INSTRUCTIONS

INSTRUCTIONS DE BASE

sérialisation

Elle est notée en C++ par un point virgule

bloc

L'instruction bloc est notée par une accolade ouvrante au début du bloc et une accolade fermante en fin de bloc.

Un bloc est présenté de la façon suivante :

```
{
    I1;
     ...
    In;
}
```

L'exécution d'un bloc est l'exécution de toutes les instructions en séquence.

choix

Instruction conditionelle :

```
if ( condition ) I1;  else I2 ;
if ( condition ) I1;
```

L'exécution de l'instruction conditionelle commence par l'évaluation de la condition. Si la condition a valeur vrai, il y a exécution de l'instruction *I1* sinon il y a exécution de *I2* (si elle existe).

Instruction switch : l'instruction *switch* permet de condenser l'écriture de certains *si alors sinon*.

```
switch (expression)
{
             case constante :  instructions;
             ...
             default instructions;
}
```

Expression et constante ci-contre doivent être des entiers. L'exécution d'un switch se comporte de la façon suivante :

- évaluation de l'expression
- recherche de la première constante ayant même valeur que l'expression, et exécution de ***toutes*** les instructions qui suivent.

exemple : supposons que nous voulions compter les voyelles d'un texte :

```
{char c;
      ...
             switch(c)
```

```
              {
                     case 'a':case 'A': cpta+=1; break;
                     case 'e':case 'E': cpte+=1; break;
                     case 'i':case 'I': cpti+=1; break;
                     case 'o':case 'O': cpto+=1; break;
                     case 'u':case 'U': cptu+=1; break;
              default cptc +=1;
              }
```

Il ne faut pas oublier l'instruction **break** pour ne pas exécuter en séquence les instructions qui suivent.

itérations

tant que

```
while (condition de continuation) I;
```

L’exécution de l’instruction *while* se comporte de la façon suivante :

1.évaluation de la condition de continuation :
- si la condition est fausse l ’exécution du *while* est terminée
- si la condition est vraie il y a exécution de *I*, puis de nouveau exécution du *while*

Dans l’écriture des programmes, la rédaction de la conditionn d’arrêt d’une itération est toujours plus « facile » à écrire que la condition de continuation : il est donc conseillé d’écrire un tant que de la façon suivante :

```
while (!(condition d’arrêt) ) I;
```

répéter

```
do{
   I;
}while (condition de continuation);
```

L’exécution de l’instruction *do while* se comporte de la façon suivante :

1. il y a exécution de *I*
2. évaluation de la condition de continuation :
   - si la condition est fausse l ’exécution du *do while* est terminée
   - si la condition est vraie il y a exécution de nouveau du *do while*

itérer

De façon plus générale, on peut écrire des itérations de la façon suivante :

```
while(true)
{
      I1;
      if (condition d’arrêt) break;
      I2;
}
```

L’instruction **continue** permet de recommencer le *while* sans terminer l’exécution de la liste des instructions.

pour

```
for( initialisation; condition de continuation; mise à jour ) I;
```

L'exécution du *for* se fait de la façon suivante :

1. exécution de l'*initialisation*
2. évaluation de la *condition de continuation*
   - si la condition de continuation est fausse l'exécution du *for* est terminée
   - si la condition de continuation est vraie il y a exécution de *I*, puis exécution de *la mise à jour*, et enfin exécution de 2.

exemples :

1. calcul de la somme des **n** premiers entiers :

```
n= ...;
somme=0;
i=1;
while(i<=n){
    somme +=i;
    ++i;
}
```

ou :

```
n= ...;
somme=0;
for(i = 1; i<=n ; ++i) somme+=i;
```

ou encore :

```
n= ...;
for(i = 1, somme=0; i<=n ; somme+=i, ++i);
```

2. soit à calculer le produit de deux nombres, le premier (x) étant positif ou nul, sans utiliser l'opérateur * :

```
i=0;
p=0;
while(i<x){
     p+=y;
     ++i;
}
```

ou :

```
for(i =0, p=0; i<x ; p+=y, ++i);
```

ACTIONS ET FONCTIONS

actions

En C++ seul existe le concept de fonction ; pour implanter le concept d'action, il faut utiliser une fonction qui retourne le type vide : **void**.

```
void nomDeLaFonction(){
      déclarations ...
      instructions ...
}
```

exemple : écrire une action qui range la plus petite valeur parmi *a, b* et *c* dans *p*, la plus grande dans *t*, et celle qui reste dans *s* :

```
void classer_abc ()
{ if ((a<b)&&(b<c)){p=a;s=b;t=c;}
  else if((a<c)&&(c<b)){p=a;s=c;t=b;}
  else if((b<a)&&(a<c)){p=b;s=a;t=b;}
   ...
}
```

paramètres en C++

Il y a trois types de paramètres les paramètres *consultés*, les paramètres *élaborés* et les paramètres *modifiés*.

Les paramètres modifiés ou élaborés sont traduit en C++ par des paramètres par référence : le nom du paramètre est précédé par &.

exemple : action qui échange les valeurs de deux variables :

```
void echanger( int &x, int &y){
     int z;
     z=x; x=y; y=z;
}
```

Les paramètres consultés peuvent être traduit de différentes façons :

- **type paramètre** c'est la forme la plus simple. Au moment de l'appel, il y a recopie de la valeur du paramètre effectif dans une variable locale de la fonction. Si la taille du paramètre est importante celà peut poser des problèmes de saturation de la pile.
- **type const paramètre** Il y a aussi recopie de la valeur du paramètre effectif, et de plus le compilateur interdit d'écrire des expressions de la forme *a*= ... si a est spécifié avec *const*.

- **type const &paramètre** le paramètre est spécifié par référence, et par conséquent ce qui est transmis à la fonction c'est l'adresse du paramètre effectif ; cependant la présence du mot *const* interdit l'écriture de *a* = ..., et donc la modification de la valeur du paramètre effectif.

exemple : l'action plus1 affecte au second paramètre la valeur du premier augmentée de 1 :

```
void plus1(int i, int& j){
   j=i+1;
}
```

paramètres en C

La spécification des paramètres en C peut se faire de deux façon suivant l'environnement :

void f(a, b); int a; float b; { ... }	void f(int a, float b) { ... }

En C, le passage de paramètre par référence n'existe pas, on utilise alors des pointeurs, mais l'écriture est plus lourde :

```
echanger(x,  y) int * x, *y;
{   int z;
    z=*x; *x=*y; *y=z;
}
```

Pour les paramètres de type pointeur (donnée-résultat, ou résultat), l'appel se fait en transmettant l'adresse du paramètre effectif : *echanger(&a, &b);*

les paramètres par défaut

Les derniers paramètres d'une fonction C++ peuvent être munis d'une valeur par défaut : si ces paramètres sont absents lors de l'appel de la fonction, c'est leur valeur par défaut qui est prise en compte.

exemple :

Définir une fonction *inc* qui augment de *n* la valeur du premier argument, si *n* est omis le premier argument est augmenté de 1 :

```
void inc (int & a, int n=1)
{ int i;
   for (i=1; i<=n; ++i) ++a;
}

void main()
{ int k;
             inc(k,8)      // k est augmenté de 8
             inc(k);       // k est augmenté de 1
}
```

En C, il n'y a pas d'arguments par défaut, ni de *const*.

Les paramètres ellipses (paramètres en nombre variable)

Il peut être intéressant de ne pas spécifier le nombre de paramètres d'une fonction C ou C++. C'est le cas par exemple de la fonction *printf* qui a un nombre variable de paramètres. cette fonction est déclarée de la façon suivante : *printf(char * , ...)*. Pour manipuler les ellipses, ou listes variables d'arguments en C++, il faut utiliser le type *va_list*, et les trois fonctions *va_start*, *va_arg*, et *va_end* qui se trouvent dans la librairie *stdarg.h*.

exemples :

La fonction *x* affiche une liste variable de paramètres terminée par un entier égal à 0, alors que *y* affiche une liste variable de paramètres dont on connait le nombre.

```
void x(int a, ...)
{  va_list ap;
   int arg;
   cout<<a<<"\n";
   va_start(ap,a);
   while((arg=va_arg(ap, int))!=0) cout<<arg<<"\n";
   va_end(ap);
}
void y(int n, ...)
{  va_list ap;
   int arg;
   va_start(ap,a);
   for(int i=1; i<=n;++i){arg=va_arg(ap, int); cout<<arg<<"\n";}
   va_end(ap);
}

void main()
{
  x(1, 2, 3, 0);x(1, 52, 0);x(1, 62, 63,64, 65, 0);
  cout<<"\n\n";
  y(2,1, 2); y(5,1,2,3,4,5);
}
```

paramètres : coercition, promotion

```
int x (int a){ return a+1;}

float y( float a){ return a+1;}

void main()
{  int a=1; long b=2; float c=3.5;
    x(a);    // transmission normale
    x(b);    // troncature
    x(c);    // coercition
    y(a);    // promotion
}
```

fonctions

```
typeDeRetour nomDeLaFonction( liste de paramètres ){
      déclarations ...
      instructions ...
}
```

exemples :

1. fonction qui calcule le maximum de deux entiers :

```
int max2( int a, int b)
{
     if(a>b) return a; else return b;
}
```

2. fonction qui a pour valeur le produit de deux nombres sans utiliser l'opérateur *, le premier nombre étant ≥ 0 :

```
int produit(int x, int y)
{ int i, p;
   i=0;
   p=0;
   while(i<x)
   {
        P+=y;
        ++i;
   }
   return p;
}
```

exercices :

1. Ecrire une définition de fonction qui calcule le maximum de trois entiers.
2. Ecrire une définition de fonction qui calcule la factorielle d'un nombre entier.
3. Ecrire une définition d'action qui calcule le quotient et le reste de la division de deux nombres entiers.

fonction retournant une référence

Lorsqu'une fonction retourne une référence il peut se poser le problème suivant :
soit la définition de fonction suivante :

Cette définition ne pose aucun problème : la valeur retournée par la fonction est recopiée.

```
type f (...){
   type loc;
   ...
   loc =
   ...
   return loc;
}
```

```
type & f (...){
   type loc;
   ...
   loc =
   ...
   return loc;
}
```

En revanche, ici le compilateur génère un message d'erreur, car *loc* meurt au sortir de la fonction, et par conséquent *f* ne peut pas retourner une référence à un objet qui n'existe plus.

Une fonction qui retourne une référence peut être utilisée comme une *l-value* dans une affectation : exemple :

```
int a;
int & entier () {return a;}
void main()
{ cout<<entier()<<"\n";    // affiche 0
  entier()=3;
  cout<<entier();          // affiche 3
  cout<<a;                 // affiche également 3;
}
```

récursivité

Les fonctions C++ peuvent évidemment s 'appeler elles-mêmes : cette possibilité simplifie le travail de définition de la fonction, mais va générer une surconsomation d'espace mémoire dans la pile des appels de fonctions. Exemple, une fonction récursive qui affiche la traduction d'un entier positif ou nul en héxadécimal :

```
void hexa( int a){
    char * hex ="0123456789abcdef";
    int b = a / 16;
    if(b!=0) hexa(b); // appel récursif
    cout<<hex[a%16];
}
```

quelques traits spécifiques à C++

ordre d'évaluation des paramètres

En C++ les paramètres sont évalués de droite à gauche avant l'appel de la fonction :

```
int f( int a, int b){
  return a+b;
}
```

```
a=3 ; cout<<f(++x, x); // affiche 7
a=3 ; cout<<f(x, ++x); // affiche 8
```

prototype, signature

La liste des types des arguments est appelée **signature** de la fonction.

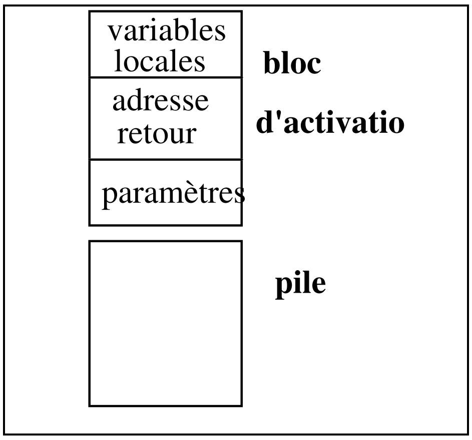
Le triplet type de retour, nom de la fonction, signature est appelé **prototype** de la fonction. Il peut y avoir plusieurs occurrences du prototype d'une fonction dans un même programme, sans qu'il y ait erreur de compilation.

fonctions inline

L'utilisation du mot *inline* devant le prototype d'une fonction permet de la rendre *inline* : c'est une recommandation au compilateur. Si c'est possible, le compilateur génère le code de la fonction au lieu d'un appel à la fonction. Cette génération n'est évidemment pas possible si la fonction est récursive. Définir une fonction *inline* permet :

1. d'accélérer l'exécution du programme : les séquences d'instruction d'appel et de retour de fonction sont supprimées.
2. d'économiser de la place dans la pile : il n'y a pas d'empilement de bloc d'activation de la fonction. Un bloc d'activation est composé essentiellement de :

- variables locales
- adresse retour
- paramètres



L'utilisation de fonctions *inline* augmente en général la taille des programmes s'il y a plusieurs appels de fonctions.

opérateur de portée

L'opérateur de portée **::** permet d'accéder à des variables globales à portée fichier, qui sont masquées par des paramètres ou des variables locales.

```
int a;
int x (int a)
{ ::a=a+1;    // la variable a de gauche est la variable du fichier
              // celle de droite est le paramètre
}
```

Il faut éviter au maximum une telle utilisation car l'utilisation des variables globales fait perdre de l'indépendance aux fonctions, et par conséquent les rend difficiles à maintenir.

static

Une variable locale à une fonction peut être définie en *static* : il y aura alors une seule occurrence de la représentation de la variable pour tous les appels de la fonction. Cette représentation se fait dans le segment de donnée. Cette variable n'est visible que dans la fonction.

pointeur sur une fonction

C++ autorise l'utilisation de pointeurs sur des fonctions

```
void f( int a){
    ....
}

 void (*pf)( int ); // déclaration d'un pointeur sur une fonction void f( int )

 pf = f;            // affectation à pf de f

 pf(30);            // appel de pf donc de f
 (*pf)(30);         // appel de pf donc de f
```

On peut également définir un tableau de pointeurs sur des fonctions de la façon suivante :

```
void (*ptf[10])(int);

ptf[0] = f;         // affectation de f à ptf[0]

ptf[0](30);         // appel de ptf[0]
```

SURCHARGE

On dit qu'une fonction ou un opérateur est surchargé ou surdéfini, lorque le nom de la fonction ou de l'opérateur peut avoir plusieurs sens : l'opérateur + de C++ est surchargé : il a un sens pour des opérandes entiers, et un autre pour des opérandes flottants :

12+13 ⇔ il s'agit d'un + entre entiers, et le résultat est entier.
12.4+13.5 ⇔ il s'agit d'un + entre réels et le résultat est un réel.

La surcharge des fonctions permet de n'utiliser qu'un seul nom pour des fonctions ayant le même sens, mais s'appliquant à des arguments de type différents : écriture d'une fonction *max* qui calcule le maximum de deux nombres de type identiques mais arbitraires (*int* ou *float*) :

```
int max (int a, int b){ return (a>b)?a:b;}

float max (float a, float b){ return (a>b)?a:b;}
```

```
int i, j; float x, y;

max (i, j) ; // max entre entiers
max(x, y);          // max entre flottants
max(i, x);          // erreur de compilation : ambiguïté
```

L'ambiguïté vient de ce que le compilateur ne sait pas s'il doit contraindre le paramètre flottant en un entier, ou au contraire promouvoir le paramètre entier en un flottant. Pour lever l'ambiguïté, on est amené à définir une autre fonction *max*, ou à forcer le bon appel :

```
max(float(i),x);
```

```
float max (int a, float b) { return a>b?a:b;}
```

Pour qu'il y ait surdéfinition de fonction, il faut que les prototypes des deux fonctions soient différents :

- Si le type de retour et la signature des deux fonctions sont identiques, le message d'erreur du compilateur est : *body has already been defined for ...*
- Si la signature est identique, et le type de retour différent, le message d'erreur est : *type mismatch in redeclaration of ...*
- On peut surcharger une foncion en jouant sur le nombre d'arguments :

```
int max (int a, int b, int c) { return max(a, max(b, c));}
```

- Le statut de *const* change la signature pour les pointeurs, mais pas pour les autres types :

```
int max ( int a, int b) { ... }
int max ( const int a, int b ) { ... } // max(const int, int) cannot
                                 // be distingueshed from max(int, int)
```

En revanche, il n'y a pas d'ambiguïté dans les déclarations suivantes :

```
void ecrire ( char * a) { cout << a;}
void ecrire( const char * a ) { cout<<"const : "<<a;}
```

surcharge des opérateurs

Comme les fonctions les opérateurs peuvent être surchargés : seuls les opérateurs ., .* :: ?: ne peuvent pas être surchargés. Pour être surchargés les opérateurs doivent être des opérateurs membres de classes, ou au moins avoir un opérande qui est un objet. On étudiera ceci en même temps que les classes.