Programmation en langage C++

Les bases du C++

Version imprimable du cours *Programmation en langage C++*

Julien Yves ROLLAND (julien.rolland@univ-fcomte.fr)
Laboratoire de Mathématiques de Besançon
Université de Franche-Comté

1.1

Buts et objectifs

Buts et objectifs de ce chapitre.

1. Situer le positionnement du C++ parmi les langages de programmation ;
2. Comprendre le fonctionnement et la structure d'un code C++ ;
3. Identifier et utiliser les types et opérations de base d'un code C++ ;
4. Préparer le terrain pour l'introduction des notions avancées.

1.2

Table des matières

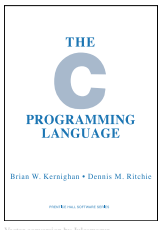
1.3

1 Introduction historique

1.4

L'héritage du C

Le langage C



— 1972 : Conception du langage C pour les besoins du développement du système Unix par Dennis Ritchie et Ken Thompson.
— 1978 : Brian Kernighan et Dennis Ritchie publient « The C Programming language ».

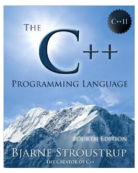
Popularisation et normalisation

— Années 80 : Popularisation du C permettant la programmation structurée et typée des systèmes.
— 1983-1989 : Définition des premières normes ANSI C (C89) puis ISO C (C90).
— 1999 et 2011 : Révisions du langage C (C99 puis C11).

1.5

Le langage C++

Le langage C++



Bjarne Stroustrup.

— Années 80 : Émergence de la « Programmation Orientée Objet »(POO).
— 1983 : Le « C with Classes » de Bjarne Stroustrup (AT&T) devient le C++.
— 1985 : Publication de « The C++ Programming Language ».

Normalisation

— 1985-1998 : Les éditions de « The C++ Programming Language » servent de norme.
— 1998 : Définition de la première norme ISO C++98
— 2011 : Troisième et actuelle révision ISO : C++11

1.6

Positionnement du C

Aujourd'hui le langage C, malgré son positionnement très bas niveaux (par rapport au Python, Ruby, Java,...) est l'un des plus utilisés.

Héritant de sa conception principalement tournée vers la programmation des systèmes, ses atouts majeurs sont :

— des composants bas niveau relativement polyvalents et succincts ;
— une excellente application dans la programmation des systèmes ;
— un déploiement possible partout et sur tout ;
— une intégration à l'environnement Unix.

1.7

Positionnement du C++

Le C++ est apparu comme supplément au C ajoutant principalement les spécificités nécessaires à la Programmation Orientée Objet.

Ses grandes caractéristiques sont :

— Une norme (ANSI/ISO) cadrée et régulièrement révisée ;
— D'excellents résultats en terme de performance, d'efficacité et de flexibilité ;
— Une flexibilité sur le choix du paradigme de programmation (POO non obligatoire).
— Une relative complexité héritée de la compatibilité avec le C ;

1.8

Pourquoi apprendre le C++ ?

De par sa flexibilité le C++ est apte à remplir des tâches appartenant à différents domaines :

— traitement local ou distribué ;
— utilisation de fonctions numériques, graphiques ;
— traitement des interactions utilisateurs et accès aux bases de données.

Cette polyvalence en fait un des langages les plus utilisés dans la recherche et le calcul scientifique, de nombreuses librairies de calcul sont écrites ou portées pour ce langage.

Enfin, il a largement influencé la conception d'autres langages (C#, Java,...). Sa maîtrise est donc utile dans le cadre général de la programmation.

1.9

Avantages du C++

Le C++ n'est pas le plus concis ni le plus « propre » des langages mais il est :

— suffisamment « propre » pour l'enseignement des concepts de base ;
— suffisamment efficace et souple pour l'élaboration de programme exigeant ;
— suffisamment ouvert pour s'intégrer dans un écosystème hétérogène ;
— suffisamment complet pour l'enseignement et l'usage des techniques de programmation avancées.

> « C++ is a language that you can grow with. »
>
> *Bjarne Stroustrup*
> *The C++ Programming Language*

1.10

2 Concepts et structures du C++

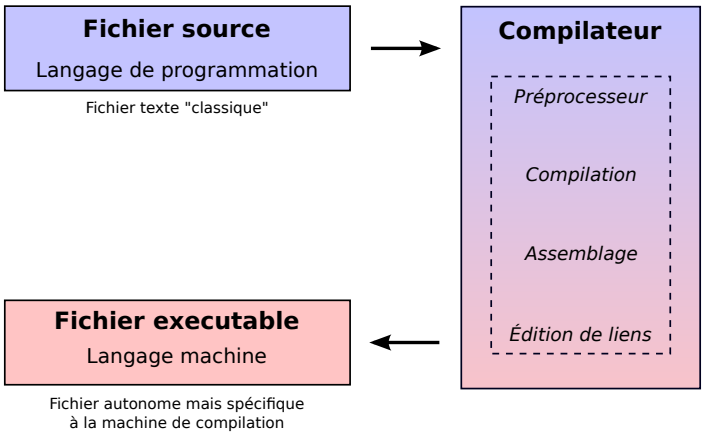
2.1 La compilation

1.11

C++, langage compilé

Un langage compilé ?

Le C++ est un langage compilé, les fichiers d'origine doivent être « transformés » pour générer des fichiers compréhensibles par la machine. Un langage interprété (Matlab, Scilab, R, Python) est directement lisible par un « interpréteur ».



1.12

Rôle du compilateur

Le compilateur est un programme (ou une chaîne d'outils logiciels) qui a pour charge de :

— Vérifier la syntaxe du fichier source ;
— Générer des erreurs compréhensibles pour l'utilisateur ;
— Optimiser tout ou partie du code selon les paramètres de l'utilisateur ;
— Regrouper les différents sous-programmes constituant le programme final ;
— Générer un code assembleur optimisé pour la machine cible.

1.13

Étapes de compilation

Préprocesseur
Il « nettoie » le code en interprétant les fichiers sources selon les directives propres (identifiées par le symbole #).

Compilation
Il traduit le fichier du préprocesseur en assembleur : Un suite d'instructions pour le processeur.

Assemblage
Le code assembleur est transformé en fichier binaire, instructions en code machine (adapté à l'architecture cible). Étape souvent regroupée dans la compilation.

Édition de liens
Regroupe les différents fragments du programme (plusieurs sources, librairies externes,…). Produit le fichier exécutable.

1.14

Faune et flore locales

Plusieurs types de fichier sont rencontrés durant la compilation.

Les fichiers sources
— .cpp : Le fichier source (implémentation)
— .hpp ou .h : Fichier en-tête (préprocesseur, macro, …)

Fichiers intermédiaires
— .o : Fichier objet (fichier temporaire à assembler)
— .a : Fichier archive (librairies pré-compilées)

Fichiers de sortie
— .exe (Windows) : Le fichier exécutable
— .so (Linux) ou .dll (Windows) : Librairie compilée dynamique
— .a : Librairie compilée statique

1.15

2.2 Exemples de code

Exemple commenté de code C++ : Hello World

1.16

```
1 // Mon premier programme C++
2 #include <iostream>
3 using namespace std;
4 int main()
5 {
6   cout<<"Hello World !"<<endl;
7   return 0;
8 }
```

1.17

Décodage

Commentaires

```
// Mon premier programme C++
```

Commentaires (supprimé par le préprocesseur).

Directive au préprocesseur

```
#include <iostream>
```

Demande d'inclure les fichiers d'une librairie standard C++ (`iostream`) du système (les symboles <>) pour permettre l'utilisation de la fonction `cout`.

Instructions

```
using namespace std;
```

Les fonctions de la librairie standard (dont `iostream`) sont déclarées dans un espace de noms : le `namespace` « std ». On demande à y avoir accès.

1.18

Plus de décodage

Déclaration

```
int main()
{
...
}
```

On déclare une fonction (symboles `{ }`) nommée `main` sans argument (parenthèses vides) et retournant un `int` (un entier).

Instructions

```
cout<<"Hello World !"<<endl;
return 0;
```

Deux lignes d'instruction, elles se terminent par un `;`

`cout` affiche une chaîne de caractères (contenue dans `"..."`) et ajoute un retour à la ligne (`endl` pour « end line »). `return` permet de sortir de `main` en retournant 0 (un entier).

1.19

Exemple commenté de code C++ : Racines carrés

1.20

```cpp
#include <iostream>
#include <cmath>
#define NFOIS 5

using namespace std;

int main(void){
  int i;
  float x;
  float racx = 0;

  cout<<"Bonjour"<<endl;
  cout<<"Je␣vais␣calculer␣"<<NFOIS<<"␣racines␣carrées␣réelles.\n";

  for(i=0; i<NFOIS; i++){
    cout<<"Donnez␣un␣nombre:␣";
    cin>>x;
    if(x<0.0){
      cout<<"Le␣nombre␣"<<x<<"␣ne␣possède␣pas␣de␣racine␣réelle\n";
    }//end if
    else{
      racx = sqrt(x);
      cout<<"La␣racine␣carrée␣de␣"<<x<<"␣est:␣"<<racx<<endl;
    }//end else
  }//end for
  cout<<"Fin␣du␣programme\n";
}//end main
```

Décodage

Directive au préprocesseur

```cpp
#define NFOIS 5
#include <cmath>
```

Le préprocesseur remplace automatiquement les occurrences de `NFOIS` par le nombre 5. On charge la bibliothèque de fonctions mathématiques (pour `sqrt()`).

Instruction de déclaration

```cpp
int i;
float x;
```

On déclare des variables (`i` et `x`) qui seront respectivement un entier et un réel.

1.21

Encore du décodage

Déclaration et initialisation

```cpp
float racx = 0.0;
```

On déclare la variable `racx` de type réel AVEC initialisation à la valeur 0.

Opérateurs de flux

```cpp
cout<<"Bonjour"<<endl;
cin>>x;
```

Les symboles `<<` et `>>` sont des opérateurs de flux. Il agissent sur des flux (des données typées) qu'ils manipulent. Ici `cout` envoie la chaîne de caractères vers l'écran, `cin` envoie la saisie utilisateur vers une variable.

1.22

Toujours plus de décodage

Instruction de contrôle : boucle

```cpp
for(i=0; i<NFOIS; i++){
}
```

Ouvre une boucle d'itération sur la variable `i`, de la valeur 0 à la valeur `NFOIS`. L'instruction `++` indique d'avancer le pas « d'une unité » (ici entière).

Instruction de contrôle : Condition

```
if(x<0.0){
}
else{
}
```

`if` ouvre un bloc d'instruction conditionnel : Si la condition est remplie le bloc est exécuté, sinon le bloc `else` est exécuté, s'il existe.

1.23

2.3 Les composantes de base

1.24

Les composantes élémentaires du C++

Un programme en C ou en C++ est constitué de 7 groupes de composantes élémentaires :

- les identificateurs
- les mots-clés
- les constantes
- les chaînes de caractères
- les opérateurs
- les signes de ponctuations
- les commentaires

1.25

Les identificateurs

L'identificateur sert à donner un nom à une entité du programme. Il peut désigner :

- un nom de variable ou de fonction ;
- un `struct`, `class`, `enum`, `union` ;
- un type défini par `typedef`.

L'identificateur est composé d'une suite de caractères parmi :

- des lettres (majuscules ou minuscules, différenciées, pas d'accent) ;
- des chiffres ;
- un blanc souligné (blanc_souligné).

L'identificateur ne peut pas commencer par un chiffre !

1.26

Les mots-clés

Certains mots sont réservés dans le langage et ne peuvent donc pas être utilisés comme identificateurs. Voici une liste non exhaustive :

```
auto        do          int         struct
bool        double      long        switch
break       else        new         typedef
case        enum        register    union
char        extern      return      unsigned
class       float       short       using
const       for         signed      void
continue    goto        sizeof      volatile
default     if          static      while
```

1.27

Les commentaires

Commentaire en ligne, bloc de commentaire

Le commentaire d'une ligne est précédé du symbole `//`. On peut créer un bloc de commentaire en le bornant des symboles `/*` et `*/`.

```
1 /* Ce code est un exemple ne servant
2    à
3    rien créé par Julien Yves Rolland.
4 */
5 int main ()
6 {
7   // Je ne sers à rien !
8   return 0;
9 }
```

1.28

2.4 La structure d'un programme

1.29

Expression et instructions

Expression
Suite de composantes élémentaires syntaxiquement correcte.

```
((i >= 0) && (i < 10)) || (P[i] != 0)
```

Instruction et bloc d'instructions
Expression suivie d'un point virgule. L'accolade permet de créer des bloc d'instructions.

```
if( x != 0 ){
z = y / x;
t = y % x;
}
```

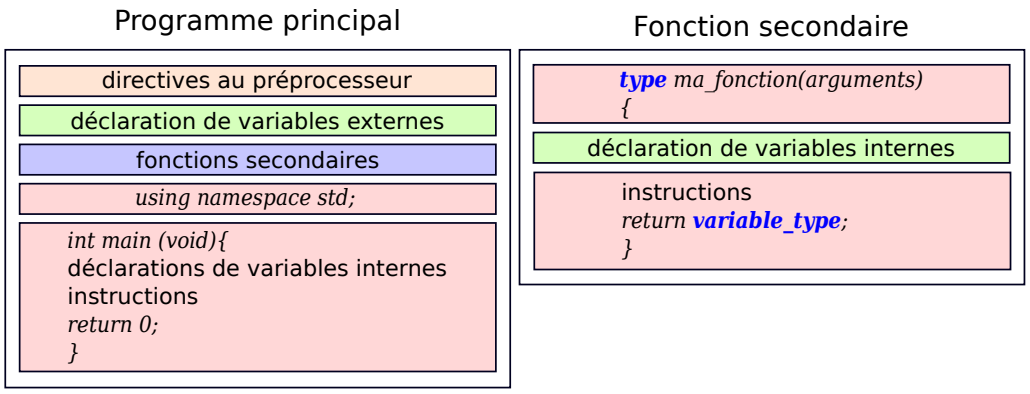
Déclaration composée

```
int a, b;
float x=1.5, y;
```

1.30

Structure classique d'un programme C++

En résumé, un programme classique et simple se présente sous la forme suivante :



1.31

Fonctions secondaires

Les fonctions secondaires sont des fonctions annexes au programme principal et qui sont souvent déclarées avant leur utilisation dans le bloc d'instruction `main` (ou la fonction secondaire l'utilisant).

```
1  int somme(int a, int b)
2  {
3  int resultat;
4  resultat = a+b;
5  return resultat;
6  }
7
8  int produit(int a, int b)
9  {
10   int resultat = 0;
11   for(int i=0; i<b; i++){
12     resultat = somme(resultat, a);
13   }
14   return resultat;
15 }
```

1.32

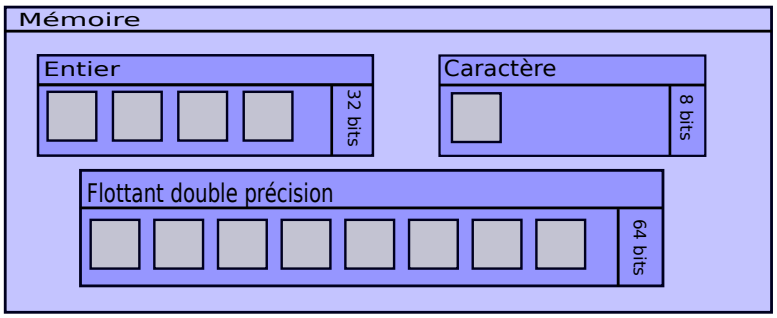
2.5 Un mot sur la mémoire

1.33

Représentation mémoire des objets

Représentation mémoire

La mémoire est une suite continue d'octets (1 octet = 8 bits). Chaque octet est identifié par son adresse qui est un entier. Deux octets contigus en mémoire ont une adresse qui diffère d'une unité. Le type définit le nombre d'octets consécutifs réservés.



1.34

Organisation de la mémoire pour un logiciel

Le langage C++ permet de gérer l'utilisation de la mémoire (langage bas niveau) avec un certain degré d'abstraction (langage haut niveau). Il permet de choisir la zone mémoire utilisée.

Le tas et la pile

La totalité de la mémoire allouée à un programme est divisée en deux zones :

— La pile, allocation/désallocation rapide car zone hiérarchisée (*first in / last out*) mais de taille limitée ;

— Le tas, le reste, zone non hiérarchisée et donc plus lente mais de taille bien plus importante.

Par exemple, sur une machine Linux moderne, la taille par défaut de la pile, définie par l'OS, est de 8MB (commande "*ulimit -a*"), le tas « peut » occuper tout le reste.

1.35

Gestion de la mémoire en C++, vie et mort d'une entité

Sans entrer dans les détails et en simplifiant, la durée de vie d'une entité dans le code est définie par la méthode de déclaration (et donc sa nature) :

— Un type de base ou une classe d'objets existe au sein d'un bloc d'instruction (`{}`). On appelle cet espace sa « portée ». Ils existent dans la pile ;

— Toute entité créée par l'instruction `new` est créée dans le tas, n'est pas limitée par un bloc et doit être détruite par l'instruction `delete` ;

— Tout ce qui n'est pas déclaré dans un bloc ou tout ce qui n'a pas été détruit par `delete` n'est désalloué qu'à la fin du programme.

Généralement le tas est utilisé pour stocker les instances des classes d'objet, mais pas toujours.

Attention lors de l'usage des `new` à ne pas oublier de `delete` (fuite mémoire).

1.36

3 Les bases du langage C++

3.1 Les types de base

1.37

C++ : Un langage typé

Langage typé ?
C/C++ sont des langages typés, c-à-d que toute variable, constante ou fonction est d'un type précis qui définit sa représentation en mémoire.

Les types de base en C/C++
- les entiers (mot-clé `int`);
- les flottants (mot-clé `float` ou `double`);
- les caractères (mot-clé `char`);
- les booléens (mot-clé `bool`).

1.38

Un mot sur les espaces de noms

Espaces de noms
Tous les objets du C++ sont distribués dans des espaces de noms (*namespace*). On peut les voir comme des librairies regroupant un jeu de fonctions, variables, classes,...

La qualification explicite, mot-clé `using` et opérateur `::`
`using` permet d'intégrer des espaces de nom dans l'espace global statique (le fichier). L'opérateur de qualification explicite `::` doit être utilisé dans le cas contraire.

```
1  using namespace std;
2  int main(){
3    cout<<"Coucou␣?"<<endl;
4    return 0;
5  }
```

À la place de :

```
1  int main(){
2    std::cout<<"Coucou␣?"<<std::endl;
3    return 0;
4  }
```

1.39

Le type entier : `int`

Le type `int`
Il existe en 3 versions :
- `short (int)` (au plus la taille d'un int)
- `int`
- `long (int)` (au moins la taille d'un int)

Gestion du signe
Usuellement, le bit de poids fort (premier) est réservé pour définir le signe. Les 3 versions d'`int` peuvent être préfixées du mot-clé `unsigned`, qui définit alors un entier positif. Le bit fort est alors utilisé pour représenter l'entier (on gagne 1 bits).

Mot-clé `sizeof`
`sizeof(`*type*`)` permet d'obtenir un entier correspondant aux nombre d'octets utilisés pour représenter le *type*.

1.40

Représentation en mémoire

Représentation d'un `int`

Si l'entier est signé, le bit de poids fort est réservé pour le signe. Les autres bits permettent de représenter l'entier en base 2. Le nombre de bits utilisé n'est pas exactement spécifié dans la norme mais un `int` est codé sur au moins 16bits. Exemple : Entier « 12 » codé sur 32bits *0 0...(25 fois 0)...01100*

Plage de représentation d'un `int`

Sur un ordinateur de nos jours (Unix64 - LP64 data model) :

`short int`	$[-2^{15};2^{15}[$	16 bits	2 octets
`int`	$[-2^{31};2^{31}[$	32 bits	4 octets
`long int`	$[-2^{63};2^{63}[$	64 bits	8 octets
`unsigned short int`	$[0;2^{16}[$	16 bits	2 octets
`unsigned int`	$[0;2^{32}[$	32 bits	4 octets
`unsigned long int`	$[0;2^{64}[$	64 bits	8 octets

1.41

Le type flottant : `float`

Le type `float`

Il existe en 3 versions qui diffèrent par leur précision :

- `float` (simple précision)
- `double` (double précision)
- `long double` (précision étendue)

Représentation des flottants

Ils représentent de manière approchée les nombres réels sous la forme :

$$(-1)^s M \cdot B^E$$

où s définit le signe, M est la mantisse, B la base (2 ou 16) et E l'exposant. Seuls signe, mantisse et exposant sont stockés dans les octets de représentation.

1.42

Le type flottant : `float` (suite)

Particularités de représentation

- Le nombre de bits utilisés n'est pas exactement spécifié dans la norme C++. Une norme existe pour les flottants en informatique (IEE 754) où le `float` du C++ est codé sur 32bits (*binary32*).
- L'exposant E est toujours codé comme positif. Un décalage de valeur $2^{\text{size}(E)-1}-1$ est utilisé pour retrouver la valeur (pouvant être négative).
- Des valeurs particulières de la mantisse M et de l'exposant E permettent de définir les infinis, le zéro exact et le `NaN` (Not a Number).

Plage de représentation des `float`

Type	Expo.	Mant.	Plage	Significatifs
`float`	8	23	$\pm 3.4 \cdot 10^{\pm 38}$	≈ 7
`double`	11	52	$\pm 1.7 \cdot 10^{\pm 308}$	≈ 16
`long double`	15+	64+	$\pm 1.2 \cdot 10^{\pm 4932}$	≈ 20

1.43

Les types `char` et `bool`

Le type `char`

Le type `char` est codé sur 1 octet.

```
char c = 'h';
cout<<"c= "<<c<<endl;
```

Il est possible de représenter une chaîne de caractères avec le type `string` nécessitant la directive `#include <string>`.

```
string s;
s = "L'écriture !";
```

Le type `bool`

Le mot-clé `bool` désigne le booléen prenant valeur `False` (0) ou `True` (entier $\neq 0$). Il est codé sur un seul octet.

1.44

Exemple

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

using namespace std;

int main()
{
  string s1, s2, s;
  cout<<"Entrer une chaîne de caractères\n";
  cin>>s1;
  cout<<"Entrer une autre chaîne de caractères\n";
  cin>>s2;
  s = s1 + s2;
  cout<<s<<endl;
  return 0;
}
```

1.45

3.2 Les constantes

1.46

Définition et types

Une constante ?
Une constante est une expression littérale dans le code (« en dur ») qui servira souvent à l'initialisation d'une variable pour une partie non interactive du code.

Types de constantes
Le type de la constante dépend de la façon dont elle est écrite lors de sa définition. Il en existe de quatre types :

- entière ;
- flottante ;
- caractère ;
- chaîne de caractères.

1.47

Constante entière

Choix de la base
La constante de type entière peut être exprimée dans 3 bases différentes. La base est définie par le préfixe utilisé :

- base 10, classique (et défaut) : `int x = 42;`
- base 8, préfixé par un `0` : `int x = 052;`
- base 16, préfixe par `0x` (ou `0X`) : `int x = 0x2a`

Choix du format
Le format de représentation de l'entier peut lui aussi être choisi lors de l'utilisation à l'aide d'un suffixe :

- suffixe `u` (ou `U`) pour `unsigned` : `x = 18U + 24u;`
- suffixe `l` (ou `L`) pour `long` : `x = 15 + 27l;`
- suffixe `ll` (ou `LL`) pour `long long` : `x = 15 + 27LL;`

Ces suffixes ont leur utilité lors de passage de types implicite (« cast »).

1.48

Constante flottante

Expression
La constante flottante est exprimée par l'usage d'un point et/ou d'un `e` (ou `E`) dans l'expression.

Choix du type
Le type par défaut est `double`. On peut, lui aussi, le choisir grâce à un suffixe :

- Type double : `x = 3.0 + 3e1` (3 et 30 en flottant)
- Type float : `x = 3f` ou `x = 3F`
- Type long double : `pi = 3.14159L`

1.49

Constantes caractères

Caractère et chaîne de caractères
Un caractère s'obtient à l'aide d'apostrophes, la chaîne de caractères avec l'apostrophe double.

```
char toto = 't';
string test = "Hello␣!";
```

Écriture et caractères non-imprimables
Tous les caractères ont aussi un code octal ou hexa (ex : `\64` ou `\x40` pour « @ »). Quelques caractères disposent d'une notation simplifiée (appelés « escape code ») :

`\n`	newline	`\a`	alert (beep)
`\r`	carriage return	`\'`	single quote (')
`\t`	tab	`\"`	double quote (")
`\v`	vertical tab	`\?`	question mark (?)
`\b`	backspace	`\\`	backslash (\)
`\f`	form feed (page feed)		

1.50

3.3 Les opérateurs

1.51

Opérateur d'affectation, conversion implicite

L'affectation (« assignement ») ou opérateur =

```
Variable = expression;
```

Le terme de gauche peut être une variable simple, un élément de tableau mais pas une constante !

```
1  #include <iostream>
2  using namespace std;
3  int main(){
4    int i, j = 2;
5    float x = 2.5;
6    i = j + x;
7    x = x + i;
8    cout<<"x=␣"<<x<<endl;
9    return 0;
10 }
```

Le code affichera `x= 6.5`, une conversion implicite a lieu en ligne 6 vers le type `int` de la variable `i` : `i = int(j + x);`

1.52

Les opérateurs arithmétiques

Liste des opérateurs standards
- l'opérateur unaire de signe : `-`
- les opérateurs binaires : `+ , - , * , /`

Attention à la division d'`int`, elle retourne un `int` !

L'opérateur binaire modulo `%`
Il n'est valable que sur des opérations entières. Le signe du résultat est (en général) celui du dividende. L'opération suivante retournera `-1` :

```
x = -5 % 2;
```

La puissance
L'opérateur de puissance n'existe pas ! Il faut employer la fonction `pow(x,y)` nécessitant la directive `#include <cmath>`.

1.53

Les opérateurs relationnels

Syntaxe
Opérateurs binaires agissant sur 2 expressions et retournant un booléen suivant la syntaxe suivante :

```
Expression_1 (op.) Expression_2
```

Les opérateurs relationnels disponibles sont :

```
> , >= , < , <= , == , !=
```

```
int main(){
  int a = 0;
  int b = 1;
  if(a=b)
    cout<<"a et b sont égaux\n";
  else
    cout<<"a et b sont différents\n";
  return 0;
}
```

La réponse sera `"a et b sont égaux"`, il ne faut pas confondre l'opérateur = et == ! Ici l'affectation étant réussie (les valeurs ne sont PAS comparées), l'opération retourne un booléen `True`.

1.54

Les opérateurs logiques

Syntaxe

```
Expr_1 (op.) Expr_2 (op.) Expr_3
```

Trois opérations logiques sont disponibles :

```
&& , || , !
```

Principe d'évaluation
L'évaluation de gauche à droite. Stop dès que résultat défini.

```
if((u >= 0) && (i <= 9) && !(P[i] == 0))
```

Dernière condition ignorée si $i \notin [0,9]$.

Opérations logiques

Opérateur `&&`

a	b	a && b
True	True	True
True	False	False
False	True	False
False	False	False

Opérateur `||`

a	b	a || b
True	True	True
True	False	True
False	True	True
False	False	False

1.55

Opérateurs logiques bit à bit

Définition
Permettent de manipuler les entiers (`short`, `int`, `long`) au niveau du bit.

`&`	ET logique	`|`	OU inclusif
$\wedge$	OU exclusif	$\sim$	complément à 1
<<	décalage à gauche	>>	décalage à droite

1.56

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

$\wedge$	0	1
0	0	1
1	1	0

Expression	Binaire	Décimal
a	01001101	77
b	00010111	23
a & b	00000101	5
a | b	01011111	95
a ∧ b	01011010	90
∼a	10110010	178
b<<2	01011100	92
b<<5	11100000	112
b>>1	00001011	11

Les opérateurs d'affectation composée

Principe de fonctionnement

```
expr_1 (op.) expr_2
```

est équivalent à :

```
expr_1 = expr_1 (op.) expr_2
```

Opérateurs disponibles :

```
+= , -= , *= , /= , %= ,
&= , ^= , |= , <<= , >>=
```

```
1 int i = 5;
2 i+=3;
3 cout<<"i = "<<i<<endl;
```

Ce bloc d'instruction retourne `i = 8`.

1.57

Opérateurs d'incrémentation et de décrémentation

Utilisation

```
++ , --
```

Incrémentent ou décrémentent la variable associée d'une unité. S'utilisent en suffixe (i++) ou en préfixe (++i).

— Le suffixe retourne l'ancienne valeur (retourne puis incrémente) ;
— Le préfixe retourne la nouvelle valeur (incrémente puis retourne).

```
1 int i = 5, a;
2 a = i++;
3 cout<<"a = "<<a<<endl;
4 cout<<"i = "<<i<<endl;
```

Retourne `a = 5` et `i = 6`.

```
1 int i = 5, a;
2 a = ++i;
3 cout<<"a = "<<a<<endl;
4 cout<<"i = "<<i<<endl;
```

Retourne `a = 6` et `i = 6`.

1.58

Autres opérateurs

Opérateur virgule : ,
Exécution de gauche à droite. La dernière expression compte pour l'évaluation.

```
a = (b=3, b+2);
```

retourne `a = 5` et `b = 3`.

Opérateur conditionnel ternaire : ? :

```
x >= 0 ? x : -x;
m = ((a > b) ? a : b);
```

sont une façon de coder la valeur absolue et le max.

Opérateur conversion de type
Appelé `cast`, il permet de modifier le type d'un objet.

```
int i=3, j=2;
cout<<(float)i/j<<endl;
```

1.59

Règle de priorité des opérateurs

Ordre de priorité :

```
::
()   []   ->   .
!    ~    ++   --   -   &(adresse)   *()   new/delete   sizeof()
.*  ->*
*    /    %
+    -(binaire)
<<   >>
<    <=   >    >=
==   !=
& (ET bit à bit)
^
|
&&
||
? :
=   +=   -=   *=   /=   %=   &=   ^=   |=   <<=   >>=
,
```

Attention aux priorités de parenthèses : `if ( (x^y) != 0 )` 1.60

3.4 Les instructions de contrôle 1.61

`if...else`

Syntaxe

```
if (expression_1)
  instruction_1
else if(expression_2){
  instruction_2_1
  instruction_2_2
  instruction_2_3
}
else if(expression_3)
  instruction_3
else
  instruction_4
```

Imbrications de `if`

Le `else` s'applique au premier `if` précédent non attribué.

```
if (a <= b)
  if(b <= c)
    cout<<"ordonné\n";
  else cout<<"non ordonné\n";
```

1.62

Instruction `switch`

Syntaxe

```
switch (expression_etiquette){
  case constante_1:
    instructions_1
    break;
  case constante_2:
    instructions_2
    break;
  ...
  case constante_n:
    instructions_n
    break;
  default:
    instructions_def
}
```

Explications
Évalue `expression_etiquette` (type `int` ou `char` uniquement). Effectue un test logique == successivement sur les constantes énumérées par les `case`. Exécute `default` si et seulement si aucun test n'est `True`.

1.63

Exemple de switch

```
using namespace std;
int main(){
  int n;
  cout<<"Entrer un entier: ";
  cin>>n;
  switch(n){
    case 0: cout<<": nul\n"<<endl;
            break;
    case 1: cout<<": un\n"<<endl;
            break;
    case 2: cout<<": deux\n"<<endl;
            break;
    default: cout<<": trop grand\n"<<endl;
  }
}
```

On peut utiliser des `char` et les comparer à des `int`, ils sont systématiquement convertis en `int`.

1.64

3.5 Les boucles

1.65

Boucle `while`

Principe
Tant qu'une condition est vérifiée (est `True`), un bloc d'instruction est exécuté. On sort de la boucle quand la condition est `False`. Le bloc n'est jamais exécuté si la condition est `False` au départ.

Syntaxe

```
int i=1;
while (i < 10){
  cout<<"i= "<<i<<endl;
  i++;
}
```

1.66

Boucle `do...while`

Principe
Variante de la boucle `while` avec test après exécution du bloc d'instruction. L'instruction sera exécutée au moins une fois.

Syntaxe

```
int a;
do {
  cout<<"Entrer un entier : ";
  cin>>a;
  cout<<endl<<"L'entier est : "<<a<<endl;
}
while((a <= 0) || (a>10));
```

1.67

Boucle `for`

Principe

```
for(expr_1 ; expr_2 ; expr_3){
  instructions
}
```

est l'équivalent des instructions :

```
expr_1;
while(expr_2){
  instructions
  expr_3;
}
```

Exemple :

```
for (int i=0 ; i < 10 ; i++){
  cout<<"i= "<<i<<endl;
}
```

À la fin de la boucle, `i` vaut 10.

1.68

`break` non conditionnel

Principe
L'instruction `break` permet de sortir d'un structure de contrôle ou d'une boucle indépendamment de l'état du test. `break` fait sortir de la boucle la plus interne.

```
using namespace std;
int main(){
  for(int i=0 ; i < 5 ; i++){
    cout<<"i= "<<i<<endl;
    if(i == 3)
      break;
    }
    cout<<"Valeur de i en sortie : "<<i<<endl;
  }
```

1.69

`continue` non conditionnel

Principe
L'instruction `continue` permet de passer directement au tour de boucle suivant. Le reste des instructions de la boucle est ignoré pour ce tour.

1.70

```
using namespace std;
int main(){
  for(int i=5 ; i > 0 ; i--){
    cout<<i<<", ";
    if(i == 2)
      continue;
    }
    cout<<"Décollage !"<<endl;
  }
```

4 Références

Références

Quelques références pour compléter cette présentation :

— « The C++ Programming language » - Bjarne Stroustrup (VF dispo)
— « The C Programming language » - Brian W. Kernighan, Dennis M. Ritchie (VF dispo)
— Cours INRIA / Projet CODES - « Programmation en langage C » - Anne Canteaut
— Site `http://www.cplusplus.com/` : Section Tutorial - « C++ Language »
— Site `http://fr.cppreference.com/` : Section « Référence C++ » (anglais dispo)

1.71