Notes sur le langage C++

Le langage C++ peut être considéré comme une formule 1 parmi les langages. Il autorise presque tous les exploits, mais ne pardonne pas la moindre erreur de conduite. À vous d'être à la hauteur !

Les commentaires

Deux type de commentaires
- de fin de ligne : commencent par // et se terminent à la fin de la ligne
- à la C : commencent par /* et se terminent au premier */

Les types élémentaires

bool	booléen (true/false)
char	caractère ('A','B',...)
short	entier court
int	entier
long	entier long
float	réel simple précision
double	réel double précision

Un caractère est en fait un petit entier qui contient le code ascii du caractère. On peut donc manipuler les caractères comme des entiers. Chaque entier est signé par défaut. On peut se passer du signe en indiquant unsigned (unsigned int par ex.) ce qui permet de gagner un bit. Les tailles des entiers peuvent varier d'une architecture à l'autre. La seule garantie que l'on ait est que sizeof(char)=1 octet et que sizeof(char)≤sizeof(short)≤sizeof(int)≤sizeof(long).

Les caractères spéciaux

Certains caractères ont une notation spéciale :

\n	saut de ligne
\t	tabulation horizontale
\v	tabulation verticale
\b	baskspace
\r	retour chariot
\f	form feed
\a	alerte
\ \	backslash
\'	guillemet
\"	guillement double
ooo	caractère de code ascii *ooo* en octal
\x*hh*	caractère de code ascii *hh* en hexadécimal

Affectation et test d'égalité

L'affectation *var*← *expression* qui consiste à calculer la valeur de l'expression et à ranger le résultat dans la variable *var* se note en utilisant le signe égal en C++ : *var=expression*. Du côté gauche de ce signe égal, on ne peut trouver que le nom de la variable qui va recevoir la valeur.

Le test d'égalité (a=b en pseudo) s'écrit a==b. Il renvoie vrai uniquement lorsque les deux valeurs sont égales.

Il ne faut pas confondre l'affectation (= en C++) avec le test d'égalité (== en C++). Le compilateur ne vous indiquera pas d'erreur si vous écrivez l'un à la place de l'autre, mais votre programme n'aura aucune chance de fonctionner.

Les structures de contrôle et déclarations

En C++, le type des variables se précise avant leur nom. Certaines structures de contrôle n'opèrent que sur une seule instruction. Pour les faire opérer sur plusieurs instructions, il suffit de mettre ces instructions entre accolades pour qu'elles n'en forment plus qu'une seule (création d'un bloc). Chaque instruction doit se terminer par un point-virgule.

en pseudo-langage	en C++
instruction1 *instruction2* *instruction3*	*instruction1* ; *instruction2* ; *instruction3* ;

debut
  instruction1
  instruction2
  instruction3
fin
	{
	  instruction1 ;
	  instruction2 ;
	  instruction3 ;
	}

si *condition* **alors** *instructions* **fin si**	**if** (*condition*) *une_seule_instruction* ;
si *condition* **alors** *instructions1* **sinon** *instructions2* **fin si**	**if** (*condition*) *une_seule_instruction1* ; **else** *une_seule_instruction2* ;

selon *expression* choisir
  quand *valeur1* **faire**
  instructions1
  fin
  quand *v1* | *v2* | *v3* **faire**
  instructions2
  fin
  par defaut faire
  instructions3
  fin
fin selon
	switch (*expression*)
	{
	  case *valeur1* :
	  instructions1 ;
	  break ;
	  case *v1* :
	  case *v2* :
	  case *v3* :
	  instructions2 ;
	  break ;
	  default :
	  instructions3 ;
	  break ;
	}

pour *c* **de** *inf* **a** *sup* **faire** *instructions* **fin pour**	**for** (*type_compteur c=inf* ; *c<=sup* ; *c++*) *une_seule_instruction* ;
pour *c* **de** *sup* **a** *inf* **decroissant faire** *instructions* **fin pour**	**for** (*type_compteur c=sup* ; *c>=inf* ; *c−−*) *une_seule_instruction* ;
tant que *condition* **faire** *instructions* **fin tant que**	**while** (*condition*) *une_seule_instruction* ;

en pseudo-langage	en C++

repeter
  instructions
tant que *condition* **fin**
	do
	{
	  instructions;
	}
	while (*condition*);

repeter
  instructions
jusque *condition* **fin**
	do
	{
	  instructions;
	}
	while (!*condition*);

programme *nom*
constantes
  nomconst : *type* = *valeur*
variables
  var1, var2 : *type*
// *définitions des fonctions et procédures*
debut
  instructions
fin
	#include <iostream.h>
	// autres includes...
	const *type nomconst=valeur*;
	// autres constantes...
	type var1, var2;
	// autres variables...
	// définitions des fonctions et procédures
	int main()
	{
	  instructions;
	}

procedure *nom*
  (**donnee** *p1,p2*:*type1*;
  resultat *p3*:*type3*;
  donnee resultat*p4*:*type4*)
// *constantes et variables locales*
debut
  instructions
fin
	void *nom*(*type1 p1*,
	  type1 p2,
	  type3 &p3,
	  type4 &p4)
	{
	  // *const. et var. locales*
	  instructions;
	}

fonction *nom*
  (**donnee** *p1,p2*:*type1*;
  resultat *p3*:*type3*;
  donnee resultat*p4*:*type4*)
  retourne *type_retour*
// *constantes et variables locales*
debut
  instructions
  retourner *valeur*
fin
	type_retour nom
	  (*type1 p1*,
	  type1 p2,
	  type3 &p3,
	  type4 &p4)
	{
	  // *const. et var. locales*
	  instructions;
	  return *valeur*;
	}

lire(*val*) afficher("valeur",*val*)	cin >> *val*; cout << "valeur=" << *val* << "\n";
// conversion de type *type*(*expr*)	// conversion de type (*type*)*expr*

Les opérateurs

Les connecteurs logiques sont systématiquement évalués de manière partielle. Il ne font pas confondre le ET, le OU et le NON logique (&&, ||, !) avec leur correspondant bit à bit.

Le tableau ci-dessous représente les principaux opérateurs par ordre de priorité décroissante. À l'intérieur d'un même cadre, les opérateurs ont des priorités identiques. Les opérateurs de même priorité sont associatifs à gauche sauf les opérateurs unaires et d'affectation qui sont associatifs à droite. Ex : a+b-c*d représente ((a+b)-(c*d)), a=b+=-b représente (a=(b+=(-b))).

plus forte priorité	
structure.*champ*	accès à un champ
pointeur->*champ*	accès à un champ
tableau[*expr*]	élément d'un tableau
nom(*parms*)	appel de fonction
var++	post-incrémentation
var--	post-décrémentation
sizeof(*type*)	taille d'un type
++*var*	pré-incrémentation
--*var*	pré-décrémentation
~*expr*	complément à 1
!*expr*	négation logique
-*expr*	signe moins
+*expr*	signe plus
&*var*	adresse de
*****expr*	déréférence
(*type*)*expr*	conversion de type
expr * *expr*	multiplication
expr / *expr*	division
expr % *expr*	modulo (reste)
expr+*expr*	addition
expr-*expr*	soustraction
expr<<*expr*	décalage à gauche
flux<<*expr*	affichage
expr>>*expr*	décalage à droite
flux>>*var*	lecture
expr<*expr*	strict. inférieur
expr<=*expr*	inférieur ou égal
expr>*expr*	strict. supérieur
expr>=*expr*	supérieur ou égal
expr==*expr*	test d'égalité
expr !=*expr*	différent
expr & *expr*	ET bit à bit
expr ^ *expr*	OU excl. bit à bit
expr | *expr*	OU bit à bit
expr && *expr*	ET logique
expr || *expr*	OU logique
var=*expr*	affectation
*var**=*expr*	multiplication et affectation
var/=*expr*	division et affectation
var%=*expr*	modulo et affectation
var+=*expr*	addition et affectation
var-=*expr*	soustraction et affectation
var<<=*expr*	décalage à gauche et affectation
var>>=*expr*	décalage à droite et affectation
var&=*expr*	ET bit à bit et affectation
var|=*expr*	OU bit à bit et affectation
var^=*expr*	OU excl. bit à bit et affectation
moins forte priorité	