

C++

Introduction

```
for(std::list<int>::iterator it=valid_index->begin(); it!=valid_index->end(); ++it)
{
    int k=*it;
    if(k!=k0 && k!=k1 && k!=k2)
    {
        const v2& x=v_vertices[k];

        if( (x-x0).det(u0)<0 && (x-x1).det(u1)<0 && (x-x2).det(u2)<0 )
        {
            std::list<int>::iterator it_temp=it; ++it;
            valid_index->erase(it_temp);
        }
    }
}
```



Premier programme

Créer un fichier

nom.**c**
ou nom.**cc**
ou nom.**cxx**
ou nom.**cpp**

toutes extensions valides

avantage **cc**, **cxx**, **cpp**:
pas de confusion avec pgm C

Rem. Pour les fichier d'en tête, on aura

nom.**h**
ou nom.**hh**
ou nom.**hxx**
ou nom.**hpp**

Pas une unique règle
Rester cohérent

Premier programme

```
#include <iostream>

int main()
{
    std::cout<<"Hello World"<<std::endl;
    return 0;
}
```

En ligne de commande:

```
$ g++ nom.cpp (-g -Wall -Wextra)
```

 *idem qu'en compilation C*

```
$ ./a.out
```

```
> Hello World
```

Analyse 1^{er} programme

en tête standard C++

Input Output Stream

(Flux d'entrée sortie = gestion affichage, écriture fichiers, ...)

```
#include <iostream>
```

Rem. Librairie standard C++
pas d'extension

```
int main()
```

```
{
```

```
std::cout<<"Hello World"<<std::endl;
```

```
return 0;
```

```
}
```

End of Line

```
std :: cout <<
```

Standard
Library

Scope Resolution Operator
(Opérateur de résolution de portée)

Common
Output

Opérateur C++

Variante 1^{er} programme

```
#include <iostream>
```

```
using std::cout;
```

```
using std::endl;
```

Permet d'appeler cout
à la place de std::cout

std est un espace de nom
(namespace)

```
int main()
```

```
{
```

```
    cout<<"Hello World"<<endl;
```

```
    return 0;
```

```
}
```

Plus de std

Variante 1^{er} programme

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

string: chaîne de caractères
(ne plus utiliser char*)

```
int main()
{
```

```
    std::string mot="Hello World";
```

```
    mot += "!!!";
```

concaténation
redimensionnement
automatique

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

```
    return 0;
```

```
}
```

Variante 1^{er} programme

```
#include <iostream>

int main()
{
    auto tableau={1.1 , 5.7 , 6.5 , 7.2 , -12.1 , 6.3};

    for(auto valeur : tableau)
        std::cout<<valeur<<std::endl;

    return 0;
}
```

\$ g++ nom.cpp **-std=c++11** (-g -Wall -Wextra)

↑
active c++ moderne

Variante 1^{er} programme

```
#include <iostream>

int main()
{
    auto tableau={1.1 , 5.7 , 6.5 , 7.2 , -12.1 , 6.3};

    for(auto valeur : tableau)
        std::cout<<valeur<<std::endl;

    return 0;
}
```

auto : déduction de type automatique

{a,b,c,d,...} : suite d'éléments

for(type nom : conteneur) : "range based loop"
{ ... }

> 1.1
5.7
6.5
7.2
-12.1
6.3

Tour d'horizon du C++

VS quelques autres langages

Affichage variables

```
#include <iostream>

int main()
{
    int a=5;
    float b=12.4;

    std::cout<<"valeur de a: "<<a<<" et b="<<b<<std::endl;

    return 0;
}
```

`std::cout<< ... << ...`

C++ est **générique**: pas besoin de désigner le type
VS C : `printf("%d %f ...`

Fonctions, surcharge

```
#include <iostream>

int addition(int a,int b)
{
    return a+b;
}
float addition(float a,float b)
{
    return a+b;
}

int main()
{
    std::cout<<addition(4,8)<<std::endl;
    std::cout<<addition(4.4f,8.4f)<<std::endl;

    return 0;
}
```

Surcharge:

fonction avec même nom
mais différents paramètres

- En C, fonction définie uniquement par son nom.
 - En C++, par son nom et ses paramètres.
- Mangling** (name decoration)

```
addition_int(...)
addition_float(...)
addition_double(...)
...
```

Fonctions, surcharge

```
int multiplication(int a,int b)
{
    return a*b;
}
matrix multiplication(matrix a,matrix b)
{
    return {a.x00*b.x00+a.x01*b.x10 , a.x00*b.x01+a.x01*b.x11,
            a.x10*b.x00+a.x11*b.x10 , a.x10*b.x01+a.x11*b.x11};
}
```

```
#include <iostream>
```

```
struct matrix
{
    int x00,x01;
    int x10,x11;
};
```

```
int main()
{
    int res=multiplication(4,8);

    matrix a={1,2,
              4,6};
    matrix b={7,-1,
              5,-2};
    auto c=multiplication(a,b);

    std::cout<<res<<std::endl;
    std::cout<<c.x00<<" "<<c.x01<<std::endl;
    std::cout<<c.x10<<" "<<c.x11<<std::endl;

    return 0;
}
```

Programmation objet

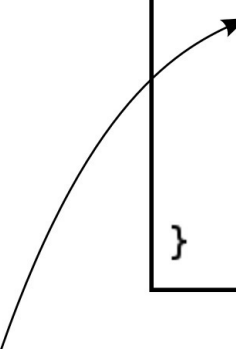
Struct similaires au C

```
struct voiture
{
    int kilometre;
    float essence;
};

int main()
{
    voiture ma_voiture;
    ma_voiture.kilometre=50000;
    ma_voiture.essence=50;

    return 0;
}
```

pas de répétition struct



Programmation objet

Fonctions membres (= méthodes en Java)

```
#include <iostream>

struct voiture
{
    int kilometre;           attributs
    float essence;

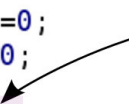
    void roule(int nbr_km)   fonction membre
    {
        kilometre += nbr_km;
        essence -= 5.0f/100 * nbr_km;
    }
};

int main()
{
    voiture ma_voiture;
    ma_voiture.kilometre=0;
    ma_voiture.essence=50;
    ma_voiture.roule(40);

    std::cout<<ma_voiture.kilometre<<std::endl;
    std::cout<<ma_voiture.essence<<std::endl;

    return 0;
}
```

Appel de la fonction membre



Programmation objet

Constructeurs

```
#include <iostream>

struct voiture
{
    int kilometre;
    float essence;

    voiture() Constructeurs
    {
        kilometre=0; essence=0;
    }
    voiture(int kilometre_initial, float essence_initiale)
    {
        kilometre=kilometre_initial;
        essence=essence_initiale;
    }

    void roule(int nbr_km) {...}
};

int main()
{
    voiture ma_voiture(5000, 10); Appel au constructeur

    ma_voiture.roule(40);

    std::cout<<ma_voiture.kilometre<<std::endl;
    std::cout<<ma_voiture.essence<<std::endl;

    return 0;
}
```

Programmation objet

Constructeurs

```
#include <iostream>

struct voiture
{
    int kilometre;
    float essence;

    voiture()
        :kilometre(0), essence(0)
    {}
    voiture(int kilometre_initial, float essence_initiale)
        :kilometre(kilometre_initial), essence(essence_initiale)
    {}

    void roule(int nbr_km) {...}
};

int main()
{
    voiture ma_voiture(5000, 10);
    ma_voiture.roule(40);

    std::cout<<ma_voiture.kilometre<<std::endl;
    std::cout<<ma_voiture.essence<<std::endl;

    return 0;
}
```

*Constructeurs avec
liste d'initialisation*

*Appel au
constructeur*

Programmation objet

Encapsulation (protection des données internes, robustesse)

```
struct voiture
{
public:
    voiture()
        :kilometre(0),essence(0)
    {}
    voiture(int kilometre_initial,float essence_initial)
        :kilometre(kilometre_initial),essence(essence_initial)
    {}
    void roule(int nbr_km)
    {
        kilometre += nbr_km;
        essence -= 5.0f/100 * nbr_km;
    }
    void affiche()
    {
        std::cout<<kilometre<<" km et "<<essence<<"L"<<std::endl;
    }
private:
    int kilometre;
    float essence;
};
```

accessible depuis l'exterieur

inaccessible en dehors de la classe

Programmation objet

Encapsulation (protection des données internes, robustesse)

```
struct voiture
{
public:
    voiture()
        :kilometre(0),essence(0)
    {}
    voiture(int kilometre_initial,float essence_initial)
        :kilometre(kilometre_initial),essence(essence_initial)
    {}
    void roule(int nbr_km)
    {
        kilometre += nbr_km;
        essence -= 5.0f/100 * nbr_km;
    }
    void affiche()
    {
        std::cout<<kilometre<<" km et "<<essence<<"L"<<std::endl;
    }
private:
    int kilometre;
    float essence;
};
```

accessible depuis l'exterieur

inaccessible en dehors de la classe

```
int main()
{
    voiture ma_voiture(50000,50);
    ma_voiture.roule(40);
    ma_voiture.affiche();

    return 0;
}
```

OK

```
21 private:
22     int kilometre;
23     float essence;
24 };
25
26 int main()
27 {
28     voiture ma_voiture(50000,50);
29     ma_voiture.roule(40);
30
31     std::cout<<ma_voiture.essence<<std::endl;
32
33     return 0;
34 }
35
```

KO

float voiture::essence is private

Programmation objet

Class vs Struct

```
struct voiture_1
{
//par default public:
    int kilometre;
    float essence;
};

class voiture_2
{
//par default private:
    int kilometre;
    float essence;
};

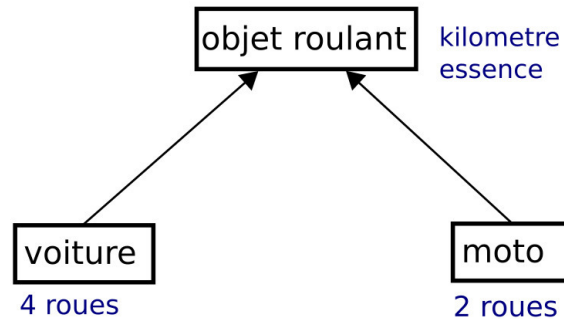
int main()
{
    voiture_1 ma_voiture_1;
    voiture_2 ma_voiture_2;

    return 0;
}
```

Programmation objet

Héritage

```
class objet_roulant
{
    int kilometre; float essence;
public:
    objet_roulant(int km, float qte) : kilometre(km), essence(qte) {}
    void roule(int km) { kilometre += km; essence -= 0.06f * km; }
};
class voiture : public objet_roulant
{
public:
    voiture(int km, float qte) : objet_roulant(km, qte) {}
    int nbr_roues() { return 4; }
};
class moto : public objet_roulant
{
public:
    moto(int km, float qte) : objet_roulant(km, qte) {}
    int nbr_roues() { return 2; }
};
```



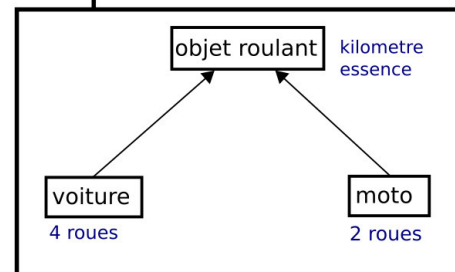
Programmation objet

Héritage

```
class objet_roulant
{
    int kilometre; float essence;
public:
    objet_roulant(int km, float qte) : kilometre(km), essence(qte) {}
    void roule(int km) { kilometre += km; essence -= 0.06f * km; }
};

class voiture : public objet_roulant
{
    voiture(int km, float qte) : objet_roulant(km, qte) {}
    int nbr_roues() { return 4; }
};

class moto : public objet_roulant
{
    moto(int km, float qte) : objet_roulant(km, qte) {}
    int nbr_roues() { return 2; }
};
```



```
int main()
{
    voiture ma_voiture(20000, 50);
    moto ma_moto(10000, 30);
    ma_voiture.roule(30);
    ma_moto.roule(30);
    std::cout << ma_voiture.nbr_roues() << std::endl;
    std::cout << ma_moto.nbr_roues() << std::endl;

    return 0;
}
```

Programmation objet

POO dynamique

- Similaire à Java (*polymorphisme, encapsulation, etc*)

Avec en +:

- Héritage multiple (+/-)
- **Surcharge d'opérateur (+++)**

$C = A + B$

$D = A * (B - C)$

...

Surcharge opérateur

```
struct vec2
{
    int x,y;

    vec2 operator+(const vec2& v) const
    {
        vec2 v_out={x+v.x,y+v.y};
        return v_out;
    }
};

int main()
{
    vec2 a={1,6};
    vec2 b={7,9};
    vec2 c=a+b;
    std::cout<<c.x<<" "<<c.y<<std::endl;

    return 0;
}
```

*Calculs mathématiques
ressemblent à des maths*



Surcharge opérateur

Exemple: calculer $5*((v0+v1)/2 + v2)$

```
struct vec2 v0={1,5},v1={8,6},v2={7,3};  
struct vec2 c=add_vec2(&v0,&v1);  
struct vec2 d=div_scalar(&c,2);  
struct vec2 e=add_vec2(&d,&v2);  
struct vec2 f=mul_scalar(&e,5);
```

C

```
vec2 v0 = new vec2(1,5);  
vec2 v1 = new vec2(8,6);  
vec2 v2 = new vec2(7,3);  
v0.add(v1).div(2).add(v2).mul(5)
```

Java

illisible

```
vec2 v0(1,5), v1(8,6), v2(7,3);  
vec2 v3=5*((v0+v1)/2+v2);
```

C++

Librairie standard très complète

STL : **S**tandard **L**ibrary

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

Conteneurs

- tableau dynamiques
- listes doublement chaînées
- files de priorités
- arbres
- tables de hachages
- ...

Algorithmes et fonctions utiles

- string
- tuple
- algorithms
- numeric
- ...

`std::`

La librairie standard est optimisée pour chaque architecture

Elle est + rapide et robuste qu'un code écrit à la main

Utiliser la librairie standard en 1er lieu!

- Librairie très technique

Tableau dynamique

std::vector=tableau d'éléments contigus en mémoire
taille s'adapte lors de l'ajout en fin

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v;
    v.push_back(4);
    v.push_back(6);
    v.push_back(9);
    v.push_back(15);

    for(int k=0;k<v.size();++k)
        std::cout<<v[k]<<std::endl;

    return 0;
}
```

en tête des std::vector

déclaration du type: std::vector

<int> est le type contenu dans le std::vector
*<x> : paramètre template
(=paramètre générique connu à la compilation)*

push_back: ajoute en fin de tableau

Accès comme un tableau C standard

Tableau dynamique

Tableau dynamique de mots

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

int main()
{
    std::vector<std::string> v;
    v.push_back("Bonjour");
    v.push_back("a tous");
    v.push_back("j'apprends");
    v.push_back("le C++.");

    for(int k=0;k<v.size();++k)
        std::cout<<v[k]<<std::endl;

    return 0;
}
```

Tableau dynamique

```
#include <iostream>
#include <vector>

struct vec2
{
    int x,y;
    vec2(int x0,int y0):x(x0),y(y0){}
};

int main()
{
    std::vector<vec2> v;
    v.push_back(vec2(1,4));
    v.push_back(vec2(7,2));
    v.push_back(vec2(9,7));
    v.push_back(vec2(3,2));

    for(int k=0;k<v.size();++k)
    {
        vec2 vec=v[k];
        std::cout<<vec.x<<" "<<vec.y<<std::endl;
    }

    return 0;
}
```

La STL est générique
S'adapte aux classes
de l'utilisateur

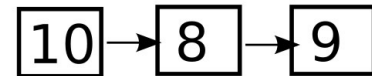
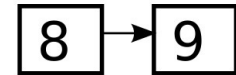
Listes doublement chaînées

```
#include <iostream>
#include <list>

int main()
{
    std::list<int> ma_liste;
    ma_liste.push_back(8);
    ma_liste.push_back(9);
    ma_liste.push_front(10);

    for(int element : ma_liste)
    {
        uniquement en C++11
        std::cout<<element<<std::endl;
    }

    return 0;
}
```



Arbre binaire de recherche + valeurs

```
#include <iostream>
#include <map>

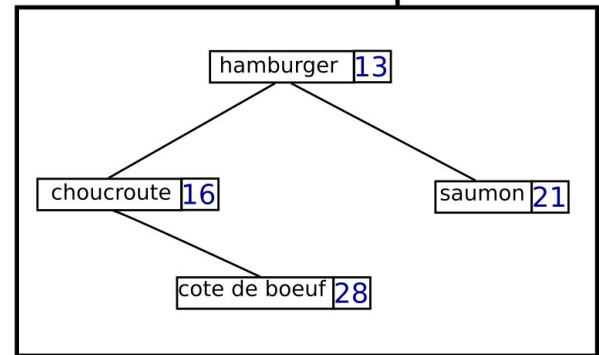
int main()
{
    std::map<std::string,int> menu;

    menu["choucroute"]=16;
    menu["hamburger"]=13;
    menu["saumon"]=21;
    menu["cote de boeuf"]=28;

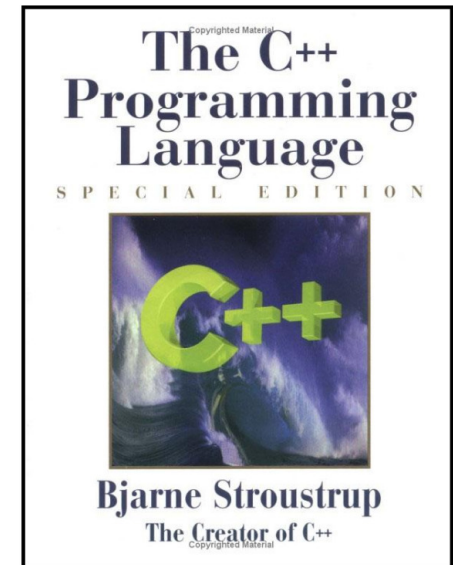
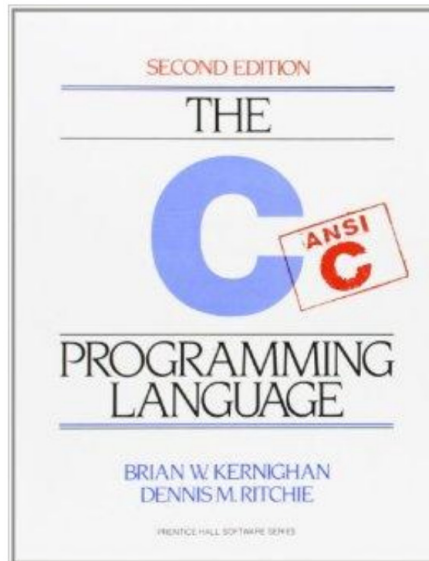
    std::cout<<menu["saumon"]<<std::endl;

    if(menu.find("truite")==menu.end())
        std::cout<<"Pas de truite au menu"<<std::endl;

    return 0;
}
```



Compatibilité avec le langage C



Compatibilité avec le C

Règle de bonne pratique C++

Evitez d'utiliser des fonctions du C si vous avez le choix

Mélange entre C et C++ = C/C++
courant sur internet
à bannir: mauvaise pratique

Sur un CV: Langage informatique:

- ~~C/C++~~
- Java
- ...

*C/C++ sur un CV = généralement
'je ne sais pas bien coder ni en C ni en C++'*

Compatibilité avec le C

Les bibliothèques C sont compatibles avec le C++

```
#include <stdio>

int main()
{
    const char* value="du C";

    printf("Je fais %s en C++\n",value);
    std::printf("Les fonctions %s sont dans le namespace std \n",value);

    return 0;
}
```

Pour les en têtes de la librairie standard

#include <cNOM> à la place de #include <NOM.h>

Inclue les fonctions dans namespace std

#include <NOM.h> fonctionne, mais problème de conflits de noms. A éviter.

Compatibilité avec le C

Deux appels pour la mémoire dynamique

1. Par malloc/free

```
#include <cstdlib>
#include <cassert>

int main()
{
    float *dynamic_memory=(float*)malloc(50*sizeof(float));
    assert(dynamic_memory!=NULL);

    for(int k=0;k<10;++k)
        dynamic_memory[k]=float(k);

    free(dynamic_memory);
    dynamic_memory=NULL;

    return 0;
}
```

cast du malloc obligatoire
En C++ le cast de void*
vers un type doit être explicite



A ne pas faire: On utilise pas malloc/free en C++

(Sauf cas exceptionnels)

Compatibilité avec le C

Deux appels pour la mémoire dynamique

2. Par new/delete

```
#include <cstdio>
#include <cassert>

int main()
{
    float *dynamic_memory=new float[50];
    assert(dynamic_memory!=NULL);

    for(int k=0;k<10;++k)
        dynamic_memory[k]=float(k);

    delete[] dynamic_memory;
    dynamic_memory=NULL;
}
```

Opérateur new
Typé: pas de cast, pas de sizeof

Ne jamais mélanger
malloc/free
et new/delete
Mécanisme de gestion
différent

Opérateur delete sur un tableau

Il est rare de devoir utiliser explicitement new/delete pour des tableaux ! (std::vector remplace avantageusement)

Casts

Cast de type C:

```
#include <iostream>

int main()
{
    int a=8;
    double b=4.5;

    b=(double)a;

    std::cout<<sizeof(b)<<std::endl;

    return 0;
}
```

Cast de type C++:

Ressemble à un
constructeur

```
#include <iostream>

int main()
{
    int a=8;
    double b=4.5;

    b=double(a);

    std::cout<<sizeof(b)<<std::endl;

    return 0;
}
```

Casts

Problème du cast: peut aboutir rapidement à de mauvaises pratiques.

```
void f(const int* ptr)
{
    int *x=(int *) (ptr);
    *x=15;
}

int main()
{
    int a=8;
    const int* ptr_a=&a;

    f(ptr_a);

    std::cout<<a<<std::endl;

    return 0;
}
```

promesse
d'être constant

promesse
oubliée

Casts

Le C++ peut spécifier quel type de cast.

- évite les oublis involontaires
- informe le lecteur du type de cast

```
int main()
{
    int a=4;
    double b=-4.51;
    a=static_cast<int>(b);

    const int* ptr_a=&a;
    int* ptr_b=const_cast<int*>(ptr_a);
    *ptr_b=12;

    const std::string* ptr_string=NULL;
    ptr_string=reinterpret_cast<const std::string*>(ptr_a);

    std::cout<<a<<std::endl;
    return 0;
}
```

static_cast: conversion standard d'un type vers un autre (safe)

const_cast: enlève le const (à éviter tant que possible)

reinterpret_cast: changement du type pointé (unsafe).
À éviter tant que possible

+ dynamic_cast: reconversion (safe) d'un pointeur à l'exécution

Pourquoi le C++ ?

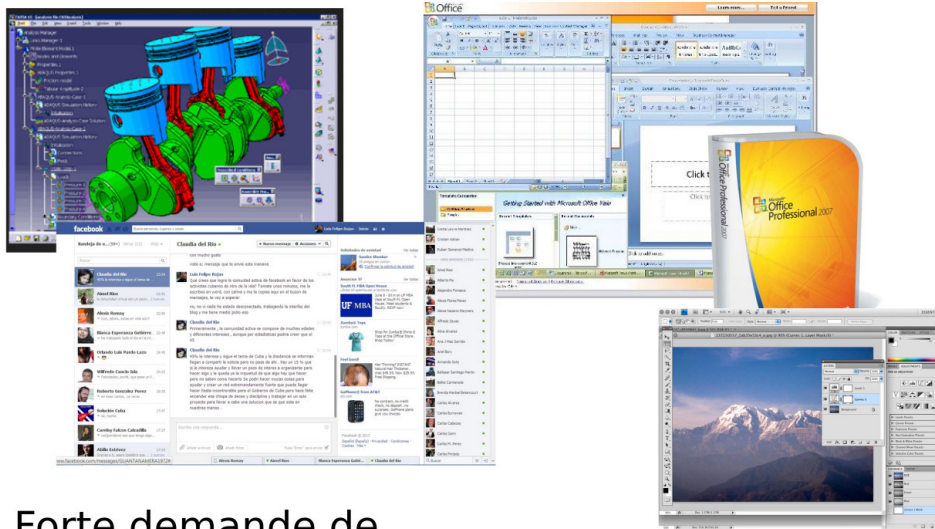


Pourquoi le C++ ?

- C++ aujourd'hui le langage le + utilisé pour

Les logiciels **importants** avec contraintes d'**efficacité**

Logiciels calculs, CAO, 3D, images, visualisation, etc.



Forte demande de
programmation technique

Catia
Maya 3D
MS Office
Outlook
Photoshop
Facebook
DirectX
Firefox
Chrome
Internet Explorer
gcc

Pourquoi le C++ ?

- C++ aujourd'hui le langage le + utilisé pour
Les logiciels **importants** avec contraintes d'**efficacité**
- L'unique langage qui permet de faire à la fois
 - De la programmation haut niveau
 - De la programmation bas niveau
 - Execution rapide
 - Compatible avec librairies existantes (C, assembleur, etc)
 - Adapté aux code de grande envergure
 - Liberté au programmeur

Avantages/Inconvénients



- Haut/bas niveau
 - Robustesse
 - Rapidité optimale
 - Bibliothèques utilisables
-



- Liberté totale programmeur
 - Compatibilité C
-



- Complexité (*Probablement le langage le + compliqué*)
- Temps de compilation