Syntaxe Élémentaire

Voire Cours de C.

Identificateurs quelques mots clefs en plus.
Instructions (if else etc)

Mots réservés

aa dynamicccast namespace reinterperttcast staticccast
unsigned and and_eq asm auto bitand
bitor bool break case catch
char class compl const const_cast
continue default delete do double
dynamic_cast else enum explicit export
extern false float for friend
goto if inline int long
mutable namespace new not not_eq
operator or or_eq private protected
public register reinterpret_cast return short
signed sizeof static static_cast struct
switch template this throw true
try typedef typeid typename union
unsigned using virtual void volatile
wchar_t while xor xor_eq

Exceptions

Les exceptions sont un mécanisme de déroutement conditionnel.

* throw lève une exception;
* catch capte l'exception.

Exemple

```
void f(int i) {
  try {
    if (i)
      throw "Help!";
    cout <<"Ok!\n";
  }
  catch(char* e) {
    cout << e << "\n";
  }
}
```

```
void main() {
  f(0);
  f(1);
}
```

Résultat

```
Ok!
Help!
```

Constantes

Un identificateur dont la valeur ne change pas peut être déclaré constant en utilisant le mot-clé const.

Le mot-clé const peut intervenir dans quatre contextes:

* dans la spécification du type d'un objet;
* dans la spécification du type d'un paramètre de fonction; il indique que cette fonction n'en modifie pas la valeur;
* dans la spécification du type d'un membre de classe;
* comme qualificatif d'une méthode pour indiquer qu'elle ne modifie pas les données membres de cette classe.

Une constante doit être initialisée, et ne peut pas être affectée.

Constantes (2)

Les constantes se substituent avantageusement aux #define.

```
const N =10;
float a[1+N];
```

La syntaxe peut prêter à confusion:

```
const int *p;
```

se lit `(const int) *p`.

Donc p est un pointeur vers une constante de type int. Avec

```
int i;
const int *p = &i;
```

la valeur de i ne peut pas être modifiée à travers le pointeur:
on peut écrire `i = 10` mais pas `*p = 10;` !

```
int i;
int * const q = &i;
```

Ici, q est un pointeur constant vers un entier; l'initialisation est nécessaire. La valeur de q (c'est-à-dire la case pointée) ne peut être modifiée, mais on peut modifier la valeur contenue dans cette case, et écrire `*q = 10;` (Noter l'analogie avec les tableaux en C).

```
int i;
const int * const r = &i;
```

Enfin, r est un pointeur constant vers un objet constant.

Références

Une entité au sens courant du terme est une place en mémoire destinée à contenir une donnée. Une variable est un entité, mais une entité peut être constante si la donnée est non modifiable.

Une référence est un nouveau nom (synonyme, alias) attribué à une entité déjà défini.

La syntaxe est

```
type& ident=expression;
```

* Une référence doit être initialisée à sa définition.
* L'expression d'initialisation doit désigner un objet (l-value).

```
int i = 1;
int& r = i; // r et i désignent le même entier
int x = r;  // x = 1;
r = 2;      // x = 1; i = 2
int & y;    // erreur : pas d'initialisation
int* p = &r // == &i : même objet
```

Usage des références

* spécification des paramètres et valeurs de retour des fonctions
* surcharge d'opérateurs

Ne pas confondre

```
int&  val = 1;      // val est reference a int
&val                // adresse de val
val & 1             // ``et'' bit a bit
```

Une référence constante ne ne permet pas de changer la valeur.

```
int n;
const int& k = n;
...
++n;   // ok
++k;   // erreur
```

Passage par référence

Passage par référence d'arguments de fonctions. Rappelons:

```
int f(int k) {...}
main() {
   ...
   j = f(i) ;
   ...
}
```

Lors de l'appel f(i)

1. il y a création d'une nouvelle variable ibis,
2. initialisation de ibis avec la valeur de i
3. les modifications subies par k dans la définition sont subies par ibis,
4. au retour, ibis est détruite et i a conservé sa valeur

Si maintenant, k est déclarée en référence

```
int f(int& k) {...}
main() {
   ...
   j = f(i) ;
   ...
}
```

Lors de l'appel f(i)

1. il y a création d'une référence k initialisée à i,
   donc k et i désignent la même place;
2. les modifications subies par k dans la définition sont subies par i.
3. au retour, k est détruite et i a conservé ses modifications.

Exemple

```
void incr(int& a) { a++;}

main() {
   int x = 1;
   incr(x);  // x = 2
}
```

En effet, l'appel incr(x) réalise

l'initialisation `int& a = x;`
l'incrémentation `a++;`

Si de plus incr est déclarée inline le code produit par la compilation de `incr(x)` et exactement identique à celui produit par `x++` ceci permet d'éviter dans de nombreux cas le coût de la syntaxe orienté objet.

```
#include <iostream.h>

struct Personne {  int age;  char* nom; };

// Passage par reference pour changer le contenu
void change(Personne& p) {
  p.age = 53; // en 2003
  p.nom = "Bjorne Stroustrup";
}

// Passage par reference pour eviter la recopie
void affiche(const Personne& p) {
  cout << p.nom <<endl;
}

void main()
{
  Personne auteur;    change(auteur);     affiche(auteur);
}
```

Le résultat est bien entendu

```
   Bjorne Stroustrup
```

Retour par référence

Passage par référence en résultat de fonctions. Rappelons:

```
int f() {... return x;}
```

* L'instruction return x retourne la valeur de x;
* il y a création d'une variable sur la pile contenant cette valeur;
* f() n'est pas une l-valeur

```
int& f() {... return x;}
```

* Ici return x retourne une référence, donc un synonyme d'un objet;
* il y a création d'une variable sur la pile contenant cette valeur;
* f() est une référence pour la variable x, et c'est une l-value.

Exemple de retour par référence

Exemple : retourner l'objet complexe de plus grand module.

```
#include <iostream.h>
#include <math.h>

struct Complexe {
  float re, im;
  Complexe(float r = 0, float i = 0) { re = r; im = i; }
  float module() const { return sqrt(re*re+im*im); }
};

Complexe& maxmod(const Complexe& x, const Complexe& y) {
  return (x.module() > y.module()) ? x : y;
}

void main()
{
  Complexe u(3,5), v(4,4);
  cout << "u.module() = " << u.module() <<" ";
  cout << "v.module() = " << v.module() << endl;
  Complexe& z = maxmod(u, v);
  cout << &z <<' '<< &u <<' '<< &v << endl;
}
```

Le résultat est

```
u.module() = 5.83095 v.module() = 5.65685
0x7ffffc80 0x7ffffc80 0x7ffffc78
```

```
int& incr(int &x) {
  x += 10;
  return x;
}

main() {
  int i = 5;
  incr(i) +=15;
  cout << i << endl;
}
```

incr(i) est synonyme de i, et main l'incrémente de 15. Le résultat est 30.

[ZZZ: Il ne faut pas faire retourner une référence à une variable locale.]

Surcharge de fonctions

Il y a surcharge lorsqu'un même identificateur désigne plusieurs fonctions.
Pour cela, ces fonctions doivent différer par la liste des types de leurs arguments.

La surcharge existe déjà en C: 3/5 et 3.5/5, mais est systématisée en C++.

Usage de la surcharge:

* dans les constructeurs;
* dans les méthodes;
* dans les opérateurs.

Donner la possibilité d'effectuer une opération avec des degrés différent d'ajustement de paramètres. L'absence d'un paramètre vaut acceptation de la valeur par défaut.

Polymorphisme typé obtenu par surcharge:

```
#include <iostream.h>

void affiche(int i)    { cout << "Entier " << i << endl; }

void affiche(char c)   { cout << "Caractere " << c << endl; }

void affiche(double d)  { cout << "Reel " << d << endl; }

void affiche(char* txt, int n) {
  for (int j = 0; j < n; j++) cout << txt[j];
  cout << endl;
}

void main()
{
 affiche(3);
 affiche('x');
 affiche(1.2);
 affiche("Bonjour, monde !", 7);
}
```

Résultat

```
Entier 3
Caractere x
Reel 1.2
Bonjour
```

Surcharge (3)

Le compilateur engendre une fonction par type, postfixée par un codage des types.

affiche__Fi affiche__Fc affiche__Fd affiche__FPci

C'est pourquoi une fonction C externe doit être déclarée pour conserver son nom « C » :

```
extern "C" f();
```

L'analyse de la signature détermine la fonction à utiliser. Les promotions et conversions usuelles s'appliquent. Le type de retour n'intervient pas. Les conversions dites ``triviales'' impliquent :

1. Pour tout type T, un T et un T& sont équivalents:

```
void f(int i) {...}
void f(int& i) {...} // erreur
```

2. De même, un T et un const T sont équivalents, sauf pour const T* et T* (type de pointeurs distincts)

3. Un type défini par typedef ne donne pas un type séparé

```
typedef int Int;
void f(int i) {...}
void f(Int i) {...} // erreur
```

4. Pointeur et tableau sont équivalents

```
void f(char* p) {...}
void f(char p[]){...} // erreur
void f(char p[12]){...} // erreur
```

Résolution de la surcharge

Règles de choix

1. Recherche d'une correspondance exacte et conversions ``triviales'':

    * T en T & et vice-versa,
    * ajout de const
    * transformation T[] en T*
2. Promotion (char en int, float en double...) ou dégradation.
3. Règles de conversion, surtout entre classes.
4. Règles de conversion définies par le programmeur.
5. Correspondance avec points de suspension.

Toute ambiguïté est une erreur (a l'intérieur d'une règle).

Resolution exemple

Prototypes

```
void f(int);            // fonction 1
void f(float);          // fonction 2
void f(int, float);     // fonction 3
void f(float, int);     // fonction 4
void f(int, int, int); // fonction 5
void f(...);            // defaut
```

Variables

```
int i,j;
float x, y;
char c;
double z;
```

Appels

```
f(c);          // R2 fonction 1
f(i, j);       // R2 erreur : en 3 ou en 4 ?
f(i, c);       // R2 erreur : en 3 ou en 4 ?
f(c, i);       // R2 erreur : en 3 ou en 4 ?
f(i, z);       // R2d fonction 3
f(z, i);       // R2d fonction 4
f(z, i, i);    // R2d fonction 5
f("Hello");    // R5 defaut
```

Exemple Factorielle

```
int factorielle(int n) {
return (!n) ? 1 : n * factorielle(n - 1);
}
```

Elle se programme aussi avec une récursivité terminale à l'aide de

```
int Fact(int n, int p) {
  return (!n) ? p : Fact(n - 1, p * n);
}
```

On a alors factorielle(n) = Fact(n, 1). On définit donc

```
int Fact(int n) {
  return Fact(n, 1);
}
```

ou mieux encore

```
int Fact(int n, int p = 1) {
  return (!n) ? p : Fact(n - 1, p * n);
}
```

Exemple une calculette +, -, *, /

Un programme simulant une calculette élémentaire, capable d'évaluer les quatre opérations +, -, *, /.
Il faut prévoir le cas d'une division par 0.

```
#include <iostream.h>

void eval (float x, float y, char op, float& r) {
  switch (op) {
    case '+': r = x+y; return;
    case '-': r = x-y; return;
    case '*': r = x*y; return;
    case '/':
      if (y == 0) throw "DIV par 0";
      r = x/y; return;
  }
  throw "Operateur inconnu";
}
```

Exécution

```
Calculette
Expression ? 5+6
5+6 = 11
Encore (o/n)? o
Expression ? 6/0
DIV par 0
Encore (o/n)? o
Expression ? 5@4
Operateur inconnu
Encore (o/n)? n
Au revoir !
```

```
void main() {
  float x, y, resultat;
  char operateur, c;

  cout << "Calculette\n";
  for(;;) {
    cout << "Expression ? ";
    cin >> x >> operateur >> y;
    try {
      eval (x, y, operateur, resultat);
      cout << x << operateur << y << " = " << resultat << endl;
    }
    catch (char* message) {
      cerr << message << endl;
    }
    cout << "Encore (o/n)? ";
    cin >> c;
    if (c != 'o')
      break;
  }
  cout << "Au revoir !\n";
}
```