ratio_less	Call
ratio_less_equal	Call
ratio_multiply	Call
ratio_not_equal	Call
ratio_subtract	Call

### 3 – Extensions de la bibliothèque standard -> <system\_error>



La bibliothèque <system\_error> définit des classes normalisées pour signaler les conditions d'erreur provenant du système d'exploitation ou d'autres opérations de bas niveau.



## 3 – Extensions de la bibliothèque standard -> <typeindex>



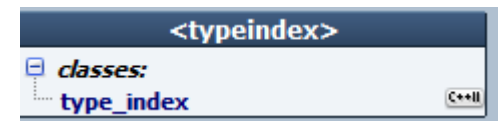
La classe `type_index` enveloppe un objet de `type_info` afin qu'il puisse être copié et / ou utilisé comme index au moyen d'une fonction de hachage standard.

### Example

```
1 // type_index example
2 #include <iostream>           // std::cout
3 #include <typeinfo>           // operator typeid
4 #include <typeindex>           // std::type_index
5 #include <unordered_map>       // std::unordered_map
6 #include <string>              // std::string
7
8 struct C {};
9
10 int main()
11 {
12     std::unordered_map<std::type_index, std::string> mytypes;
13
14     mytypes[typeid(int)]="Integer type";
15     mytypes[typeid(double)]="Floating-point type";
16     mytypes[typeid(C)]="Custom class named C";
17
18     std::cout << "int: " << mytypes[typeid(int)] << '\n';
19     std::cout << "double: " << mytypes[typeid(double)] << '\n';
20     std::cout << "C: " << mytypes[typeid(C)] << '\n';
21
22     return 0;
23 }
```

Output:

```
int: Integer type
double: Floating-point type
C: Custom class named C
```



## 3 – Extensions de la bibliothèque standard C++11-> <codecvt> ( KO en C++17)



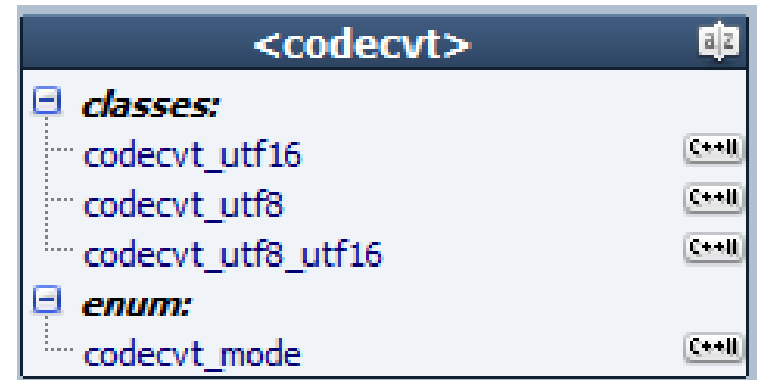
La bibliothèque <codecvt> contient des classes standard permettant de faire des conversions entre les différents codages de caractères UTF.

### Example

```
1 // codecvt_utf8: writing UTF-32 string as UTF-8
2 #include <iostream>
3 #include <locale>
4 #include <string>
5 #include <codecvt>
6 #include <fstream>
7
8 int main ()
9 {
10     std::u32string str ( U"U00004f60U0000597d" ); // ni hao (你好)
11
12     std::locale loc (std::locale(), new std::codecvt_utf8<char32_t>);
13     std::basic_ofstream<char32_t> ofs ("test.txt");
14     ofs.imbue(loc);
15
16     std::cout << "Writing to file (UTF-8)... ";
17     ofs << str;
18     std::cout << "done!\n";
19
20     return 0;
21 }
```

### Output

Writing to file (UTF-8)... done!



### 3 - Complément : Plain Old Data

#### Extensions de la bibliothèque standard -> `<type_traits>`



Plain Old Data en C ++ Object , est défini en tant que type scalaire ou classe PDS tels que la classe n'a pas d'opérateur d'affectation de copie défini par l'utilisateur, pas de destructeur défini par l'utilisateur et pas de données membres non statiques qui ne sont pas elles-mêmes PDS.

De plus, elle ne doit comporter aucun constructeur déclaré par l'utilisateur, aucune donnée non statique privée ou protégée, aucune classe de base virtuelle et aucune fonction virtuelle.

La bibliothèque `type_traits` de la bibliothèque standard C ++ fournit un modèle nommé `is_pod` qui peut être utilisé pour déterminer si un type donné est un POD.

## 3 - Complément : Plain Old Data

### Extensions de la bibliothèque standard -> <type\_traits>



```
1 // is_pod example
2 #include <iostream>
3 #include <type_traits>
4
5 struct A { int i; };           // C-struct (POD)
6 class B : public A {};        // still POD (no data members added)
7 struct C : B { void fn(){} }; // still POD (member function)
8 struct D : C { D(){} };       // no POD (custom default constructor)
9
10 int main() {
11     std::cout << std::boolalpha;
12     std::cout << "is_pod:" << std::endl;
13     std::cout << "int: " << std::is_pod<int>::value << std::endl;
14     std::cout << "A: " << std::is_pod<A>::value << std::endl;
15     std::cout << "B: " << std::is_pod<B>::value << std::endl;
16     std::cout << "C: " << std::is_pod<C>::value << std::endl;
17     std::cout << "D: " << std::is_pod<D>::value << std::endl;
18     return 0;
19 }
```

Output:

```
is_pod:
int: true
A: true
B: true
C: true
D: false
```

## 3 - Complément : sizeof (C++11)



Dans le C++98, il n'est pas possible de faire référence dans sizeof à une donnée membre non statique sans instancier cette classe.

Le C++11 supprime cette limitation

```
struct C {  
    static T1 m1;  
    T2 m2;  
    void foo() const;  
};  
  
sizeof(C::m1); // ok, variable membre statique  
sizeof(C::m2); // erreur en C++98, valide en C++11  
  
C c;  
sizeof(c.m2); // ok, la classe est instanciée  
  
void C::foo() const { sizeof(m2); } // erreur en C++98, valide en C++11  
  
sizeof(((C*) 0)->m2); // hack possible pour contourner en C++98
```

## 3 - Complément : Classe Type trait



Une classe de trait est une classe (ou structure) qui associe à un type donné d'autres types (grâce à des typedef) ainsi que des fonctions membres statiques.

La puissance des traits est due au fait que cela ajoute un niveau d'abstraction et permet d'ajouter un niveau de généricité.

```
template <typename T>
struct TypeDescriptor
{
    typedef T type;
    typedef T* pointer;
    typedef T& reference;
    typedef const T const_type;
    // ...
};

// Plus loin dans le code
int i = 42;
TypeDescriptor<int>::pointer pi = &i;
*pi = 24;
```

```
template <typename T>
struct is_int
{
    static const bool value = false;
};
```

```
template <>
struct is_int<int>
{
    static const bool value = true;
};
```

<https://alp.developpez.com/tutoriels/traitspolicies/>



## 3 - Complément : Classe Type trait



```
template <typename T>
struct TypeDescriptor
{
    typedef T type;
    typedef T* pointer;
    typedef T& reference;
    typedef const T const_type;
    // ...
};

// Plus loin dans le code
int i = 42;
TypeDescriptor<int>::pointer pi = &i;
*pi = 24;
```

```
template <typename T>
struct TypeDescriptor<T&>
{
    typedef T& type;
    // ..
    typedef T& reference;
    // ..
};
```

```
class A;
A myinstance;
TypeDescriptor<A&>::reference ra = myinstance;
```

<https://alp.developpez.com/tutoriels/traitspolicies/>

## 3 - Complément : Classe Type trait



```
template <bool is_a_pointer>
void f()
{
    std::cout << "Je ne sais pas qui je suis" << std::endl;
}

template <>
void f<true>()
{
    std::cout << "Je suis un pointeur!" << std::endl;
}

struct Foo
{
    template <typename T>
    void Bar()
    {
        f< is_pointer<T>::value >();
        /* affichera "Je ne sais pas qui je suis" si T n'est pas un pointeur et
        "Je suis un pointeur!" dans le cas contraire */
    }
};

// ...
Foo f;
f.Bar<int>(); // affiche "Je ne sais pas qui je suis"
f.Bar<int*>(); // affiche "Je suis un pointeur!"
```

<https://alp.developpez.com/tutoriels/traitspolices/>

## 3 - Complément : Classe Type Politiques



Les classes de politique (policy classes) ressemblent aux classes de traits, mais les classes traits à ajoutent des informations à des types, les classes de politiques servent à définir des comportements

```
template <typename T>
struct Addition
{
    static void Accumuler(T& Resultat, const T& Valeur)
    {
        Resultat += Valeur;
    }
};

template <typename T, typename Operation>
T Accumulation(const T* Debut, const T* Fin)
{
    T Resultat = 0;
    for ( ; Debut != Fin; ++Debut)
        Operation::Accumuler(Resultat, *Debut);

    return Resultat;
}
```

<https://alp.developpez.com/tutoriels/traitspolitiques/>

# 3 – Complément : Extensions de la bibliothèque standard -> <cmath>

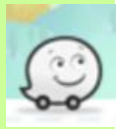


## Opérations de base

<b>abs</b> (int) <b>labs</b> <b>llabs</b> <b>imaxabs</b> (C++11)	calcule la valeur absolue d'une valeur intégrale ( $ x $ ) (fonction)
<b>abs</b> (float) <b>fabs</b>	valeur absolue d'une valeur à virgule flottante ( $ x $ ) (fonction)
<b>div</b> <b>ldiv</b> <b>lldiv</b> <b>imaxdiv</b> (C++11)	le quotient et le reste de la division entière (fonction)
<b>fmod</b>	reste de l'opération de division à virgule flottante (fonction)
<b>remainder</b> (C++11)	signé reste de l'opération de division (fonction)
<b>remquo</b> (C++11)	signé reste ainsi que les trois derniers bits de l'opération de division (fonction)
<b>fma</b> (C++11)	fusionné multiply-add opération (fonction)
<b>fmax</b> (C++11)	plus grande des deux valeurs à virgule flottante (fonction)
<b>fmin</b> (C++11)	plus petite des deux valeurs à virgule flottante (fonction)
<b>fdim</b> (C++11)	différence positive de deux valeurs à virgule flottante ( $\max(0, x-y)$ ) (fonction)
<b>nan</b> (C++11) <b>nanf</b> (C++11) <b>nanl</b> (C++11)	pas-un nombre-(NaN) (fonction)

<https://fr.cppreference.com/w/cpp/numeric/math>

## 3 – Complément : Extensions de la bibliothèque standard -> <cmath>



### Fonctions exponentielles

<b>exp</b>	retours $e$ à la puissance donnée ( $e^x$ ) (fonction)	[edit]
<b>exp2</b> (C++11)	retours 2 élevé à la puissance donnée ( $2^x$ ) (fonction)	[edit]
<b>expm1</b> (C++11)	retours $e$ à la puissance donnée, moins un ( $e^x - 1$ ) (fonction)	[edit]
<b>log</b>	calcule naturel (base $e$ ) logarithme (de base $e$ ) ( $\ln(x)$ ) (fonction)	[edit]
<b>log10</b>	calcule commune (base 10) Logarithme ( $\log_{10}(x)$ ) (fonction)	[edit]
<b>log1p</b> (C++11)	logarithme naturel (à la base $e$ ) de 1, plus le nombre donné (fonction)	[edit]
<b>log2</b> (C++11)	logarithme en base 2 d'un nombre donné (fonction)	[edit]

### Fonctions puissance

<b>sqrt</b>	calcule la racine carrée ( $\sqrt{x}$ ) (fonction)	[edit]
<b>cbrt</b> (C++11)	calcule la racine cubique ( $\sqrt[3]{x}$ ) (fonction)	[edit]
<b>hypot</b> (C++11)	calcule la racine carrée de la somme des carrés des deux nombres donnés ( $\sqrt{x^2 + y^2}$ ) (fonction)	[edit]
<b>pow</b>	soulève un certain nombre à la puissance donnée ( $x^y$ ) (fonction)	[edit]

<https://fr.cppreference.com/w/cpp/numeric/math>

# 3 – Complément : Extensions de la bibliothèque standard -> <cmath>



## Fonctions hyperboliques

<b>sinh</b>	calcule sinus hyperbolique ( $sh(x)$ ) (fonction)
<b>cosh</b>	calcule le cosinus hyperbolique ( $ch(x)$ ) (fonction)
<b>tanh</b>	tangente hyperbolique (fonction)
<b>asinh</b> (C++11)	arc sinus hyperbolique (fonction)
<b>acosh</b> (C++11)	arc cosinus hyperbolique (fonction)
<b>atanh</b> (C++11)	arctangente hyperbolique (fonction)

## Fonctions erreurs et gamma

<b>erf</b> (C++11)	fonction d'erreur (fonction)
<b>erfc</b> (C++11)	fonction d'erreur complémentaire (fonction)
<b>lgamma</b> (C++11)	logarithme naturel de la fonction gamma (fonction)
<b>tgamma</b> (C++11)	fonction gamma (fonction)

## Fonction d'entiers les plus proches

<b>ceil</b>	entier le plus proche n'est pas inférieure à la valeur donnée (fonction)
<b>floor</b>	entier le plus proche n'est pas supérieure à la valeur donnée (fonction)
<b>trunc</b> (C++11)	entier le plus proche n'est pas supérieure en grandeur à la valeur donnée (fonction)
<b>round</b> (C++11) <b>lround</b> (C++11) <b>llround</b> (C++11)	entier le plus proche, arrondi en s'éloignant de zéro dans les cas de transitivité (fonction)
<b>nearbyint</b> (C++11)	entier le plus proche en utilisant le mode d'arrondi courant (fonction)
<b>rint</b> (C++11) <b>lrint</b> (C++11) <b>llrint</b> (C++11)	entier le plus proche en utilisant le mode d'arrondi courant à l'exception si le résultat est différent (fonction)

<https://fr.cppreference.com/w/cpp/numeric/math>

# 3 – Complément : Extensions de la bibliothèque standard -> <cmath>



## Fonctions de manipulation des nombres flottants

<b>frexp</b>	se décompose en un certain nombre mantisse et d'une puissance de 2 (fonction)	[edit]
<b>ldexp</b>	multiplie par un nombre 2 élevé à une puissance (fonction)	[edit]
<b>modf</b>	décompose un nombre en parties entières et fractionnaires (fonction)	[edit]
<b>scalbn</b> (C++11) <b>scalbln</b> (C++11)	multiplie par un nombre FLT_RADIX élevé à une puissance (fonction)	[edit]
<b>ilogb</b> (C++11)	extraît exposant du nombre (fonction)	[edit]
<b>logb</b> (C++11)	extraît exposant du nombre (fonction)	[edit]
<b>nextafter</b> (C++11) <b>nexttoward</b> (C++11)	prochaine valeur représentable en virgule flottante vers la valeur donnée (fonction)	[edit]
<b>copysign</b> (C++11)	copie le signe d'une valeur à virgule flottante (fonction)	[edit]

## Classification et comparaison

<b>fpclassify</b> (C++11)	catégorise la valeur du point donné flottante (fonction)	[edit]
<b>isfinite</b> (C++11)	vérifie si le nombre donné a une valeur finie (fonction)	[edit]
<b>isinf</b> (C++11)	vérifie si le nombre donné est infini (fonction)	[edit]
<b>isnan</b> (C++11)	vérifie si le nombre donné est NaN (fonction)	[edit]
<b>isnormal</b> (C++11)	vérifie si le nombre donné est normal (fonction)	[edit]
<b>signbit</b> (C++11)	vérifie si le nombre donné est négatif (fonction)	[edit]
<b>isgreater</b> (C++11)	vérifie si le premier argument de virgule flottante est supérieure à la seconde (fonction)	[edit]
<b>isgreaterequal</b> (C++11)	vérifie si le premier argument de virgule flottante est supérieure ou égale à la seconde (fonction)	[edit]
<b>isless</b> (C++11)	vérifie si le premier argument de virgule flottante est inférieure à la seconde (fonction)	[edit]
<b>islessequal</b> (C++11)	vérifie si le premier argument de virgule flottante est inférieure ou égale à la seconde (fonction)	[edit]
<b>islessgreater</b> (C++11)	vérifie si le premier argument de virgule flottante est inférieure ou supérieure à la seconde (fonction)	[edit]
<b>isunordered</b> (C++11)	vérifie si deux valeurs à virgule flottante ne sont pas ordonnés (fonction)	[edit]





## 4 - Les éléments dépréciés C++17



Eléments dépréciés :

- **std::result\_of** remplacé par un nouveau trait **std::invoke\_result**.
- l'entête **<codecvt>**
- **std::wstring\_convert** et **std::wbuffer\_convert** de l'entête **<locale>**

## 4 - Les éléments supprimés C++17



### Éléments supprimées:

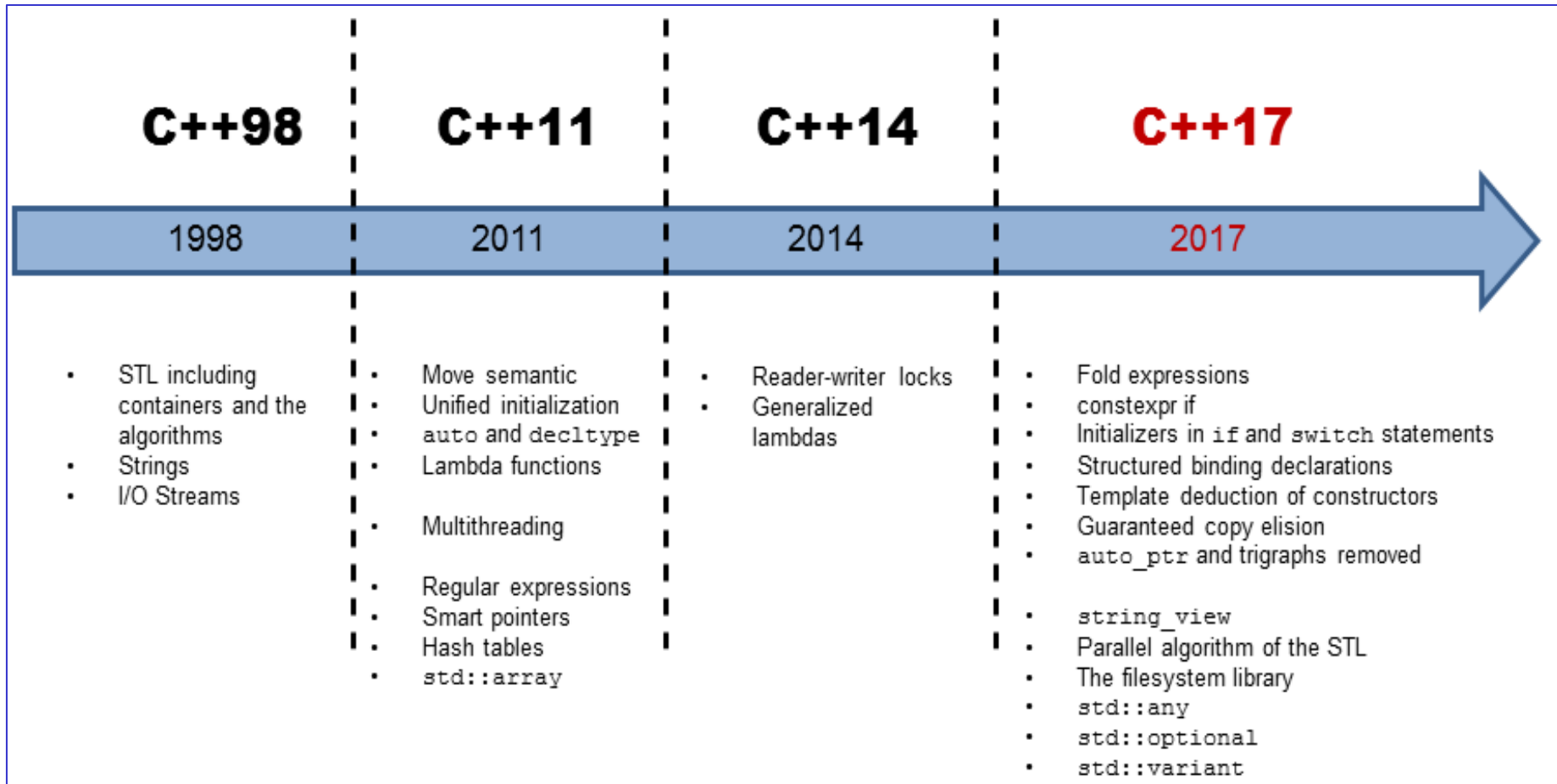
- **Les trigraphes** les trigraphes sont hérités du langage C et de l'époque où certains claviers ne possédaient pas un jeu de caractères couvrant tous les symboles du langage
- Le mot-clef **register** car aucun effet
- **operator++(bool)**
- **std::auto\_ptr** remplacé par **std::weak\_ptr**
- **std::function** propose des allocateurs pour certains de ses constructeurs et fonctions membres qui ont été supprimés .
- **std::random\_shuffle**
- **Les reliques de <functional>**
  - **bind1st** et **bind2nd**, remplacés par **std::bind**
  - **std::unary\_function** et **std::binary\_function** n'ont plus vraiment d'utilité avec les ajouts de **std::function**
  - **std::mem\_fun** et **std::mem\_fun\_ref** sont supprimés étant donné l'introduction de **std::mem\_fn**
  - **std::ptr\_fun** car les pointeurs de fonctions pouvant être directement passés à **std::bind** ou **std::function**

Symbole	Digraphe	Trigraphe
{	<%	??<
}	%>	??>
[	<:	??{
]	:>	??}
#	%:	??=
\		??/
^		??'
		??!
~		??~





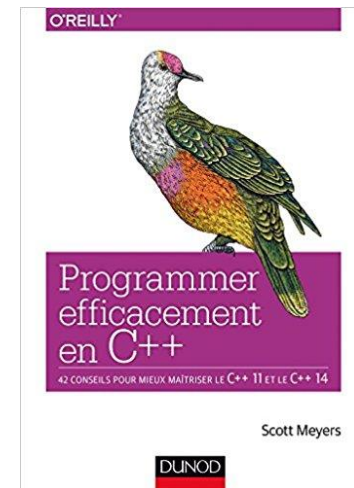
# 5 - Conclusion



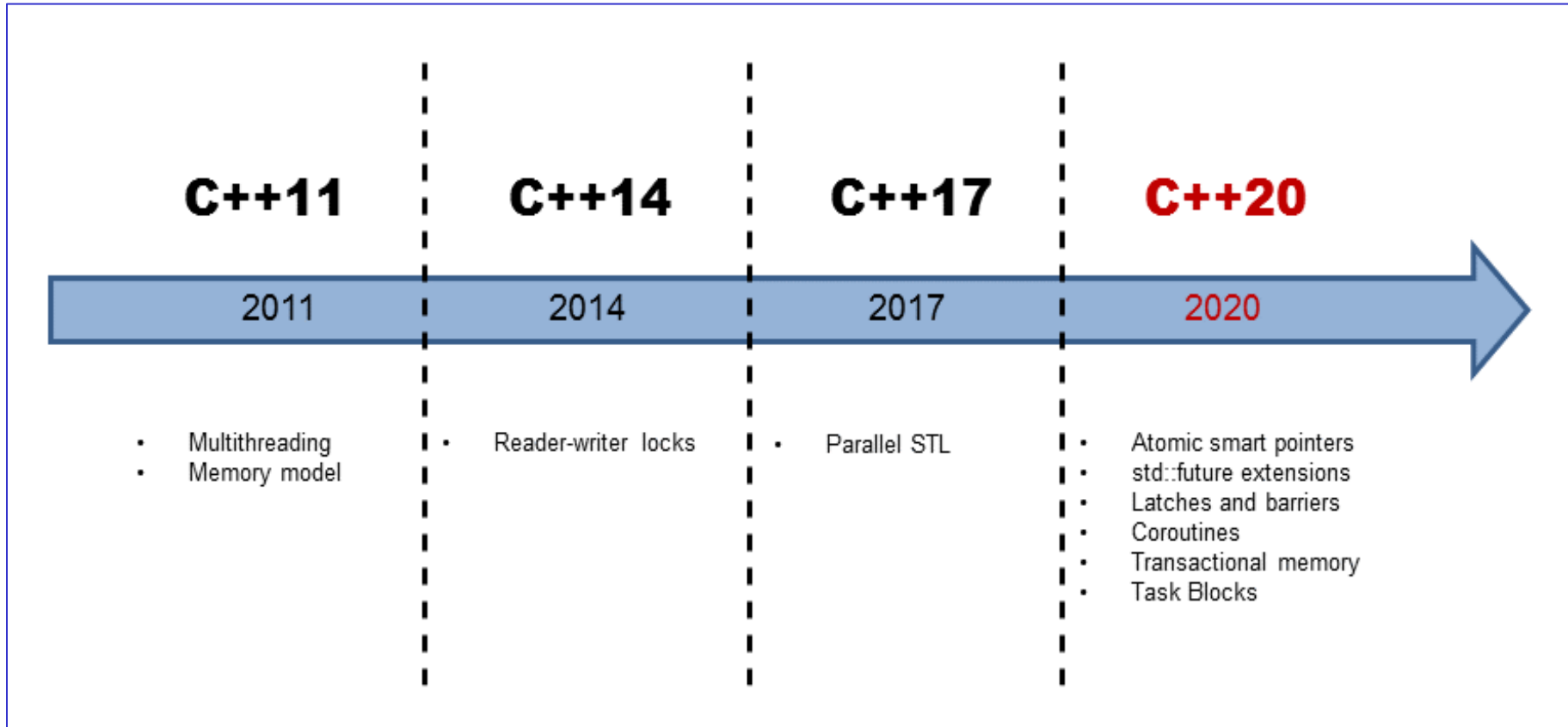
<http://www.modernescpp.com/index.php/c-17-new-algorithm-of-the-standard-template-library>  
<https://isocpp.org/std/standing-documents/sd-6-sg10-feature-test-recommendations#recs.cpp17>



- Conseil n° 13. Préférer les `const_iterator` aux `iterator`
- Conseil n° 42. Envisager le placement plutôt que l'insertion



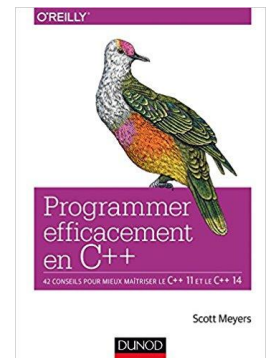
## 5 - Conclusion



<http://www.modernescpp.com/index.php/multithreading-in-c-17-and-c-20>



- Conseil n° 22. Avec l'idiome Pimpl, définir des fonctions membres spéciales dans le fichier d'implémentation
- Conseil n° 27. Se familiariser avec les alternatives à la surcharge sur les références universelles
- Conseil n° 28. Comprendre la réduction de référence
- Conseil n° 29. Supposer que les opérations de déplacement sont absentes, onéreuses et inutilisées
- Conseil n° 30. Se familiariser avec les cas d'échec de la transmission parfaite
- Conseil n° 41. Envisager un passage par valeur pour les paramètres copiables dont le déplacement est bon marché et qui sont toujours copiés





## Les éléments suivants sont des fonctionnalités clés dans C++20 :

- Les Modules.
  - Les Coroutines.
  - Les Concepts.
  - Les Ranges.
  - Les constexprification: `constexpr`, `constexpr`, `std::is_constant_evaluated`, `constexpr` allocation, `constexpr` `std::vector`, `constexpr` `std::string`, `constexpr` union, `constexpr` try and catch, `constexpr` `dynamic_cast` and `typeid`.
  - `std::format("For C++{}", 20)`
  - `operator<=>`
  - Macros de test de fonctionnalités.
  - `std::span`
  - `std::source_location`.
  - `std::atomic_ref`.
  - `std::atomic::wait`, `std::atomic::notify`, `std::latch`, `std::barrier`, `std::counting_semaphore`, etc.
  - `std::jthread` and `std::stop_*`
- <https://cpp.developpez.com/actu/295043/Le-comite-ISO-Cplusplus-a-propose-une-feuille-de-route-pour-Cplusplus-23-et-finalise-la-nouvelle-version-du-langage-Cplusplus-20-la-norme-devrait-etre-publiee-dans-les-mois-a-venir/>

**<concepts>** : definit des prédicats portant sur un ou plusieurs type, qui peuvent être utiliser dans la définition d'un patron(template)

```
template<class T>
concept integral = is_integral_v<T>;

template<class T>
concept signed_integral = integral<T> && is_signed_v<T>;

template<class T>
concept unsigned_integral = integral<T> && !signed_integral<T>;

template<class T>
concept floating_point = is_floating_point_v<T>;
```

<https://en.cppreference.com/w/cpp/language/constraints>

<https://www.modernescpp.com/index.php/c-20-concepts-predefined-concepts>

# C++20 : Concepte & Contrainte -> Exemple

```
template<typename T>
concept EqualityComparable = requires(T a, T b) {
    { a == b } -> std::boolean;
    { a != b } -> std::boolean;
};
```

```
void f(const EqualityComparable auto&); // constrained function template declaration
```

```
f(42); // OK, int satisfies EqualityComparable
```

# C++20 : Concepte & Contrainte -> Exemple

```
#include <string>
#include <cstdint>
#include <concepts>
using namespace std::literals;

// Declaration of the concept "Hashable", which is satisfied by
// any type T such that for values a of type T,
// the expression std::hash<T>{}(a) compiles and its result is convertible to std::size_t
template<typename T>
concept Hashable = requires(T a) {
    { std::hash<T>{}(a) } -> std::convertible_to<std::size_t>;
};

struct meow {};

template<Hashable T>
void f(T); // constrained C++20 function template

// Alternative ways to apply the same constraint:
// template<typename T>
//     requires Hashable<T>
// void f(T);
//
// template<typename T>
// void f(T) requires Hashable<T>;

int main() {
    f("abc"s); // OK, std::string satisfies Hashable
    f(meow{}); // Error: meow does not satisfy Hashable
}
```

<https://en.cppreference.com/w/cpp/language/constraints>

# C++20 : Concepte & Contrainte -> Exemple

```
template<class T> constexpr bool is_meowable = true;
template<class T> constexpr bool is_cat = true;

template<class T>
concept Meowable = is_meowable<T>;

template<class T>
concept BadMeowableCat = is_meowable<T> && is_cat<T>;

template<class T>
concept GoodMeowableCat = Meowable<T> && is_cat<T>;

template<Meowable T>
void f1(T); // #1

template<BadMeowableCat T>
void f1(T); // #2

template<Meowable T>
void f2(T); // #3

template<GoodMeowableCat T>
void f2(T); // #4

void g(){
    f1(0); // error, ambiguous:
           // the is_meowable<T> in Meowable and BadMeowableCat forms distinct
           // atomic constraints that are not identical (and so do not subsume each other)

    f2(0); // OK, calls #4, more constrained than #3
           // GoodMeowableCat got its is_meowable<T> from Meowable
}
```

<https://en.cppreference.com/w/cpp/language/constraints>

# C++20 : Concepte & Contrainte -> Exemple

```
template <class T>
concept Integral = std::is_integral<T>::value;
template <class T>
concept SignedIntegral = Integral<T> && std::is_signed<T>::value;
template <class T>
concept UnsignedIntegral = Integral<T> && !SignedIntegral<T>;
```

```
template<typename T>
constexpr bool get_value() { return T::value; }

template<typename T>
    requires (sizeof(T) > 1 && get_value<T>())
void f(T); // #1

void f(int); // #2

void g() {
    f('A'); // OK, calls #2. When checking the constraints of #1,
            // 'sizeof(char) > 1' is not satisfied, so get_value<T>() is not checked
}
```

```
template <class T = void>
    requires EqualityComparable<T> || Same<T, void>
struct equal_to;
```

<https://en.cppreference.com/w/cpp/language/constraints>

# C++20 : Concepte & Contrainte -> <type\_traits>

---

La bibliothèque Ranges permet d'appliquer directement des algorithmes sur des collections, sans passer par des itérateurs, dans un style de programmation fonctionnelle. On peut composer les algorithmes grâce à l'opérateur pipe |. On peut aussi appliquer les algorithmes sur des flux infinis de données (streams ou pipelines).

Un `std::range` est un groupe d'éléments sur lequel on peut itérer.

Une `std::view` est une adaptation composable d'un range, qui ne possède pas les données. La fonction d'adaptation est appliquée lorsque l'on itère sur la view.

## Example

Run this code

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

int main()
{
    std::vector<int> ints{0,1,2,3,4,5};
    auto even = [](int i){ return 0 == i % 2; };
    auto square = [](int i) { return i * i; };

    for (int i : ints | std::views::filter(even) | std::views::transform(square)) {
        std::cout << i << ' ';
    }
}
```

Output:

0 4 16

<https://en.cppreference.com/w/cpp/ranges>



Les coroutines sont des fonctions asynchrones, c'est-à-dire qui peuvent suspendre et reprendre leur exécution tout en conservant leur état.

Elles permettent de faire :

- des tâches coopératives ;
  - des boucles d'évènements (event loops) ;
  - des flux de données (streams) infinis ;
  - des pipelines.
- 
- `co_yield` : suspend temporairement l'exécution en renvoyant la valeur
  - `co_return` : termine l'exécution en renvoyant la valeur
  - `co_await` : suspend temporairement l'exécution et rend le contrôle à l'appelant

<https://www.programmez.com/actualites/les-fonctionnalites-de-c20-sont-definies-28649>

<https://www.it-swarm.dev/fr/c++/que-sont-les-coroutines-en-c-20/830706244/>

<http://www.xavierlamorlette.fr/programmation/cpp/cpp20/>

```
Generator<int> getNext(int start = 0, int step = 1){
    auto value = start;
    for (int i = 0;; ++i){
        co_yield value;           // 1
        value += step;
    }
}

int main() {

    std::cout << std::endl;

    std::cout << "getNext():";
    auto gen = getNext();
    for (int i = 0; i <= 10; ++i) {
        gen.next();               // 2
        std::cout << " " << gen.getValue();
    }

    std::cout << "\n\n";

    std::cout << "getNext(100, -10):";
    auto gen2 = getNext(100, -10);
    for (int i = 0; i <= 20; ++i) {
        gen2.next();              // 3
        std::cout << " " << gen2.getValue();
    }

    std::cout << std::endl;

}
```

Les modules permettent de :

- ne plus séparer déclaration et définition, et donc outrepasser les limitations des fichiers d'en-tête ;
- remplacer les directives pré-processeur `#include` ;
- construire plus facilement des packages ;
- accélérer les builds.

# The three-way Comparison operator $\lt\Rightarrow\gt$

Comparaison trilatérale :

- si  $a < b$  alors  $(a \lt\Rightarrow b) < 0$
- si  $a$  équivaut à  $b$  alors  $(a \lt\Rightarrow b) == 0$
- si  $a > b$  alors  $(a \lt\Rightarrow b) > 0$

Cet opérateur doit être une relation d'ordre total (weak ordering), c'est-à-dire vérifiant  $(a == b \parallel a < b \parallel a > b) == \text{true}$ .  $==$  signifie ici que les éléments sont équivalents mais pas forcément identiques.

Avec une relation d'ordre strict total (strict weak ordering),  $==$  signifie que les éléments sont identiques.

Cet opérateur peut être construit automatiquement :

```
auto operator<=>(const A &) = default;
```

# String Literals as Template Parameters

Avant C ++ 20, vous ne pouvez pas utiliser un string comme paramètre d'un template . Avec C ++ 20, vous pouvez l'utiliser.

```
template<std::basic_fixed_string T>
class Foo {
    static constexpr char const* Name = T;
public:
    void hello() const;
};

int main() {
    Foo<"Hello!"> foo;
    foo.hello();
}
```

## New Attributes: `[[likely]]` and `[[unlikely]]`

---

Les deux attributs permettent de donner un indice à l'optimiseur, que le chemin d'exécution soit plus ou moins probable.

```
for(size_t i=0; i < v.size(); ++i){  
    if (unlikely(v[i] < 0)) sum -= sqrt(-v[i]);  
    else sum += sqrt(v[i]);  
}
```

# constexpr and constinit Specifier

Le nouveau spécificateur constexpr crée une fonction immédiate. Pour une fonction immédiate, chaque appel à la fonction doit produire une expression constante de compilation. Une fonction immédiate est implicitement une fonction constexpr .

```
constexpr int sqr ( int n) {  
    return n * n;  
}  
constexpr int r = sqr ( 100 ); // D'ACCORD  
  
int x = 100 ;  
int r2 = sqr (x); // Erreur
```

# std::source\_location

C ++ 11 a deux macros pour `__LINE__` et `__FILE__` pour obtenir les informations lorsque les macros sont utilisées. Avec C ++ 20, la classe `source_location` vous donne le nom de fichier, le numéro de ligne, le numéro de colonne et le nom de fonction sur le code source.

```
#include <iostream>
#include <string_view>
#include <source_location>

void log(std::string_view message,
        const std::source_location& location = std::source_location::current())
{
    std::cout << "info:"
               << location.file_name() << ":"
               << location.line() << " "
               << message << '\n';
}

int main()
{
    log("Hello world!"); // info:main.cpp:15 Hello world!
}
```

# Calendar and Time-Zone

se compose de types, qui représentent une année, un mois, un jour d'un jour de la semaine et un nième jour de la semaine d'un mois. Ces types élémentaires peuvent être combinés à des types complexes tels que par exemple `year_month`, `year_month_day`, `year_month_day_last`, `years_month_weekday` et `year_month_weekday_last`. L'opérateur "/" est surchargé pour la spécification pratique des points temporels. De plus, nous obtiendrons avec C++ 20 nouveaux littéraux: `d` pour un jour et `y` pour un an.

```
auto d1 = 2019y/oct/28;  
auto d2 = 28d/oct/2019;  
auto d3 = oct/28/2019;
```

```
#include "date.h"  
#include <iostream>  
  
int  
main()  
{  
    using namespace date;  
    using namespace std::chrono;  
    auto now = system_clock::now();  
    std::cout << "The current time is " << now << " UTC\n";  
    auto current_year = year_month_day{floor<days>(now)}.year();  
    std::cout << "The current year is " << current_year << '\n';  
    auto h = floor<hours>(now) - sys_days{jan/1/current_year};  
    std::cout << "It has been " << h << " since New Years!\n";  
}
```

```
Start  
The current time is 2017-10-26 02:11:17.930301323 UTC  
The current year is 2017  
It has been 7154h since New Years!  
0  
Finish
```



std::span est une vue sur une séquence contigue d'objets.

Il n'est pas propriétaire de la mémoire. La mémoire contigue peut-être, par exemple, celle d'un tableau C, d'un array ou d'un vector.

`std::string` et `std::vector` peuvent être `constexpr`.

# Designated initializers

```
// aggregateInitialisation.cpp

#include <iostream>

struct Point2D{
    int x;
    int y;
};

class Point3D{
public:
    int x;
    int y;
    int z;
};

int main(){

    std::cout << std::endl;

    Point2D point2D {1, 2};
    Point3D point3D {1, 2, 3};

    std::cout << "point2D: " << point2D.x << " " << point2D.y << std::endl;
    std::cout << "point3D: " << point3D.x << " " << point3D.y << " " << point3D.z << std::endl;

    std::cout << std::endl;

}
```

# Designated initializers

```
// designatedInitializer.cpp

#include <iostream>

struct Point2D{
    int x;
    int y;
};

class Point3D{
public:
    int x;
    int y;
    int z;
};

int main(){

    std::cout << std::endl;

    Point2D point2D {.x = 1, .y = 2};
    // Point2D point2d {.y = 2, .x = 1};           // (1) error
    Point3D point3D {.x = 1, .y = 2, .z = 2};
    // Point3D point3D {.x = 1, .z = 2}           // (2) {1, 0, 2}

    std::cout << "point2D: " << point2D.x << " " << point2D.y << std::endl;
    std::cout << "point3D: " << point3D.x << " " << point3D.y << " " << point3D.z << std::endl;

    std::cout << std::endl;

}
```

# Designated initializers

---

```
class My_class {  
public:  
    int a = 0;  
    int b = 0;  
    int c = 0;  
};  
  
{  
    My_class object{.a = 1, .c = 3};  
}
```

# Various Lambda Improvements

---

```
struct Lambda {  
    auto foo() {  
        return [=] { std::cout << s << std::endl; };  
    }  
  
    std::string s;  
};  
  
struct LambdaCpp20 {  
    auto foo() {  
        return [=, this] { std::cout << s << std::endl; };  
    }  
  
    std::string s;  
};
```

std::format permet de formater les chaînes de caractères.

Cela remplace notamment l'utilisation de iomanip

```
std::string message = std::format("The answer is {}. ", 42);
```

```
fmt::memory_buffer buf;  
format_to(buf, "{}", 42);    // replaces itoa(42, buffer, 10)  
format_to(buf, "{:x}", 42);  // replaces itoa(42, buffer, 16)  
// access the string with to_string(buf) or buf.data()
```

# `std::atomic_ref<T>`

---



# `std::atomic<std::shared_ptr<T>>` and `std::atomic<std::weak_ptr<T>>`

---





# Semaphores, Latches and Barriers

---

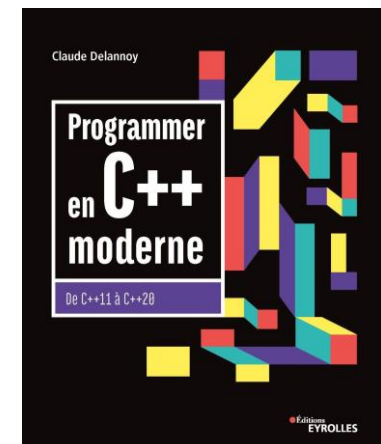
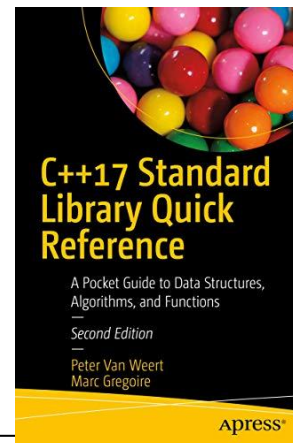
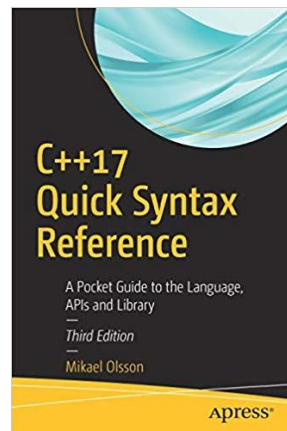
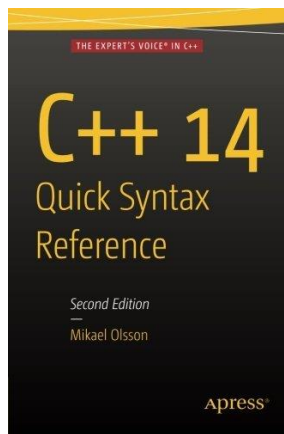
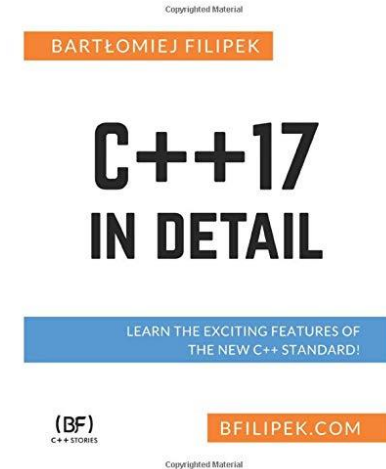
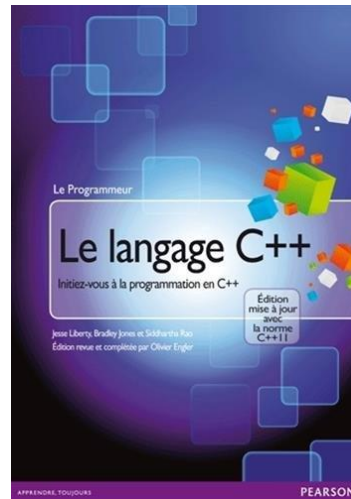
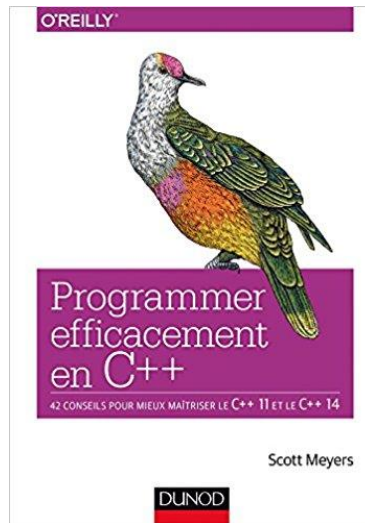


```
auto foo = []<typename T>(const  
std::vector<T> & vector_of_stuff) {  
  
    ...  
  
};
```

- On peut désormais utiliser des méthodes virtuelles dans des expressions constantes.



<https://cpp.developpez.com/redaction/data/pages/users/gbdivers/cpp11/>







- <http://www.cplusplus.com>
- <https://msdn.microsoft.com/fr-fr/library/hh875057.aspx>
- <https://www.developpez.com/>
- <http://fr.cppreference.com/w/cpp/language>
- <https://wikimonde.com/article/C%2B%2B11>
- <http://guillaume.belz.free.fr/doku.php?id=references>
- <https://zestedesavoir.com/tutoriels/474/la-deduction-de-type-en-c/>
- <https://etienne-boespflug.fr/cpp/10-larrivee-du-c17-partie-1-presentation-generale/>
- <http://www.cplusplus2017.info/category/uncategorized/>
- <http://cpp.sh/>
- <https://h-deb.clg.qc.ca/Liens/Caracteristiques-Cplusplus--Liens.html>



