Correspondances algo/C++

Voici un tableau qui va vous présenter la correspondance entre les syntaxes algorithmiques et le langage C++.

Algo	**C++**

	`// commentaire d'une ligne`
	`/* commentaire`
	`de plusieurs lignes */`

Entrées/sorties
`Afficher nomVariable`
`Lire nomVariable`
	
	`cout << nomVariable << endl;`
	`cin >> nomVariable;`

Les types : entier réel caractère Chaîne de caractères booléen	 int // -2147483648 à 2147483647 (4 octets) float // $-3,4 \cdot 10^{-38}$ à $3,4 \cdot 10^{38}$ char string bool

Déclaration de variables :
`nomVariable : typeVariable`
exemple :
`k : entier`
	
	`typeVariable nomVariable ;`
	exemple :
	`int k ;`
	Une variable peut être déclarée n'importe où et ne concerne que le bloc où elle est déclarée (un bloc est délimité par des accolades)

Affectation :
`nomVariable` ← `valeur`
	
	`nomVariable = valeur ;`
	Il est possible de déclarer et en même temps initialiser une variable.
	Exemple :
	`int k = 0 ;`

Tableaux 1D :
`nomTableau[1..N] : typeTableau`
exemple :
`tab[1..10] : entier`
 	
`tab[i]` ← 3
	
	`typeTableau nomTableau[N] ;`
	exemple :
	`int tab[10] ;`
	Remarque : attention, en C++, les indices de tableaux commencent à 0. Donc, ici le tableau `tab` déclaré avec 10 cases possède des cases numérotées de 0 à 9.
	`tab[i]` = 3
	(la case n° `i` de `tab` prend la valeur 3)

Tableaux 2D :
`nomTableau[1..N, 1..M] : typeTableau`
exemple :
`tab[1..10, 1..3] : entier`
`tab[i][j]` ← 3
	
	`typeTableau nomTableau[N][M] ;`
	exemple :
	`int tab[10][3] ;`
	`tab[i][j]` = 3
	(la case ligne `i` colonne `j` de `tab` prend la valeur 3)

Chaînes de caractères :
`chaine1 <- "hello"`
`chaine2 <- "world"`
`chaine <- chaine1 + " " + chaine2`
`taille(chaine)`
`extraire(chaine, 6, 5)`
`extraire(chaine, 6)`
`position("r", ch1)`
`uneVariableChaine <- str(3)`
`uneVariableNumerique <- val("3")`
 	
Insérer, remplacer...
	
	`chaine1 = "hello"`
	`chaine2 = "world"`
	`chaine = chaine1 + " " + chaine2`
	`chaine.length()`
	`chaine.substr(6,5)` (extraire 5 car. à partir du n° 6)
	`chaine.substr(6)` (extraire du car. n° 6 jusqu'à la fin)
	`chaine.find("r")`
	`uneVariableChaine = to_string(3)`
	`unEntier = stoi("3")` (vers un entier)
	`unFlottant = stof("3")` (vers un flottant)
	
	Voir la documentation officielle :
	`http://www.cplusplus.com/reference/string/`

signes de comparaison et logique : $=, \neq, <, >, \leq, \geq$, non, et, ou	(dans le même ordre :) `==, !=, <, >, <=, >=, !, &&, ||` Remarque : attention de bien mettre `==` lors d'une comparaison (dans un if, par exemple) car le `=` tout seul ne va pas provoquer d'erreur, mais fera une affectation à la place
opérateurs logiques : et, ou, oux, $\sim$ (inverse)	(dans le même ordre :) `&, |, ^, ~` Remarque : ces opérateurs sont utilisables sur des valeurs numériques, l'ordinateur appliquera l'opération sur la valeur binaire du nombre.
opérations : `+, -, *, /, div, mod`	(dans le même ordre :) `+, -, *, /, /, %` Remarque : il n'y a pas de division entière (div), mais une division simple avec une affectation dans une variable de type entier donne le même résultat.

Quelques raccourcis d'écriture :
$A \leftarrow A + 1$
$A \leftarrow A + B$
$A \leftarrow A$–1
$A \leftarrow A$–B
$A \leftarrow A \times B$
$A \leftarrow A/B$
	
	`A++`
	`A += B`
	`A--`
	`A -= B`
	`A *= B`
	`A /= B`

si *condition* **alors**
| action
fin
	`if (condition) {`
	`  action ;`
	`}`
	
	Remarque : attention de ne pas oublier les parenthèses autour de la condition (obligatoire pour toutes les conditions, que ce soit dans les `if`, `while` etc.)

si *condition* **alors**
| action1
sinon
| action2
fin
	`if (condition) {`
	`  action1 ;`
	`}`
	`else {`
	`  action2 ;`
	`}`

tant que *condition* **faire**
| action
fin
	`while (condition) {`
	`  action ;`
	`}`

répéter
| action
jusqu'à *condition*
	`do {`
	`  action ;`
	`}while(condition) ;`

répéter
| action
jusqu'à *condition*
	`do {`
	`  action ;`
	`}while(condition) ;`

pour *k* ***de*** *deb* **à** *fin* **faire**
| action
fin
	`for (k = deb ; k <= fin ; k++) {`
	`  action ;`
	`}`

Procédure `nomProc(`*param1 : type1, ..., paramN : typeN*`)`
déclarations
début
| traitement
fin
	`void nomProc (type1 param1, ..., typeN paramN) {`
	`  déclarations ;`
	`  traitements ;`
	`}`
	
	Remarque : `void` signale l'absence de type retourné, donc la procédure.

Fonction `nomFonc(...)` : typeRetour
déclarations
début
| traitement
| retourner valeur
fin
	`typeRetour nomFonc (...) {`
	`  déclarations ;`
	`  traitements ;`
	`  return valeur ;`
	`}`