



Du C au C++

Groupe étudiants : CIR2

Nils Beaussé

E-Mail : nils.beausse@isen-ouest.yncrea.fr

Numéro de bureau : A2-78

N° de téléphone : 0230130573

Informations pratiques

- Volume horaire C++ : 60h (30 séances de 2h) → votre plus gros module
- Ce cours est issu du cours d'Ayoub Karine, lui-même inspiré des cours de M. Soullignac et M. Aron
- **DS et examen :**
 - Les cours pourront commencer par un petit QCM noté sur machine, qui pourra être systématique ou non selon le temps que j'ai. Il portera toujours sur ce qu'on a vu dans la séance précédente.
 - Au moins 2 DS → Semestre 1
 - Au moins 1 DS → Semestre 2
 - Quelques TPs pourront aussi être notés et intégrés dans la note du DS
 - Note finale = DS + TP + votre note d'algorithmique pour le S1

Historique

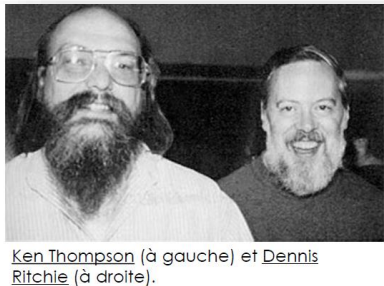
- **Le langage C** a été inventé :
 - en 1972
 - par Dennis Ritchie et Ken Thompson
 - dans le laboratoire AT&T (Laboratoire Bell)
 - en même temps et pour UNIX
 - Introduit notamment les Types par rapport au B



Ken Thompson (à gauche) et Dennis Ritchie (à droite).

Historique

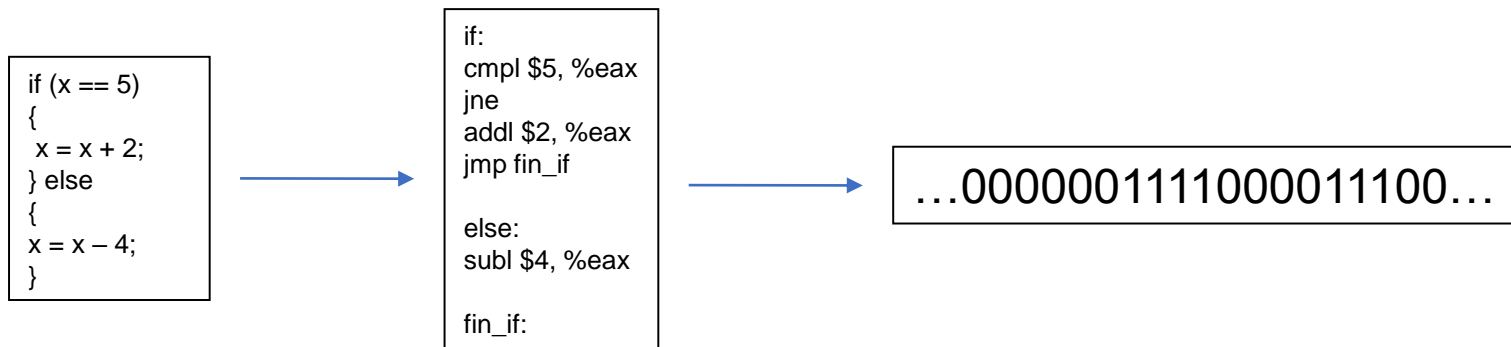
- **Le langage C** a été inventé :
 - en 1972
 - par Dennis Ritchie et Ken Thompson
 - dans le laboratoire AT&T (Laboratoire Bell)
 - en même temps et pour UNIX
 - Introduit notamment les Types par rapport au B



→ En route pour quelques petits rappels

Rappels

- Le C est un langage **compilé** de **bas niveau** :
→ Une instruction est traduite directement en code machine par le compilateur (gcc/clang etc.)



Chaque instruction assembleur à un sens précis et simple

Rappels

- Par exemple :

```
if:
  cmpl $5, %eax
  jne
  addl $2, %eax
  jmp fin_if

else:
  subl $4, %eax

fin_if:
```

CMPL

...0000001111000011100...

Un code binaire qui est une
succession d'instruction
élémentaire

Bits	Value
0-5	31
6-8	BF
9	/
10	L
11-15	RA
16-20	RB
21-30	32
31	/

Rappels

- Par exemple :

Chacune est comprise par un sous-circuit du processeur et effectue une opération directement au niveau électronique

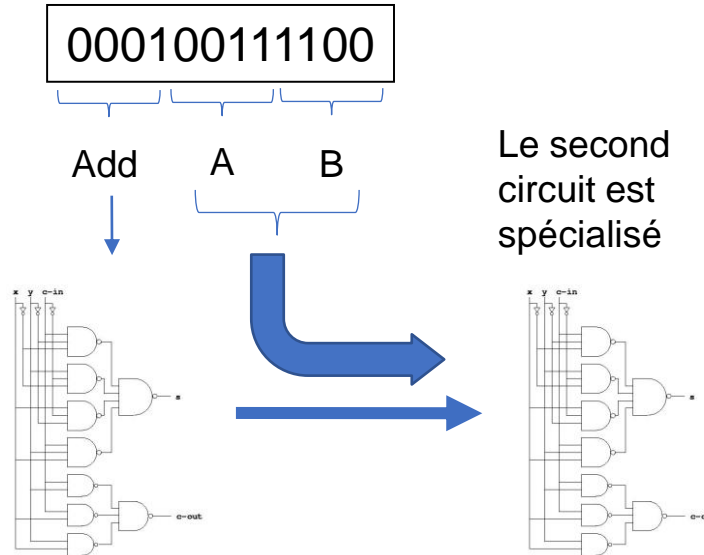
Bits	Value
0-5	31
6-8	BF
9	/
10	L
11-15	RA
16-20	RB
21-30	32
31	/



Rappels

- Grossièrement :

Un circuit comprends la première instruction et redirige la suite à un autre circuit (ici un circuit d'addition par exemple)



Résultat



En conclusion

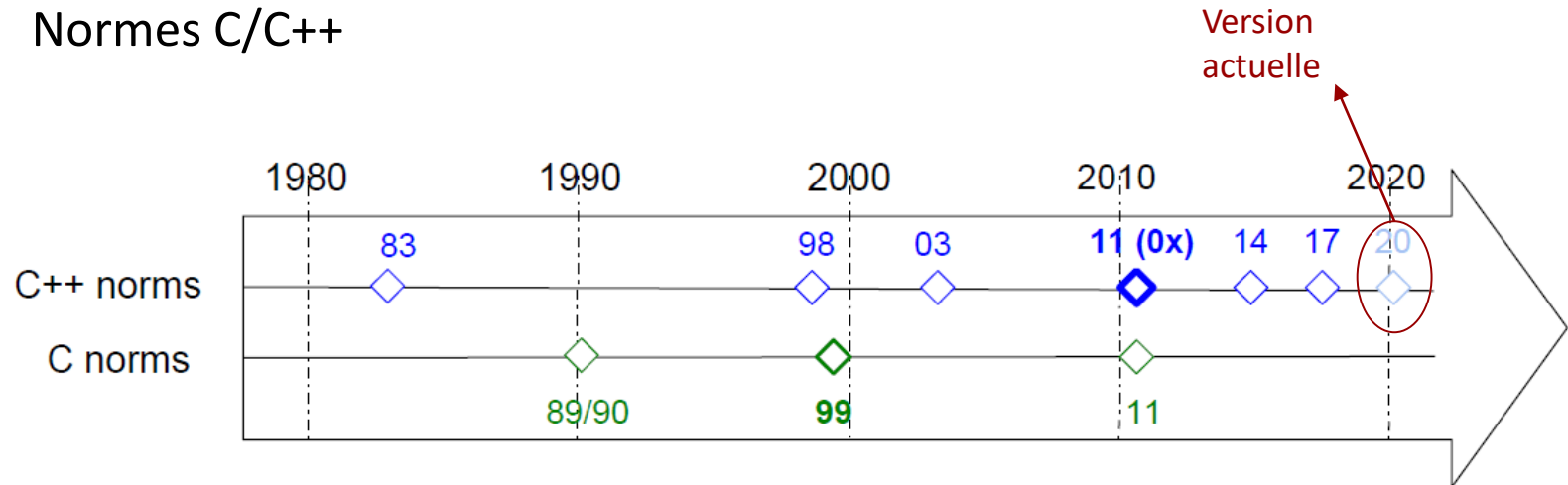
- La programmation dans un langage compilé permet d'interagir directement avec l'électronique du processeur (CPU).
 - Ce point permet aux langages compilés d'être très « proche » du matériel :
 - On parle de langage « **bas niveau** ».
 - On parle aussi de programme « **natif** » pour les programmes qui en résultent.
 - De fait : Programmes extrêmement rapides par rapport à une autre approches.
 - En opposition aux langages dit « interprétés » : Java / Python etc.

Les défauts du C

- C est un langage compilé rapide et bas niveau MAIS :
 - Trop bas niveau pour certaines applications, il manque de l'abstraction ce qui demande d'écrire parfois **beaucoup de code redondant**
 - L'aspect bas niveau le rends très susceptible aux erreurs de programmation.
- Ces deux points ont donné naissance à une évolution du C : **le C++** .

C VS C++

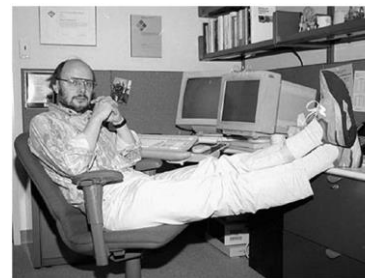
- **C++ est une amélioration du C**
- Un même programme peut combiner C et C++
- Un compilateur C++ peut compiler C
- Normes C/C++



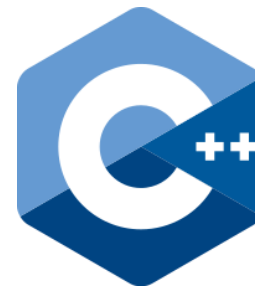
Le C++

- **Le langage C++** a été inventé

- en 1983
- par Bjarne Stroustrup
- dans le même labo que le C (laboratoire Bell)



Bjarne Stroustrup (source Wikipédia)



- Le C++ est conçu pour résoudre les problèmes d'abstraction et d'erreurs du C en développant le concept d'**objet** du C
- Et en introduisant de nombreuses autres différences plus ou moins importantes que l'on va voir durant ce cours.

Pourquoi C++ ?

- Très répandu (forme avec le C le groupe de langage le plus utilisé)
 - Documentation disponible
 - Questions/réponses dans les forums
- Rapide (possibilité de la programmation temps réel)
- Portable (avec plus de travail que pour un langage interprété)
 - Même code C++ peut, théoriquement, être transformé en exécutable sous Windows, Mac OS et Linux
- Richesse en bibliothèques
- Programmation orientée objet

Pourquoi C++ ?

- Applications développées en C++



macOS



Office



Pourquoi C++ ?

- Applications développées en C++



Etc...

+ La quasi-totalité des jeux-vidéos et outils de développement de jeux-vidéo

Outils/liens utiles

- Outils et applications utiles aux développeurs

GitHub



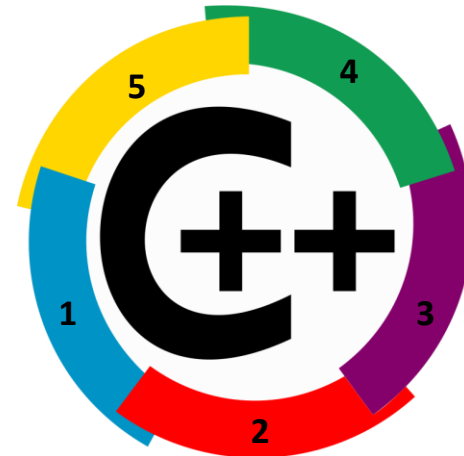
stackoverflow



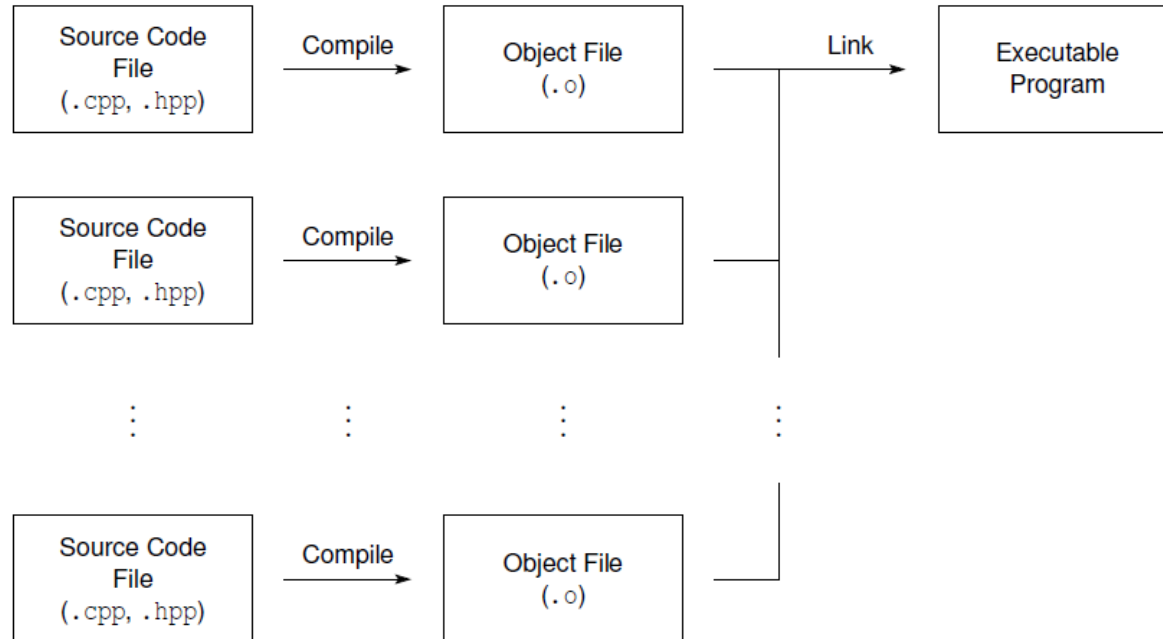
slack

Étapes de la programmation en C++

1. Coder le programme C++ (.hpp, .cpp, ...)
2. Compiler le programme
3. Correction des erreurs → retour à l'étape 1
4. Exécuter le programme
5. Debugger → retour à l'étape 1



Génération de l'exécutable

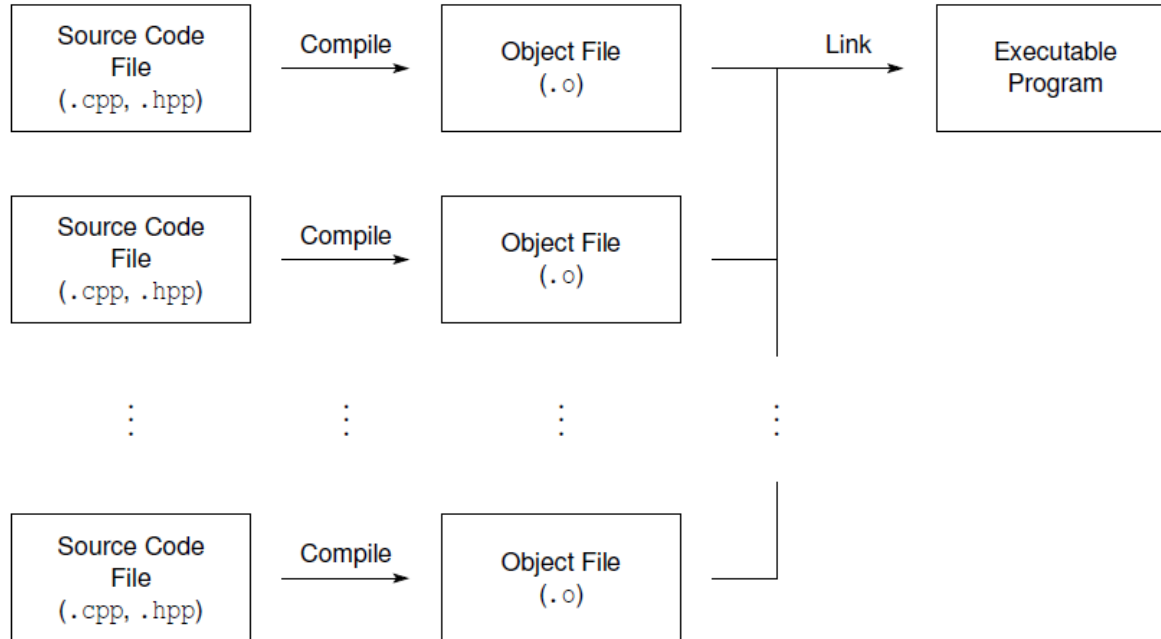


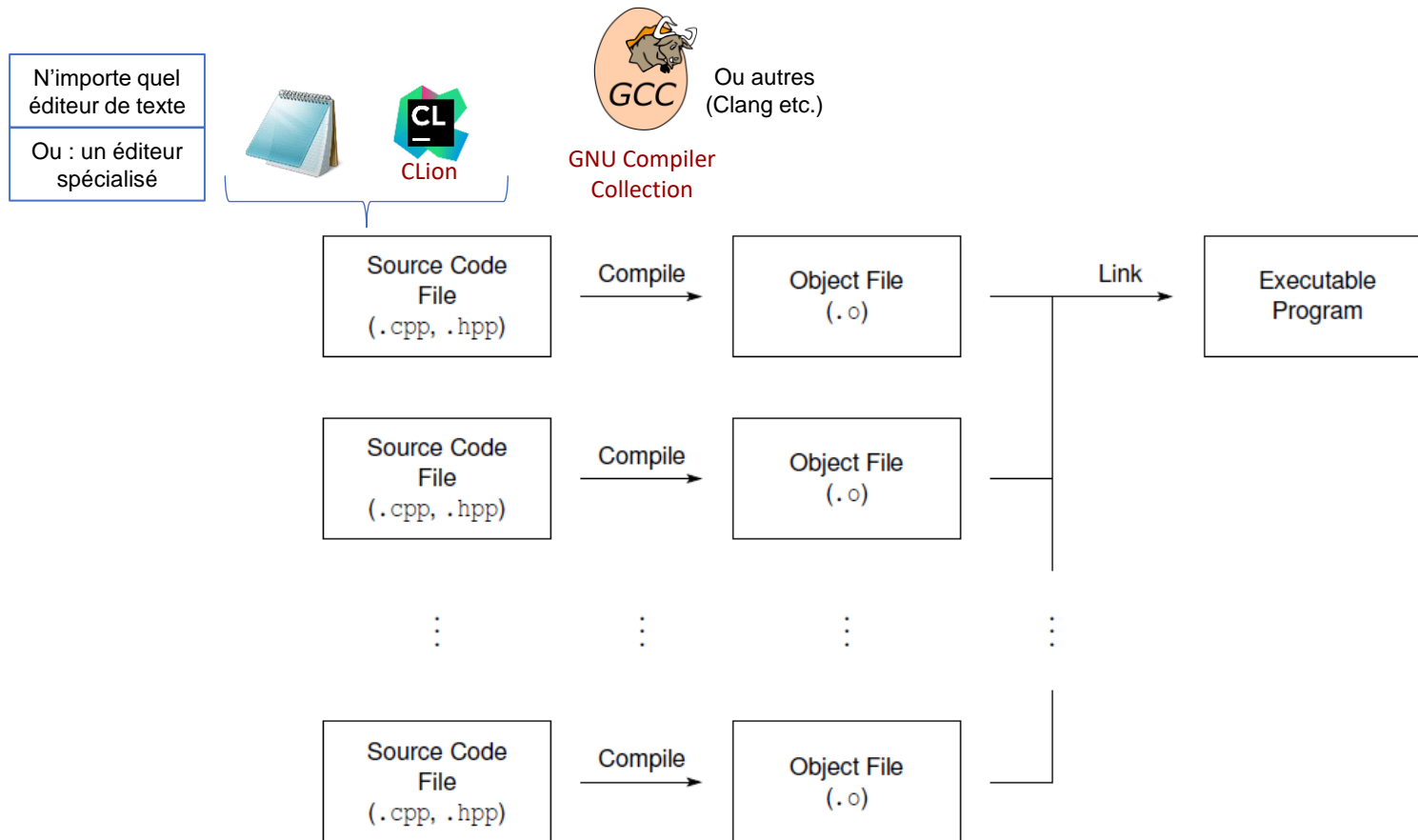
N'importe quel
éditeur de texte

Ou : un éditeur
spécialisé



CLion





N'importe quel éditeur de texte
Ou : un éditeur spécialisé



CLion

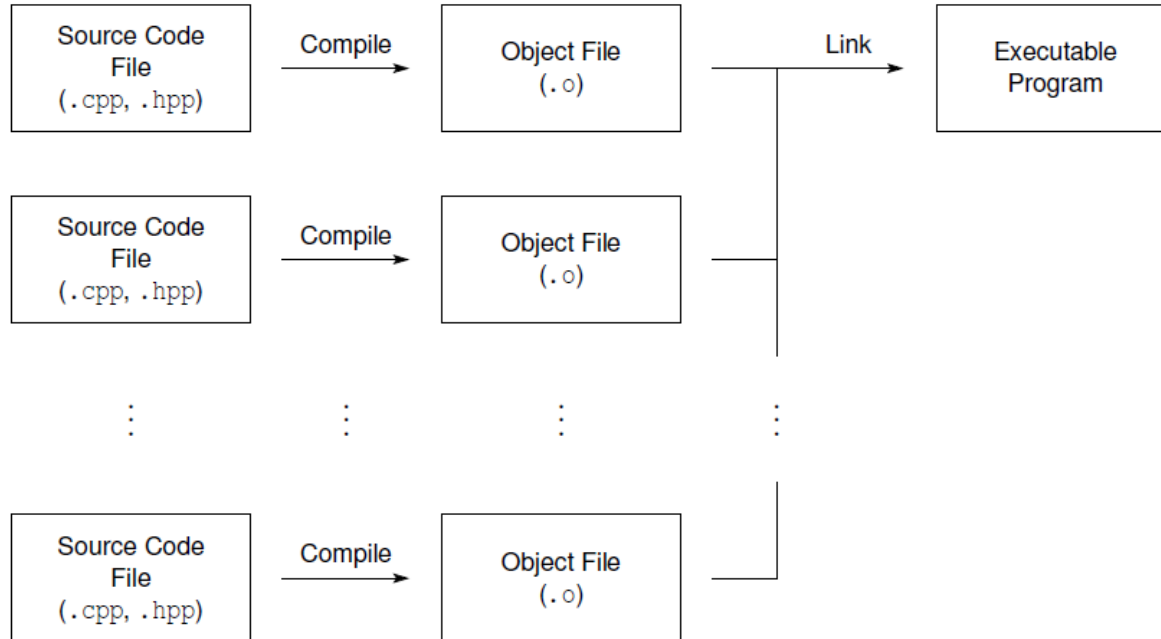


GNU Compiler
Collection

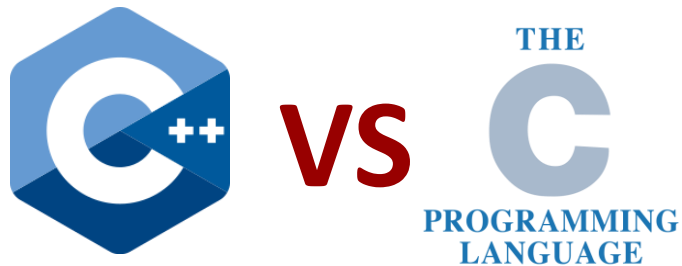
Ou autres
(Clang etc.)



GNU Linker (appelé
automatiquement par GCC)



Différences entre C et C++



Différences entre C et C++



Les petits détails...

1- Headers

- Règle générale

C	C++
<code>#include <directive.h></code>	<code>#include <directive></code>

- Exemple

C	C++
<code>#include <stdio.h></code>	<code>#include <cstdio></code>
<code>#include <stdlib.h></code>	<code>#include <cstdlib></code>
<code>#include <string.h></code>	<code>#include <cstring></code>

2- Commentaires

- **En C < 99**

- Commentaires en plusieurs lignes : `/* */`
- Commentaires dans une seule ligne : `/* */`

- **En C++ et C > 99**

- Commentaires en plusieurs lignes : `/* */`
- Commentaires dans une seule ligne : `//`

3- Variables booléennes

▪ En C > 99

- définies dans `<stdbool.h>`
ou
- Via les macros et `define`

```
typedef int bool;  
  
#define true 1  
  
#define false 0
```

▪ En C++

- Nouveau type ***bool*** → deux valeurs sont possibles : **true** ou **false**
- Conversion possible vers/depuis Integer, Float, ...
- Pour afficher le contenu d'une variable booléenne, on utilise **`std::boolalpha`**

3- Variables booléennes

- Exemple de la conversion Boolean <-> Entiers

```
bool b;  
int i;  
  
b = true;  
i = b; // i = 1  
  
b = false;  
i = b; // i = 0
```

```
bool b;  
int i;  
  
i = 3;  
b = i; // b = true  
  
i = -2;  
b = i; // b = true  
  
i = 0;  
b = i; // b = false
```

4- Inférence de type

- **En C** : obligation de déclarer le type de la variable

```
int count = 0;  
char ch = 'Z';  
double limit = 100.0;
```

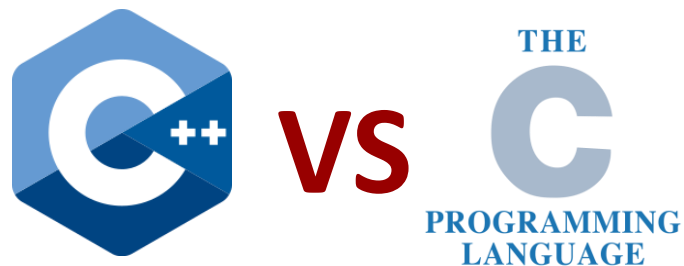
- **En C++** : utilisation du mot-clé **auto** qui laisse au compilateur la tâche de déduire le type de la variable (l'initialisation de la variable est obligatoire)

```
auto count = 0;  
auto ch = 'Z';  
auto limit = 100.0;
```

4- Inférence de type

- **NE PAS EN ABUSER !!!!**
- Pratique quand ça évite d'écrire des types à rallonge, mais peut rendre le code illisible ! ! !

Différences entre C et C++



Les espaces de noms

Espace de noms

- En C toute fonction qui est incluse via un Header (.h en C, .hpp en C++) sera disponible dans le programme
- Si on a un code lourd on peut se passer de certains header non nécessaire dans un certain fichier pour éviter de se tromper de noms de fonctions
- MAIS : ce n'est pas pratique, en C++ on introduit alors la notion de « namespace ».

Espace de noms

- Les espaces de nom (*namespace*) permettent d'associer un nom à un ensemble de variables, types, fonctions ...

- Création d'un espace de noms :

```
namespace mynamespace  
{  
    // identifiants  
}
```

- Utilisation d'un espace de noms :

```
mynamespace::identifiant
```

- Pour éviter d'écrire les noms complets, on utilise
- (au début du fichier, après les includes)

```
using namespace super_nom;
```


Espace de noms

- Exemple :

```
namespace f2
{
    int i=0;
    int inc(int n) {
        return n+1;
    }
}
```

```
int main(){
    int j=3;
    int c=f2::inc(j);
}
```

OU

```
using namespace f2;
int main(){
    int j=3;
    c=inc(j)
}
```

Différences entre C et C++



Les flux d'entrée/sortie

1- Flux de sortie

```
// code en C
#include <stdio.h>
void main(){
printf("Hello World !\n");
}
```

1- Flux de sortie



```
// code en C
#include <stdio.h>
void main(){
printf("Hello World !\n");
}
```

```
// code en C++
#include <iostream>
int main() {
std::cout << "Hello world !" << std::endl;
return 0;
}
```

1- Flux de sortie

```
// Il est possible de combiner C et C++  
#include <iostream> // les librairies E/S (I/O) C++  
#include <cstdio>    // les librairies E/S (I/O) C  
int main() {  
    std::cout << "Hello world !" << std::endl;  
    printf("ISEN\n");  
    return 0;  
}
```

`std::` = préfixe utilisé par tous les mots clés de la librairie standard du C++

1- Flux de sortie

```
// Il est possible de combiner C et C++  
#include <iostream> // les librairies E/S (I/O) C++  
#include <stdio.h>   // les librairies E/S (I/O) C  
int main() {  
    std::cout << "Hello world !" << std::endl;  
    printf("ISEN\n");  
    return 0;  
}
```

`std::` = préfixe utilisé par tous les mots clés de la librairie standard du C++

On peut également
éviter d'utiliser ce
préfixe grâce aux
espaces de nom
(namespace)



```
#include <iostream>  
using namespace std;  
int main(){  
    cout << "Hello world!" << endl;  
    return 0;}
```

1- Flux de sortie

- L'opérateur **<<** prends à sa gauche ce qu'on appelle un « flot » ou « flux »
 - Ici c'est **cout** , mais il peut exister d'autres flux
- À droite l'opérateur prends n'importe quelles valeurs classiques (int, float, char etc.)

1- Flux de sortie

Modification de l'affichage de l'opérateur ?

- Avec **printf** on fait classiquement comme ceci pour modifier l'affichage :

```
int main() {  
    float a = 0.25;  
    printf("a est affiché comme un int : %d",a);  
    printf("a est affiché comme un float : %f",a);  
    printf("a est affiché comme un float : %9.6f",a);  
    // un float de 9 caractères (. compris, 6 chiffres après la virgules et 2 avant)  
  
    return 0;  
}
```


1- Flux de sortie

Modification de l'affichage de l'opérateur ?

- Avec **cout** ce n'est plus du tout la même philosophie :

- On introduit des modificateurs !

```
cout << modificateur  
cout << "Hello world!" << endl;
```

- Exemple :

```
bool mon_booleen = true;  
cout << boolalpha  
cout << mon_booleen << endl;
```

Affiche « **True** » grâce
au modificateur
« boolalpha »
Sinon : affiche « 0 »



1- Flux de sortie

```
// code en C
#include <stdio.h>
void main(){ // affichage de CIR2 Nantes ISEN
int a = 2
char e[] = "isen"
printf("CIR %d Nantes %s\n", a, e);
return 0; }
```

1- Flux de sortie



```
// code en C
#include <stdio.h>
void main(){ // affichage de CIR2 Nantes ISEN
int a = 2
char e[] = "isen"
printf("CIR %d Nantes %s\n", a, e);
return 0; }
```

```
// code en C ++
#include <iostream>
using namespace std;
void main(){ // affichage de CIR2 Nantes ISEN
int a = 2;
char e[] = "isen";
cout << "CIR" << a << " Nantes " << e << endl;
return 0; }
```

2 - Flux d'entrée

- **cout** est l'équivalent de **printf** en C++, mais quel est l'équivalent de **scanf** ?
- C'est ... fort logiquement : **cin**

- Usage →


```
int age;  
cin >> age;
```

- Tout simplement !

2 - Flux d'entrée


```
// code en C
#include <stdio.h>
void main(){ int age;
printf("Quel est ton age ?");
scanf("%d",&age);
printf("Vous avez %d ans", age);
}
```

2 - Flux d'entrée



```
// code en C
#include <stdio.h>
void main(){ int age;
printf("Quel est ton age ?");
scanf("%d",&age);
printf("Vous avez %d ans", age);
}
```

2 - Flux d'entrée



```
// code en C
#include <stdio.h>
void main(){ int age;
printf("Quel est ton age ?");
scanf("%d",&age);
printf("Vous avez %d ans", age);
}
```

```
// code en C ++
#include <iostream>
using namespace std;
int main(){
int age;
cout << "Quel est ton age ?" << endl;
cin >> age;
cout << "Vous avez " << age << " ans" << endl;
return 0;}
```

2 - Flux d'entrée

- **En C**

- `#include <stdlib.h>`
- Fonctions : **`malloc()`**, **`free()`**

```
int main()
{
    int* variable = NULL;
    variable = malloc(sizeof(int)); // Allocation de mémoire
    free(variable); // Libération de mémoire
    // *** cas d'un tableau ** //
    int* tableau = NULL;
    tableau = malloc(20 * sizeof(int)); // Allocation
    free(tableau); // Libération de mémoire
    return 0;
}
```

- **En C++**

- Sans include
- Opérateurs : **`new`** et **`delete`**

```
int main()
{
    int* variable = new int; // Allocation de mémoire
    delete variable; // Libération de mémoire
    // *** cas d'un tableau ** //
    int* tableau = new int[20]; // Allocation de mémoire
    delete[] tableau; // Libération de mémoire
    return 0;
}
```

On expliquera plus tard pourquoi ce changement !

Différences entre C et C++



Les références

1 - Rappel sur les pointeurs

- Adresse d'une variable = emplacement dans la mémoire d'une variable.
 - Analogie avec l'adresse d'une maison : l'adresse est l'ensemble de symbole qui permet de dire où est la maison. Mais l'adresse n'est PAS la maison.

1 - Rappel sur les pointeurs

- Adresse d'une variable = emplacement dans la mémoire d'une variable.
 - Analogie avec l'adresse d'une maison : l'adresse est l'ensemble de symbole qui permet de dire où est la maison. Mais l'adresse n'est PAS la maison.

*11 Avenue du Champ de Manœuvre
44470 Carquefou*

!=



1 - Rappel sur les pointeurs

- En informatique l'emplacement dans la mémoire est plus simple que dans la vraie vie.
 - C'est comme si il n'y avait qu'une très longue rue
 - **On peut donc se contenter de donner le numéro** sans le nom de la rue
 - « 6411 » tout seul est donc une adresse valide en informatique, qui désigne bien un emplacement de la mémoire.
 - Par habitude en informatique on préfère utiliser la base hexadécimale.

« 6411 » = « 0x0000190B »

1- Rappel sur les pointeurs

« 6411 » = « 0x0000190B »

- En C on peut stocker ce numéro dans n'importe quelle variable de type nombre assez grande pour la contenir :

int i = 6411 est valide

char b = 6411 est invalide (car char ne va que de -128 à 127)

- Problème, certains système ont une « rue » plus longue que d'autres.
 - En clair : certains système informatique peuvent accéder à plus de case mémoire que d'autres.

1- Rappel sur les pointeurs

- Exemple :

- Intel 8080 :



= Architecture 8 bits => Maximum = 11111111 = 255

1- Rappel sur les pointeurs

- Exemple :

- Intel 8080 :



= Architecture 8 bits => Maximum = 11111111 = 255

Les architectures 8 bits
compte encore pour 50%
des processeurs vendu
dans le monde !



1- Rappel sur les pointeurs

▪ Exemple :

- Intel 8080 :



= Architecture 8 bits => Maximum = 11111111 = 255

- Intel i386 :

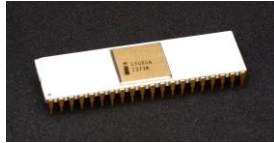


= Architecture 32 bits => Maximum = 111111.... (32 fois) = 4 294 967 296
= 4Giga de case

1- Rappel sur les pointeurs

▪ Exemple :

- Intel 8080 :



= Architecture 8 bits => Maximum = 11111111 = 255

- Intel i386 :



= Architecture 32 bits => Maximum = 111111.... (32 fois) = 4 294 967 296
= 4Giga de case

- AMD Athlon 64



= Architecture 64 bits => Maximum = 111... (64 fois) = 2^{64} = BEAUCOUP !

1- Rappel sur les pointeurs

- Problème :

char **adresse** -> suffisant pour un processeur 8 bits car 255 valeurs possibles sur char

Mais insuffisant sur un système 32 bits.

Idem : un entier de 32 bits est suffisants sous un Windows XP (32 bits) mais pas sous Windows 10 (64 bits) !

- Comment écrire le même code pour toutes les machines ?
 - Pour éviter de se tromper et faire du code portable !

1- Rappel sur les pointeurs

▪ Solution

En C on a un type « pointeur » qui fait toujours **la taille adéquate** selon le système et qui permet de contenir une adresse.

- On le déclare en rajoutant une étoile à un type quelconque :

Int* a -> fera 8 bits sous un système 8 bits, et 64 bits sous un système 64 bits etc.

Char* b -> idem

Bidule* c -> idem

- Le type avant l'étoile **n'est qu'une indication** sur ce que contiendra la case à l'adresse. Un peu comme si on écrivait « no 355 -> immeuble » « no 356 -> maison ». C'est pour éviter de faire des erreurs, mais en vrai ça ne change **RIEN**.
 - D'ailleurs on peut stocker des pointeurs sans indication, dans un type vide si on veut : **void*** truc. C'est juste aussi !
 - Et on peut transférer les pointeurs de l'un à l'autre si on veut malgré des indications différentes : a = b ; ou b = c ou truc = a -> tout ça ça marche ! (mais c'est sale !)

1- Rappel sur les pointeurs

▪ Solution

- Comment avoir accès à l'adresse d'une variable quelconque ?

- Grace au symbole « **&** » !

➤ Exemple :

On déclare une
variable quelconque

`int b = 23;`



1- Rappel sur les pointeurs

▪ Solution

- Comment avoir accès à l'adresse d'une variable quelconque ?
 - Grace au symbole « **&** » !

➤ Exemple :

```
int b = 23;  
void* a;
```

On déclare une
variable quelconque

On déclare un
pointeur

1- Rappel sur les pointeurs

▪ Solution

- Comment avoir accès à l'adresse d'une variable quelconque ?

- Grace au symbole « **&** » !

➤ Exemple :

```
int b = 23;  
void* a;  
  
a = &b;
```

On déclare une
variable quelconque

On déclare un
pointeur

Le pointeur
récup
l'adresse de b.

1- Rappel sur les pointeurs

▪ Solution

- Comment avoir accès à l'adresse d'une variable quelconque ?

- Grace au symbole « **&** » !

➤ Exemple :

On déclare une
variable quelconque

On déclare un
pointeur

Le pointeur
récup
l'adresse de b.

```
int b = 23;  
void* a;  
  
a = &b;
```

Pour aider le compilateur : On lui
donne une indication sur ce qui
existera à l'emplacement de l'adresse
ET on initialise à 0 (NULL).

```
int b = 23;  
int* a = NULL;  
  
a = &b;
```

1- Rappel sur les pointeurs

- Et à quoi ça sert ?
 - Le truc c'est qu'on peut modifier la variable qui est à l'adresse contenu dans le pointeur très simplement !

- Exemple :

```
int a = 3 ;  
int* b = NULL ;  
  
b = &a ;  
  
*b = 2 ;  
  
cout << "La valeur de a est " << a << endl ;
```

Et paf ! Dans
« a » on a 2.



1- Rappel sur les pointeurs

- OK, on peut modifier la valeur d'une variable indirectement, mais ça sert à quoi ?!

```
int a = 3 ;  
int* b = NULL ;  
  
b = &a ;  
  
*b = 2 ;  
a = 3;  
  
cout << "la valeur de a est " << a << endl ;
```

Et paf ! Dans
« a » on a 2.

Oui mais bon,
c'est quand
même plus simple
comme ça !

1- Rappel sur les pointeurs

- En fait ça sert surtout à passer à une fonction une variable que l'on veut modifier.
- En effet en C, si j'écris

Il faut se souvenir que a, b et c sont des COPIES de ce qui a été passé à la fonction

```
int fonction_addition(int a, int b, int c)
{
    a = 2 ;
    b = 3 ;
    c = b + a ;
    return c ;
}
```

```
int main()
{
    int a = 4 ;
    int b = 7 ;
    int c = 9 ;
    int d = 1;
    d = fonction_addition(a,b,c) ;
}
```

1- Rappel sur les pointeurs


- En fait ça sert surtout à passer à une fonction une variable que l'on veut modifier.
- En effet en C, si j'écris

Il faut se souvenir que a, b et c sont des COPIES de ce qui a été passé à la fonction

```
int fonction_addition(int a, int b, int c)
{
    a = 2 ;
    b = 3 ;
    c = b + a ;
    return c ;
}
```

```
int main()
{
    int a = 4 ;
    int b = 7 ;
    int c = 9 ;
    int d = 1 ;
    d = fonction_addition(a,b,c) ;
}
```

A = 4
B = 7
C = 9
D = 5



1- Rappel sur les pointeurs

- En fait ça sert surtout à passer à une fonction une variable que l'on veut modifier.
- En effet en C, si j'écris

Il faut se souvenir que a, b et c sont des **COPIES** de ce qui a été passé à la fonction

```
int fonction_addition(int a, int b, int c)
{
    a = 2 ;
    b = 3 ;
    c = b + a ;
    return c ;
}
```

```
int main()
{
    int a = 4 ;
    int b = 7 ;
    int c = 9 ;
    int d = 1;
    d = fonction_addition(a,b,c) ;
}
```

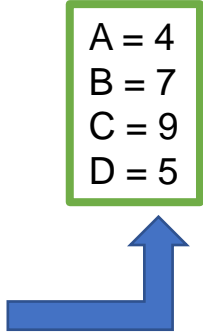
A = 4
B = 7
C = 9
D = 5

1- Rappel sur les pointeurs

- Avec un pointeur le problème disparaît.
- Disons, que je veut modifier a, je fais ainsi :

```
int fonction_addition(int* a, int b, int c)
{
    *a = 2 ;
    b = 3 ;
    c = b + *a ;
    return c ;
}
```

```
int main()
{
    int a = 4 ;
    int b = 7 ;
    int c = 9 ;
    int d = 1 ;
    d = fonction_addition(&a,b,c) ;
}
```





A = 4
B = 7
C = 9
D = 5



- Les pointeurs sont une manière indirecte de modifier des valeurs, utile dans ce cas ! **L'adresse est copiée, mais la destination est toujours la même !**

1- Rappel sur les pointeurs

- En résumé :

int x; 

x = 4; 

int *p;  

p = &x;   **& : Accès à l'adresse**

*p = 7;   *** : Accès au contenu**

1- Rappel sur les pointeurs

Rappel sur les pointeurs

```
int i = 42;
```

Adresse	Contenu	Utilisation
107	?	disponible
106	?	disponible
105	?	disponible
104	?	disponible
103	42	i
102		
101		
100		

1- Rappel sur les pointeurs

Rappel sur les pointeurs

```
int i = 42;  
int* pi;
```

Adresse	Contenu	Utilisation
107	?	pi
106		
105		
104		
103	42	i
102		
101		
100		

1- Rappel sur les pointeurs

Rappel sur les pointeurs

```
int i = 42;  
int* pi = &i;
```

Adresse	Contenu	Utilisation
107	100	pi
106		
105		
104		
103	42	i
102		
101		
100		

1- Rappel sur les pointeurs

Rappel sur les pointeurs

```
int i = 42;  
int* pi = &i;  
  
pi++;  
  
// pi vaut 104  
// i vaut 42
```

Adresse	Contenu	Utilisation
107	104	pi
106		
105		
104		
103	42	i
102		
101		
100		

1- Rappel sur les pointeurs

Rappel sur les pointeurs

```
int i = 42;  
int* pi = &i;  
  
(*pi)++;  
  
// pi vaut 100  
// i vaut 43
```

Adresse	Contenu	Utilisation
107	100	pi
106		
105		
104		
103	43	i
102		
101		
100		

1- Rappel sur les pointeurs

Rappel sur les pointeurs

```
int i = 42;  
int* pi;  
  
(*pi)++;  
  
/* a toutes les chances de  
planter (segmentation fault) */
```

Adresse	Contenu	Utilisation
107	?	pi
106		
105		
104		
103	42	i
102		
101		
100		

2- Initialisation des pointeurs

- **En C :** `char* chaîne = NULL;`
 - NULL est un entier (vaut réellement 0)
 - `int i = NULL;`
 - Cette ligne ne déclenche pas d'erreur ! (mais peut déclencher un warning)

2- Initialisation des pointeurs

- **En C :** `char* chaîne = NULL;`
 - NULL est un entier (vaut réellement 0)
 - `int i = NULL;`
 - Cette ligne ne déclenche pas d'erreur ! (mais peut déclencher un warning)
- **En C++ >= 11 :** `char* chaîne = nullptr;`
 - Le type de nullptr est spécial **et ne s'applique qu'aux pointeurs.**
 - `int i = nullptr;` donne ainsi une erreur à la compilation.

→ Permet d'éviter les erreurs !

3- Les références

- **En C :**
 - Les pointeurs sont une notion très puissante et légère qui permet de nombreuses choses.
 - **Mais pour les cas simples, comme le passage d'une variable que l'on souhaite modifier à une fonction -> c'est un peu lourd**
 - On introduit une nouvelle notion en C++ en plus des pointeurs
→ : **Les références**

3- Les références

- Une **référence** est un synonyme (alias) d'une autre variable
 - La variable originale et sa référence ont le même contenu et la même adresse
- **Objectif** : ne plus utiliser les pointeurs dans certains cas, notamment le passage d'une variable à modifier dans une fonction.
- **Syntaxe** :

```
int i = 20;  
int& refi = i;
```
- Ensuite c'est auto-magique ! On peut l'utiliser tel quel partout !



3- Les références

- Exemple de passage par référence :

```
int fonction_addition(int& u, int v, int w)
{
    u = 2 ;
    v = 3 ;
    w = u + v ;
    return w ;
}
```

```
int main()
{
    int a = 4 ;
    int b = 7 ;
    int c = 9 ;
    int d = 1;
    d = fonction_addition(a,b,c) ;
}
```

a = 2
b = 7
c = 9
d = 5



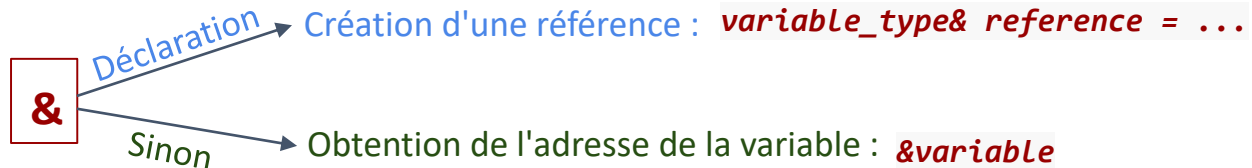
3- Les références

- Quelques règles



1. Une référence doit être initialisée dès sa déclaration
2. La même référence ne doit pas faire référence à deux différentes variables
3. Si la référence est modifiée, la variable initiale sera aussi modifiée

- référence \neq adresse



3- Les références

```
#include <stdio.h> // printf()

int main(){
    int i = 1;
    int& ref = i;

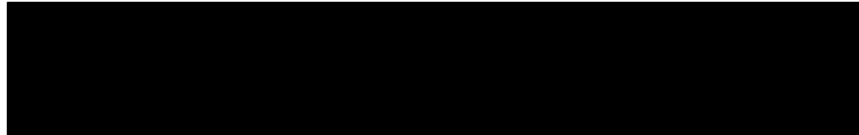
    printf("&i: %p, &ref: %p\n", &i, &ref);

    i++;
    printf("i: %d, ref: %d\n", i, ref);

    ref++;
    printf("i: %d, ref: %d\n", i, ref);

    return 0;
}
```

Standard output



3- Les références

On peut tout à fait accéder à l'adresse d'une référence, et c'est la même adresse que celle de la variable d'origine !

```
#include <stdio.h> // printf()

int main(){
    int i = 1;
    int& ref = i;

    printf("&i: %p, &ref: %p\n", &i, &ref);

    i++;
    printf("i: %d, ref: %d\n", i, ref);

    ref++;
    printf("i: %d, ref: %d\n", i, ref);

    return 0;
}
```

Standard output

```
&i: 0x7fffd0898a0c, &ref: 0x7fffd0898a0c
i: 2, ref: 2
i: 3, ref: 3
```

3- Les références

Code équivalent avec les pointeurs

```
#include <stdio.h> // printf()

int main() {
    int i = 1;
    int* ref = &i;

    printf("&i: %p, ref: %p\n", &i, ref);

    i++;
    printf("i: %d, *ref: %d\n", i, *ref);

    (*ref)++;
    printf("i: %d, *ref: %d\n", i, *ref);

    return 0;
}
```

Standard output

```
&i: 0x7fffd0898a0c, ref: 0x7fffd0898a0c
i: 2, *ref: 2
i: 3, *ref: 3
```

3- Les références

Qu'affiche le programme
suivant ?

```
#include <iostream>

int main() { int x = 5; int y = x; int& r = x;

    std::cout << "x = " << x << '\n';
    std::cout << "y = " << y << '\n';
    std::cout << "r = " << r << '\n';
    std::cout << "*****" << '\n';

    x = 7;

    std::cout << "x = " << x << '\n';
    std::cout << "y = " << y << '\n';
    std::cout << "r = " << r << '\n';
    std::cout << "*****" << '\n';

    y = 8;

    std::cout << "x = " << x << '\n';
    std::cout << "y = " << y << '\n';
    std::cout << "r = " << r << '\n';
    std::cout << "*****" << '\n';

    r = 2;

    std::cout << "x = " << x << '\n';
    std::cout << "y = " << y << '\n';
    std::cout << "r = " << r << '\n';

}
```

3- Les références

Solution

```
x = 5  
y = 5  
r = 5  
*****  
  
x = 7  
y = 5  
r = 7  
*****  
  
x = 7  
y = 8  
r = 7  
*****  
  
x = 2  
y = 8  
r = 2
```

4- Résumé sur les passages

- Le passage de paramètres dans une fonction se fait :
 - a. **par valeur**
ou
 - b. **par adresse**
ou
 - c. **par référence**



4- Résumé sur les passages

1. Le passage de paramètre par valeur = passage par recopie (valable en C et C++)

→ c'est le comportement par défaut

```
void increment(int i){  
    i++;  
}
```

```
int i = 42;  
increment(i);
```

Adresse	Contenu	Utilisation
107	?	disponible
106	?	disponible
105	?	disponible
104	?	disponible
103	?	disponible
102	?	disponible
101	?	disponible
100	?	disponible

4- Résumé sur les passages

1. Le passage de paramètre par valeur = passage par recopie (valable en C et C++)

→ c'est le comportement par défaut

```
void increment(int i){  
    i++;  
}
```

→ `int i = 42;`
`increment(i);`

Adresse	Contenu	Utilisation
107	?	disponible
106	?	disponible
105	?	disponible
104	?	disponible
103	?	i
102		
101		
100		

4- Résumé sur les passages

1. Le passage de paramètre par valeur = passage par copie (valable en C et C++)

→ c'est le comportement par défaut

```
void increment(int i){  
    i++;  
}
```

→

```
int i = 42;  
increment(i);
```

Adresse	Contenu	Utilisation
107	?	disponible
106	?	disponible
105	?	disponible
104	?	disponible
103	42	i
102		
101		
100		

4- Résumé sur les passages

1. Le passage de paramètre par valeur = passage par recopie (valable en C et C++)

→ c'est le comportement par défaut

```
→ void increment(int i) {  
    i++;  
}
```

```
→ int i = 42;  
   increment(i);
```

Adresse	Contenu	Utilisation
107	?	i
106		
105		
104		
103	42	i
102		
101		
100		

4- Résumé sur les passages

1. Le passage de paramètre par valeur = passage par recopie (valable en C et C++)

→ c'est le comportement par défaut

```
→ void increment(int i){  
    i++;  
}
```

```
→ int i = 42;  
    increment(i);
```

Adresse	Contenu	Utilisation
107	43	i
106		
105		
104		
103	42	i
102		
101		
100		

4- Résumé sur les passages

1. Le passage de paramètre par valeur = passage par recopie (valable en C et C++)

→ c'est le comportement par défaut

```
void increment(int i){  
    i++;  
}
```

→

```
int i = 42;  
increment(i);
```

Adresse	Contenu	Utilisation
107	43	disponible
106		disponible
105		disponible
104		disponible
103	42	i
102		
101		
100		

4- Résumé sur les passages

2. Le passage de paramètre par adresse (valable en C et C++)

→ c'est le comportement par défaut

```
void increment(int* pi){  
    (*pi)++;  
}
```

→

```
int i = 42;  
increment(&i);
```

Adresse	Contenu	Utilisation
107	?	disponible
106	?	disponible
105	?	disponible
104	?	disponible
103	42	i
102		
101		
100		

4- Résumé sur les passages

2. Le passage de paramètre par adresse (valable en C et C++)

```
→ void increment(int* pi){  
    (*pi)++;  
}  
  
→ int i = 42;  
   increment(&i);
```

Adresse	Contenu	Utilisation
107	100	pi
106		
105		
104		
103	42	i
102		
101		
100		

4- Résumé sur les passages

2. Le passage de paramètre par adresse (valable en C et C++)

```
→ void increment(int* pi){  
    (*pi)++;  
}
```

```
→ int i = 42;  
   increment(&i);
```

Adresse	Contenu	Utilisation
107	100	pi
106		
105		
104		
103	43	i
102		
101		
100		

4- Résumé sur les passages

2. Le passage de paramètre par adresse (valable en C et C++)

```
void increment(int* pi){  
    (*pi)++;  
}
```

```
int i = 42;  
increment(&i);
```



Adresse	Contenu	Utilisation
107	100	disponible
106		disponible
105		disponible
104		disponible
103	43	i
102		
101		
100		

4- Résumé sur les passages

3. Le passage de paramètre par référence (SPÉCIFIQUE AU LANGAGE C++)

```
void increment(int& var){  
    var++;  
}
```

```
int i = 42;  
increment(i);
```

Adresse	Contenu	Utilisation
107	?	disponible
106	?	disponible
105	?	disponible
104	?	disponible
103	?	disponible
102	?	disponible
101	?	disponible
100	?	disponible

4- Résumé sur les passages

3. Le passage de paramètre par référence (SPÉCIFIQUE AU LANGAGE C++)

```
void increment(int& var){  
    var++;  
}
```

```
→ int i = 42;  
   increment(i);
```

Adresse	Contenu	Utilisation
107	?	disponible
106	?	disponible
105	?	disponible
104	?	disponible
103	42	i
102		
101		
100		

4- Résumé sur les passages

3. Le passage de paramètre par référence (SPÉCIFIQUE AU LANGAGE C++)

```
void increment(int& var){  
    var++;  
}
```

```
int i = 42;  
increment(i);
```

Adresse	Contenu	Utilisation
107	?	disponible
106	?	disponible
105	?	disponible
104	?	disponible
103	42	i var
102		
101		
100		

4- Résumé sur les passages

3. Le passage de paramètre par référence (SPÉCIFIQUE AU LANGAGE C++)

```
void increment(int& var){  
    var++;  
}
```

```
int i = 42;  
→ increment(i);
```

Adresse	Contenu	Utilisation
107	?	disponible
106	?	disponible
105	?	disponible
104	?	disponible
103	43	i var
102		
101		
100		

4- Résumé sur les passages

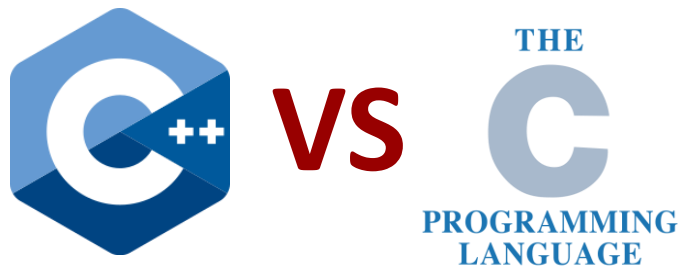
3. Le passage de paramètre par référence (SPÉCIFIQUE AU LANGAGE C++)

```
void increment(int& var){  
    var++;  
}
```

```
int i = 42;  
increment(i);
```

Adresse	Contenu	Utilisation
107	?	disponible
106	?	disponible
105	?	disponible
104	?	disponible
103	43	i var
102		
101		
100		

Différences entre C et C++



Les fonctions en C++

1- La surcharge des fonctions

- **En C** : toutes les fonctions doivent avoir des noms différents
- **En C++** :
 - **Les fonctions peuvent avoir le même nom** mais le type/nombre des arguments doivent être différents → **Fonctions surchargées**
 - La fonction à utiliser est choisie, par le compilateur, en se basant sur son prototype :

```
void printParams(int i1, int i2); ✓  
void printParams(int i, float f); ✓  
void printParams(float f); ✓  
int printParams(int i1, int i2); X printParams() already exists  
with parameters (int, int)
```

1- La surcharge des fonctions

- Exemple :

```
float fonction_addition(float a, float b)
{
    float resultat ;
    resultat = a + b ;
    return resultat ;
}

int fonction_addition(int a, int b)
{
    int resultat ;
    resultat = a + b ;
    return resultat ;
}
```

```
int main()
{
    int a = 1 ;
    int b = 1 ;

    float c = 1.5 ;
    float d = 1.5 ;

    float resulf;
    int resuld;

    resuld = fonction_addition(a,b) ;
    resulf = fonction_addition(c,d) ;
}
```

1- La surcharge des fonctions

- Exemple :

```
float fonction_addition(float a, float b)
{
    float resultat ;
    resultat = a + b ;
    return resultat ;
}

int fonction_addition(int a, int b)
{
    int resultat ;
    resultat = a + b ;
    return resultat ;
}
```

```
int main()
{
    int a = 1 ;
    int b = 1 ;

    float c = 1.5 ;
    float d = 1.5 ;

    float resulf;
    int resuld;

    resuld = fonction_addition(a,b) ;
    resulf = fonction_addition(c,d) ;
}
```

Ça, ça ne marche pas en C !



2- nullptr et surcharge

- **En C :** `char* chaine = NULL;`
 - NULL est un entier (vaut réellement 0)
 - Pour les deux fonctions suivantes :

```
void f(int* pi);  
void f(int i);
```

L'appel `f(NULL)` génère l'erreur

call of overloaded f(NULL) is ambiguous

- **En C++ >= 11 :** `char* chaine = nullptr;`
 - Le type de nullptr est `nullptr_t`
 - L'appel `f(NULL)` fonctionne parfaitement

3- Arguments par défauts

- Une valeur par défaut est assignée à l'argument d'une fonction si aucune valeur n'est affectée au moment de l'appel à la fonction
- Un argument par défaut doit apparaître au moins comme dernier argument de la fonction

```
int increment(int i, int step = 1);
```



```
int increment(int i = 0, int step = 1);
```



```
int increment(int i = 0, int step);
```



3- Arguments par défauts

- Les valeurs par défaut sont définies dans le prototype de la fonction **ou** dans l'implémentation (**mais pas dans les deux**)

```
// prototype
int increment(int i, int step = 1);

// implementation
int increment(int i, int step = 1){
    return i + step;
}
```

```
// prototype
int increment(int i, int step = 1);

// implementation
int increment(int i, int step){
    return i + step;
}
```



3- Arguments par défauts

- L'idée :

```
int fonction_addition(int a, int b = 0.0, int c = 0.0)
{
    int resultat ;
    resultat = a + b + c ;
    return resultat ;
}
```

```
resultat1 = 1
resultat2 = 11
resultat3 = 14
```

```
int main()
{
    int resultat1 ;
    int resultat2 ;
    int resultat3 ;

    resultat1 = fonction_addition(1) ;
    resultat2 = fonction_addition(5,6) ;
    resultat3 = fonction_addition(5,6,3) ;
}
```



3- Arguments par défauts

- L'utilisation du même nom de fonction avec des arguments par défaut peut générer des erreurs :

```
// prototypes
void f1(int i, int j = 1);
void f1(float i, float j = 1.0);

// call
int main(){
    int i = 0;
    float f = 0;

    f1(i);
    f1(f);

    return 0;
}
```

**Exemple
correct**

```
// prototypes
void f1(int i, int j = 1);
void f1(int i, float j = 1.0);

// call
int main(){
    int i = 0;

    f1(i);

    return 0;
}
```

**Exemple
incorrect**

Car le compilateur ne peut pas savoir qui appeler !

Différences entre C et C++

...