Sommaire principal
__ utilité d'un tel aide mémoire de nos jours ? (internet, autocomplétion...)

Aide mémoire sur la syntaxe du C++

Ce document est destiné à retrouver facilement une syntaxe en C++ et à copier-coller le code d'exemple. Il est organisé selon le plan de l'ouvrage « Le langage C++ » de Bjarne Stroustrup . Les syntaxes spécifiques au C++11 sont indiquées à l'aide du tag [C++11]. Ce document ne donne pas d'explication sur les fonctionnalités présentées, pour cela, veuillez-vous référérer à un cours de C++.

Hello world

```
#include <iostream>
using namespace std ;
int main()
{
    cout << "Salut tout le monde !" << endl;
}
```

Les base du langage C++

Les types

- Entier : int
- Réel : float, double
- Caractères : char, wchar_t, char16_t, char32_t
- Booléen : bool
- Modificateurs : short, long, long long
- Modificateur de signe : [signed], unsigned

Les littérales

Entiers

```
Int main() {
    cout << 123 << endl; // int
    cout << 123L << endl; // long int
    cout << 123LL << endl; // long long int
    cout << 123U << endl; // unsigned int
    cout << 123UL << endl; // unsigned long int
    cout << 123ULL << endl; // unsigned long long int
}
```

Réels

```
int main() {
    cout << 12.34 << endl; // double (64 bits)
    cout << 12.34f << endl; // float (32 bits)
    cout << 12.3L << endl; // long double
    cout << 12.3L << endl; // long double
}
```

Caractères

```
int main() {
    cout << 'a' << endl; // char
    cout << L'a' << endl; // wchar_t
    cout << u'a' << endl; // char16_t
    cout << U'a' << endl; // char32_t
}
```

Booléen

```
int main() {
    cout << boolalpha << true << endl; // char
    cout << boolalpha << false << endl; // wchar_t
}
```

Les variables

Variantes :

```
auto
auto const
auto &
auto const&
```

Autres syntaxe :

```
int i{}; // 0 initialisation
int x(3.141592);
int y = 3.141592;
int x = 10 + 20; // expression évaluée lors de la
compilation
```

Obsolète

```
int x; // toujours initialiser ses variables
```

Modificateurs :

```
unsigned
short
long
long long
```

Les types littérales

```
char tab = '\t'; // tabulation
char endl = '\n'; // fin de ligne
const char* s = "une chaine de caractères";
// les user string litterale [C++11]
```

Remarques

```
char* s1 = "une chaine"; // const char* (compatiblité avec C)
s1[0] = 'U'; // erreur

char s2[] = "une chaine"; // copie de la chaine dans le
```

```
tableau
s2[0] = 'U'; // ok
```

chaîne longue

```
char* s = "abcdef"
          "ghijkl"; // s vaut "abcdzfghijkl"
```

Types réelles

```
float
double d = 123.456e-789;
long double
1.23456789f ou 1.23456789F // littérale float
```

Déclaration dans une condition

```
for (int i=0; i<10; ++i) { }
if (double d = foo()) { /* si d n'est pas nul */ }
```

Opérateur sizeof :

```
sizeof A;
sizeof(A);

1 ≡ sizeof(char) ≤ sizeof(short) ≤ sizeof(int) ≤ sizeof(long)
1 ≤ sizeof(bool) ≤ sizeof(long)
sizeof(char) ≤ sizeof(wchar_t) ≤ sizeof(long)
sizeof(float) ≤ sizeof(double) ≤ sizeof(long double)
sizeof(N) ≡ sizeof(signed N) ≡ sizeof(unsigned N)
```

Obtenir des informations sur un type (http://www.cplusplus.com/reference/std/limits/numeric_limits/)

```
#include <limits>
int m1 = numéric_limits<int>::min();
int m2 = numéric_limits<int>::max();
bool b1 = numéric_limits<int>::in_signed();
bool b2 = numéric_limits<int>::in_integer();
```

Cast

```
static_cast<A>(a); // vérification à la compilation
dynamic_cast<A>(a); // vérification à l'exécution

reinterpret_cast<A>(a); // pas de vérification
const_cast<A>(a); // conversion de const

A a = b; // cast ou conversion implicite
A a(b); // cast ou conversion implicite
A a = (A) b; // cast explicite
```

Modificateur de déclaration

```
const char tab = '\t'; // doit être initialisé lors de la
déclaration
static int i;
mutable double d; // uniquement dans classe ?
extern float f;
```

Constantes

```
const int i = 10;
const int[] v = { 1, 2, 3, 4 };
```

Pointeurs et références constantes

```
const int* p1 = &i; // pointeur vers constante
int* const p2 = &i; // pointeur constant
const int* const p3 = &i; // pointeur constant vers une
constante
int const& r1 = i; // référence vers consante
```

Les énumération

```
enum { ITEM1, ITEM2, ITEM3 }; // ITEM1 = 0, ITEM2 = 1; ITEM3
= 2
enum E1 { ITEM1, ITEM2, ITEM3 }; // ITEM1 = 0, ITEM2 = 1;
```

```
ITEM3 = 2
enum E2 { ITEM1 = 1, ITEM2 = 2, ITEM3 = 4 };
    // ITEM1 = 1, ITEM2 = 2; ITEM3 = 4
E2 e1 = ITEM1; // e == ITEM1
E2 e2 = E(1); // e == ITEM1
E3 e3 = E(1 | 2); // e == E2(3)
// strong enum [C++11]
// constexpr avec flag
```

La définition de type Typedef

```
typedef unsigned char uchar;
```

Les pointeurs

Déclaration

```
void* p1 = 0;
void* p2 = NULL;
void* p3 = nullptr; [C++11]
```

Allocation et initialisation

```
float* p2 = new float;
float* p3 = new float(123.456f);
new (&a) int; // placement
new (&a) A(123); // placement et initialisation
```

Destruction

```
delete p3;
```

Récupérer l'adresse d'une variable

```
int* pi = &i;
```

Manipuler les pointeurs

```
int x = *pi;
++pi;
pi++;
pi + 3;
pi – 3;
```

Cast

```
int* pi;
float* pf = pi; // erreur, conversion implicite interdit
float* pf = static_cast<float>(pi); // conversion explicite,
mais non sur
```

Les pointeurs sur void

```
int* pi;
void* pv = pi; // ok, conversion implicite
*pv; // erreur
++pv; // erreur
int* pi2 = static_cast<int*>(pv); // ok, conversion vers le
type d'origine
```

Surcharge des opérateurs new et delete

```
void* operator new (size_t);
void operator delete (void);
void* operator new[] (size_t);
void operator delete[] (void*);
```

Exception lancée par new

```
try { for (;;) new char[1000]; } // provoque une erreur dès
que la mémoire sature
catch (bad_alloc) { cout << "erreur bad alloc" << endl; }

void foo() { }
set_new_handler(foo);
for (;;) new char[1000]; // provoque une erreur dès que la
mémoire sature => foo()
```

Les références

```
int i = 1;
int& r1 = i; // r est un alias de i
++r1; // i vaut 2 maintenant
extern int& r2; // ok, initialisation ailleurs
```

Intialisation avec une litérrale :

```
int& r3 = 123; // erreur
const int& r3 = 123; // Ok, lvalue
```

2.9. Les tableaux Les tableaux à une dimension

```
int a1[10];
int a2[] = { 1, 2, 3, 4 }; // tableau de 4 éléments
int a3[4] = { 1, 2, 3, 4 };
int a3[8] = { 1, 2, 3, 4 }; // ajout de 4 éléments 0 à la
fin
```

Pointeur sur tableau

```
int v[] = { 1, 2, 3, 4 };
int* p1 = v;
int* p1 = &v[0];
int* p2 = &v[2];
int x = *p2; // x = 3
int* p3 = ++p2; // p3 pointe sur 4
int* p4 = p1 + 2; // p4 pointe sur 3
int d = &v[2] - &v[0]; // d = 2
```

Parcourir un tableau

```
int v[] = { 1, 2, 3, 4 };
for (int i = 0; i < 4; ++i) { int x = v[i]; }
for (int* p = &v[0]; p != &v[4]; ++p) { int x = *p; }
```

Les tableaux à N dimensions

```
int a2[15][20];
```

Tableaux dynamiques

```
int* pi;
int i = pi[i]; // cast
int* table = new[10] int;
table[i];
delete[] table;
```

2.10. Les espaces de nom

```
namespace std {}
using namespace std;
using std::string;

::nom; // espace de nom global
```

Ordre de priorité des opérateurs

Priorité : de haut en bas, et de gauche à droite pour les opérateurs unaires et de droite à gauche pour les binaires.

Directives de compilation

Les directives d'inclusion

```
#include "file/nomfichier.h"
#include <vector> // classe de la STL
```

3.2. Les commentaires

```
// commentaire sur une seule ligne
/* commentaire
sur
```

```
plusieurs
lignes */
```

Fichier header

dans mon_fichier.h #ifndef MONFICHIER_H #ifndef MONFICHIER_H #include <...> liste des inclusion de la STL #include "..." class ...;

... *déclaration des classes et fonctions #endif* MONFICHIER_H Hello world *main.cpp #include <iostream> int main() { std::cout « "Salut tout le monde !" « std::endl; } ===== Les fonctions ===== ==== Précisions sémantiques ==== void f(int i) {}* i est un paramètre f(j); *j est un argument void f();* délaration void f() { } *définition ⇒ doit être unique* signature d'une fonction : nom de la fonction, les types de paramètres *en entrée et les modificateurs (const) int foo (int i, double d) { } double foo (int i, double d) { }* erreur

```
  // le paramètre de retour n'apparatient pas à la signature
```

int foo (int j, double f) { } *erreur* le nom de paramètres n'appartient pas à la signature int foo (int j, double f) const { } *ok* la fonction const et non const n'ont pas les mêmes signatures *static ? Inline ? 4.2. Déclaration d'une fonction void f();* sans paramètre de retour

void f(int i) { *un paramètre en entrée cout « i « endl;* utilisation de i }

void f(int i, int j) { *plusieurs paramètres en entrée cout « i « j « endl;* utilisation de i }

float f() { *retourne une valeur return 123.456; } void* f();* retourne un pointeur

Passage de paramètres

```
int i = 0;
```

```
void f1(int& i) { ++i; } // passage par référence
f1(i);

void f2(int* i) { ++(*i); } // passage par pointeur
f2(i);

int f3(int i) { return i+1; } // passage par copie
i = f3(i);

int f4(int const& i) { return i+1; } // ou f2(const int& i)
i = f4(i];
Attention
int const& f() { }
int& f() { int i = 10; return i; } // Erreur
4.4.  NRVO et RVO
4.5.  Fonction main()
int main() {}
int main(int, char**) {}
int main(int argc, char* argv[]) {
    for (int i=0; i<argc; ++i) {
        char* arg = argv[i];
    }
}
```

Fonction static

```
static void foo() {}
```

Fonction variadique

```
void f(... args) {
}
```

Les pointeurs de fonctions

```
int (f*) (int a);
```

Les classes et structures

Syntaxe de bases

```
struct A { int x, y; }; // définition de la structure
A a; // déclaration de la variable a de type A
struct A { int x, y; } a; // définition et déclaration de A
et a
Déclaration anticipée (forward déclaration)
stuct B; // déclaration anticipée
struct A { B b; }; // déclaration de A
struct B {}; // déclaration de B

// Nom de structure utilisable tant qu'on n'a pas besoin de
la taille ou
// du nom d'un membre
struct A; // déclaration
extern A a; // ok
A f(A a); // ok
A* g(A* a); // ok
A a; // erreur
a.x; // erreur
f(); // erreur
g(a); // erreur
```

Forme canonique de Coplien

```
struct A {
    A() {} // constructeur par défaut
    ~A(); // destructeur
    A(A const& a); // constructeur par copie
    A& operator= (A const& a); // opérateur d'affectation
    A(A&& a); // constructeur par déplacement [C++11]
```

```
    A& operator= (A&& a); // affectation par déplacement
[C++11]
};
```

Constructeurs

```
struct A {
    A(); // constructeur par défaut
    A(B const& b); // constructeur à un argument
    A(B const& b, C const& c); // constructeur à deux
arguments

    A(int);
    A() : A(42) {} // constructeur délégué [C++11]
};
A a; // construction de a par défaut
A a2(a); // constructeur par copie
A a3 = a; // opérateur de copie
```

Initialisation

```
struct A { int x; };
A a1; // initialisation avec les valeurs par défaut

struct A {
    int x;
    A() : x(10) {}
};
A a; // a.x = 10

class A {
    static const int m1 = 7; // ok
    const int m2 = 7; // [C++11]
    static int m3 = 7; // erreur : non constant
    static const int m4 = var; // erreur : n'est pas une
expression constante
    static const string m5 = "odd"; // erreur : n'est pas un
type intégral
    std::string s{"Constructor run"}; // [C++11]
    int x {5}; // [C++11]
    int y { 2 * x }; // [C++11]
    std::string id = { defaultID() }; // [C++11]
```

```
};

// Tableau d'initialisation

struct A { int x, y, z; };
A a = { 1, 2, 3 };
```

Accès aux variables et fonctions membres

```
A a;
a.x;
a.f();

A* a;
a->x; // ou (*a).x;
a->f(); // ou (*a).f();

struct A { int* pi; } a, *pa;
a.*pi;
pa->*pi;
```

Conversion

```
struct A {
    A(B const& b); // conversion implicite de B vers A
    explicit A(B const& b); // conversion explicite de B
vers A
    operator B() const; // conversion implicite de A vers B
    explicit operator B() const; // conversion implicite de
A vers B [C++11]
};
```

Les variables membres

```
a.x; // accès
```

Fonctions membres

```
a.x; // accès
```

Opérateurs

```
Foncteurs (objets fonctions)
struct A {
    void operator() (); // foncteur sans argument
    void operator() (B const& b); // foncteur à un argument
    void operator() (B const& b, C const& c); // foncteur à
deux argments
};
```

Déréfencement

```
class PtrA { // classe pointeur sur A
    A* p;
public:
    A* operator-> () { return p; }
    A& operator* () { return *p; }
    A& operator[] (int i) { return p[i]; }
    // operator. interdit !
};
```

Incrémentation et décrémentation

```
struct A {
    A& operator++ (); // préfixe : ++a
    A operator++ (int); // postfixe : a++
    A& operator-- (); // préfixe : --a
    A operator-- (int); // postfixe : a--
};
```

Héritage

Héritage simple

struct A {}; struct B1 : A {}; *héritage privé struct B : public A {};* héritage publique struct B : protected A {}; *héritage protected struct B : private A {};* héritage privé

Héritage multiple

struct A {}; struct B {}; struct C : public A, public B {}; *hérite de A et B Constructeur struct A {}; struct B1 : public A { B() : A() {} }; Fonctions virtuelles struct A { virtual ~A();* obligatoire si au moins une fonction virtuelle

```
  virtual f() = 0; // fonction virtuelle pure
  virtual f(); // fonction virtuelle
```

};

Classe abstraite

struct A { *ne possède que des fonctions virtuelles pures virtual f() = 0;* fonction virtuelle pure };

Les structures de contrôle

break, continue, return

if

if (condition) statement if (condition) { statement } if (condition) statement else statement

switch

swicth (key) {

```
  case ITEM1 : (…) break;
  case ITEM2 : (…) break;
  default : (…)
```

}

while et do

while (condition) statement do f() while (condition) do { ... } while (condition)

for

int v[] = { 1, 2, 3, 4 }; for (;;) {} *forever for (int i = 0; i < 4; ++i) { int x = v[i]; } for (int* p = &v[0]; p != &v[4]; ++p) { int x = *p; } ==== try et catch ==== try {} handler_list ==== goto ==== goto identifier; identifier : statement ===== La surcharge des opérateurs ===== void* operator new(size_t); void operator delete(void*); void* operator new[](size_t); void operator delete[](void*); ===== Les templates ===== Les fonctions template template<class T> f() {} Les classes template template<class T> class A {}; ===== Les exceptions ===== try { ... } catch (bad_alloc) { ... } ===== La bibliothèque standard ===== Utiliser une classe de la STL directement : std::vector<int> v; Utiliser une classe de l'espace de nom std : using std::vector; vector<int> v; Utiliser toutes les classes de l'espace de nom std : using namespace std; vector<int> v; Quelques fichiers de la bibliothèque standard <iostream> <cctype> ⇒ fonction isalpha(), etc. ==== Les conteneurs ==== === Les séquences === == vector == vector<int> v1;* vecteur de 0 élément vector<int> v2(100); *vecteur de 100 éléments initialisés avec 0 vector<int> v3(100, -1);* vecteur de 100 éléments initialisés avec -1 int i = v[0]; *copie int& i = v[0]; int* p = &v[0];* pointeur vers le premier élément

list

deque

stack

queue

priority_queue

Les conteneurs associatifs

map

map<string, double> table; table[“pi”] = 3.14159265358979;

multimap

set

multiset

Les autres conteneurs

bitset

valarray

C++11

unordered_set<T> unordered_multiset<T> unordered_map<T> unordered_multimap<T> forward_list<T> array<T, N>

Créer son propre conteneur

Les algorithmes

Les algorithmes non modifiant

10.2.1.1. foreach

10.2.1.2. find

10.2.1.3. find_if

10.2.2.14. generate_n

10.2.2.15. remove

10.2.2.16. remove_if

10.2.2.17. remove_copy

10.2.2.18. remove_copy_if

10.2.2.19. unique

10.2.2.20. unique_copy

10.2.2.21. reverse

10.2.2.22. reverse_copy

10.2.2.23. rotate

10.2.2.24. rotate_copy

10.2.2.25. random_shuffle

C++11

bool all_of(Iter first, Iter last, Pred pred) bool any_of(Iter first, Iter last, Pred pred) bool none_of(Iter first, Iter last, Pred pred) Iter find_if_not(Iter first, Iter last, Pred pred) OutIter copy_if(InIter first, InIter last, OutIter result, Pred pred) OutIter copy_n(InIter first, Size n, OutIter result) uninitialized_copy_n(InIter first, Size n, OutIter result) OutIter move(InIter first, InIter last, OutIter result) OutIter move_backward(InIter first, InIter last, result) s is_partitioned(InIter first, InIter last, Pred pred) pair<OutIter1, OutIter2> partition_copy(InIter first, InIter last, OutIter1 out_true, OutIter2 out_false, Pred pred) Iter partition_point(Iter first, Iter last, Pred pred) RAIter partial_sort_copy(InIter first, InIter last, RAIter result_first, RAIter result_last) RAIter partial_sort_copy(InIter first, InIter last, RAIter result_first, RAIter result_last, Compare comp) bool

is_sorted(Iter first, Iter last) bool is_sorted(Iter first, Iter last, Compare comp) Iter is_sorted_until(Iter first, Iter last) Iter is_sorted_until(Iter first, Iter last, Compare comp) bool is_heap(Iter first, Iter last) Vrai si [first, last) bool is_heap(Iter first, Iter last, Compare comp) Iter is_heap_until(Iter first, Iter last) Iter is_heap_until(Iter first, Iter last, Compare comp) T min(initializer_list<T> t) T min(initializer_list<T> t, Compare comp) T max(initializer_list<T> t) T max(initializer_list<T> t, Compare comp) pair<const T&, const T&> minmax(const T& a, const T& b) pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp) pair<const T&, const T&> minmax(initializer_list<T> t) pair<const T&, const T&> minmax(initializer_list<T> t, Compare comp) pair<Iter, Iter> minmax_element(Iter first, Iter last) pair<Iter, Iter> minmax_element(Iter first, Iter last, Compare comp) void iota(Iter first, Iter last, T value) 10.3. Iterateurs et allocateurs

Les chaînes de caractères

Opérations sur les caractères

Char c = ' '; bool t = isspace©; *test si est une espace bool t = isdigit©;* test si est un chiffre bool t = isalpha©; *test si est une lettre === Manipuler les chaînes de caractères === bool t = isalnum©;* test si est un chiffre ou une lettre string s; s = “une chaine de caractères”; string s2 = “une autre chaine”; 10.4.3. manipuler les chaînes style C strlen() strcpy() strcmp()

Les flux

Flux de sortie

cout « “un message” « endl; cerr « “une erreur” « endl; clog « “une information” « endl;

Flux d'entrée

cin » i; *lecture char c; cin.get©; === Flux sur les chaines de caractères === #include <sstream> istream* input; input = new istringstream(argv[1]) ; === Flux sur les fichiers === ifstream ===== Les fonctions numériques ==== fsqrt() ===== A trier ===== typeid(int);* évaluation à la compilation typeid(a); *évaluation à l'éxécution extern namespace*