

Formation C++

Aimery Tauveron--Jalenques

Septembre 2018

ViaRézo

Introduction

C++ est un langage de programmation compilé permettant la programmation sous de multiples paradigmes (comme la programmation procédurale, orientée objet ou générique). [...]

Créé initialement par Bjarne Stroustrup dans les années 1980, le langage C++ est aujourd'hui normalisé par l'ISO. Sa première normalisation date de 1998, ensuite amendée par l'erratum technique de 2003. Une importante mise à jour a été ratifiée et publiée par l'ISO en septembre 2011 sous le nom de C++11. Depuis, des mises à jour sont publiées régulièrement : en 2014 puis en 2017.

Extrait adapté de Wikipédia

Ce n'est pas :

- une formation C (pas de `(int*)malloc(...)`),
- une formation compilation,
- une formation POO,
- un tutoriel.

C++ 101

Programmation orientée objet en C++

Le C++ moderne

Introduction à la librairie standard

C++ 101

```
#include <iostream>

int main () {
    std::cout << "Hello world!\n";
    return 0;
}
```

Il faut compiler !

- GCC, Clang, Microsoft Visual C++

```
g++ hello.cpp
```

On peut exécuter :

```
./a.out  
Hello world!
```



```
g++ -O3 -Wall hello.cpp -o hello
```

C'est mieux!

```
./hello  
Hello world!
```

```
#include <iostream>

int main () {
    int a;
    int b (0);
    int c = 0;
    std::cout << "a is: ";
    std::cin >> a;
    b = 2*a;
    std::cout << "a is: " << a << "\n";
    std::cout << "b is: " << b << "\n";
    std::cout << "2*a is: " << 2*a << "\n";
    return 0;
}
```

Des types :

- numériques : `int` (unsigned, long, short), `float`, `double`
- non-numériques : `char`, `bool`
- `void`
- autres (objets...)

Trois catégories de variables :

- variables : `type var`
- références : `type &var`
- pointeurs : `type *var`

Un qualificateur : `const`

```
int main () {  
    int a (2);  
    int b (a);  
    int &c (a);  
    int *d = a;  
    std::cout << "a is: " << a << "\n";  
    std::cout << "b is: " << b << "\n";  
    std::cout << "c is: " << c << "\n";  
    std::cout << "d is: " << d << "\n";  
    return 0;  
}
```

```
int main () {  
    int a (2);  
    int b (a);  
    int &c (a);  
    int *d = &a; // address of a  
    std::cout << "a is: " << a << "\n";  
    std::cout << "b is: " << b << "\n";  
    std::cout << "c is: " << c << "\n";  
    std::cout << "d is: " << d << "\n";  
    return 0;  
}
```

```
int main () {  
    int a (2);  
    int b (a);  
    int &c (a);  
    int *d = &a; // address of a  
    std::cout << "a is: " << a << "\n";  
    std::cout << "b is: " << b << "\n";  
    std::cout << "c is: " << c << "\n";  
    std::cout << "d is: " << *d << "\n"; /* the  
        value pointed to by d */  
    return 0;  
}
```

```
int a [8];  
char w [256];  
a[0] = 0;  
a[1] = 4;  
std::cout << a[0];
```

- Hérités de C
- Indexés de 0 à $n - 1$
- Des pointeurs \implies pas de vérification des indices

```
int *b = a; // Correct  
a[11] = 32; // Undefined behaviour
```

- Création à la déclaration
- Destruction à la fin du bloc {...}

```
int main () {  
    // maybe some code  
    int a (0);  
    unsigned int b;  
    {  
        double c (.5);  
    }  
    // c doesn't exist here  
    return 0;  
}
```


Deux couples d'opérateurs : new et delete, malloc et free

```
int *a = new int(2);  
delete a;  
a = nullptr;
```

Pour les tableaux : new[] et delete[]

```
int *a = new int[4];  
delete[] a;  
a = nullptr;
```

```
int main () {  
    int *a;  
    {  
        int *b = new int(4);  
        a = b;  
    }  
    // b doesn't exist here  
    // but a does  
    std::cout << "a is: " << *a << "\n";  
    delete a;  
    a = nullptr;  
    return 0;  
}
```

```
if ( condition ) {  
    // code  
} else if ( condition ) {  
    // code  
} else {  
    // code  
}
```

```
if ( condition )  
    // single line of code  
else  
    // single line of code
```

```
while ( condition ) {  
    // code  
}
```

```
for ( initialisation ; condition ; increase ) {  
    // code  
}
```

```
for ( int i=0 ; i<10 ; ++i ) {  
    // code  
}
```

```
for ( n=0, i=100 ; n!=i ; ++n, --i ) {  
    // code  
}
```

```
switch ( expression ) {  
    case constant:  
        // code  
        break;  
    case constant:  
        // code  
        break;  
    default:  
        //code  
}
```

Définition :

```
return_type function_name (arguments) {  
    // code  
    return return_value;  
}
```

Appel :

```
variable = function_name (arguments);
```

Surcharge de fonctions possible

Trois manières de passer un argument :

- par valeur

```
int f (int a) {...}
```

- par référence

```
int f (int &a) {...}
```

- par pointeur

```
int f (int *a) {...}
```



```
int main (int argc, char **argv) {  
    // code  
    return 0;  
}
```

ou bien

```
int main (int argc, char *argv[]) {  
    // code  
    return 0;  
}
```

Dans le désordre :

- arithmétiques : `+`, `-`, `*`, `/`, `%`
- de comparaison : `==`, `!=`, `<`, `>`, `<=`, `>=`
- logiques : `&&`, `||`, `!`
- binaires : `&`, `|`, `^`, `~`
- mémoire : `&`, `*`

En C++, le contexte est important !

Pourquoi `std::cout` et non `cout` ? `int` et non `std::int` ?

int est dans le namespace global, `cout` dans le namespace `std`.

`std` contient la bibliothèque standard.

On peut utiliser :

```
using namespace std ;

int main () {
    cout << "Hello world!\n";
    return 0;
}
```

```
namespace foo {  
    bool var (true);  
}  
namespace bar {  
    double var (1.0);  
}  
int main () {  
    std::cout << "foo::var is:" << foo::var << "\n";  
    std::cout << "bar::var is:" << bar::var << "\n";  
    return 0;  
}
```

Programmation orientée objet en C++

string : chaîne de caractères améliorée

```
#include <string>

int main () {
    std::string str ("some");
    std::string str2 ("text");
    str.append(str2); // str = str + str2;
    const char *cstr (str.c_str());
    return 0;
}
```

```
struct atom {  
    int atomic; // Atomic number  
    int mass; // Mass number  
};
```

Membres d'une structure :

- variables (attributs),
- fonctions (méthodes)

```
atom carbon12;  
carbon12.atomic = 6;  
carbon12.mass = 12;  
  
atom *oxygen16 = new atom;  
oxygen16->atomic = 8;  
oxygen16->mass = 16;
```



```
class Atom {  
private:  
    int m_atomic;  
    int m_mass;  
public:  
    Atom(int a, int m) {  
        m_atomic = a; m_mass = m;  
    }  
};  
  
Atom carbon12 (6,12);  
carbon12.m_mass = 13; // Forbidden!
```

Les attributs d'une classe sont toujours **privés**.

```
class A {  
private:  
    // attributes  
    // private methods  
public:  
    ~A (...) {...} // destructor  
    A (const A &a) {...} // copy constructor  
    A& operator= (const A &a) {...} // copy  
        assignment operator  
    // public methods  
};
```

```
class GaussInt {  
private:  
    int m_real;  
    int m_imag;  
public:  
    GaussInt (int a=0, int b=0): m_real(a),  
        m_imag(b) {}  
    ~GaussInt () {}  
    GaussInt (GaussInt &a): m_real(a.m_real),  
        m_imag(a.m_imag) {}  
    double norm () { return std::sqrt(m_real*  
        m_real + m_imag*m_imag); }  
};
```

```
int main () {  
    GaussInt a;  
    GaussInt b (4,3);  
    GaussInt c (b);  
    GaussInt d = a;  
    std::cout << "Norm of c: " << c.norm() << std  
        ::endl;  
    std::cout << "Norm of d: " << d.norm() << std  
        ::endl;  
    return 0;  
}
```

- Définition implicite du constructeur de copie, de l'opérateur d'affectation et du destructeur
- Copie membre à membre

```
class GaussInt {  
    // ...  
public:  
    GaussInt (int a=0, int b=0): m_real(a),  
        m_imag(b) {}  
    double norm () { return std::sqrt(m_real*  
        m_real + m_imag*m_imag); }  
};
```

Implémentation seulement si nécessaire !

- Surcharger les opérateurs
- Accès aux membres : `private`, `protected` et `public`
- Héritage (multiple)
- Méthodes `const`, attributs `static` et mutable
- Fonctions et classes amies (`friend`)

- Fichiers *header* (extension `h` ou `hpp`) : déclarations
- Fichiers source (extension `cpp`) : définitions

On inclut des fichiers *header* dans les fichiers source ou dans d'autres fichiers *header*.


```
#ifndef GaussInt_included
#define GaussInt_included
class GaussInt {
private:
    int m_real;
    int m_imag;
public:
    GaussInt (int a=0, int b=0);
    ~GaussInt ();
    GaussInt (const GaussInt &);
    GaussInt& operator= (const GaussInt &);
    double norm ();
};
#endif
```

```
#include "GaussInt.h"
#include <cmath>
GaussInt::GaussInt (int a=0, int b=0): m_real(a
    ), m_imag(b) {}
GaussInt::~GaussInt () {}
GaussInt::GaussInt (const GaussInt &a): m_real(
    a.m_real), m_imag(a.m_imag) {}
```

```
GaussInt& GaussInt::operator= (const GaussInt &
    a) {
    m_real = a.m_real; m_imag = a.m_imag;
    return *this;
}
double GaussInt::norm () {
    return std::sqrt(m_real*m_real + m_imag*
        m_imag);
}
```

```
#include <iostream>
#include "GaussInt.h"
int main () {
    GaussInt a;
    GaussInt b (4,3); GaussInt c (b);
    GaussInt d; d = c;
    std::cout << "Norm of a: " << a.norm() << std
        ::endl;
    std::cout << "Norm of c: " << c.norm() << std
        ::endl;
    std::cout << "Norm of d: " << d.norm() << std
        ::endl;
    return 0;
}
```

```
g++ -Wall -c GaussInt.cpp -o GaussInt.o  
g++ -Wall -c main.cpp -o main.o  
g++ -Wall main.o GaussInt.o -o gauss.out
```

On peut exécuter :

```
./gauss.out  
Norm of a: 0  
Norm of c: 5  
Norm of d: 5
```

```
template <typename T> bool isPositive (T a) {  
    if (a>=0) {  
        return true;  
    } else {  
        return false;  
    }  
}
```

Mot-clé : typename ou class

```
int n = 4;  
double x = 2.5;  
isPositive(n);  
isPositive(x);  
isPositive<double>(3.0);
```

```
template <typename T> class complex {  
private:  
    T m_real;  
    T m_imag;  
public:  
    complex (const T &a = T(), const T &b = T());  
    ~complex();  
    complex (const complex<T> &);  
    complex& operator= (const complex<T> &);  
    double norm ();  
};
```



```
complex<double> z (3, 5.5);  
complex<int> a;  
complex<int> b (0, 4);
```

- Des patrons de classe ou de fonction pour générer automatiquement du code
- Plusieurs paramètres, avec des valeurs par défaut
- Spécialisation de *templates*

Déclaration et définition **dans le même fichier** \implies dans un *header*

- Fuites : mémoire, fichiers, connections. . .
- gdb, valgrind. . .
- En cas d'interruption du programme (par exemple exception) ?

- RAI : Ressource Acquisition Is Initialisation
 - Encapsulation des ressources dans des objets
- Initialisation dans le constructeur, nettoyage dans le destructeur
- Appel du destructeur garanti par le langage
- Exception renvoyée par un destructeur : *undefined behaviour*

Le C++ moderne

1972 : première version de C

1983 : première version de C++ (alors C with classes)

1989 : premier standard pour C (ANSI C)

1998 : premier standard pour C++

2003 : norme C++03

2011 : norme C++11 ← début du C++ moderne

2014 : norme C++14

2017 : norme C++17

```
---std=c++11
```

```
std::array<int,5> a;  
a[1] = 2;  
size_t a_size = a.size();  
  
a[6] = 4; // runtime error
```

```
std::vector<int> a;  
a.push_back(10);  
a[0];  
  
std::vector<int> b (5);  
b.pop_back();  
size_t b_size = b.size(); // now 4
```



```
std::array<int,5> a;  
// code  
for (int &it : a) {  
    // code  
}
```

range for pour tous les itérables

- begin et end sur l'objet
- operator!=, operator++ et operator* sur l'itérateur

auto : inférence de type

```
for (auto &iterator : iterable) {  
    // code  
}
```

Que se passe-t-il si j'exécute le code ci-dessous ?

```
std::string f () {  
    // code  
}  
  
int main () {  
    std::string x, y;  
    // put some text in x and y  
    std::string a(x);  
    std::string b (x+y);  
    std::string c (f());  
    return 0;  
}
```

Que se passe-t-il si j'exécute le code ci-dessous ?

```
std::vector<std::string> f () {  
    // return a large vector of long strings  
}  
  
int main () {  
    std::vector<std::string> a (f());  
    return 0;  
}
```

Pourquoi faire une copie ?

⇒ Indiquer au compilateur que l'objet affecté est détruit immédiatement après !

- Possible depuis C++11
- Utilisation automatique
- Utilisation manuelle avec `std::move`

```
new_var = std::move(old_var)
```

```
class A {  
    // stuff  
public:  
    ~A ();  
    A (const A &);  
    A (A &&); // move constructor  
    A& operator= (const A &);  
    A& operator= (A &&); // move assignment  
    operator  
    // more stuff  
};
```

Autant que possible, utiliser les méthodes par défaut !


```
class blob {  
private:  
    size_t m_size;  
    char *m_data;  
public:  
    blob () : m_size(0), m_data(nullptr) {}  
    ~blob () { delete[] m_data; }  
};
```

```
blob (const blob& other): m_size(other.m_size)
{
    // copy constructor
    if (m_size) {
        m_data = new char[m_size];
        memcpy(m_data, other.m_data, m_size);
    } else {
        m_data = nullptr;
    }
}
```

Avec *copy-and-swap* :

```
friend void swap (blob& first , blob& second) {  
    using std::swap;  
    //  
    swap(first.m_size , second.m_size);  
    swap(first.m_data , second.m_data);  
}  
  
blob (blob&& other): blob() {  
    // move constructor  
    swap(*this , other);  
}
```

```
blob& operator= (blob other) {  
    swap(*this, other);  
    return *this;  
}
```

- Ressource (encapsulée dans un objet) « unique »
- Interdire la copie ?

```
class A {  
    // stuff  
public:  
    ~A ();  
    A (const A &) = delete;  
    A (A &&) = delete;  
    A& operator= (const A) = delete;  
};
```

delete utilisable aussi avec l'héritage

Impossible d'utiliser une *factory* ?

```
class A {  
    // stuff  
public:  
    ~A ();  
    A (const A &) = delete;  
    A (A &&);  
    A& operator= (const A);  
};
```

```
int main () {  
    A a = factory_for_A();  
    A b, c;  
    b = a; // Forbidden  
    c = std::move(a); // a may end up unusable  
    return 0;  
}
```

```
void f (int i) {  
    std::cout << i << ' '  
}  
  
int main () {  
    std::vector<int> vec;  
    vec.push_back(10);  
    // ...  
    for_each(vec.begin(), vec.end(), f);  
    std::cout << '\n';  
    return 0;  
}
```



```
int main () {  
    std::vector<int> vec;  
    vec.push_back(10);  
    // ...  
    for_each(vec.begin(), vec.end(), [](int i) {  
        std::cout << i << ' ';  
    });  
    std::cout << '\n';  
    return 0;  
}
```

```
[](double x) -> double {  
    if (x>0) {  
        return 0.1*x;  
    } else {  
        return 0;  
    }  
}
```

Capture de variables :

- [*var*] pour capturer par valeur
- [&*var*] pour capturer par référence
- [=] pour capturer toutes les variables par valeur
- [&] pour capturer toutes les variables par référence
- [=, &*var*]
- [&, *var*]

- `shared_ptr`
- `unique_ptr`

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

int main () {
    std::shared_ptr<int> foo (new int(5));
    std::unique_ptr<int> bar (new int(3));
    std::cout << *foo << ' ' << *bar << '\n';
    return 0;
}
```

Que se passe-t-il si j'exécute le code suivant ?

```
int main () {  
    int *ptr = new int(5);  
    std::shared_ptr<int> foo (ptr);  
    std::cout << *foo << '\n';  
    {  
        std::shared_ptr<int> bar (ptr)  
    }  
    std::cout << *foo << '\n';  
    return 0;  
}
```

Ne pas initialiser de pointeur intelligent à partir d'un pointeur classique !

Ou mieux, utiliser :

- `shared_ptr` avec `make_shared`
- `unique_ptr` avec `make_unique` (C++14)

```
int main () {  
    std::shared_ptr<int> foo = make_shared<int>  
        >(5);  
    {  
        std::shared_ptr<int> bar (foo);  
    }  
    std::unique_ptr<int> foobar = make_unique<  
        double>(3.14);  
    std::cout << *foo << ' ' << *foobar << '\n';  
    return 0;  
}
```

Introduction à la librairie standard

Vous rappelez-vous du *namespace* `std` ?

Des directives `#include <...>` ?

- `string` pour gérer les chaînes de caractères
- `iostream` pour les flux standards d'entrée et de sortie
- `fstream` pour les opérations d'entrée et de sortie sur des fichiers
- `cmath` pour les mathématiques
- `memory` pour les pointeurs intelligents

- `array`, `vector`
- `queue`, `dequeue`
- `list`, `forward_list`
- `map`, `unordered_map`
- `set`, `unordered_set`
- `stack`

- `thread`
- `atomic`
- `mutex`, `condition_variable`
- `future` pour l'asynchronisme

```
#include <iostream>
#include <exception>

void f () { throw std::runtime_error("an error
    occured in function f"); }
int main () {
    try {
        f(); std::cout << "No error\n";
    } catch (std::exception &e) {
        std::cout << "Error detected: " << e.what()
            << '\n';
    }
    return 0;
}
```

Conclusion

Des ressources pour plus d'informations :

- cplusplus.com
- cppreference.com
- Stack Overflow

Une série d'articles intéressante : Modern C++ for C Programmers