Programme simple en C++

```cpp
#include <iostream>

using namespace std;

int main()
{
  cout << "Hello world" << endl;
  return 0;
}
```

- `std` Namespace standard
- `cout` Flux de sortie
- `<<` Opérateur d'insertion
- `::` "Scope resolution operator" (opérateur de portée)
  - `class_name::identifier`
  - `namespace::identifier`

Boolean

En C++ on a le type boolean

```cpp
bool isOk = true;
isOk = false;
```

Conversion implicite `int` / `pointer`
`true` -> `1` (`int`)
`true` -> `0` (`int`)

`0` ou `nullptr` -> `false`
`!=0` ou `!=nullptr` -> `true`

> En pratique on n'utilise pas ces conversion implicite
> `if (x != 0)` plutôt que `if (x)`

Namespace

headers

```cpp
namespace X
{
```

```
  class MyClass {}
}

// Nested
namespace A::B
{
  class MyClass {}
}
```

cpp

```
namespace X
{
  MyClass::MyClass() {}
}

// Nested
namespace A::B
{
  MyClass::MyClass() {}
}
```

main.cpp

```
// includes ...

using namespace std;
using namespace A::B;

main()
{
  X::MyClass x = new X::MyClass();
  MyClass ab = new MyClass();
  cout << x;

  return 0;
}
```

IO

On utilise les opérateurs
>> *Extraction* (`cin`)
<< *Insertion* (`cout`)

`cin`

Entrée de flux de données

```
int n;

cin >> n;
```

cout

Sortie de flux de données

```
int n;

cin >> n;
cout << "val " << n;
```

Manipulateur d'IO

`endl`
Fin de ligne

`dec`, `oct`, `hex`
Changement de base

`uppercase`, `nouppercase`
Changement de casse

`showpose`, `noshowpos`
Force l'affichage du signe `+`

`boolalpha`, `noboolalpha`
Affichage booléen (persistant)

```
int i = 10;

cout << nouppercase << hex << i << endl;
cout << uppercase << hex << i << endl;
cout << showpos << dec << i << endl;
```

<iomanip>

Les manipulateurs qui prennent des arguments font partis de la librairie `<iomanip>`

`setprecision(int)`
Nombre de chiffres significatifs

`fixed`
Représentation à virgule fixe

`scientific`
Représentation scientifique

`defaultfloat`
Représentation par défaut

`setw(int)`
Nombre de caractères utilisés

`setfill(char)`
Caractère de remplissage

Condition avec >>

L'expression `(cin >> integer)` est une référence sur le flux.
Elle est nulle (`false` ou `0`) en cas d'échec.

```
int n;

if (!(cin >> n))
{
  cout << "Error"
}
```

Contrôle de buffer

`cin.fail()`
Problème dans le flux d'entrée, saisie incorrecte

`cin.clear()`
Remet les bots de contrôle de flux à OK

`cin.eof()`
Fin du flux d'entrée

Remise en état du buffer

Après une erreur on remet en état le buffer avec la méthode `ignore()` de `cin`

```
#include <limits>
#include <iostream>

// ...

// numeric_limits<streamsize>::max()
```

```
// retourne la taille max du buffer
cin.ignore(numeric_limits<streamsize>::max(), '\n');
```

Exemple d'IO

```
int n;
cout << "Entrez un chiffre entre 1 et 6: ";

while (!( cin >> n ) || n < 1 || n > 6)
{
  if (cin.fail())
  {
    cout << "Saisie incorrecte, recommencez: ";
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
  }
  else
  {
    cout << "Le chiffre n'est pas entre 1 et 6: ";
  }
}
```

Passage de paramètres

Référence &

C'est une alternative aux pointeurs pour manipuler des adresses.

Le seul endroit où l'adresse d'une référence peut être modifié est à sa déclaration

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

// r est une référence sur i
int &r = i;

// i = 0
r = j;

// i == 10
```

Différents passages de paramètres

Par valeur

```
int value;

void fn(int arg)
{
  // arg est une copie de value
}

fn(value)
```

Par pointeur-adresse

```
int value;

void fn(int *arg)
{
  // arg est un pointeur sur value
}

fn(&value)
```

Par référence-adresse

Si `arg` est modifié alors `value` l'est aussi

> Fonctionne un peu comme l'exemple si dessus à mais on a pas besoin de déréférencer `arg` (`*arg`) pour accéder à sa valeur

```
int value;

void fn(int &arg)
{
  // arg est une référence sur value
}

fn(value)
```

Par référence sur une constante

```
int value;

void fn(const int &arg)
{
  // arg est une référence constante sur value (non modifiable)
}

fn(value)
```

Le symbole &

Il a 3 significations dépendant du contexte

Déclaration d'une référence

```
int &r = i
```

Récupération de l'adresse d'une variable

```
p = &i
```

AND bit à bit

```
z = a&b
```

Valeur par défaut des paramètres

Les paramètres avec une valeur par défaut vont en dernier dans la liste des paramètres

```
void fn(int a, int b, int c = 10, int d = 100);

fn(10, 10);
fn(10, 10, 20);
fn(10, 10, 20, 30);
```

Surcharge de fonctions

On peut recharger une fonction avec le même nom mais une signature différente

```
int minimum(int a, int b)
{
  return a < b ? a : b;
}

int minimum(int a, int b, int c)
{
  return minimum(a, minimum(b, c));
}
```

Allocation dynamique

Equivalent de `malloc`

L'équivalent de `malloc` en C est le mot clé `new`, qui renvoi un pointeur

```
double *ptr = new double;
```

Equivalent de `free`

L'équivalent de `free` en C est le mot clé `delete`, qui renvoi un pointeur

```
double *ptr = new double;
delete ptr;
```

Attention aux tableaux

Si c'est un tableau on doit préfixer la nom de la variable par `[]` lors d'un `delete`

```
int *arr = new int[10];
delete []arr;
```

Erreur d'allocation

En cas d'échec de `new`, l'exception `std::bad_alloc` est levée

L'opérateur `new(std::nothrow)` renvoi un `nullptr` en cas d'échec

```
ptr = new(nothrow) int[10];

if (ptr == nullptr)
{
  //...
}
```

Le type `string`

Les opérateurs suivants sont utilisables:
`= + += << >> []`

Les méthodes suivantes permettent de manipuler ces `string`

- `myString.capacity()`
- `myString.size()`
- `myString.find()`
- `myString.erase()`

- `myString.insert()`
- ...

> Est dans le namespace `std`

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

using namespace std;

int main()
{
  string x = "toto";
  string y = x;
  string z = x + "_" + y;

  cout << x << " " << y << " " << z << endl;
  cout << "capacity: " << z.capacity() << endl;
  cout << "size: " << z.size() << endl;

  return 0;
}
```

```
toto toto toto_toto
capacity: 16
size: 9
```

Fonction `inline`

Macro en C

```cpp
#define MAX(x, y) ((x) > (y) ? (x) : (y))

int x = MAX(1, 2);
```

En C++

```cpp
inline int max(int x, int y) {
  return x > y ? x : y;
}

int x = max(1, 2);
```

Les struct

On a aussi les `struct` en C++

```
struct Vector
{
  float x;
  float y;

  float size()
  {
    return sqrt(x * x + y * y);
  }
}

// Pas besoin de mentionner struct comme en C
// struct Vector v;
Vector v;
v.x = 3;
v.y = 4;

v.size();
```

Range-based `for` loop

```
int values[] = {10, 20, 30};

for(const int& value : values)
{
  cout << value << endl;
}
```

Avec `auto` (inférence de type)

```
int values[] = {10, 20, 30};

// auto& à la place de int&
for(const auto& value : values)
{
  cout << value << endl;
}
```