Cours de C++

Learn to prog :

Deuxième partie II

Structures de base du C++

Plan

Programmation

Les briques de base du langage:

- Ce qui permet de stocker des données: types, variables, tableaux, etc.
- Les expressions qui permettent de manipuler les données.
- Les instructions pour construire les algorithmes.

Plan

Types élémentaires

- `int` : entiers (au min 16 bits, pour des valeurs ≤ 32767)
  (d'autres formats existent: `long int` (min 32 bits), `short int`, `unsigned int`) ...
- (`float`), `double` et `long double` : nombres à virgule flottante (en général 15 chiffres sign. pour `double`).
  Par ex. 23.3 ou $2.456e12$ ou $23.56e-4$
- `char` : caractères ('a','b',... 'A',...,':',...).
- `bool` : booléens ('true' ou 'false').
  (Et: 0 ≡ false ; tout entier non nul équivaut à false)

(Et aussi: des types particuliers pour les tailles de tableaux, de chaînes de caractères, d'objets etc. `size_t`, `::size_type`)

Définition de variable

Syntaxe : *type v* ;

```
int p ;
double x;
```

- Toute variable doit être définie avant d'être utilisée !
- Une définition peut apparaître n'importe où dans un programme.
- Une variable est définie jusqu'à la fin de la première instruction composée (marquée par }) qui contient sa définition.
- (Une variable définie en dehors de toute fonction – et de tout espace de nom – est une variable globale).

Définition de variable

Une variable peut être initialisée lors de sa déclaration, deux notations sont possibles :

```
int p=34 ;
double x=12.34;
```

```
int p (34) ;
double x (12.34);
```

Une variable d'un type élémentaire qui n'est pas initialisée, n'a pas de valeur définie: elle peut contenir n'importe quoi.

Constantes symboliques

Syntaxe : `const` *type nom = val ;*

Par exemple: `const int Taille = 100;`

Il ne sera pas possible de modifier `Taille` dans le reste du programme (erreur à la compilation)...

Chaînes de caractères

Il existe une classe `string`, ce n'est un pas un type élémentaire.

Pour l'utiliser, il faut placer tête du fichier :

```
# include <string>
```

- `string t;` définit `t` comme une variable...
- `string s(25,'a');`
- `string mot = "bonjour";`
- `s.size()` représente la longueur de `s`
- `s[i]` est le i-ème caractère de `s` (`i` = 0,1,...`s.size()`-1)
- `s+t` est une nouvelle chaîne correspondant à la concaténation de `s` et `t`.

NB: il existe une autre sorte de chaîne de caractères en C/C++

Tableaux

Pour utiliser la classe `vector`, il faut placer en tête du fichier :

```
# include <vector>
```

Un tableau est typé:

```
vector<int> Tab(100,5);
vector<int> Tab(50);
vector<double> T;
```

Structure générale: `vector< type > Nom(n,v);`

- `vector< type > Nom1 = Nom2;`
  → les valeurs de *Nom*2 sont alors recopiées dans *Nom*1.
- `T.size()` correspond à la taille de `T`.
  NB: `size()` renvoie en fait un entier non signé, son type exact est `vector<type>::size_type`

Tableaux

- `T[i]` désigne le i-ème élément avec i = 0, ...`T.size()`-1.
- `vector<vector<int> > T` définit un tableau à deux dimensions.

  Pour l'initialiser, on peut utiliser l'instruction suivante :

```
vector<vector<int> >
T2(100,vector<int>(50,1));
```

  ...on initialise chacune des 100 cases de T1 avec un tableau de taille 50 rempli de 1.

Plan

Affectation

En C/C++, l'affectation est une expression:

Soient *v* une variable (au sens large) et *expr* une expression.

v = *expr* affecte la valeur de *expr* à la variable *v* et retourne la valeur affectée à *v* comme résultat.

Par exemple, `i = (j = 0)` affecte 0 à `j` puis à `i` et retourne 0 !!

Opérateurs classiques

- Opérateurs arithmétiques:
  *, +, -, / (division entière et réelle), % (modulo)

- Opérateurs de comparaison
  < (inférieur), <= (inférieur ou égal), == (égal), > (supérieur), >= (supérieur ou égal) et != (différent)

- Opérateurs booléens
  && représente l'opérateur "ET", || représente le "OU", et ! représente le "NON".
  Par exemple, `((x<12) && ((y>0) || !(z>4)))`

Pré et Post incrément

- `++`*var* incrémente la variable *var* et retourne la nouvelle valeur.
  (`++i` équivaut à `i=i+1`)
- *var*`++` incrémente la variable *var* et retourne l'ancienne valeur.
  (`i++` équivaut à `(i=i+1)-1`)

En dehors d'une expression, `i++` et `++i` sont donc équivalentes.

Pré et Post décrément

- L'expression --*var* décrémente la variable *var* et retourne la nouvelle valeur.
- L'expression *var* -- décrémente la variable *var* et retourne l'ancienne valeur.

Plan

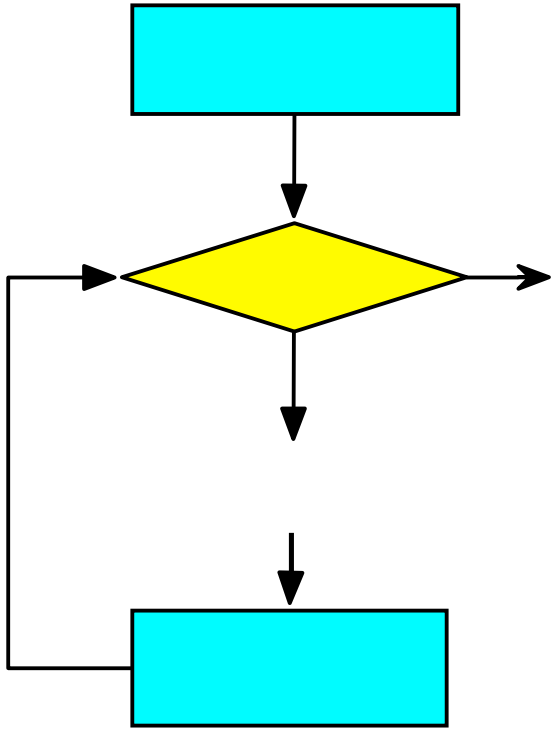
Instructions usuelles

- définition de variables, fonctions, types etc.
- *expr* ;
- { liste d'instructions } : instruction composée.
- `if (`*expr*`)` *instr*
  `if (`*expr*`)` *instr*$_1$ `else` *instr*$_2$

```
if (v == 3) i =i+4 ;

if ((v==3) && (i<5))
     { i=i+4 ;
       v=v*2 ;}
   else v=i ;
r = r*3 ;
```

La boucle FOR

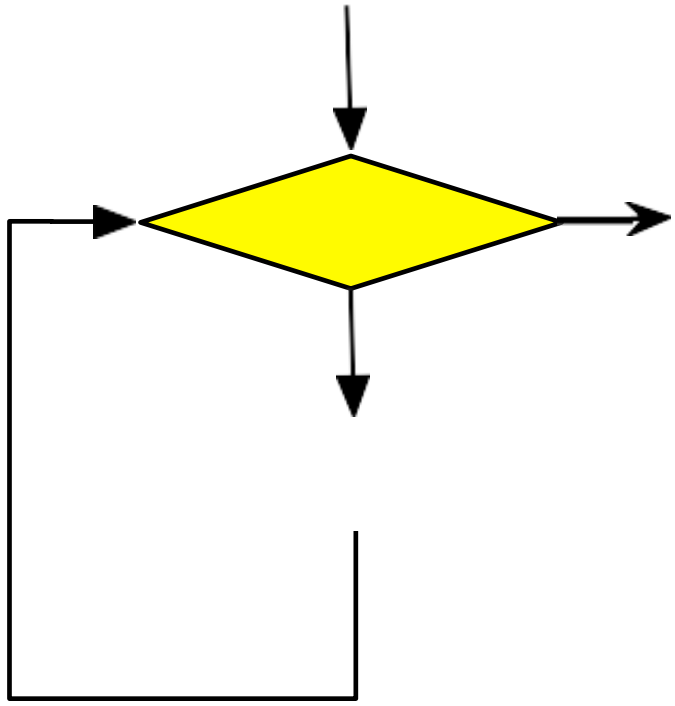
for (*expr*$_1$;*expr*$_2$;*expr*$_3$) *instr*



Ex: for(i=0 ;i<235 ;i=i+1) cout << T[i] ;
for(i=0,j=1 ;i<235 ;i=i+1,j=j+3) cout << T[i][j] ;

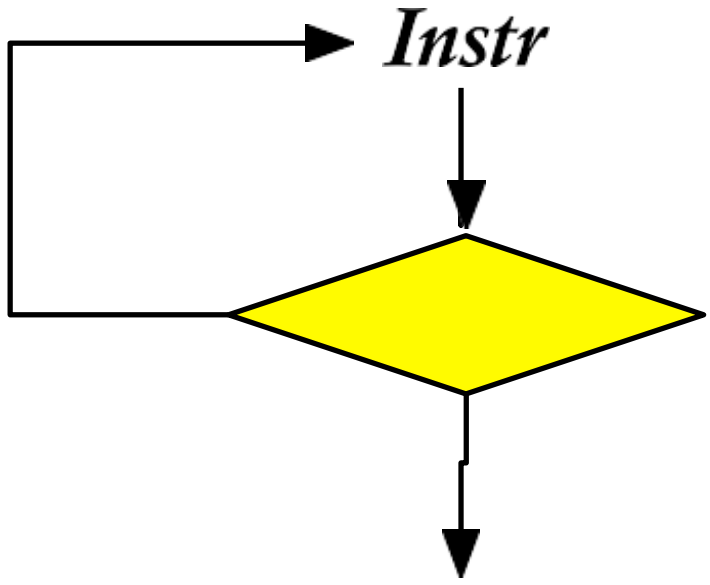
La boucle WHILE

`while (`*expr*`)` *instr*



La boucle DO

```
do instr while (expr);
```



Plan

Afficher à l'écran

Pour utiliser les entrées/sorties, il faut ajouter:

```
# include <iostream>
```

Syntaxe : `cout << expr1 << ... << exprn ;`

Cette instruction affiche $expr_1$ puis $expr_2$...

Afficher un saut de ligne se fait au moyen de `cout << endl`.

```
int i=45 ;
cout << "la valeur de i est " << i << endl ;
```

Afficher à l'écran

Syntaxe : `cout` << *expr*$_1$ << . . . << *expr*$_n$;

- `cout` (ou `std::cout`) désigne le "flot de sortie" standard.
- `<<` est un opérateur binaire:
  - le premier opérande est `cout` (de type "flot de sortie")
  - le second opérande est l'expression à afficher
  - le résultat est de type "flot de sortie'
- `<<` est associatif de gauche à droite
- `<<` est surchargé (ou sur-défini): on utilise le même opérateur pour afficher des caractères, des entiers, des réels ou des chaînes de caractères etc.

Lire au clavier

Syntaxe : `cin` >> *var*$_1$ >> `...` >> *var*$_n$;

Cette instruction lit (au clavier) des valeurs et les affecte à *var*$_1$ puis *var*$_2$ `...`

`cin` est le flot d'entrée standard, et >> est un opérateur similaire à <<.

Les caractères tapés au clavier sont enregistrés dans un buffer dans lequel les `cin` viennent puiser des valeurs. Les espaces, les tabulations et les fins de lignes sont des séparateurs.

Plan

Exemples...

- chaine.cc
- tab.cc
- es.cc

Plan

Evaluation des expressions

Priorité des opérateurs :

- x[y] x++ x--
- ++x --x !x -x
- x*y x/y x% y
- x+y x-y
- x >> y x << y
- x < y x > y x >= y x <= y
- x == y x != y
- x && y
- x || y
- x = y x op= y
- x ? y : z

Evaluation des expressions booléennes

- Dans `e1 && e2`, la sous-expression `e2` n'est évaluée que si `e1` a été évaluée à 'true'.

  `if (i >=0 && T[i] > 20)` blabla

- Dans `e1 || e2`, la sous-expression `e2` n'est évaluée que si `e1` a été évaluée à 'false'.

  `if (i<0 || T[i] > 20)` blabla

Evaluation des expressions arithmétiques

Si une (sous-)expression mélange plusieurs types, c'est le type le plus large qui est utilisé.

```
int i=3,j=2,m;
double r=3.4;
m = (i/j)*r;
```

- D'abord l'expression `(i/j)` est évaluée: `/` désigne ici la division entière, cela donne donc 1.
- Pour évaluer le produit `1*r`, il faut convertir 1 en `double` (1.0) et faire le produit sur les doubles, cela donne 3.4
- Pour l'affectation, comme `m` est entier, 3.4 est converti en `int`. Finalement on a `m = 3`.

Evaluation des expressions arithmétiques

· Pour éviter les erreurs, il est possible de convertir explicitement des données d'un certain type en un autre.

Par exemple:

```
int i=3,j=2,m;
double r=3.4;
m = (double(i)/j)*r ;
```

Donne...

Evaluation des expressions arithmétiques

· Pour éviter les erreurs, il est possible de convertir explicitement des données d'un certain type en un autre.

Par exemple:

```
int i=3,j=2,m;
double r=3.4;
m = (double(i)/j)*r ;
```

Donne... 5 !

Evaluation des expressions arithmétiques

· Pour éviter les erreurs, il est possible de convertir explicitement des données d'un certain type en un autre.

Par exemple:

```
int i=3,j=2,m;
double r=3.4;
m = (double(i)/j)*r ;
```

Donne... 5 !

· L'évaluation d'une expression arithmétique ne se fait pas toujours de gauche à droite !

Ex: `(i/j)*(r/3)`

(démo)

Plan

Structure générale d'un programme

Un programme C++ est réparti dans un ou plusieurs fichiers.

Chacun peut contenir des définitions/déclarations de fonctions, des définitions de types et des définitions de variables globales.

Il existe une seule fonction `main`: c'est la fonction qui sera exécutée après la compilation.

Le profil de `main` est : `int main()` ou `int main( int argc, char ** argv )` pour passer des arguments.
Exemple de programme complet. . .

Structure générale d'un programme

Un programme complet:

```
#include <iostream>

void test(int  j)
{  cout << j << endl ; }

int main()
{
int i =20 ;
cout << "bonjour" << endl;
test(i) ;
}
```

Plan

Définition de fonction

type `nom(` liste des paramètres`)` `{` corps `}`

- *type* est le type du résultat de la fonction.
  (`void` si il s'agit d'une procédure)
- La liste des paramètres (*paramètres formels*):
  type$_1$ `p1, ..., ` *type*$_n$ `pn`
- Le corps décrit les instructions à effectuer.

  Le corps utilise ses propres variables locales, les éventuelles variables globales et les paramètres formels.
- Si une fonction renvoie un résultat, il doit y avoir (au moins) une instruction `return` *expr* `;`

  Dans le cas d'une procédure, on peut utiliser: `return ;`

Exemple

```
int max(int a,int b)
{
int res=b ;
if (a>b) res = a ;
return res ;
}
```

Appel d'une fonction

`nom(liste des arguments)`

La liste des arguments (*paramètres réels*) est $expr_1$, $expr_2$, ... $expr_n$ où chaque $expr_i$ est compatible avec le type $type_i$ du paramètre formel `pi`.

Exemple :

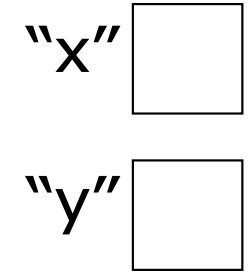
```
 int k=34, t=5, m;
m = max(k,2*t+5);
```

Exemple

```
int max(int a,int b)
{int res=b;
if (a>b) res = a ;
return res ; }
...

int main() {
int x,y;
x=5 ;
y=10 ;
int z = max(y,x);
cout<<" z = "<<z; }
```

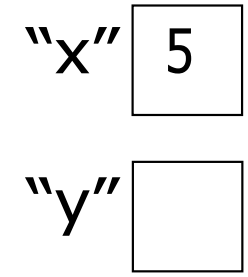
Exemple

```
int max(int a,int b)
{int res=b;
if (a>b) res = a ;
return res ; }
...

int main() {
int x,y;
x=5 ;
y=10 ;
int z = max(y,x);
cout<<" z = "<<z; }
```



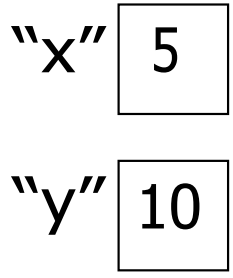
Exemple

```
int max(int a,int b)
{int res=b;
if (a>b) res = a ;
return res ; }
...

int main() {
int x,y;
x=5 ;
y=10 ;
int z = max(y,x);
cout<<" z = "<<z; }
```



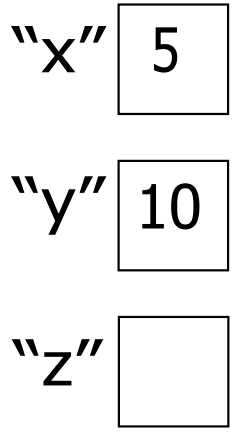
Exemple

```
int max(int a,int b)
{int res=b;
if (a>b) res = a ;
return res ; }
...

int main() {
int x,y;
x=5 ;
y=10 ;
int z = max(y,x);
cout<<" z = "<<z; }
```



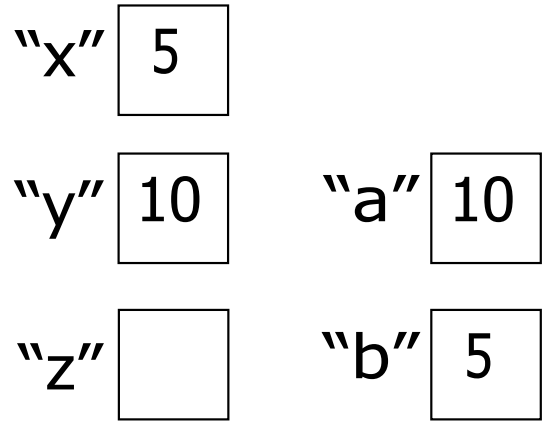
Exemple

```
int max(int a,int b)
{int res=b;
if (a>b) res = a ;
return res ; }
...

int main() {
int x,y;
x=5 ;
y=10 ;
int z = max(y,x);
cout<<" z = "<<z; }
```



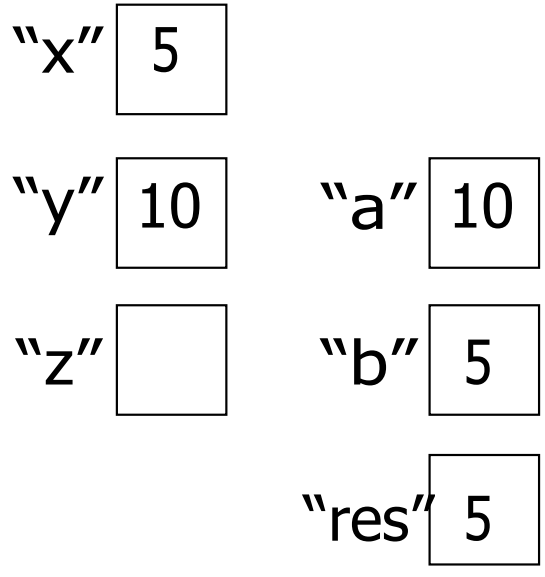
Exemple

```
int max(int a,int b)
{int res=b;
if (a>b) res = a ;
return res ; }
...

int main() {
int x,y;
x=5 ;
y=10 ;
int z = max(y,x);
cout<<" z = "<<z; }
```



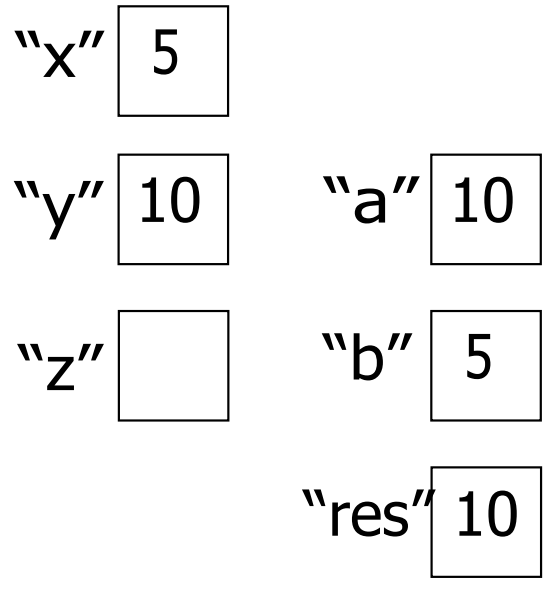
Exemple

```
int max(int a,int b)
{int res=b ;
if (a>b) res = a ;
return res ; }
...

int main() {
int x,y ;
x=5 ;
y=10 ;
int z = max(y,x) ;
cout<<" z = "<<z ; }
```



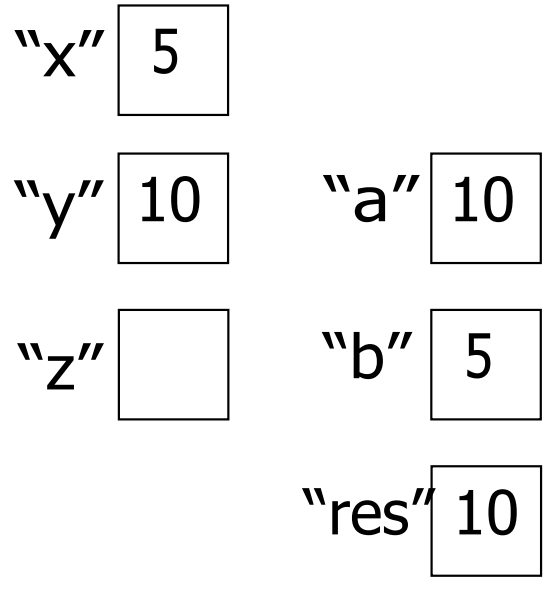
Exemple

```
int max(int a,int b)
{int res=b;
if (a>b) res = a ;
return res ; }
...

int main() {
int x,y;
x=5 ;
y=10 ;
int z = max(y,x);
cout<<" z = "<<z; }
```



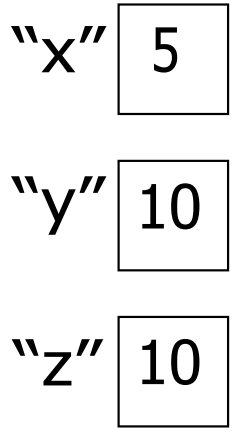
Exemple

```
int max(int a,int b)
{int res=b;
if (a>b) res = a ;
return res ; }
...

int main() {
int x,y;
x=5 ;
y=10 ;
int z = max(y,x);
cout<<" z = "<<z; }
```



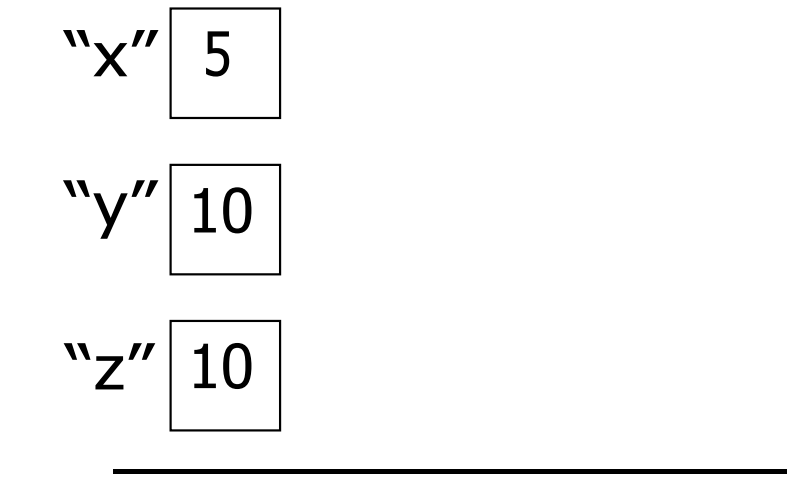
Exemple

```
int max(int a,int b)
{int res=b;
if (a>b) res = a ;
return res ; }
...

int main() {
int x,y;
x=5 ;
y=10 ;
int z = max(y,x);
cout<<" z = "<<z; }
```



Exemple

```
int max(int a,int b)
{int res=b;
if (a>b) res = a ;
return res ; }
...

int main() {
int x,y;
x=5 ;
y=10 ;
int z = max(y,x);
cout<<" z = "<<z; }
```



z = 10

Exemple

```
int max(int a,int b)
{int res=b;
if (a>b) res = a ;
return res ; }
...

int main() {
int x,y;
x=5 ;
y=10 ;
int z = max(y,x);
cout<<" z = "<<z; }
```

- Fin

Afficher le contenu d'un tableau d'entiers

```
 void AfficherTab(vector<int> T)
{
for (int i=0; i< T.size(); i++)
   cout << T[i] << " ";
}
```

Saisie d'un tableau d'entiers

Saisie d'un tableau d'entiers

```
 vector<int> SaisieTab()
{
int taille ;
cout << " Entrer une taille : " ;
cin >> taille ;
vector<int> res(taille,0) ;

for (int i=0 ; i< taille ; i++) {
   cout << " val = " ;
   cin >> res[i] ;
}
return res ;
}
```

Recherche du plus grand élément

Recherche du plus grand élément

```
 int Recherche(vector<int> T)
{
if (T.size()==0) {
   cout << ''Erreur ! Tableau vide !'' << endl ;
return -1 ;}

int res=T[0] ;

for (int i=1 ; i<T.size() ;i++)
   if (T[i] > res) res=T[i] ;

return res ;
}
```

Recherche de l'indice du plus grand élément

Recherche de l'indice du plus grand élément

```
int RechercheInd(vector<int> T)
{
if (T.size()==0) {
   cout << ''Erreur ! Tableau vide !'' << endl ;
return -1 ;}

int res=0 ;

for (int i=1 ; i<T.size() ;i++)
   if (T[i] > T[res]) res=i ;

return res ;
}
```

Portée des identificateurs

Un identificateur XXX peut être utilisé à la ligne *l* de la définition d'une fonction *F* ssi:

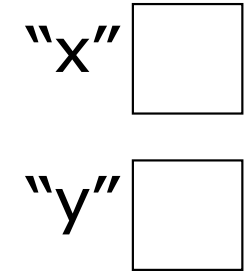
- XXX est une variable définie dans une instruction composée contenant *l* .
- XXX est un paramètre de *F*.
- (XXX est une variable globale, *i.e.* définie hors de toute fonction.)

Exemple

```
int max(int a,int b)
{int res=b ;
if (a>b) res = a ;
return res ; }
...

int main() {
int x,y ;
x=5 ;
y=10 ;
int z = max(y,x) ;
cout<<" z = "<<z ; }
```

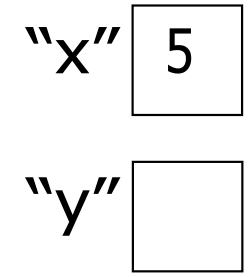
Exemple

```
int max(int a,int b)
{int res=b;
if (a>b) res = a ;
return res ; }
...

int main() {
int x,y;
x=5 ;
y=10 ;
int z = max(y,x);
cout<<" z = "<<z; }
```



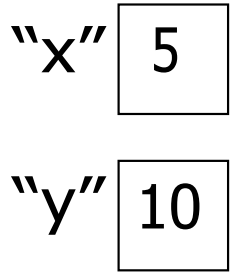
Exemple

```
int max(int a,int b)
{int res=b;
if (a>b) res = a ;
return res ; }
...

int main() {
int x,y;
x=5 ;
y=10 ;
int z = max(y,x);
cout<<" z = "<<z; }
```



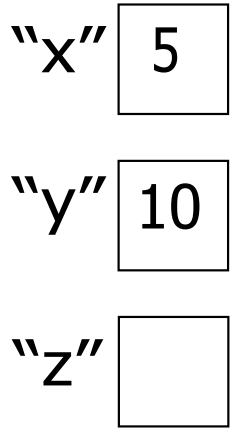
Exemple

```
int max(int a,int b)
{int res=b;
if (a>b) res = a ;
return res ; }
...

int main() {
int x,y;
x=5 ;
y=10 ;
int z = max(y,x);
cout<<" z = "<<z; }
```



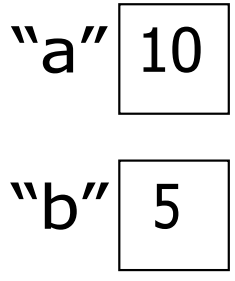
Exemple

```
int max(int a,int b)
{int res=b;
if (a>b) res = a ;
return res ; }
...

int main() {
int x,y;
x=5 ;
y=10 ;
int z = max(y,x);
cout<<" z = "<<z; }
```



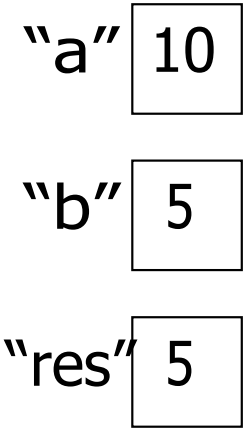
Exemple

```
int max(int a,int b)
{int res=b;
if (a>b) res = a ;
return res ; }
...

int main() {
int x,y;
x=5 ;
y=10 ;
int z = max(y,x);
cout<<" z = "<<z; }
```



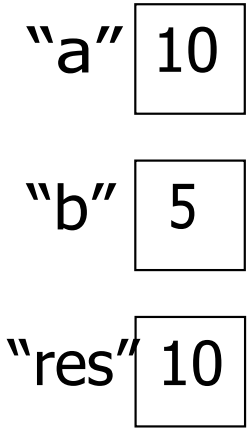
Exemple

```
int max(int a,int b)
{int res=b ;
if (a>b) res = a ;
return res ; }
...

int main() {
int x,y ;
x=5 ;
y=10 ;
int z = max(y,x) ;
cout<<" z = "<<z ; }
```



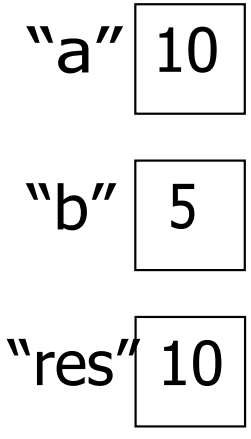
Exemple

```
int max(int a,int b)
{int res=b;
if (a>b) res = a ;
return res ; }
...

int main() {
int x,y;
x=5 ;
y=10 ;
int z = max(y,x);
cout<<" z = "<<z; }
```



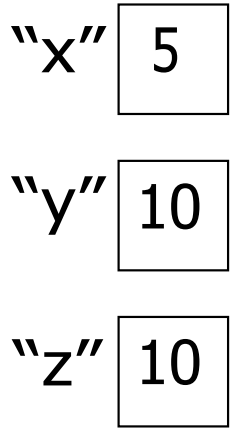
Exemple

```
int max(int a,int b)
{int res=b;
if (a>b) res = a ;
return res ; }
...

int main() {
int x,y;
x=5 ;
y=10 ;
int z = max(y,x);
cout<<" z = "<<z; }
```



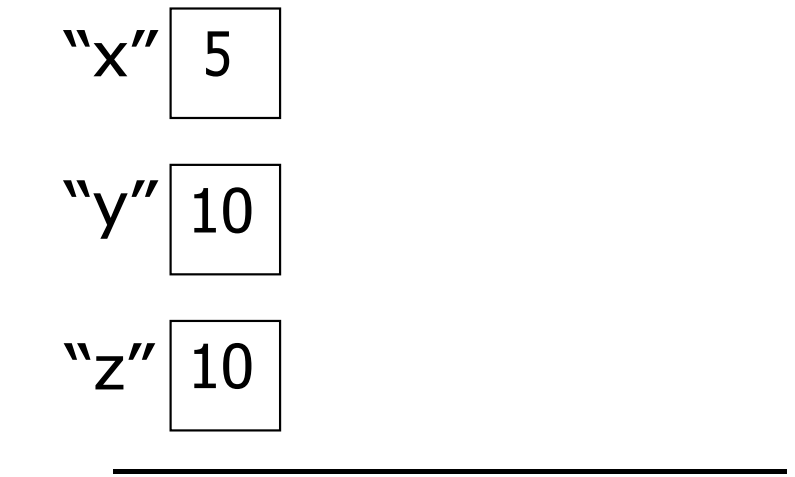
Exemple

```
int max(int a,int b)
{int res=b;
if (a>b) res = a ;
return res ; }
...

int main() {
int x,y;
x=5 ;
y=10 ;
int z = max(y,x);
cout<<" z = "<<z; }
```



Exemple

```
int max(int a,int b)
{int res=b;
if (a>b) res = a ;
return res ; }
...

int main() {
int x,y;
x=5 ;
y=10 ;
int z = max(y,x);
cout<<" z = "<<z; }
```



z = 10

Exemple

```
int max(int a,int b)
{int res=b;
if (a>b) res = a ;
return res ; }
...

int main() {
int x,y;
x=5 ;
y=10 ;
int z = max(y,x);
cout<<" z = "<<z; }
```

- Fin

Passage de paramètres par valeur

Par défaut, les paramètres d'une fonction sont initialisés par une copie des valeurs des paramètres réels.

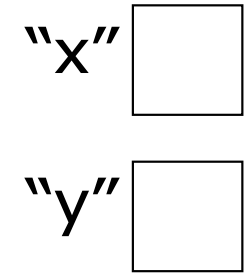
Modifier la valeur des paramètres formels dans le corps de la fonction ne change pas la valeur des paramètres réels.

Essai de permutation

```
void permut(int
a,int b)
{int aux = b ;
b = a ;
a = aux ;}
...

int main() {
int x,y;
x=5 ;
y=10 ;
permut(y,x) ;
cout<<" x = "<<x ; }
```

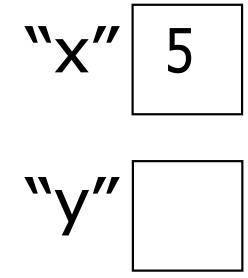
Essai de permutation

```
void permut(int
a,int b)
{int aux = b ;
b = a ;
a = aux ;  }
...

int main() {
int x,y;
x=5 ;
y=10 ;
permut(y,x) ;
cout<<" x = "<<x ; }
```



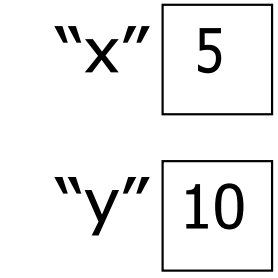
Essai de permutation

```
void permut(int
a,int b)
{int aux = b ;
b = a ;
a = aux ;  }
...

int main() {
int x,y;
x=5 ;
y=10 ;
permut(y,x) ;
cout<<" x = "<<x ; }
```



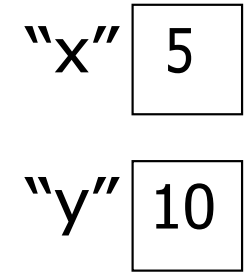
Essai de permutation

```
void permut(int
a,int b)
{int aux = b ;
b = a ;
a = aux ;  }
...

int main() {
int x,y;
x=5 ;
y=10 ;
permut(y,x) ;
cout<<" x = "<<x ; }
```



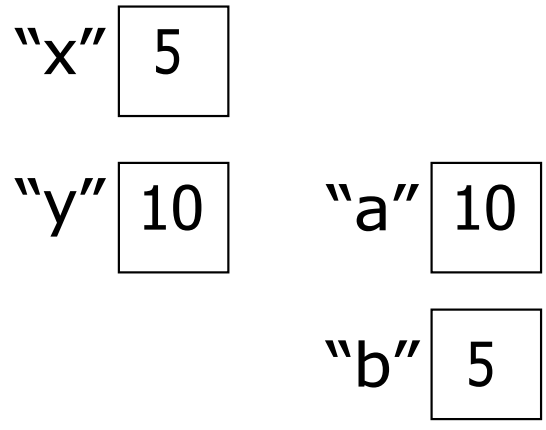
Essai de permutation

```
void permut(int
a,int b)
{int aux = b ;
b = a ;
a = aux ;  }
...

int main() {
int x,y;
x=5 ;
y=10 ;
permut(y,x) ;
cout<<" x = "<<x ; }
```



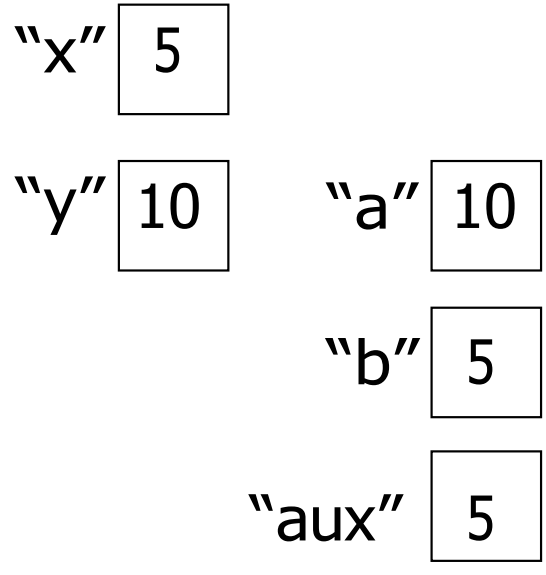
Essai de permutation

```
void permut(int
a,int b)
{int aux = b ;
b = a ;
a = aux ;}
...

int main() {
int x,y;
x=5 ;
y=10 ;
permut(y,x) ;
cout<<" x = "<<x ; }
```



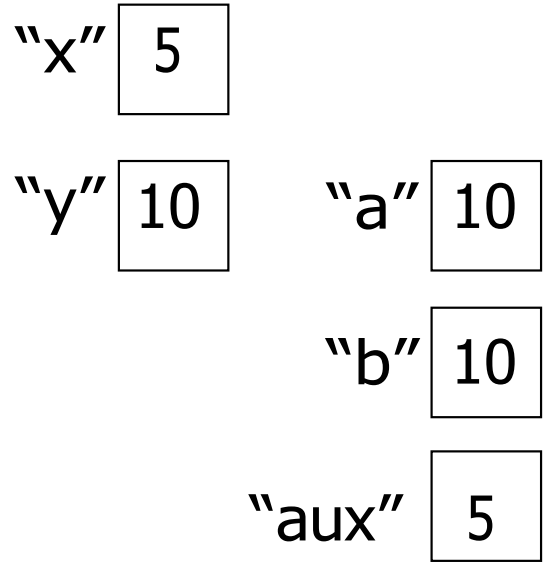
Essai de permutation

```
void permut(int
a,int b)
{int aux = b ;
b = a ;
a = aux ;}
...

int main() {
int x,y;
x=5 ;
y=10 ;
permut(y,x) ;
cout<<" x = "<<x ; }
```



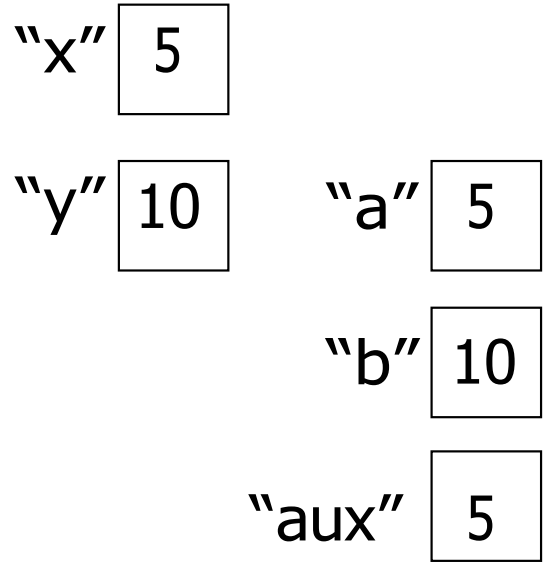
Essai de permutation

```
void permut(int
a,int b)
{int aux = b ;
b = a ;
a = aux ;}
...

int main() {
int x,y;
x=5 ;
y=10 ;
permut(y,x) ;
cout<<" x = "<<x ; }
```



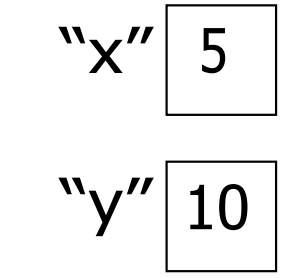
Essai de permutation

```
void permut(int
a,int b)
{int aux = b ;
b = a ;
a = aux ; }
...

int main() {
int x,y;
x=5 ;
y=10 ;
permut(y,x) ;
cout<<" x = "<<x ; }
```



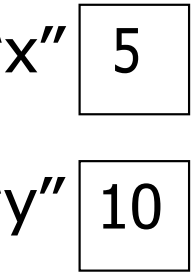
Essai de permutation

```
void permut(int
a,int b)
{int aux = b ;
b = a ;
a = aux ;  }
...

int main() {
int x,y;
x=5 ;
y=10 ;
permut(y,x) ;
cout<<" x = "<<x ; }
```



Essai de permutation

```
void permut(int
a,int b)
{int aux = b ;
b = a ;
a = aux ;  }
...

int main() {
int x,y;
x=5 ;
y=10 ;
permut(y,x) ;
cout<<" x = "<<x ; }
```



x = 5

Essai de permutation

```
void permut(int
a,int b)
{int aux = b ;
b = a ;
a = aux ;}
...

int main() {
int x,y;
x=5 ;
y=10 ;
permut(y,x) ;
cout<<" x = "<<x ; }
```

Passage des paramètres par référence

Pour modifier la valeur d'un paramètre réel dans une fonction, il faut passer ce paramètre par référence.

Une référence sur une variable est un synonyme de cette variable, c'est-à-dire une autre manière de désigner le même emplacement de la mémoire.

On utilise le symbole & pour la déclaration d'une référence:

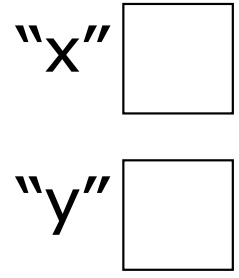
Dans la liste des paramètres de la définition d'une fonction, *type* & p_i déclare le paramètre p_i comme étant une référence sur le i^{eme} paramètre réel v_i.

Permutation

```
void permut(int & a,int & b)
{int aux = b ;
b = a ;
a = aux ;}
...

int main() {
int x,y;
x=5 ;
y=10 ;
permut(y,x) ;
cout<<" x = "<<x ; }
```

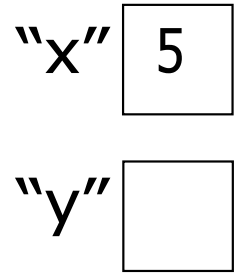
Permutation

```
void permut(int & a,int & b)
{int aux = b ;
b = a ;
a = aux ;  }
...

int main() {
int x,y;
x=5 ;
y=10 ;
permut(y,x) ;
cout<<" x = "<<x ; }
```



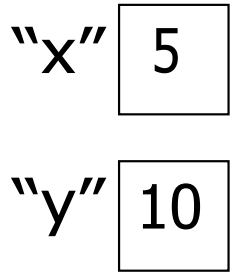
Permutation

```
void permut(int & a,int & b)
{int aux = b ;
b = a ;
a = aux ;  }
...

int main() {
int x,y;
x=5 ;
y=10 ;
permut(y,x) ;
cout<<" x = "<<x ; }
```



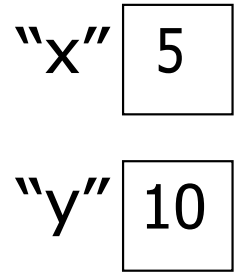
Permutation

```
void permut(int & a,int & b)
{int aux = b ;
b = a ;
a = aux ;  }
...

int main() {
int x,y;
x=5 ;
y=10 ;
permut(y,x) ;
cout<<" x = "<<x ; }
```



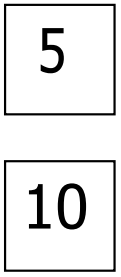
Permutation

```
void permut(int & a,int & b)
{int aux = b ;
b = a ;
a = aux ;  }
...

int main() {
int x,y;
x=5 ;
y=10 ;
permut(y,x) ;
cout<<" x = "<<x ; }
```



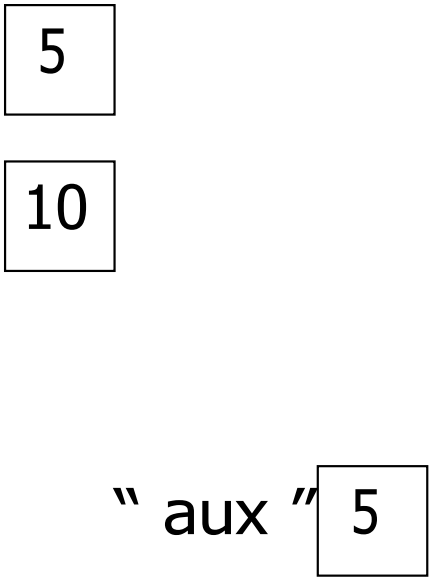
Permutation

```
void permut(int & a,int & b)
{int aux = b ;
b = a ;
a = aux ;  }
...

int main() {
int x,y;
x=5 ;
y=10 ;
permut(y,x) ;
cout<<" x = "<<x ; }
```



Permutation

```
void permut(int & a,int & b)
{int aux = b ;
b = a ;
a = aux ;  }
...

int main() {
int x,y;
x=5 ;
y=10 ;
permut(y,x) ;
cout<<" x = "<<x ; }
```



Permutation

```
void permut(int & a,int & b)
{int aux = b ;
b = a ;
a = aux ;  }
...

int main() {
int x,y;
x=5 ;
y=10 ;
permut(y,x) ;
cout<<" x = "<<x ; }
```

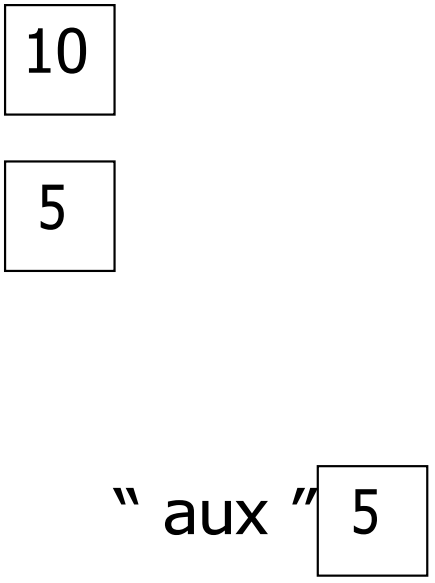
"b"^"x" 10

"a"^"y" 10

" aux " 5

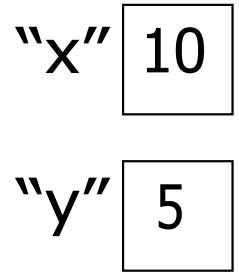
Permutation

```
void permut(int & a,int & b)
{int aux = b ;
b = a ;
a = aux ;  }
...

int main() {
int x,y;
x=5 ;
y=10 ;
permut(y,x) ;
cout<<" x = "<<x ; }
```



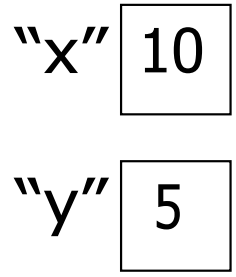
Permutation

```
void permut(int & a,int & b)
{int aux = b ;
b = a ;
a = aux ;  }
...

int main() {
int x,y;
x=5 ;
y=10 ;
permut(y,x) ;
cout<<" x = "<<x ; }
```



Permutation

```
void permut(int & a,int & b)
{int aux = b ;
b = a ;
a = aux ;  }
...

int main() {
int x,y;
x=5 ;
y=10 ;
permut(y,x) ;
cout<<" x = "<<x ; }
```



x = 10

Permutation

```
void permut(int & a,int & b)
{int aux = b ;
b = a ;
a = aux ;}
...

int main() {
int x,y;
x=5 ;
y=10 ;
permut(y,x) ;
cout<<" x = "<<x ; }
```

- Fin

Passage par référence

- Il faut que les paramètres réels soient compatibles avec un passage par référence...

  `permut(x,5)` n'est pas possible !

- Usage "bizarre" des références dans la liste des paramètres: `const` *type* `&` *p*

  Idée: le passage par référence est plus efficace que le passage par valeur car il évite la recopie des arguments car seule l'adresse mémoire de l'argument est communiquée à la fonction.

Exercice …

Algorithmes de tri
Comment trier un tableau d’entiers ?

Exercice …

Algorithmes de tri
Comment trier un tableau d'entiers ?

- rechercher le plus grand élément et le placer à la fin du tableau
- recherche le deuxième plus grand et le placer en avant-dernière position
- etc.

Exercice ...

Algorithmes de tri

Comment trier un tableau d'entiers `Tab` ?

Pour `IndFin` = `Tab.size()-1` ... `1` faire

- rechercher l'indice `IMAX` du plus grand élément de `Tab` entre les indices `0` à `IndFin`.
- Permuter les éléments placés en `IMAX` et `IndFin`.

Exercice ...

Algorithmes de tri

Comment trier un tableau d'entiers `Tab` ?

Pour `IndFin` = `Tab.size()-1 ... 1` faire

- rechercher l'indice `IMAX` du plus grand élément de `Tab` entre les indices `0` à `IndFin`.
- Permuter les éléments placés en `IMAX` et `IndFin`.

A la première étape, on sélectionne le plus grand élément et on le place à la fin du tableau ; puis on trouve le deuxième plus grand et on le place à l'avant-dernière place etc.

Fonction de sélection

```
int RechercheInd(vector<int> T, int imax)
{
  if (T.size() < imax-1) {
   cout << "Erreur ! Tableau trop petit ! " << endl ;
   return -1 ; }

  int res=0 ;

  for (int i=1 ; i<= imax ;i++)
   if (T[i] > T[res]) res=i ;

  return res ;
}
```

Fonction de sélection

```
 vector<int> Trier(vector<int> T)
{
  vector<int> Taux = T ;
  int aux ;

  for (int i=Taux.size()-1 ; i>0 ; i--) {
   aux = RechercheInd(Taux,i) ;
   Permuter(Taux[aux],Taux[i]) ;
   }

  return Taux ;
}
```

Fonction de sélection

Tri *in situ* avec passage par référence

```
void Trier(vector<int> & T)
```

Fonction de sélection

Tri *in situ* avec passage par référence

```
 void Trier(vector<int> & T)
{
   int aux ;

   for (int i=T.size()-1; i>0; i--) {
    aux = RechercheInd(Taux,i);
    Permuter(T[aux],T[i]);
    }
}
```

Plan

Fonctions récursives

La factorielle...

```
 int fact(int  n)
{
  assert(n>=0) ;
  if (n==0) return 1 ;
  return n*fact(n-1) ;
}
```

Les tours de Hanoi

```
 void Hanoi(int nb,int p1, int p2)
{

if (nb > 0) {
  int p3 = 6-p1-p2 ;
  Hanoi(nb-1,p1,p3) ;
  cout << "On bouge le Pion " << nb << " vers " <<
p2 ;
  Hanoi(nb-1,p3,p2) ;
}
}
```

Plan

Valeurs par défaut des paramètres

On peut attribuer des valeurs par défaut aux paramètres d'une fonction.

```
int max(int a,int b =0)
{
if (a>b) return a ;
else return b ;
}
```

Recherche dichotomique

```
 int RechDicho(vector<double> Tab,int bg, int
bd,double x)
{ if (bg > bd) return -1 ;
int M = (bg+bd)/2 ;
if (Tab[M]==x) return M;
if (Tab[M] > x)
  return RechDicho(Tab,bg,M-1,x) ;
else return RechDicho(Tab,M+1,bd,x) ;
}
 int Recherche(vector<double> Tab,double x)
{ int taille = Tab.size() ;
if (taille == 0) return -1 ;
return RechDicho(Tab,0,taille-1,x) ;
}
```

Recherche dichotomique

```
 int Recherche(vector<double> Tab, double x,
int bg =-1, int bd =-1)
{ if((bg==-1)&&(bd ==-1))
  return Recherche(Tab,x,0,Tab.size()-1) ;
if (bg > bd) return  -1 ;
int M = (bg+bd)/2 ;
if (Tab[M]==x) return M;
if (Tab[M] > x)
  return Recherche(Tab,x,bg,M-1) ;
else return Recherche(Tab,x,M+1,bd) ;
}
```

Appel directement avec: Recherche(T,10.3)

Fonctions inline

Pour de petites fonctions, il peut être plus efficace (= rapide) d'éviter un appel de fonction et à la place, expanser le corps de la fonction en lieu et place de chaque appel.

C'est possible avec le mot clé `inline`.

```
inline int max(int a,int b =0)
{
if (a>b) return a ;
else return b ;
}
```

Le tri fusion

Principe:

Pour trier un tableau T entre les indices a et b:

- On trie entre a et $\frac{a+b}{2}$
- On trie entre $\frac{a+b}{2}$ et b
- On construit un tableau trié à partir des sous-parties triées.

```
 void Tri_fusion(vector<int> & T,int bg =-1, int bd
=-1)
{
  if ((bg==-1) && (bd==-1)) {
    bg=0 ;
    bd=int(T.size()) -1 ;
  }
  int k ;
  if (bg < bd) {
    k = (bg+bd)/2 ;
    Tri_fusion(T,bg,k) ;
    Tri_fusion(T,k+1,bd) ;
    Fusion(T,bg,k,bd) ;
  }
```

```
}
```

Plan

Portée d'un identificateur

La portée d'un identificateur correspond aux parties du programme où cet identificateur peut être utilisé sans provoquer d'erreur à la compilation.

La portée d'une variable globale ou d'une fonction globale est égale au programme.

La portée d'une variable locale va de sa définition jusqu'à la fin de la première instruction composée ({...}) qui contient sa définition.

Deux variables peuvent avoir le même nom mais dans ce cas, elle doivent avoir deux portés différentes.

Portée

```
  {
int i=3 ;
  {
    int i=5 ;
    cout << i ; // affiche 5
  }
cout << i ; // affiche 3
}
```

Les deux variables ont des portées différentes !

Exemple. . .

Visibilité

· La visibilité d'une variable dit quand elle est accessible

La visibilité est inclue dans la portée...

· La durée de vie d'une variable correspond à la période depuis création à sa destruction.

Durée de vie et portée coïncident pour toutes les variables que nous avons évoquées jusqu'ici.

mais . . .

Variables static

L'espace mémoire d'une variable locale déclarée `static` est alloué une seule fois et n'est détruit qu'à la fin du programme !

```
 void Test(int v)
{
static int compteur=0;

compteur++;
cout << compteur << "-eme appel de Test"<< endl;

cout << " v = " << v << endl;

}
```

Sa durée de vie est celle du programme.

Hanoi (suite)

Obj: compter le nombre d'appels à la procédure Hanoi.

```
 void Hanoi(int nb,int p1, int p2, int affichage=0)
{   static int NB = 0 ;
NB++ ;
if (affichage == 1) {
cout << NB << " appels de Hanoi "
return ; }

if (nb > 0) {
  int p3 = 6-p1-p2 ;
  Hanoi(nb-1,p1,p3) ;
  cout << "On bouge le Pion " << nb << " vers le
piquet " << p2 << endl ;
  Hanoi(nb-1,p3,p2) ;
}
}
```

Espace de noms

Un programme important utilise de nombreuses bibliothèques, fonctions etc. Pour éviter les problèmes de conflit de noms, on utilise des espaces de noms (namespace) :

- on associe un nom à un ensemble de variables, types, fonctions.
- leur nom complet est: leur nom d'espace suivi de `::` et de leur nom.

  Ex. la fonction `int fct(){...}` définie dans le domaine `A` aura pour nom complet `A::fct()`

Les noms complets de `cout` et `cin` sont `std::cout` et `std::cin`.

Pour éviter d'écrire les noms complets , on utilise:
`using namespace std;`

Créer un espace de nom

```
namespace toto {

 int N = 10 ;

 int Test()
 {
   cout << "test... " ;
 }

}
```

Les objets N et Test ont pour nom toto::N et toto::Test

Plan

Compléments - 1

Instructions supplémentaires:

- `break` provoque l'arrêt de la première instruction `do`, `for`, `switch` ou `while` englobant.
- `continue` provoque (dans une instruction `do`, `for`, ou `while`) l'arrêt de l'itération courante et le passage au début de l'itération suivante.
- `switch (`*exp*`) {`
  `case` *cste*$_1$: liste d'instructions$_1$
  ...
  `case` *cste*$_n$: liste d'instructions$_n$
  `default:` liste d'instructions
  `}`

Compléments - 2

- *(exp) ? exp1 : exp2* vaut *exp1* si *exp* est évaluée à vrai et *exp2* sinon.

  `m = (i < j) ? i : j`

- *var op= expr* désigne l'expression *var = var op expr* (*op* ∈ {+,-,*,/,%})

```
int i=2,j=34;
i += 2;
j *= 10;
j *= (i++);
```

```
int i=2,j=34;
i = i+2;
j = j*10;
j = j*i;
i = i+1;
```

Compléments - 2

- *(exp) ? exp1 : exp2* vaut *exp1* si *exp* est évaluée à vrai et *exp2* sinon.

```
m = (i < j) ? i : j
```

- *var op= expr* désigne l'expression *var = var op expr* (*op* ∈ {+,-,*,/,%})

```
int i=2,j=34 ;
i += 2;
j *= 10 ;
j *= (i++) ;
```

```
int i=2,j=34 ;
i = i+2 ;
j = j*10;
j = j*i ;
i = i+1 ;
```

A la fin, on a: i==5 et j==1360

Troisième partie III

Classes C++

Plan

Introduction

Un algorithme n'utilise pas que des objets de type int, double ou bool...

- états et transitions lorsqu'on parle de graphes.
- Piquet et pions pour Hanoi
- Fractions
- Piles

Un programme doit "coller" à l'algorithme !

De nouveaux types

Il faut pouvoir définir de nouveaux types de données.
Définir des variables de ce types, les manipuler dans les fonctions etc.

De nouveaux types

Il faut pouvoir définir de nouveaux types de données.
Définir des variables de ce types, les manipuler dans les fonctions etc.

Un type de données = regroupement de données + des opérations

Comme les types de base !

Plan

Regroupement de données

Exemple: une base de données pour les notes des étudiants.

Regroupement de données

Exemple: une base de données pour les notes des étudiants.

Chaque fiche d'étudiant contient :

- un nom et un prénom
- un numéro d'inscription
- un ensemble de notes

Une "base de données" pour une promo = un ensemble de fiches

Fiche d'étudiants...

Définir le type fiche:

```
struct FicheEtudiant{
   string Nom;
   string Prenom;
   int Numero;
   vector<double> Notes;
};
```

Ce n'est pas ordonné...

Fiche d'étudiants...

Autre définition équivalente pour le type fiche:

```
struct FicheEtudiant{
   string Nom, Prenom;
   int Numero;
   vector<double> Notes;
};
```

Utilisation des fiches

`fiche` est un type comme les autres:

```
fiche F ;
```

définit une variable `F` de type fiche.

Une fiche contient plusieurs parties (champs), on y accède par leur nom précédé d'un point "." :

`F.Nom` désigne le champ (de type `string`) `Nom` de la fiche `F`.
`F.Notes[i]` désigne le champ (de type `double`) `Notes[i]` de la fiche `F`.
etc.

Utilisation des fiches

Définir les champs d'une fiche:

```
FicheEtudiant F;
F.Nom = "Takis";
F.Prenom = "Jean";
F.Numero = 1234;

F.Notes[2] = 12.3;
```

Utilisation des fiches

Une fonction afficher...

```
voidAfficher(FicheEtudiantv)
{
   cout  << "No : " <<   v.Numero  << " -" <<   v.Prenom <<"
   for (int i=0; i< v.Notes.size() ; i++)
      cout <<v.Notes[i]<<"";
   cout <<endl;
}
```

demo 1-2-3

Les “struct”

On utilise le type défini pat un `struct` comme n’importe quel autre type: dans les fonctions (paramètre ou résultat), les variables, les tableaux etc.

Les affectations (paramètres de fonction, variable, etc.) se font champ à champ.

Plan

Les classes C++

Lorsqu'on définit un type de données, on fournit des fonctions de manipulation et *impose* leur usage: les détails d'implémentation n'ont pas à être connus par l'utilisateur.

On veut donc offrir une interface et protéger certaines données.

Les classes permettent de

- regrouper des données
- associer des fonctions aux objets de la classe
- restreindre l'accès à certaines données

Fichier d'étudiants

On veut construire des bases de données pour stocker les fiches d'étudiants de chaque promo.

Pour chaque promo, on veut disposer des fonctions suivantes:

- saisir les noms, prénoms et numéros des étudiants : (initialisation de la base)
- saisir des notes pour un étudiant
- afficher les notes d'un étudiant
- afficher la moyenne d'un étudiant
- afficher la moyenne pour un cours
- afficher la moyenne générale

Fichier d'étudiants

On veut construire des bases de données pour stocker les fiches d'étudiants de chaque promo.

Pour chaque promo, on veut disposer des fonctions suivantes:

- saisir les noms, prénoms et numéros des étudiants : (initialisation de la base)
- saisir des notes pour un étudiant
- afficher les notes d'un étudiant
- afficher la moyenne d'un étudiant
- afficher la moyenne pour un cours
- afficher la moyenne générale

Et c'est tout !

Fichier d'étudiants

Chaque promo doit contenir les données suivantes:

- le nombre d'étudiants
- le nombre de cours
- la liste des cours
- la fiche de chaque étudiant

Fichier d'étudiants

On intègre donc les données suivantes dans une promo:

```
class Promo {
 private:
  int Nbc;
  int Nbe;
  vector<string> Cours;
  vector<FicheEtudiant> Etudiants;

...

};
```

Comment définir les fonctions associées

On va utiliser une autre approche que celle présentée dans la partie A.

- On va protéger certains aspects du type `Promo`: la structure interne, le nom des champs...
- On va définir des fonctions particulières (fonctions membre, méthodes) qui seules pourront accéder à la partie cachée du type Promo.

Ces méthodes seront l'interface de la classe.
On les définit dans la classe.

Méthode - fonction membre

Ces fonctions s'appliquent toujours à un objet de la classe, plus éventuellement à d'autres paramètres.

Cet objet joue un rôle particulier: on dit que l'on appelle la méthode sur lui.

Le mécanisme d'appel de ces fonctions tient compte de cette différence et sépare cet objet des autres paramètres:

Plutôt que de faire `Test(o,p1,...,pk)`, on fait:

```
o.Test(p1,...,pk)
```

On appelle `Test` comme un champ...
Vision: `o` reçoit le "message" ou l'"ordre" `Test`.

Méthode - fonction membre

- Les méthodes sont déclarées dans la classe.
- Le nom complet de la méthode `Test` de la classe `Toto` est `Toto::Test`
- Les méthodes peuvent accéder aux champs des objets de la classe: ce sont les seules à pouvoir le faire (ou presque) !
- L'objet sur lequel elle sont appelées, n'apparaît explicitement dans la définition des méthodes.
- Par défaut, les champs apparaissant dans la définition d'une méthode sont ceux de l'objet sur lequel la fonction sera appelée.

Initialisation

```
class Promo {
 private:
  int Nbc;
  int Nbe;
  vector<string> Cours;
  vector<FicheEtudiant> Etudiants;

 public:
  void Initialisation();
  void SaisieNotes(int n);
  void AfficheNotesE(int n);
  double MoyenneE(int n);
  double MoyenneC(int c);
  double MoyenneG();
};
```

Initialisation

```
void Promo::Initialisation()
{
  cout << "Nb d'etudiants: ";     cin >> Nbe;
  cout << "Nb de cours: ";     cin >> Nbc;
  Cours = vector<string>(Nbc);
  Etudiants = vector<FicheEtudiant>(Nbe);

  for(int i=0;i<Nbc;i++){
    cout << "Cours "<< i << " : ";
    cin >> Cours[i];}

  for(int i=0;i<Nbe;i++){
    cout << "Nom: ";        cin >> Etudiants[i].Nom;
    cout << "Prenom : ";      cin >> Etudiants[i].Prenom;
    cout << "Numero : ";      cin >> Etudiants[i].Numero;

  Etudiants[i].Notes = vector<double>(Nbc);
  }
}
```

Constructeurs

- Méthodes particulières appelées automatiquement à chaque définition d'un objet de la classe.
- Pour passer des arguments à un constructeur, il suffit de faire suivre le nom de la variable par la liste des arguments :
  `MaClasse A(i,f) ;`
  Lorsqu'aucun argument n'est donné, le constructeur sans argument est appelé.
- Si aucun constructeur n'est défini explicitement, le compilateur crée un constructeur sans argument `MaClasse::MaClasse()` qui se contente d'appeler les constructeurs (sans argument) de chaque champ de l'objet.

Constructeur copieur

`MaClasse (const MaClasse &)`
: utilisé pour le passage des paramètres et les initialisations.

- Si une fonction a un paramètre `A` du type `MaClasse`, un appel de cette fonction avec une variable `V` de `MaClasse` conduit à l'*initialisation* de `A` avec `V` via le constructeur copieur.
- Lorsqu'on définit une variable `V` et qu'on l'initialise à une valeur `E`, par l'instruction "`MaClasse V=E ;`", on utilise le constructeur copieur.
  Différence avec : "`MaClasse V ;  V=E ;`"

Le constructeur copieur créé par défaut fait une initialisation champ à champ avec les champs de l'argument.

Constructeur / destructeur

Il existe aussi des destructeurs : on peut définir un (et un seul) destructeur qui est appelé automatiquement lorsqu'une variable locale est détruite
(surtout utiles pour les structures de données dynamiques)

Pour comprendre tous ces mécanismes d'appel...
il faut les essayer !!

cf. Constructeurs.cc

Fonctions amies

Si une fonction F est "amie" (friend) d'une classe C1, alors F peut accéder aux champs privés de C1.

Si une classe C2 comme est "amie" de C1, toutes les fonctions membres de C2 peuvent accéder aux champs privés de C1.

Ces déclarations se font dans la définition de C1 :

```
class C1 {
   ...
  friend type-de-F F(param-de-F);
  friend class C2;
   ...
};
```

Conversions entre types

Comme pour les types simples, on peut définir des conversions entre les classes.

- Un constructeur de `T1` qui prend un unique paramètre de type `T2` définit implicitement une conversion de `T2` dans `T1`.
  (pas de composition: ne marche que pour des conversions en une étape)
- Une fonction membre `operator T1()` dans la classe `T2` définit aussi une conversion de `T2` dans `T1`.
  (évite de modifier la définition de `T1`)

Par exemple :

```
classfraction{
public:
    ...
    fraction(int v)     {num=v;den=1;}
    double(){returndouble(num)/den;}
};
```

Plan

Surcharge d'opérateurs

Il est possible de surcharger les opérateurs de C++ (+, -, [ ], =, ==, . . .): les arités et les priorités ne changent pas.

Deux points de vue:

- fonction globale: "A op B" est vue comme l'application d'une fonction "op" à deux arguments "A" de type TA et "B" de type TB.

```
fraction operator*(fraction f, fraction g)
{return fraction(f.Numerateur()*g.Numerateur(),
                 f.Denominateur()*g.Denominateur());}

fraction operator-(fraction f)
{return fraction(-f.Numerateur(),
                 f.Denominateur());}
```

Surcharge d'opérateurs

- fonction membre: "A op B" est vue comme l'application d'une fonction "op" à un argument "B" de type TB sur l'objet "A" de type TA.

```
class fraction {
private :
  int num,den;
public :
  ...
  fraction operator -();
  fraction operator*(fractionh);
};

fraction fraction::operator-()
  {returnfraction(-num,den );}

  fraction fraction::operator *(fraction h)
    {return fraction(num*h.num,den*h.den);}
```

Surcharge de cout ≪

L'opérateur ≪ ne peut être surchargé que de manière globale.
Profil pour un type T:

ostream & ≪ (ostream &, T)

Exemple:

```
//-----------------------------------------------------------
ostream & operator<<(ostream &os,fractionf)
{
os <<f.Numerateur() <<"/" << f.Denominateur();
return os;
}
//-----------------------------------------------------------
```

Quatrième partie IV

Structures de données dynamiques

Plan

Pointeurs

Une variable permet de désigner – par un nom – un emplacement dans la mémoire.

Il est aussi possible de désigner directement les cases mémoire par leur adresse.

Un pointeur permet de stocker ces adresses mémoire.

Pointeurs, *, &

- Un pointeur est typé : on distingue les pointeurs sur un `int`, sur un `double`, sur une `fraction` etc.
- Définir un pointeur:
  type * *nom* ;
  nom est un pointeur sur *type*, i.e. pouvant recevoir une adresse désignant un objet de type *type*.
- `&V` représente l'adresse mémoire d'une variable `V`.
- `*P` désigne l'objet pointé par le pointeur `P`.

```
int i, * pi, k;
i=3;
pi = &i;          // pi pointe sur i
k = (*pi)+5;      // identique à :   k=i+5
(*pi) = k+2;      // identique à :  i = k+2
(*pi)++;
// ici, i=11, k=8
```

Pointeurs et références

Au lieu du passage par référence, il est possible de passer (par valeur) les adresses des arguments d'une fonction et de modifier les données contenues à ces adresses via l'opérateur *.

Fonction permut :

```
void permut(int* pa, int* pb)
{int aux = (*pa);
(*pa) = (*pb);
(*pb) = aux;}
```

Pour échanger les valeurs de i et j, on écrit permut(&i,&j); (seule solution en C)

Plan

Tableaux à la "C"

type *nom*`[`*taille*`];`
→ définit un tableau d'éléments de type *type* et de taille *taille* (indices de 0 à *taille* — 1). L'élément d'indice *i* est désigné par *nom*`[`*i*`]`.

Dans "`int Tab[10]`", `Tab` désigne l'adresse de la première case du tableau (i.e. `&Tab[0]`) et le type de `Tab` est "`const int *`".

Qu'est-ce que `Tab[i]` ?

Tableaux à la "C"

type *nom*[*taille*] ;
→ définit un tableau d'éléments de type *type* et de taille *taille* (indices de 0 à *taille* — 1). L'élément d'indice *i* est désigné par *nom*[*i*].

Dans "`int Tab[10]`", `Tab` désigne l'adresse de la première case du tableau (i.e. `&Tab[0]`) et le type de `Tab` est "`const int *`".

Qu'est-ce que `Tab[i]` ? c'est `(*(Tab+i))`.

```
int T[10];
int* p;
p = &T[4];
p[0] = 100;  //  équivaut à T[4] = 100
p[1] = 200;  //  équivaut à T[5] = 100
```

Pointeurs et tableaux

- Passer un tableau "C" en paramètre d'une fonction revient à passer l'adresse de la première case ⇒ passage par référence.

  Pour déclarer un paramètre `T` de type tableau d'entiers, on écrit: `int T[]`.

  La taille doit être passée avec un autre paramètre.
  `void Tri(int T[],int taille)`
- autres chaînes de caractères : tableaux (simples) de caractères se terminant par le caractère spécial `'¥0'`.
  `char T[8] = "exemple";`
  définit et initialise le tableau `T` avec `T[0]='e'`, `T[1]='x'`, ..., `T[7]='¥0'`.

Plan

new et delete

obj : allouer et désallouer de la mémoire sans dépendre la durée de vie des variables.

- "`new` *type* ;" alloue un espace mémoire pour contenir un objet de type *type* et retourne l'adresse ce cette zone mémoire.
- "*P* = `new` *type*`[`*N*`]` ;" alloue un espace mémoire capable de contenir *N* objets de type *type* ...
- "`delete` *P* ;" désalloue l'espace mémoire d'adresse *P*. (ou "`delete [ ]` *P* ;")

Une zone allouée par `new` n'est désallouée que par l'appel de `delete`...

Un peu de cinéma ...

Plan

Structures de données dynamiques

Une structure de données dynamique évolue au cours du temps.

Exemple de la pile: utiliser un tableau oblige à borner la taille de la pile.

On préfère allouer de l'espace mémoire en fonction des besoins :

- ajouter un élément à la pile correspondra à allouer (avec `new`) de la mémoire, et
- supprimer reviendra à désallouer (avec `delete`) la case contenant l'objet.

⇒ Le contenu de la pile sera donc éclaté dans la mémoire
Pour ne pas perdre d'éléments, on indique à chaque élément où se trouve le prochain élément.
Cela correspond à la notion de *liste chaînée*

Listes chaînées

Un maillon d'une liste chaînée:

```
classelement {
public :
  intval;
  element * suivant;
};
```

Le champ suivant contient l'adresse du prochain élément de la liste (ou 0).

On utilise l'adresse du premier élément de la liste...

Parcours de liste, inversion, longueur. . .
Listes doublement chaînées, circulaires, . . .

Plan

Classes et pointeurs

- `p->champ` raccourci pour `(*p).champ`
- `p->fct()` raccourci pour `(*p).fct()`
- Lorsqu'on alloue de la mémoire pour stocker un objet de type `T`, un constructeur est appelé :
  - soit celui sans argument (dans le cas d'un "`new T`"),
  - soit un avec argument si des arguments sont présents ("`new T(a1,...,aN);`").
- Dans la définition des fonctions membre, on peut accéder à l'adresse de l'objet courant: pointeur `this`.

Cinquième partie V

Compléments

Compléments

- Héritage
- Programmation orientée objet
- Généricité (classes, fonctions)
- GDB
- Exceptions
- ...