Mise à niveau
Langage C

Licence Informatique

3ème année

Christophe Després

Maître de Conférences

Deuxième partie

Les types fondamentaux du C

Types fondamentaux du C

- ***Les entiers***
  - *Les caractères* — ***char***
  - *Les entiers courts* — ***short int***
  - *Les entiers long* — ***long int***
  - *Les entiers classiques* — ***int***
- ***Les réels***
  - *Les réels simple précision* — ***float***
  - *Les réels double précision* — ***double***
  - *Les réels quadruple précision* — ***long double***

Les caractères

- *En C un caractère est un entier car il s'identifie à son code ASCII (le plus souvent).*
- *Un caractère peut donc être codé sur un octet.*
- *On peut donc appliquer toutes les opérations entières sur les caractères (addition, soustraction, etc.).*

```
c = 'a'  =>  c = 97
     c = c + 1
c = 'b'  =>  c = 98
```

Les entiers

- ***Un entier correspond généralement à un mot machine***
- ***Les attribut short, long, unsigned peuvent qualifier un entier***
  - *long int x; /* x est un entier long (>=32 bits) */*
  - *short int x; /* x est un entier court */*
  - *unsigned int x; /* x est un entier non signé */*
  - *unsigned short int x; /* x est un entier court non signé */*

Les entiers

- ***Il n'y a pas de taille fixée par le langage, mais le C garanti que :***
  - *1 = T_C <= T_S <= T_I <= T_L*
- ***Lorsque l'on utilise les attributs short, long ou unsigned, on peut omettre le nom du type int***
  - *long x;*
  - *short x;*
  - *unsigned x;*
  - *unsigned short x;*

Les réels

- *Les float sont des réels codés de manière interne sous forme de mantisse/exposant*
- *Les double sont des float plus long et les long double sont des doubles plus long*
- *Plus la taille est grande plus la mantisse est grande et plus la précision est grande*
- *Comme pour les entiers ces tailles sont dépendantes de la machine*

Les réels

- ***Le C garanti juste que***
  - *T_F <= T_D <= T_LD*
- ***Tailles minimales***
  - *char 1 octet*
  - *short int2 octets*
  - *int 2 octets*
  - *long int 4 octets*
  - *float 4 octets*
  - *double 8 octets*

Les constantes

- *Constantes de type entier*
- *Constantes de type char*
- *Constantes de type logique*
- *Constantes de type réel*
- *Constantes de type chaîne de caractères*

Constantes de type entier

- ***On dispose de 3 notations pour les constantes entières***
  - *décimale, qui ne commence pas par un 0, ex : 234*
  - *octale (base 8), qui commence par un 0 suivi d'un chiffre, ex : 0234 (=156 en base 10)*
  - *héxadécimale (base 16), qui commence par un 0 suivi d'un x ou d'un X, ex : 0xC (=12 en base 10)*

Constantes de type entier

- *Une constante numérique en base 10 est normalement un int. Si le nombre est trop grand le compilateur essaiera dans l'ordre les types long int, unsigned long int*
- *Une constante numérique en base 8 ou 16 est normalement un int. Si le nombre est trop grand le compilateur essaiera dans l'ordre les types unsigned int, long int, unsigned long int*
- *Les constantes de type short int et unsigned short int n'existent pas*

Constantes de type char

- *Elles n'existent pas réellement en C*
- *Une constante caractère est de type int et a pour valeur le code du caractère dans le codage utilisé par la machine*
- *Une constante caractère s'écrit entourée du signe '*
- *La constante caractère correspondant au caractère a s'écrit 'a'*
- *+ s'écrit '+'*
- *1 s'écrit '1'*

Constantes de type char

- ***Caractères non-imprimable***

sémantique	caractère
newline	`'\n'`
horizontal tabulation	`'\t'`
vertical tabulation	`'\v'`
back space	`'\b'`
carriage return	`'\r'`
audible alert	`'\a'`

Constantes de type char

- ***Caractères spéciaux***

sémantique	caractère
'	'\''
"	'\"'
\	'\\'

Constantes de type logique

- ***Les constante de type logique n'existent pas en tant que telle en C. On utilise la convention suivante sur les entiers***
  - *0 <=> faux et tout le reste est vrai*

Constantes de type réel

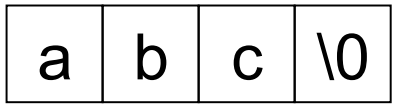
- ***Les constantes réelles sont de la forme <partie entière>.<partie fractionnaire><e ou E><partie exposant>***
- ***On peut omettre***
  - *soit <partie entière>* *: .475e-12*
  - *soit <partie fractionnaire>* *: 482.*
  - *Mais pas les deux*
- ***On peut omettre***
  - *soit .* *: 12e7*
  - *soit <partie exposant>* *: 12.489*
  - *mais pas les deux*

Constantes de type réel

- *Une constante réelle non suffixée a le type double*
- *Une constante réelle suffixée par f ou F a le type float*
- *Une constante suffixée par l ou L a le type long double*

Constantes de type chaîne de caractères

- *Elles sont constituées de caractères quelconques encadrés par des guillemets, ex : "abc"*
- *Elle sont stockées en mémoire statique sous la forme d'un tableau de caractères terminé par la valeur '\0' (valeur entière 0)*



Les constantes nommées

- ***Il y a trois façons de donner un nom à une constante : soit en utilisant les possibilités du préprocesseur, soit en utilisant des énumérations, soit avec le mot-clé const***
  - *#define*
  - *enum*
  - *const (depuis la norme ANSI)*

#define

- ***#define identificateur reste-de-la-ligne***
- ***Le préprocesseur lit cette ligne et remplace dans toute la suite du source, toute nouvelle occurrence de identificateur par reste-de-la-ligne***
- ***#define PI 3.14159***
  - *et dans la suite du programme on pourra utiliser le nom PI pour désigner la constante 3.14159*
- ***Il s'agit d'une transformation d'ordre purement textuel***

#define

- ***Une telle définition de constante n'est pas une déclaration mais une commande du préprocesseur. Il n'y a donc pas de ; à la fin***
- ***Si on écrit :***

```
#define PI 3.14159;
```

  - *le préprocesseur remplacera toute utilisation de PI par 3.14159; et par exemple remplacera l'expression PI / 2 par 3.14159; / 2 ce qui est une expression incorrecte.*
  - *Dans une telle situation, le message d'erreur ne sera pas émis sur la ligne fautive (le #define), mais sur une ligne correcte (celle qui contient l'expression PI / 2), ce qui gênera la détection de l'erreur.*

Les énumérations

- ***On peut définir des constantes de la manière suivante :***

  enum { liste-d'identificateurs }

- ***enum {LUNDI, MARDI, MERCREDI, JEUDI};***
  - *définit les identificateurs LUNDI,…, JEUDI comme étant des constantes de type int, et leur donne les valeur 0,1, 2, 3.*
- ***enum {FRANCE = 10, ESPAGNE = 20};***
- ***enum {FRANCE = 10, ITALIE, ESPAGNE = 20};***
  - *ITALIE à la valeur 11*
- ***enum {ESPAGNE = 'E', FRANCE}***

Déclaration des variables

- ***Les noms de variables (et de constantes) sont constitués de lettres et de chiffres***
  - *le premier caractère doit être une lettre*
  - *le caractère souligné noté "_" est considéré comme une lettre*
  - *les caractères majuscules et minuscules sont différents, en langage C on utilise généralement les majuscules pour les constantes et les minuscules pour les variables*
- ***Une déclaration indique un certain type et regroupe derrière une ou plusieurs variables***
  - *<Type> <NomVar1>, <NomVar2>, ...,<NomVarN>;*
- ***Les variables doivent être déclarées avant d'être utilisées***

Déclaration des variables

- ***Exemple 1 :***
  - *int compteur,X,Y;*
  - *float hauteur,largeur;*
  - *double masse_atomique;*
- ***Exemple 2 :***
  - *int compteur*
  - *int X;*
  - *int Y;*
  - *float hauteur;*
  - *float largeur;*
  - *double masse_atomique;*

Initialisation des variables

- ***L'initialisation des variables peut se faire au moment de la déclaration***
- ***Exemples :***
  - *int max = 1023;*
  - *char tab = '\t';*
  - *float x = 1.05e-4;*

Troisième partie

Expressions & opérateurs

Expressions

- *Une expression est une notation de valeur*
- *L'évaluation d'une expression est le processus par lequel le programme obtient la valeur désignée.*
- *Les constantes et les variables sont des expressions (elles retournent une valeur)*
- *Une expression (complexe) est constitué d'opérandes et d'opérateurs*
- *Les opérandes dénotent les valeurs à composer. Ce sont elles-mêmes des expressions (qui peuvent être des constantes ou des variables)*

Les opérateurs

- *Opérateur d'incrémentation et de décrémentation*
- *Opérateurs arithmétiques*
- *Opérateurs relationnels*
- *Opérateurs logiques*
- *Opérateurs binaires*
- *Opérateur d'affectation*
- *Opérateur conditionnel*
- *Opérateur de cast*

Opérateur d'in(dé)crémentation

- ***L'opérateur ++ ajoute 1 à son opérande***
- ***L'opérateur -- retranche 1 à son opérande***
- ***Lorsque l'opérateur est placé devant l'opérande, l'in(dé)crémentation a lieu avant l'utilisation de la valeur de l'opérande***
- ***Exemples :***
  - *n = 5;*
    x = ++n;
    => x vaut 6 et n vaut 6;
  - *n = 5;*
    x = n--;
    => x vaut 5 et n vaut 4;

Les opérateurs arithmétiques

- \+ *Addition*
- \- *Soustraction (ou opérateur unaire)*
- * *Multiplication*
- / *Division entière et réelle*
- % *reste de la division entière (Modulo)*

- ***Ces opérateurs (sauf le %) sont applicables aussi bien à des entiers qu'à des réels***
- ***Dans le cas de 2 opérandes de type entier, le résultat de la division est entier, dans tous les autres cas, il est réel***

Opérateurs relationnels

- `==` *égalité (<> de l'affectation)*
- `!=` *différence*
- `>` *supérieur*
- `>=` *supérieur ou égal*
- `<` *inférieur*
- `<=` *inférieur ou égal*

Opérateurs relationnels

- *Les deux opérandes doivent avoir le même type arithmétique. Si ce n'est pas le cas, des conversions sont effectuées automatiquement*
- *Le type BOOLEEN n'existe pas explicitement en C : les opérateurs de relation fournissent les valeurs 0 ou 1 (respectivement FAUX et VRAI) du type int*
- *En C, on peut écrire A < B < C car cette expression correspond à (A < B) < C ce qui n'est probablement pas le résultat escompté par le programmeur. En effet si A < B est vrai, l'expression équivaut à 1 < C et sinon à 0 < C*

Opérateurs logiques

- *! Négation unaire d'une valeur logique*
- *&& ET de 2 valeurs logiques*
- *|| OU de 2 valeurs logiques*
- *Ces opérateurs interviennent sur des valeurs de type int (0 => FAUX, toutes les autres => VRAI)*
- *Les valeurs produites sont 0 (FAUX) ou 1 (VRAI)*
- *L'opérande gauche est évalué avant celui de droite pour les opérateurs && et ||*

Opérateurs logiques

- ***L'opérande de droite peut ne pas être évalué si celui de gauche suffit à déterminer le résultat***
  - *0 à gauche d'un && implique FAUX*
  - *1 à gauche d'un || implique VRAI*
- ***Exemple :***

  ```
  int tab[10];
  soit le test : (k<10) && (tab[k]!=v)
  ```

- ***Si k est supérieur ou égal à 10, l'expression tab[k]!=v ne sera pas évaluée et il vaut mieux car pour k supérieur ou égal à 10 tab[k] est indéterminée***

Opérateurs binaires

- ***Ces opérateurs sont au ras de la machine, ils servent à manipuler des mots bit à bit pour faire des masques par exemple.***
- ***a & b : et binaire => mettre des bits à 0***
  - *c = n & 127 : c sera constitué des 7 bits de poids faible de n et complété à gauche par des 0*
- ***a \|b : ou binaire => mettre des bits à 1***
  - *c = n | 127 : c sera constitué de 7 bits de poids faible à 1 et des bits de poids fort de n*
- ***a ^ b : ou exclusif binaire => inverser des bits***
  - *c = n ^ 127 : c sera constitué des bits de n avec les 7 bits de poids faible inversés*

Opérateurs binaires

- **`a << b  : décalage à gauche`**
  - *Les bits de a sont décalés de b positions vers la gauche, les bits de poids fort sont perdus, des 0 arrivent sur la droite*
- **`a >> b  : décalage à droite`**
  - *Les bits de a sont décalés de b positions vers la droite, les bits de poids faible sont perdus, des bits X arrivent sur la gauche*
    - Si a est non signé ou signé positif : X = 0
    - Sinon (a négatif) : pas de norme
- **`~a  : complément à 1`**
  - *les bits de a sont inversés*

Opérateur d'affectation

- ***= est le symbole d'affectation :***
  - *lvalue = expression*
- ***L'affectation est une expression => elle renvoie une valeur égale à la valeur de l'objet affecté***
- ***Exemples :***
  - *i = 1;* => *affecte la valeur 1 à la variable i et renvoie 1*
  - *j = i + 1;* => *affecte la valeur 2 à la variable j et renvoie 2*
  - *while ((c = getchar()) != EOF)*

Opérateur d'affectation

- ***On peut combiner l'affectation avec l'un des dix opérateurs suivants : + - * / % << >> & | ^***
- ***Si op est l'un de ces dix opérateur alors :***

  expression1 op= expression2

  est équivalent à

  expression1 = expression1 op expression2

- ***Exemples :***
  - *i += 3;* *est équivalent à* *i = i + 3;*
  - *i /= j;* *est équivalent à* *i = i / j;*

Opérateur conditionnel

- *e1 ? e2:e3 est une expression qui vaut e2 si e1 est vrai et e3 sinon*
- *Exemple :*

```
taille<1.80 ? "petit" : "grand"

printf("Il est %s", taille<1.80 ? "petit" : "grand");
```

Opérateur de cast

- *(t) a*
- *Le cast sert à forcer le type d'un objet, c'est à dire convertir un objet d'un type vers un autre*
- *Exemples :*

```
(long int) x*y/(1.+sin(x))
int carre(float x) {return (int) x*x;}
```

Priorité et ordre d'évaluation

Opérateur	Associativité
() [] -> .	de gauche à droite
! ~ ++ -- - (t) * & sizeof	de droite à gauche
* / %	de gauche à droite
+ -	de gauche à droite
<< >>	de gauche à droite
< <= > >=	de gauche à droite
== !=	de gauche à droite
&	de gauche à droite
^	de gauche à droite
|	de gauche à droite
&&	de gauche à droite
||	de gauche à droite
?:	de droite à gauche
= += -= etc.	de droite à gauche

Priorité et ordre d'évaluation

- ***Certains choix de priorité sont plutôt mauvais (les concepteurs du langage en conviennent)***
- ***La précédence des opérateurs bits à bits (c-a-d les opérateurs binaires : &,^ et |) est inférieure à celle des opérateurs de comparaison (comme == et !=)***
  - *((x & MASK) == 0)* *différent de : (x & MASK == 0)*
    correspond à : x & (MASK == 0)
- ***La précédence des opérateurs de décalage est plus petite que celle des opérateurs de + et -***
  - *a << 4 + b* *différent de : (a << 4) + b*
    correspond à : a << (4 + b)

Priorité et ordre d'évaluation

- ***Le langage C ne précise pas dans quel ordre sont évalués les opérandes d'un opérateur***
  - *x = f() + g();*
- ***L'ordre dans lequel sont évalués les arguments d'une fonction n'est pas précisé***
  - *printf("%d %d", i++, i);*

Priorité et ordre d'évaluation

- *On en déduit que l'écriture des instructions dont le résultat dépend de l'ordre de l'évaluation est un exemple de mauvaise programmation*

Conversions numériques

- *Quand des opérandes de types différents interviennent dans une même expression ils sont tous convertis en un seul type*
- *En général les seules conversions qui se font automatiquement sont celles qui donnent un sens aux expressions comme la conversion d'un entier en réel dans une expression du genre : f + i*
- *Les expressions qui n'ont aucun sens telles qu'utiliser un nombre du type float comme indice sont rejetées à la compilation*

Conversions numériques

- ***Le langage C prévoit deux mécanismes de conversion de type :***
  - *la conversion implicite*
  - *la conversion explicite (utilisation de l ’opérateur de casting)*

Conversion de type

- *Elle concerne les variables de types numériques : int, long, float, double et leurs dérivés non signés*
- *Il est toujours possible d 'attribuer une variable de l 'un de ces types à une variable d 'un autre type*
- *S 'il n 'y a pas de perte d 'information, la conversion est implicite, sinon elle est explicite (utilisation de l 'opérateur de casting)*
- *L 'opérateur de cast peut être utilisé dans le cas d 'une conversion implicite*

Quatrième partie

Les principales instructions

Les instructions et les blocs

- *Une expression de la forme x=0 ou i++ ou printf(…) devient une instruction quand elle est suivie d'un ;*
- *Le ; fait partie de l'instruction, c'est un terminateur et non un séparateur comme en Pascal*
- *Une instruction composée ou bloc est une suite d'instructions encadrée par { et }*

if

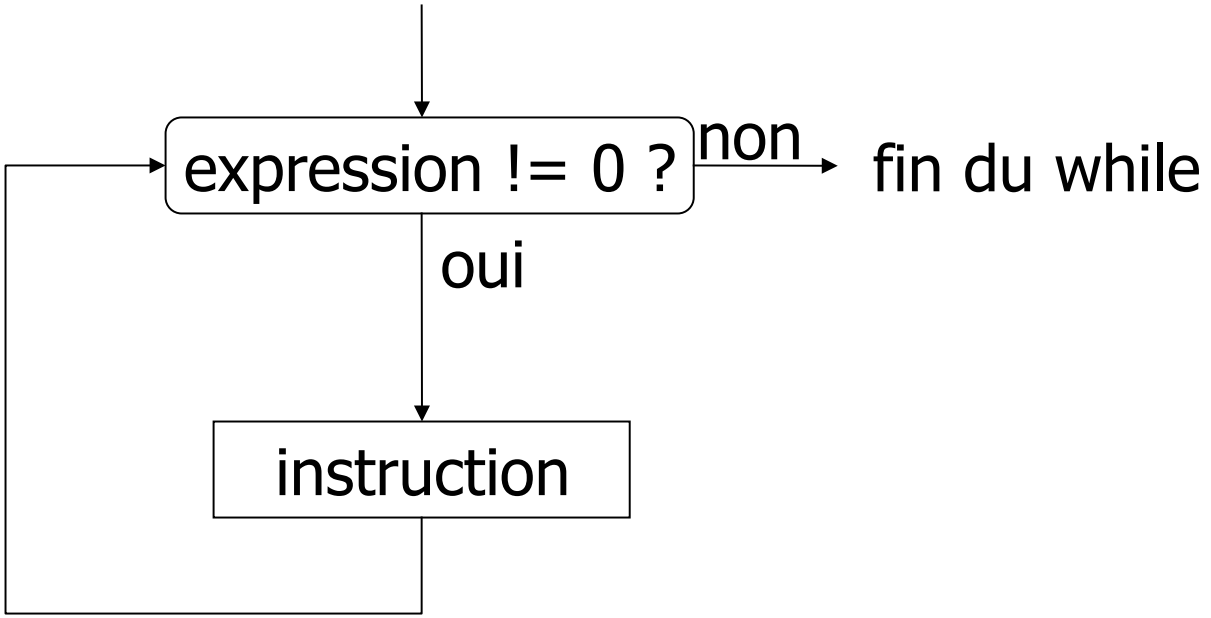
- ***Syntaxe :***
  - *if (expression) instruction*
- ***Si expression est vraie (<>0) instruction est exécutée***
- ***Attention au piège de l'affectation***

if ... else

- ***Syntaxe :***
  - *if (expression) instruction1 else instruction2*
- ***Attention à la portée du else***

while

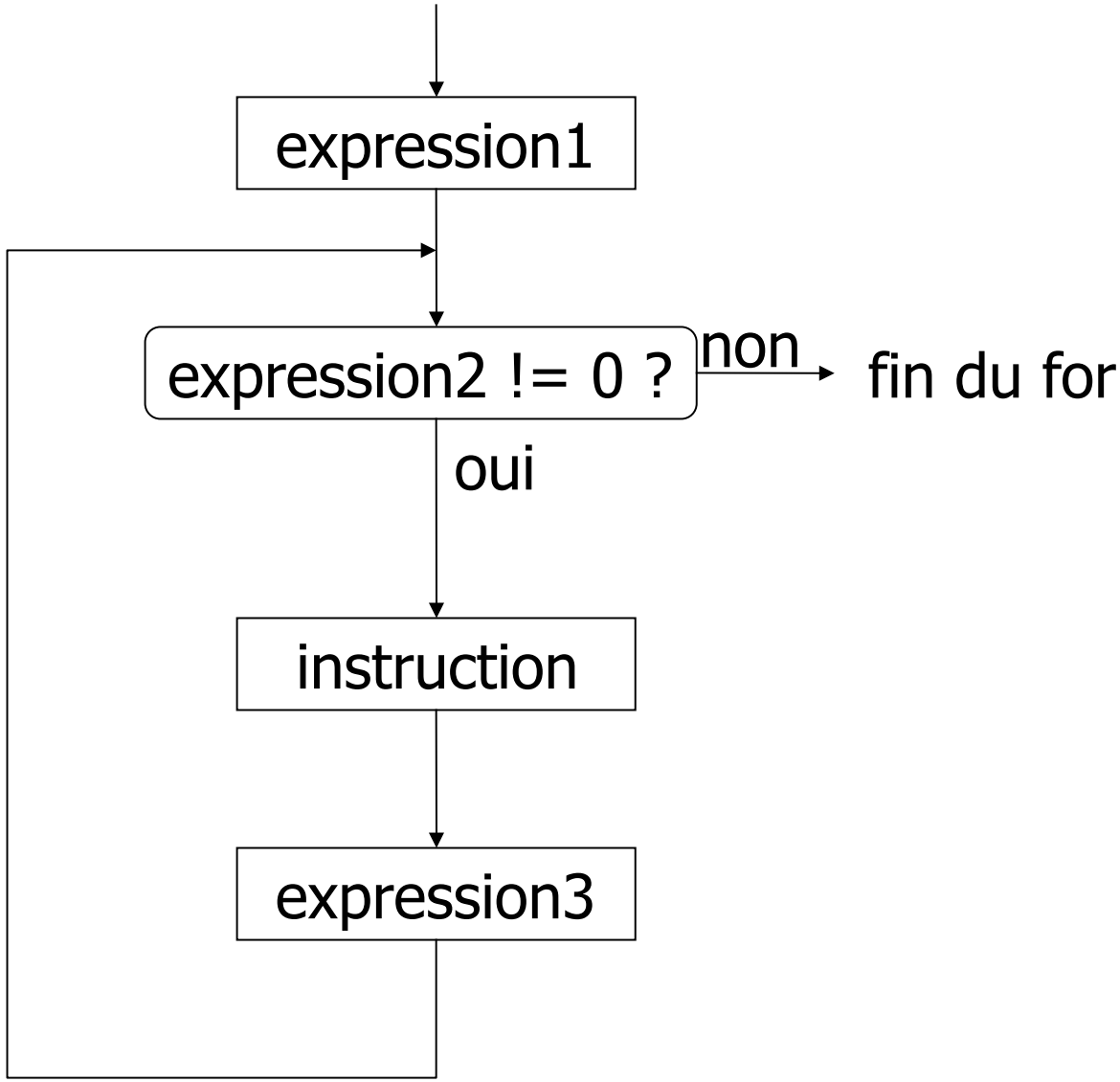
- ***Syntaxe :***
  - *while (expression) instruction*



for

- ***Syntaxe :***
  - *for (expression1; expression2; expression3)*
    instruction
  - *instruction est une instruction simple ou composée*
  - *expression1 sert à initialiser*
  - *expression2 est la condition de rebouclage*
  - *expression3 est l'expression d'incrémentation*

for


expression1
expression2 != 0 ?
non
fin du for
oui
instruction
expression3

for

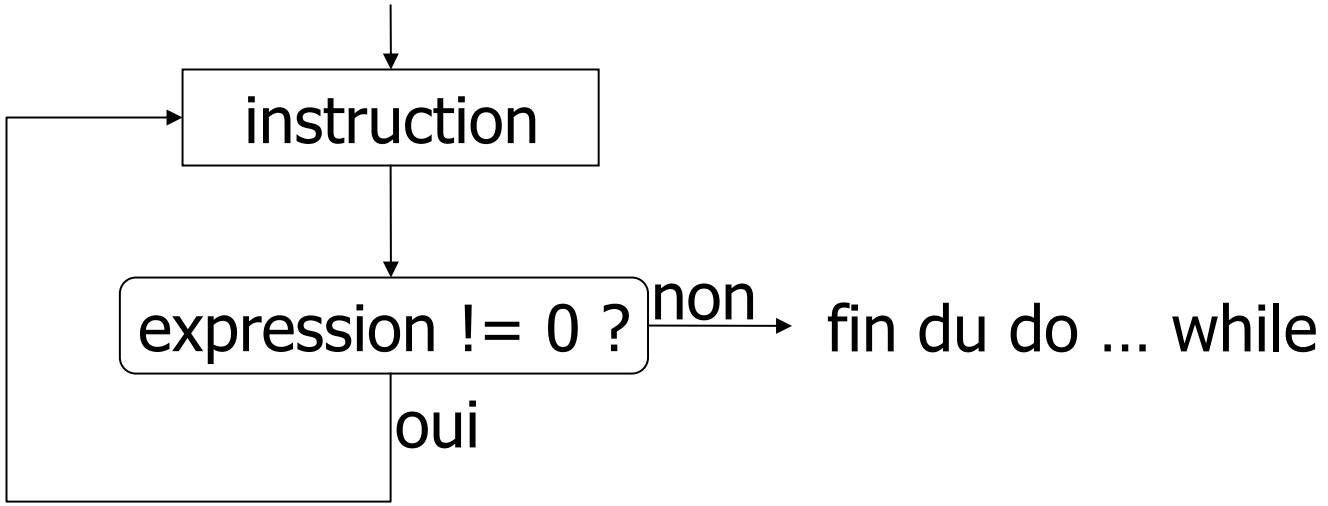
- ***Le for est un tantque traditionnel des autres langages.***
- ***le for peut dans la plupart des cas être réécrit de la façon suivante :***
  - *expression1*
    while (expression2)
    {
    instruction
    expression3;
    }

for

- ***Les expressions peuvent comporter plusieurs instructions***
  - *exemple*
- ***Rien n'oblige en C la présence des trois expressions :***
  - *for(;;) est valide et équivalent à while (1)*

do ... while

- ***Syntaxe :***
  - *do instruction while (expression);*
- ***do … while est équivalent à :***
  - *instruction*
    while (expression) instruction



break

- *Cette instruction provoque la fin de l'instruction switch, while, do ou for qui la contient (au premier niveau)*
- *Elle est à utiliser avec précaution et devra toujours être justifiée par des commentaires*

continue

- *Cette instruction a pour but de provoquer le rebouclage immédiat de la boucle do, while, ou for qui la contient*
- *Dans le cas d'un for(e1;e2;e3), e3 est évaluée avant le rebouclage. C'est pour cette raison que l'équivalence entre le for et le while n'est pas totale.*

return

- ***`return;`***
  - *L'exécution de la fonction qui contient le return est interrompu, le contrôle est rendu à la fonction appelante.*
- ***`return expression;`***
  - *Idem avec une valeur de retour égale à l'évaluation de l'expression.*

goto étiquette;

- ***NE PAS UTILISER SOUS PEINE DE MORT !!!***

switch

- ***Syntaxe :***
  - *switch (expression)*
    {
      case c1 : instructions
      …
      default : instructions
    }
- ***L'expression et les différentes constantes (ci) doivent être de type entiers, ou entiers définis par énumération.***

switch

- ***Cette instruction est différente du case Pascal, car les valeurs de la constante sont vue comme des étiquettes de point d'entrée :***

```
switch (i)
{
   case 1 : a=s[j];
   case 2 : b++;
   case 3 : a=s[j-1];
}
```

switch

- ***Utilisation de l'instuction break :***
  - *switch (i)*
    {
      case 1 : a=s[j];
        break;
      case 2 : b++;
        break;
      case 3 : a=s[j-1];
    }
- ***Il n'y a pas de possibilité de donner d'énumération de valeurs ou d'intervalles***

Cinquième partie

Les fonctions

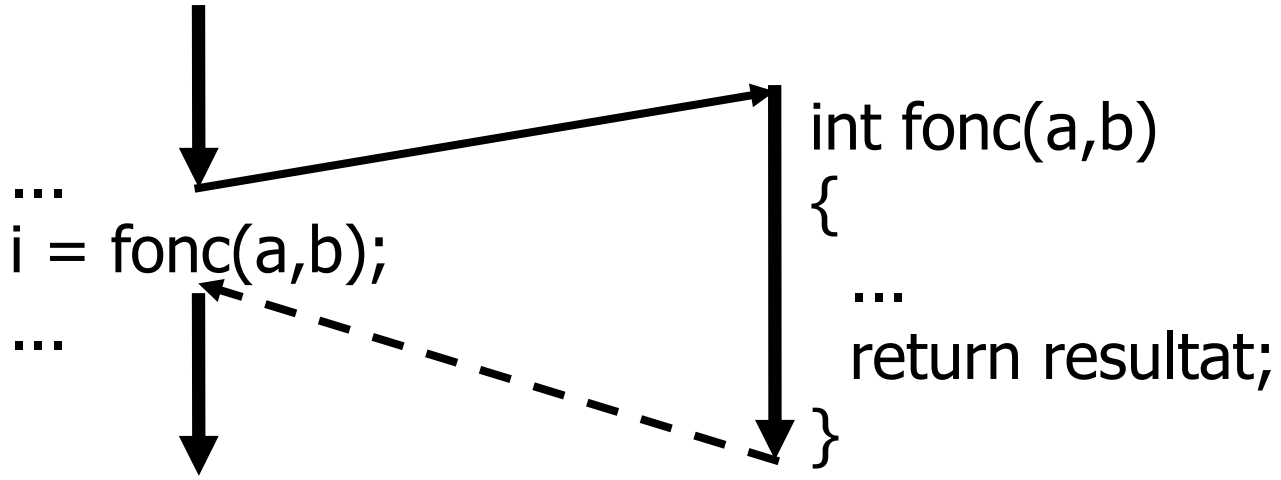
Présentation générale

- *Une fonction est une portion de programme formant un tout homogène, destiné à remplir une certaine tâche bien délimitée*
- *Lorsqu'un programme contient plusieurs fonctions, l'ordre dans lequel elles sont écrites est indifférent, mais elles doivent être indépendantes.*

Présentation générale

- *En règle générale, une fonction appelée a pour rôle de traiter les informations qui lui sont passées depuis le point d'appel et de retourner une valeur.*
- *La transmission de ces informations se fait au moyen d'identificateurs spécifiques appelés arguments et la remontée du résultat par l'instruction return.*

Présentation générale



- ***Certaines fonctions reçoivent des informations mais ne retournent rien (par exemple printf), certaines autres peuvent renseigner un ensemble de valeurs (par exemple scanf).***

Définition de fonction

- ***La définition d 'une fonction repose sur trois éléments :***
  - *son type de retour*
  - *la déclaration de ses arguments formels*
  - *son corps*
- ***De façon générale, la définition d'une fonction commence donc par :***
  - *type_retour nom (type_arg1 arg1, type_arg2 arg2 …)*

Définition de fonction

- ***L'information retournée par une fonction au programme appelant est transmise au moyen de l'instruction return, dont le rôle est également de rendre le contrôle de l'exécution du programme au point où a été appelée la fonction.***
- ***La syntaxe générale de l'instruction return est la suivante :***
  - *return expression;*

Définition de fonction

- *Exemples :*

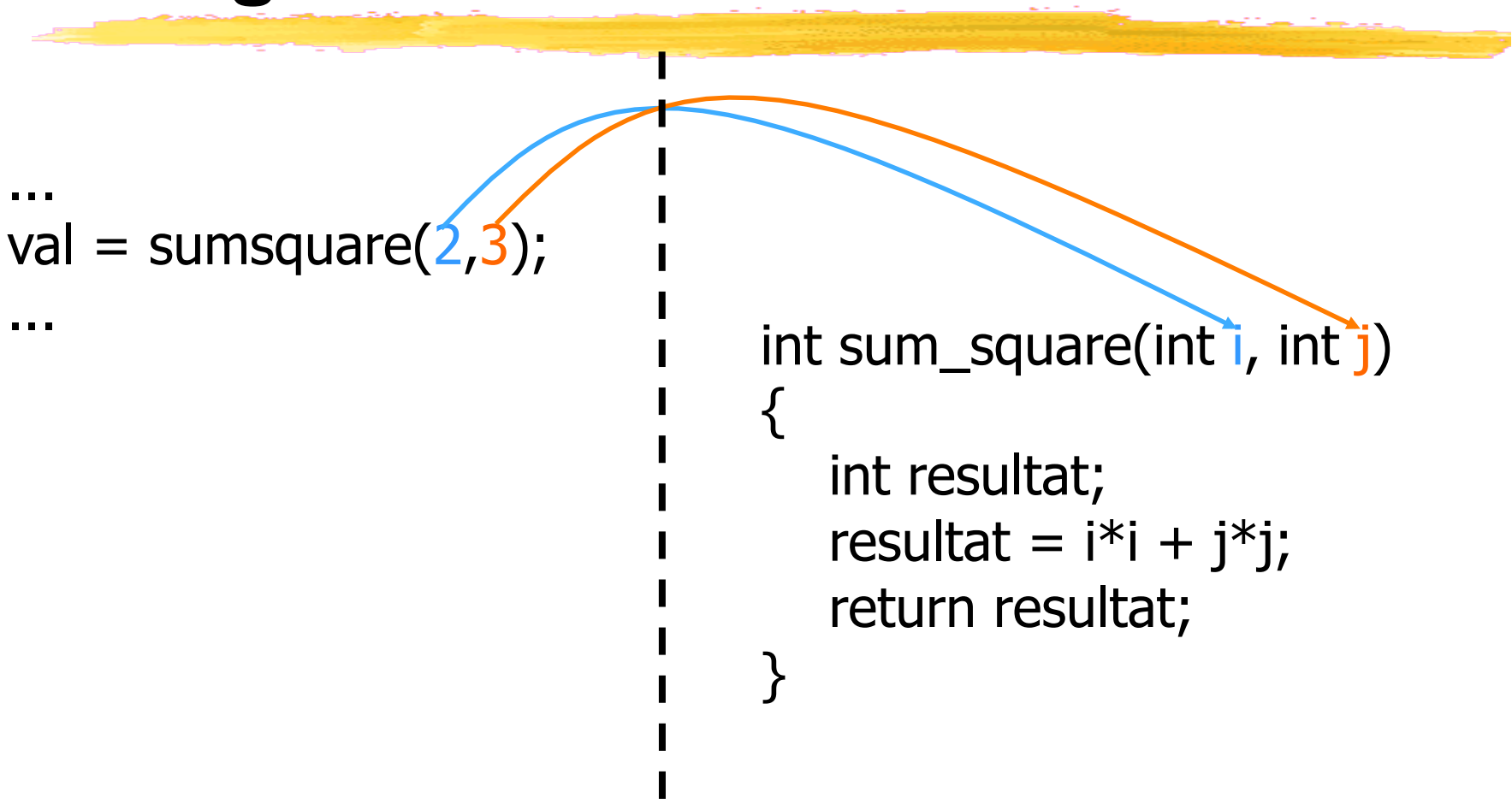
```
int sum_square(int i, int j)
{
   int resultat;
   resultat = i*i + j*j;
   return resultat;
 }

int max(int i, int j)
{
   if(i>j)
     return i;
   else
     return j;
}
```

Les arguments

- *Les arguments formels permettent le transfert d'informations entre la partie appelante du programme et la fonctions.*
- *Ils sont locaux à la fonction, et lors de l'appel ils seront mis en correspondance avec les arguments effectifs.*

Les arguments



Les arguments

- *La liste des arguments formels peut être vide s'il n'y a aucun argument à passer à la fonction.*
- *La syntaxe des déclarations pour les paramètres formels et les variables n'est pas la même.*
- *Pour déclarer des variables :*

```
int i;          est équivalent à :  int i,j;
int j;
```

- *Pour les arguments d'un fonction :*

```
int max(int i,j)          /*  est incorrecte
*/
{
...
}
```

Les arguments

- *Contrairement à d'autres langage, le C ne permet pas la modification des arguments d'une fonction.*
- *Le seul mode de passage des paramètres est le mode par valeur. Cela signifie que les valeurs des paramètres effectifs sont copiées dans les paramètres formels.*

Appel d'une fonction

- *nom_fonction (liste_d'expressions)*
- *Les expressions de liste_d'expressions sont évaluées, puis passées en tant qu'arguments effectifs à la fonction de nom nom_fonction, qui est ensuite exécutée.*
- *L'appel d'une fonction est une expression et non une instruction. La valeur rendue par la fonction est l'évaluation de l'expression appel de fonction.*
- *Exemples :*

```
s = sum_square(a,b);
m = max(a,b);
```

Appel d'une fonction

- *Dans le cas d'une fonction sans paramètre, la liste des paramètres doit être vide.*

- *L'ordre d'évaluation des paramètres effectifs n'est pas spécifiés :*

  ```
  sum_square(f(x),g(y));
  ```

  -> La fonction g sera peut-être exécutée avant f.

Déclaration de fonction

- *Lorsque l'appel d'une fonction figure avant sa définition, la fonction appelante doit contenir une déclaration de la fonction appelée. On appelle cela prototype de la fonction.*
- *Attention a ne pas confondre définition et déclaration de fonctions. La déclaration est une indication pour le compilateur quant au type de résultat renvoyé par la fonction et éventuellement au type des arguments, et rien de plus.*

Déclaration de fonction

- ***Dans sa forme la plus simple la déclaration d 'une fonction peut s 'écrire :***
  - *type_de_retour nom();*
- ***Les prototypes de fonctions sont souvent regroupés par thèmes dans des fichiers dont l 'extension est généralement h (ex stdio.h).***
- ***Ces fichiers sont inclus dans le source du programme par une directive du préprocesseur : #include.***

Récusivité

- *La récursivité est la caractéristique des fonctions capables de s'appeler elles-mêmes de façon répétitive, jusqu'à ce que soit vérifiée une condition d'arrêt.*
- *Il n'y a rien de spécial à faire pour qu'une fonction puisse être appelée de manière récursive.*
- *Exemple :*

```
int facto(int n)
{
  if (n==1) return 1;
  else return (n*facto(n-1));
}
```

Les procédures

- *Il n'y a pas de concept de procédure à proprement parler en C.*
- *Pour cela on déclare une fonction qui ne retourne aucune valeur grâce au mot-clé void comme type de retour.*
- *Exemple :*

```
void print_add(int a, int b)
{
  printf("%i",a+b);
}
```

Les fonctions imbriquées

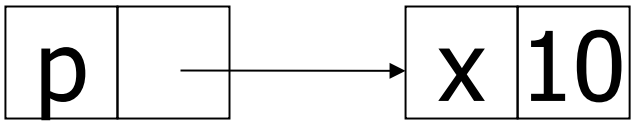
- *A l'inverse de certains langages, les fonctions imbriquées n'existent pas dans le langage C. Il n'est donc pas possible qu'une fonction ne soit connue qu'à l'intérieur d'une autre fonction.*

Sixième partie

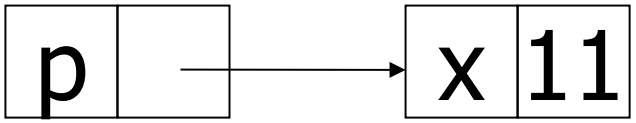
Les pointeurs et les tableaux

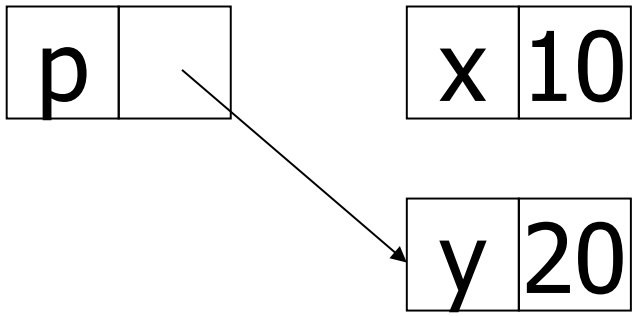
Pointeurs et adresses

- *Une variable désigne une valeur.*
- *On peut être amener parfois à vouloir désigner une variable par une autre variable.*
- *Par exemple p désigne la variable x qui a pour valeur 10*

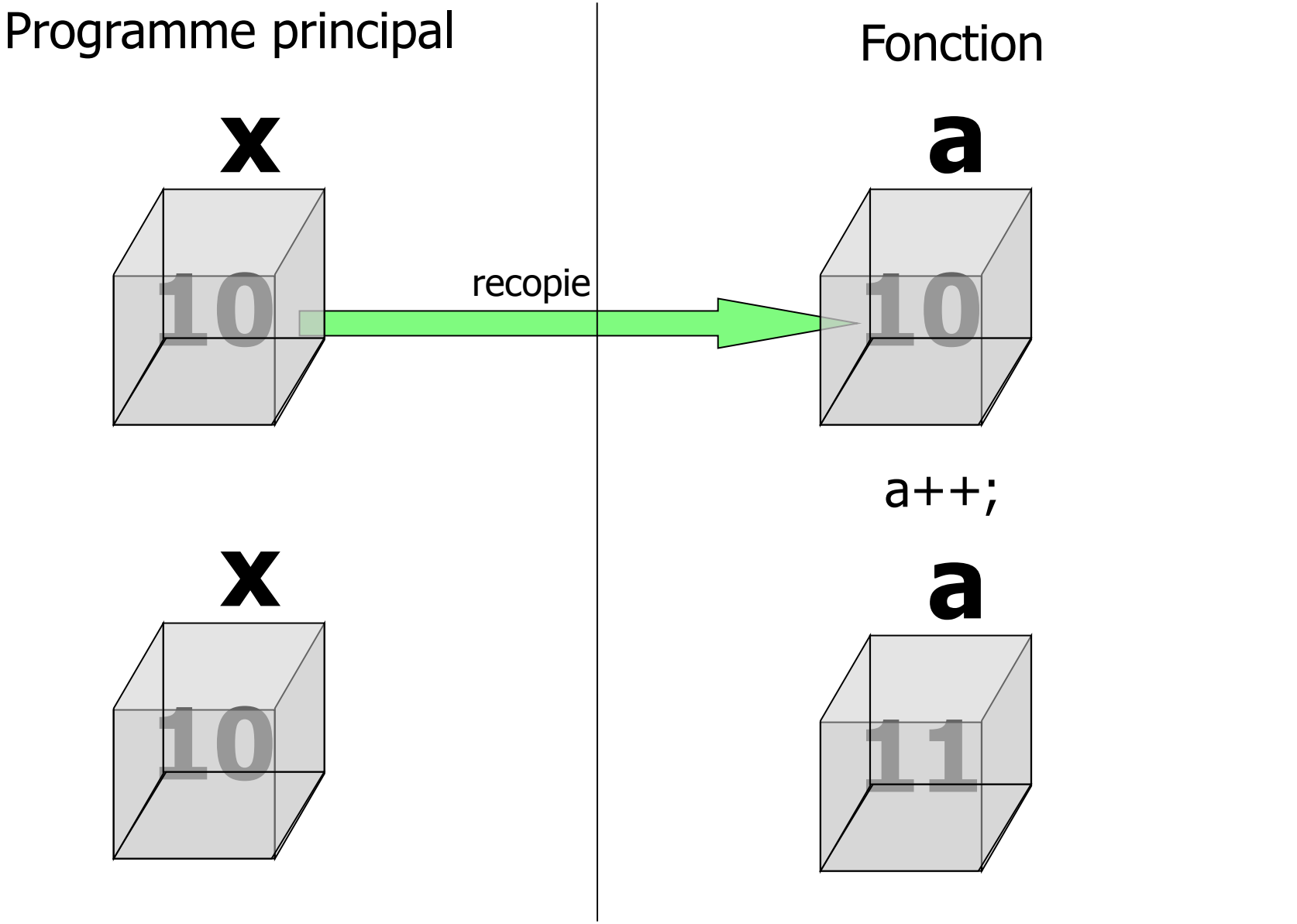


x++;

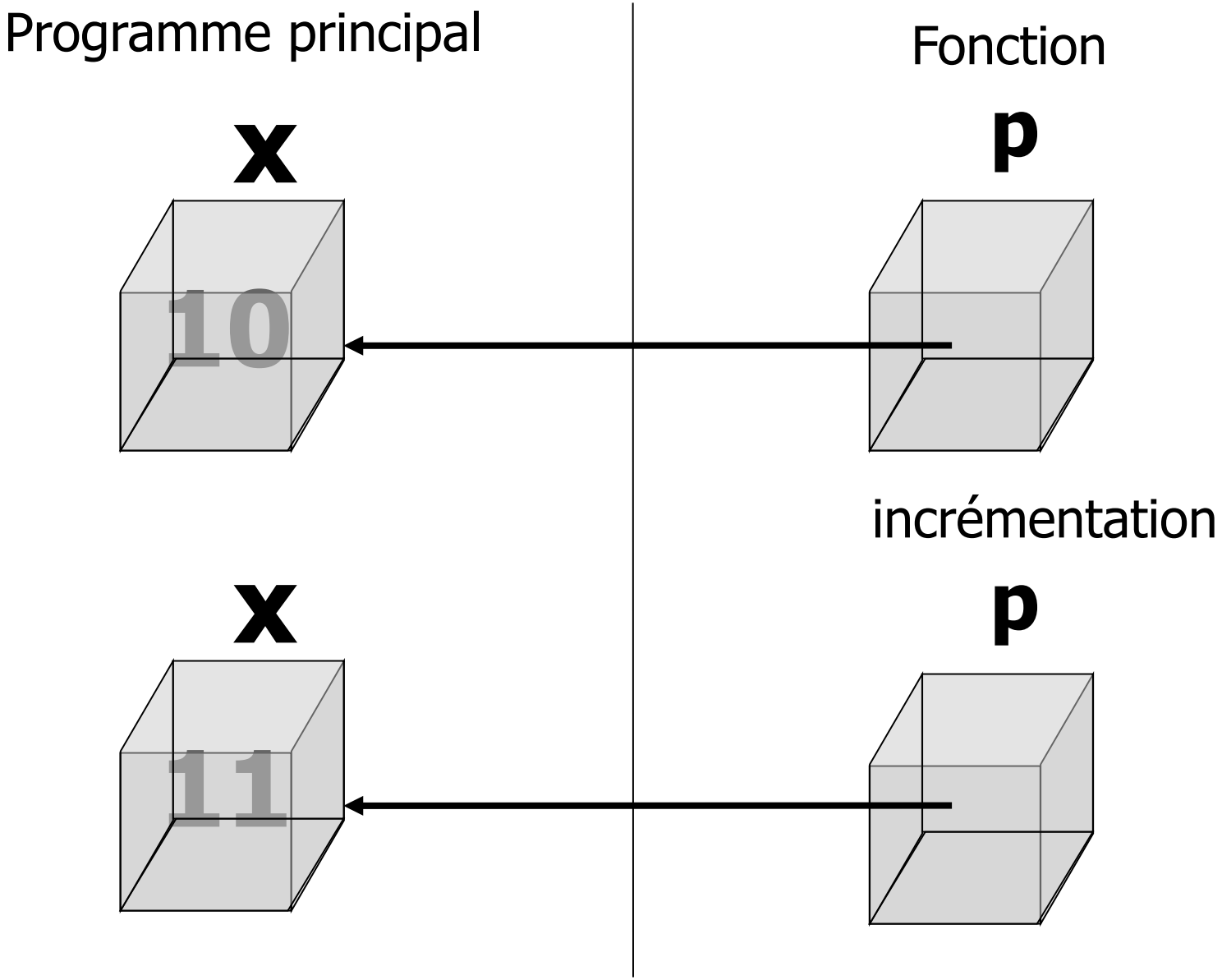




Pointeurs et adresses



Pointeurs et adresses



Pointeurs et adresses

- *Un pointeur est une adresse permettant de désigner un objet (une variable ou une fonction) en mémoire centrale.*
- *Par extension, on appelle pointeur la variable qui contient cette adresse.*

Pointeurs et adresses

- ***Un pointeur définissant l'adresse d'un objet, l'accès à cet objet peut alors être réalisé par une indirection sur le pointeur.***
  - *L'opérateur unaire & fournit une adresse de l'objet opérande (qui doit donc être une lvalue). On dit que cet opérateur est l'opérateur de **référence**.*
  - *L'opérateur unaire * considère son opérande comme un pointeur et retourne l'objet pointé par celui-ci. On dit aussi que cet opérateur est l'opérateur **d'indirection**.*

Pointeurs et adresses

- *Exemple :*

```
int x,y;
/* Soit px un pointeur sur des int */
px = &x;      /* px <- l'adresse de x */
y = *px;      /* y <- l'objet pointé par px
*/
```

- *Ceci équivaut donc à y=x;*

Pointeurs et adresses

- ***La déclaration d'un pointeur doit spécifier le type de l'objet pointé. On dit alors que le pointeur est typé.***
- ***Exemple :***

```
int *px;
```

  - *signifie que px pointera des objets de type int. (Littéralement : *px sera un int)*

Pointeurs et adresses

- *La définition d'un pointeur ne fait rien d'autre que ce qu'elle doit faire, c'est à dire qu'elle réserve en mémoire la place nécessaire pour mémoriser un pointeur, et en aucune façon de la place pour mémoriser un objet du type pointé*
- *L'objet atteint par l'intermédiaire du pointeur possède toutes les propriétés du type correspondant.*

Pointeurs et adresses

- ***Exemples :***
  - *px = &x;*
  - *y = *px + 1;* */* y = x + 1 */*
  - **px = 0;* */* x = 0 */*
  - **px += 10;* */* x = x + 10 */*
  - *(*px)++;* */* x = x + 1 */*

Pointeurs et adresses

- *Un pointeur pouvant repérer un type quelconque, il est donc possible d'avoir un pointeur de pointeur.*
- *Exemple :*

```
int **ppx;
      /* ppx pointe un pointeur d'entiers
*/
```

Opérations sur les pointeurs

- ***La valeur NULL, est une valeur de pointeur, constante et prédéfinie dans stddef.h.***
- ***Elle vaut 0 et signifie "Aucun objet".***
- ***Cette valeur peut être affectée à tout pointeur, quel que soit son type.***
  - *Dans ce cas, ce pointeur ne pointe sur rien...*
  - *Bien entendu, l'utilisation de cette valeur dans une indirection provoquera une erreur d'exécution.*

Opérations sur les pointeurs

- *L'affectation d'un pointeur à un autre n'est autorisée que si les 2 pointeurs pointent le même type d'objet (ie ont le même type)*
- *Exemple :*

```
p = NULL;
p = q;              /* p et q sont des pointeurs
sur le même type */
p = 0177300;            /* illégal */
p = (int *) 0177300;      /* légal */
```

Opérations sur les pointeurs

- *L'incrémentation d'un pointeur par un entier n est autorisée.*
- *Elle ne signifie surtout pas que l'adresse contenue dans le pointeur est incrémentée de n car alors cette adresse pourrait désigner une information non cohérente : être à cheval sur 2 mots par exemple…*
- *L'incrémentation d'un pointeur tient compte du type des objets pointés par celui-ci : elle signifie "passe à l'objet du type pointé qui suit immédiatement en mémoire".*

Opérations sur les pointeurs

- *Ceci revient donc à augmenter l'adresse contenue dans le pointeur par la taille des objets pointés.*
- *Dans le cas d'une valeur négative, une décrémentation a lieu.*
- *Les opérateurs combinés avec l'affectation sont autorisés avec les pointeurs.*
- *Il en va de même pour les incrémentations / décrémentations explicites.*

Opérations sur les pointeurs

- ***Exemples :***

```
int *ptr, k;     /* ptr est un pointeur , k est un int */

ptr++;
ptr += k;
ptr--;
ptr -= k;
```

Comparaison de pointeurs

- *Il est possible de comparer des pointeurs à l'aide des relations habituelles : < <= > >= == !=*
- *Ces opérations n'ont de sens que si le programmeur a une idée de l'implantation des objets en mémoire*
- *Un pointeur peut être comparé à la valeur NULL.*

Comparaison de pointeurs

- *Exemples :*

```
int *ptr1, *ptr2;
...
if (ptr1 <= ptr2)
...
if (ptr1 == 0177300)                /* illégal */
...
if (ptr1 == (int *)0177300)         /* légal */
...
if (ptr1 != NULL)
```

Soustraction de pointeurs

- *La différence de 2 pointeurs est possible, pourvu qu'ils pointent sur le même type d'objets.*
- *Cette différence fournira le nombre d'unités de type pointé, placées entre les adresses définies par ces 2 pointeurs.*
- *Autrement dit, la différence entre 2 pointeurs fournit la valeur entière qu'il faut ajouter au deuxième pointeur (au sens de l'incrémentation de pointeurs vue plus haut) pour obtenir le premier pointeur.*

Affectation de chaînes de caractères

- ***L'utilisation d'un littéral chaîne de caractères se traduit par :***
  - *la réservation en mémoire de la place nécessaire pour contenir ce littéral (complété par le caractère '\0')*
  - *et la production d'un pointeur sur le premier caractère de la chaîne ainsi mémorisée.*
- ***Il est alors possible d'utiliser cette valeur dans une affectation de pointeurs.***

Affectation de chaînes de caractères

- ***Exemple :***

```
char *message;
message = "Langage C"; /* est autorisé */
```

- *La chaîne est mémorisée et complétée par '\0'.*
- *La variable message reçoit l'adresse du caractère 'L'.*

Pointeurs et tableaux

- *On rappelle que la déclaration d'un tableau dans la langage C est de la forme :*

  int tab[10];

- *Cette ligne déclare un tableau de valeurs entières dont les indices varieront de la valeur 0 à la valeur 9.*

Pointeurs et tableaux

- ***En fait, cette déclaration est du "sucre syntaxique" donné au programmeur. En effet, de façon interne, elle entraîne :***
  - *La définition d'une valeur de pointeur sur le type des éléments du tableau, cette valeur est désignée par le nom même du tableau (Ici Tab)*
  - *La réservation de la place mémoire nécessaire au 10 éléments du tableau, alloués consécutivement. L'adresse du premier élément (Tab[0]) définit la valeur du pointeur Tab.*

Pointeurs et tableaux

- *La désignation d'un élément (tab[5] par exemple) est automatiquement traduite, par le compilateur, en un chemin d'accès utilisant le nom du tableau (Dans notre exemple ce serait donc *(tab + 5))*

Pointeurs et tableaux

- ***Exemples :***

  ```
  int *ptab, tab[10];
  ```

- ***On peut écrire :***
  - *ptab = &tab[0];*
  - *ptab = tab;* */* Equivalent au précédent */*
  - *tab[1] = 1;*
  - **(tab + 1) = 1;* */* Equivalent au précédent */*
- ***Donc, pour résumer, on a les équivalences suivantes :***
  - *tab + 1* est équivalent à *&(tab[1])*
  - **(tab + 1)* est équivalent à *tab[1]*
  - **(tab + k)* est équivalent à *tab[k]*

Pointeurs et tableaux

- *Aucun contrôle n'est fait pour s'assurer que l'élément du tableau existe effectivement :*
- *Si pint est égal à &tab[0], alors (pint - k) est légal et repère un élément hors des bornes du tableau tab lorsque k est strictement positif.*
- *Le nom d'un tableau n'est pas une lvalue, donc certaines opérations sont impossibles (exemple : tab++)*

Pointeurs et tableaux

```
#include <stdio.h>
main()
{
 char Tab[32], *Ptr;
                        /* Initialisation des données */
Tab[0] = 'Q'; Tab[1] = 'W'; Tab[2] = '\0';
Ptr = "ASDFGHJKL";
                        /* Edition des chaines */
printf("Contenu des chaines : \n");
printf("  Tab : %s\n  Ptr : %s\n", Tab, Ptr);
                        /* Utilisation des tableaux */
printf("Edition de l'élément de rang 1 des tableaux\n");
printf("  Tab : %c\n  Ptr : %c\n", Tab[1], Ptr[1]);
                        /* Utilisation des pointeurs */
printf("Edition du caractère pointé\n");
printf("  Tab : %c\n  Ptr : %c\n",*Tab, *Ptr);
                        /* Utilisation des pointeurs incrémentés */
printf("Edition du caractère suivant le caractère pointé\n");
printf("  Tab : %c\n  Ptr : %c\n", *(Tab + 1), *(Ptr + 1));
                        /* Règle des priorités */
printf ("Edition identique non parenthésée \n");
printf("  Tab : %c\n  Ptr : %c\n", *Tab + 1, *Ptr + 1);
}
```

Passage des paramètres

- *Le seul mode de passage des paramètres à une fonction est le passage par valeur.*
- *Le problème est que lorsqu'un sous-programme doit fournir une réponse par un de ses paramètres, ou modifier ceux-ci, le mode de passage par valeur ne peut plus convenir.*

Passage des paramètres

- ***De plus, dans le cas de paramètres structurés de taille importante, la copie de ceux-ci entraîne une consommation mémoire parfois superflue.***
- ***C'est à la charge du programmeur de gérer tous ces problèmes en fournissant un pointeur sur l'objet en question.***
- ***Ceci permet au sous-programme :***
  - ➢ *de travailler directement sur l'objet réel et donc de le modifier,*
  - ➢ *mais aussi réduit la taille des échanges, puisqu'il n'y a copie que d'une adresse.*

Passage des paramètres

```
void Echange_1 (int a,int b)
{
        int c = a;
        a = b;
        b = c;
}
```

```
void Echange_2 (int *a,int *b)
{
        int c = *a;
        *a = *b;
        *b = c;
}
```

```
main()
{
        int x, y;
        printf("Valeur de x : ");
        scanf ("%d",&x);
        printf("\nValeur de y : ");
        scanf ("%d",&y);
        Echange_1 (x,y)
        printf("\nAprès appel de Echange_1 :\n x = %d\n y = %d\n", x, y);
        Echange_2 (&x,&y)
        printf("Après appel de Echange_1 :\n x = %d\n y = %d\n", x, y);
}
```

Passage des paramètres

- *Quand un tableau est transmis comme paramètre effectif à un sous-programme, c'est en fait l'adresse de base de ce tableau qui est transmise par valeur.*
- *Ceci a pour effet de transmettre le tableau par référence sans intervention du programmeur.*
- *De plus, à l'intérieur de la fonction appelée, le paramètre tableau devient une lvalue, certaines opérations sont donc possibles.*

Passage des paramètres

```
#include <stdio.h>
void Test_Tab (int t[])
{
        printf(" Editions dans la fonction \n");
        printf(" Valeur du paramètre tableau : %o \n",t);
        printf(" Valeur du contenu : %d\n",t[0]);
        printf(" Test d'incrémentation paramètre \n");
        t++; /* t est devenue une lvalue */
        printf(" Valeur du paramètre tableau : %o \n",t);
}

main()
{
        int tab[5];
        tab[0] = 10;
        printf("Editions avant appel de fonction\n");
        printf(" Valeur du paramètre tableau : %o\n",tab);
        printf(" Valeur du contenu : %d\n",tab[0]);
        Test_Tab(tab);
}
```

Pointeurs et tableaux multidimensionnels

- *Un tableau unidimensionnel peut se représenter grâce à un pointeur (le nom du tableau) et un décalage (l'indice).*
- *Un tableau à plusieurs dimensions peut se représenter à l'aide d'une notation similaire, construite avec des pointeurs.*
- *Par exemple, un tableau de dimension 2, est en fait un ensemble de deux tableaux à une seule dimension.*
- *Il est ainsi possible de considérer ce tableau à deux dimensions comme un pointeur vers un groupe de tableaux unidimensionnels consécutifs.*

Pointeurs et tableaux multidimensionnels

- ***On peut donc écrire la déclaration suivante:***
  - *type-donnée (*varpt)[expression 2];*
- ***au lieu de la déclaration classique:***
  - *type-donnée tableau [expression l][expression 2];*
- ***Ce style de déclaration peut se généraliser à un tableau de dimension n de la façon suivante:***
  - *type-donnée (*varpt)[expression 2][expression 3] ... [expression n];*
- ***qui remplace la déclaration équivalente:***
  - *type-donnée tableau[expression 1][expression 2] ... [expression n];*

Pointeurs et tableaux multidimensionnels

- *Exemple : On suppose que x est un tableau d'entiers de dimension 2, ayant 10 lignes et 20 colonnes.*
- *On peut le déclarer de la façon suivante:*

  ```
  int (*x) [20];
  ```

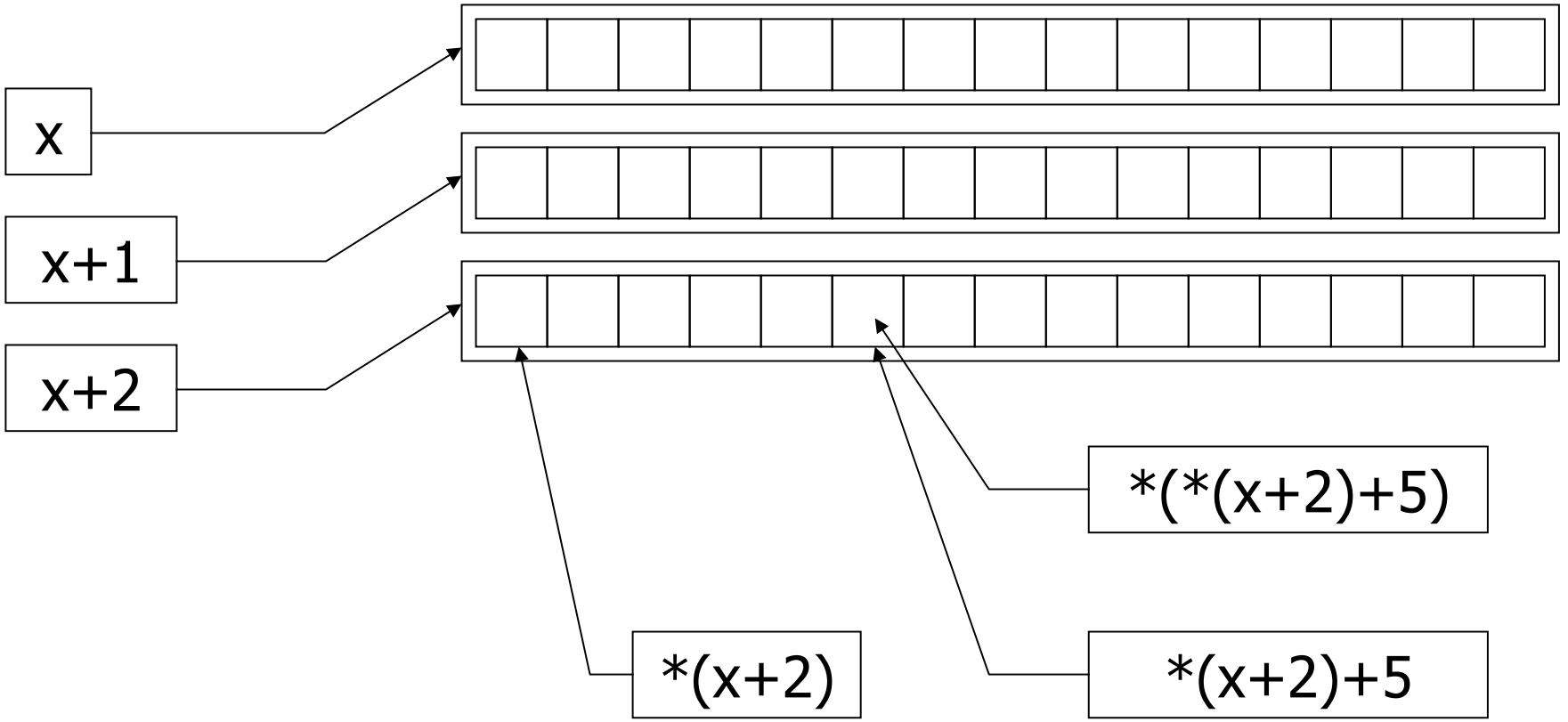
- *au lieu de :*

  ```
  int x[10] [20];
  ```

Pointeurs et tableaux multidimensionnels

- *Dans la première déclaration, on dit que x est un pointeur vers un groupe de tableaux d'entiers contigus, possédant chacun une dimension et 20 éléments.*
- *Ainsi, x pointe le premier tableau de 20 éléments, qui constitue en fait la première ligne (ligne 0) du tableau d'origine de dimension 2.*
- *De manière analogue, (x+1) pointe vers le deuxième tableau de 20 éléments, qui représente la seconde ligne (ligne 1) du tableau d'origine, et ainsi de suite*

Pointeurs et tableaux multidimensionnels



Pointeurs et tableaux multidimensionnels

- *Si l'on considère à présent un tableau tridimensionnel de réels, t, on peut le définir par:*

  float (*t) [20] [30];

- *au lieu de :*

  float t[10] [20] [30];

- ***La première forme définit t comme un pointeur vers un groupe contigu de tableaux de réels, de dimension 20 x 30.***

- ***Dans ce cas t pointe le premier tableau 20 x 30, (t+1) pointe le second tableau 20x30, etc.***

Pointeurs et tableaux multidimensionnels

- *Pour accéder à un élément donné d'un tableau à plusieurs dimensions, on applique plusieurs fois l'opérateur d'indirection. Mais cette méthode est souvent plus pénible d'emploi que la méthode classique.*
- *Exemple : x est un tableau d'entiers de dimension 2, à 10 lignes et 20 colonnes. L'élément situé en ligne 2 et colonne 5 peut se désigner aussi bien par :*

  x[2] [5] *que par* ** (* (x+2) +5)*

Pointeurs et tableaux multidimensionnels

- *Les programmes mettant en jeu des tableaux multidimensionnels peuvent s'écrire de différentes manières.*
- *Choisir entre ces formes relève surtout des goûts personnels du programmeur.*

Tableaux de pointeurs

- ***Les pointeurs étant des informations comme les autres, ils peuvent être mis dans des tableaux.***
  - *Exemple : char *ptr_char[7];*
- ***Cette déclaration se lit comme : ptr_char est un tableau dont les éléments sont du type char*.***
- ***Les 7 éléments de ptr_char sont donc des pointeurs vers des caractères.***

Tableaux de pointeurs

- *D'après ce qui a été vu précédemment cela signifie que chaque élément du tableau peut pointer vers une chaîne de caractères.*
- *On conçoit qu'avec une telle représentation, une permutation de chaîne est très facile puisqu'il suffit de permuter les pointeurs correspondants.*
- *De plus, les différentes chaînes peuvent avoir des tailles différentes, ce qui ne serait pas possible si l'on déclarait un tableau de caractères à 2 dimensions comme :*

```
char tab_char[7][10];
```

Initialisation des tableaux multidimensionnels

- ***On peut initialiser un tableau d'entiers à deux dimensions de la façon suivante :***
  - *int t[2][3] = {{1,2,3},{4,5,6}};*
- ***Ou encore :***
  - *int t[2][3] = {1,2,3,4,5,6};*
- ***Car les éléments sont placés dans l'ordre suivant : t[0][0],t[0][1],t[0][2],t[1][0],t[1][1],t[1][2], c'est-à-dire ligne après ligne.***

Pointeurs de fonctions

- *Une fonction ne peut être considérée comme une variable; cependant elle est implantée en mémoire et son adresse peut être définie comme l'adresse de la première instruction de son code.*
- *Lorsque le compilateur reconnaît, dans une expression, l'identificateur d'une fonction qui ne correspond pas à un appel de fonction (absence de parenthèses), il engendre alors l'adresse de cette fonction.*
- *Il est donc possible de définir un pointeur sur une fonction, ce qui sera particulièrement utile lorsqu'on désirera passer une fonction en paramètre.*

Pointeurs de fonctions

- ***Exemple :***
  int (*f)(); /* pointeur de fonction retournant un entier */
  - *Ici (*f) signifie : f est un pointeur; *f définit l'objet pointé*
  - *(*f)() signifie : l'objet pointé est une fonction*
  - *int (*f)() signifie : l'objet pointé est une fonction qui retourne un entier*
- ***ATTENTION !!! à ne pas confondre :***
  int (*f)() avec : int *f ()

Pointeurs de fonctions

```
int fonc1()
{
  return 1;
}
int fonc2()
{
  return 2;
}

int main()
{
  int (*f)();

  f=fonc1;
  printf("resultat : %i\n",f());
  f=fonc2;
  printf("resultat : %i\n",f());
  return 0;
}
```

Pointeurs de fonctions

```
int add(int a, int b)
{
  return a+b;
}
int mul(int a, int b)
{
  return a*b;
}

int main()
{
  int (*f)(int, int);

  f=add;
  printf("resultat : %i\n",f(2,3));
  f=mul;
  printf("resultat : %i\n",f(2,3));
  return 0;
}
```

```
int max(int a,int b)
{
  return(a>b?a:b);
}
int min(int a,int b)
{
return(a<b?a:b);
}
void main(void);
{
  int (*calcul)(int,int);
  char c;
  printf("utiliser mAx (A) ou mIn (I) ?");
  do
    c=getchar();
  while ((c!='A')&&(c!='I'));
  calcul=(c=='A')?max:min;
  printf("%d\n",calcul(10,20));
}
```

Allocation dynamique

- *La bibliothèque standard C dispose de 3 fonctions d'allocation : malloc, calloc et realloc qui ressemblent au new pascal, et une fonction de désallocation : free*
- *Elles ont leur prototype dans stdlib.h.*

Allocation dynamique

- *void *malloc (size_t tob) ;*
- *Le type size_t est défini dans stdlib.h et est équivalent à unsigned int ou unsigned long int (ça dépend des systèmes).*
- *tob représente une taille d'objet en octets.*
- *malloc renvoie un pointeur universel (void *) vers le début de la zone allouée de tob octets.*
- *malloc renvoie NULL si l'allocation a échoué.*

Allocation dynamique

```
#include <stdio.h>

int main()
{
  char chaine[80];
  printf("Entrez une chaîne :");
  scanf("%s",chaine);
  printf("Voici la chaîne : %s",chaine);
  return 0;
}
```

Allocation dynamique

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
  char *chaine;
  chaine = (char *)malloc(sizeof(char)*80);
  if(chaine)
  {
    printf("Entrez une chaîne :");
    scanf("%s",chaine);
    printf("Voici la chaîne : %s",chaine);
  }
  else
    printf("Plus de mémoire !!!");
  return 0;
}
```

Allocation dynamique

- *void *calloc (size_t nob, size_t tob) ;*
- *nob représente un nombre d'objets.*
- *calloc renvoie un pointeur universel (void *) vers le début de la zone allouée de nob objets de taille tob octets (tableau dynamique).*
- *calloc renvoie NULL si l'allocation a échoué.*

Allocation dynamique

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
  char *chaine;
  chaine = (char *)calloc(80,sizeof(char));
  if(chaine)
  {
    printf("Entrez une chaîne :");
    scanf("%s",chaine);
    printf("Voici la chaîne : %s",chaine);
  }
  else
      printf("Plus de mémoire !!!");
    return 0;
}
```

Allocation dynamique

- *void *realloc (void *ptr, size_t tob) ;*
- *realloc permet de modifier la taille d'une zone précédemment allouée.*
- *realloc renvoie NULL si l'allocation a échoué.*

Allocation dynamique

```
int main()
{
  char *chaine;
  chaine = (char *)calloc(1000,sizeof(char));
  if(chaine)
  {
    printf("Entrez une chaîne :");
    scanf("%s",chaine);
    chaine = (char*)realloc(chaine,sizeof(char)*(strlen(chaine)+1));
    printf("Voici la chaîne : %s\n",chaine);
  }
  else
  {
    printf("Plus de mémoire !!!");
  }
  return 0;
}
```

Allocation dynamique

- *void free (void *) ;*
- *free rend au système (désalloue) la zone pointée par le pointeur paramètre. Si ce pointeur a déjà été désalloué ou s'il ne correspond pas à une valeur allouée par malloc, calloc ou realloc, il y a une erreur.*
- *Le comportement du système est alors imprévisible.*

Allocation dynamique

```
int main()
{
  char *chaine;
  chaine = (char *)calloc(1000,sizeof(char));
  if(chaine)
  {
     ...
     free(chaine);
  }
  else
  {
    printf("Plus de mémoire !!!");
  }
  return 0;
}
```

Allocation dynamique

- ***Exemple : Une fonction qui reçoit en argument un pointeur à initialiser par malloc et qui renvoie un booléen indiquant si l'allocation s'est bien passée.***

```
int alloc (int **pp) /* pp est un pointeur de pointeur d'entier */
{
        *pp=(int *) malloc (sizeof (int)) ;
        return *pp != NULL ;
}
main ()
{
        ...
        int *p ;
        p=NULL ;
        if (alloc (&p))    /* adresse du pointeur          */
                *p=5 ;  /* p a été modifié par alloc    */
        .......            /*  il n'est plus à NULL        */
}
```

Septième partie

Portée des déclarations

Tableaux de pointeurs

- *On peut initialiser un tableau de pointeur de la façon suivante :*

```
char *Ptr_Char[7] = {"Lundi", "Mardi", "Mercredi"};
```

- *et on pourrait l'utiliser ainsi :*

```
for (i = 0; i<7; i++)
    printf(" Jour %d = %s\n",i, Ptr_Char[i]);
```

Structure des programmes

- *Dans certains contextes il peut être nécessaire de disposer de variables globales utilisables sur la totalité, ou une partie seulement du programme.*
- *Ce type de variable doit être définie différemment des variables dites "locales", dont la portée ne dépasse pas la fonction dans laquelle elles sont définies.*

Structure des programmes

- *Il peut être également possible de conserver certaines informations dans des variables locales, aussi bien de façon statique que dynamique.*
- *En effet, par défaut la valeur d'une variable locale est perdue à l'issue de la fonction dans laquelle elle est définie.*

Structure des programmes

- *Enfin il peut être nécessaire de développer des programmes de grande taille, faisant appel à de multiples fonctions réparties dans divers fichiers sources.*
- *Dans ce type de programmes, les fonctions peuvent être définies et appelées aussi bien de façon locale, dans un même fichier, que globale, depuis des fichiers distincts.*

Classe de mémorisation

- *Une variable est caractérisée par deux éléments qui sont : le type de donnée et la classe de mémorisation.*
- *Le type de la donnée fait référence à la nature des informations que contiendra la variable. (un entier, un réel ...)*
- *La classe de mémorisation fait référence à la pérennité (permanence) d'une variable, et à sa visibilité dans le programme.*

Classe de mémorisation

- *Il existe en C, quatre classes de mémorisation : automatique, externe, statique ou registre, identifiées respectivement par les mots clés auto, extern, static et register*
- *La classe de mémorisation attachée à une variable peut parfois être déterminée par la seule localisation de la déclaration de cette variable dans le programme. Dans les autres cas le mot clé explicite la classe de mémorisation.*

Classe de mémorisation

- ***Exemples :***

  Variables automatiques entières
  auto int a, b, c ;

  Variables externes de type réelles
  extern float racine1, racine2 ;

  Variable entière statique initialisée à 0
  static int compteur = 0 ;

  Variable externe caractère
  extern char etoile ;

Variable de classe automatique

- *Les variables de classe automatique sont toujours déclarées au sein d'une fonction, pour laquelle elle joue le rôle de variable locale.*
- *Leur portée est limitée à cette seule fonction.*
- *C'est le mode par défaut, donc le mot clé auto est facultatif.*
- *Exemple :*

```
void fonc1()
{
   int i;
   ...
}
i++;          /* erreur i n'est plus connu */
```

Variable de classe automatique

- *Ces variables peuvent être initialisées en incluant les expressions adéquates dans leur déclaration. void fonc1()*

```
{
   int i=1;
   ...
}
```

- *La valeur d'une variable de classe auto n'est pas conservée d'un appel à l'autre de la fonction.*

Variable de classe automatique

- *Il est possible de réduire la portée d'une variable automatique à une étendue inférieure à celle de la fonction. Il faut la définir pour cela dans une instruction composée, sa durée de vie est alors celle de l'instruction.*
- *Exemple :*

```
void fonc1()
{
  if (...)
  {
    int i=1;
    ...
  }
}
```

Variables de classe externe

- *Les variables de classe externe au contraire des variables de classe automatique ne sont pas limitées à la seule étendue d'une fonction.*
- *Leur portée s'étend de l'endroit où elles sont définies à la fin du programme.*
- *Elles constituent en C ce qu'on appelle une variable globale dans d'autres langages.*

Variables de classe externe

- *Exemple :*

```
int i;
int j;
                /* i et j sont connues */
void fonc1()
{
  ...               /* i et j sont connues */
}

int main()
{
  ...               /* i et j sont connues */
}
                /* i et j sont connues */
```

Variables de classe externe

- *Une variable externe peut se voir affecter une valeur dans une fonction, valeur qui pourra être accédée depuis une autre fonction. (Attention danger !!!)*
- *Exemple :*

```
int compteur;
void initialisation()
{
  compteur = 10;
}
void incrementation()
{
  compteur++;
}
```

Variables de classe externe

- *Lorsque l'on utilise des variables externes il est important de différencier leur définition et leur déclaration.*
- *La définition d'une variable externe s'écrit comme la définition de n'importe qu'elle autre variable ordinaire.*
- *La définition d'une variable externe doit se situer en dehors et en amont des fonctions qui utiliseront la variable.*
- *La définition d'une variable externe ne doit pas contenir le mot clé extern*

Variables de classe externe

- *Toute fonction utilisant une variable externe doit comporter une déclaration explicite de cette variable, (obligatoire si la fonction est définie avant la variable, facultatif si la fonction est définie après).*
- *Cette déclaration se traduit par la répétition de la définition précédée du mot clé extern.*

Variables de classe externe

- *Aucune allocation de mémoire n'est faite à la suite d'une déclaration puisque cette allocation est faite une fois pour toute lors de la définition de la variable.*
- *Ceci explique qu'il n'est pas possible d'initialiser une variable externe au moment de sa déclaration, mais uniquement lors de la définition.*

Variables de classe externe

- *Les variables externes peuvent être initialisées dans leur définition.*
- *En l'absence de toute initialisation explicite, les variables externes sont automatiquement initialisées à 0.*
- *Il est cependant préférable d'expliciter dans la définition, l'initialisation à 0 lorsqu'elle est utilisée.*

Variables de classe externe

- *Il est important de souligner les risques inhérents à l'usage des variables externes, en particulier les effets de bords.*
- *Ces variables sont souvent la source d'erreurs difficiles à identifier.*
- *Le choix par le programmeur des classes de mémorisation doit donc être effectué de façon réfléchie, pour répondre à des situations bien déterminées.*

Variable de classe statique

- *Les variables statiques sont définies dans les fonctions comme en automatique et ont donc la même portée (locale à la fonction).*
- *Par contre, ces variables conservent leur valeur durant tout le cycle du programme. Ce sont des variables rémanentes.*
- *On les définit de la même façon que les variables automatique sauf que leur déclaration doit commencer par le spécificateur de classe static.*

Variable de classe statique

- *Exemple :*

```
int foo()
{
  static int var_stat = 3;
  var_stat++;
  printf("var_stat=%i\n",var_stat);
  return 0;
}

int main()
{
  foo();
  foo();
  return 0;
}
```

Variable de classe statique

- *Les définitions de variables statiques peuvent contenir des affectations de valeurs initiales.*
- *En l'absence de valeur initiale, une variable statique est toujours initialisée à 0.*

register

- *Ce spécificateur n'est autorisé que pour les déclarations de variables locales à une instruction composée, et pour les déclarations de paramètres de fonctions.*
- *Sa signification est celle de auto avec en plus une indication pour le compilateur d'allouer pour la variable une ressource à accès rapide.*

register

- *Le programmeur est supposé mettre une variable dans la classe register quand elle est fortement utilisée par l'algorithme.*
- *Il y a cependant une contrainte : une telle variable n'a pas d'adresse, impossible donc de lui appliquer l'opérateur &.*
- ***En l'absence de valeur initiale, une variable register n'est pas initialisée à 0 et contient généralement une valeur parasite.***

Retour sur la récursivité

- *Exemple :*

```
int facto(int n)
{
  if (n==1) return 1;
  else return (n*facto(n-1));
}
```

- *n est une variable automatique.*
- *Quand une fonction fait appel à elle-même, chaque appel crée un nouvel ensemble comprenant toutes les variables automatiques qui est alors indépendant de celui engendré précédemment.*

Communication entre modules

- ***Une application peut être conçue comme une collection de modules, chaque module pouvant être :***
  - *une collection de sous-programmes,*
  - *une collection de données partagées,*
  - *un objet abstrait possédant une représentation et offrant des opérateurs de manipulation*
- ***De tels modules peuvent constituer des unités de programmes en C chacune mémorisée dans un fichier.***
- ***Une unité de programme constitue une unité de compilation.***

Communication entre modules

- *Après compilation de toutes les unités constituant l'application, l'éditeur de liens a pour tâche de rassembler les divers modules résultats des compilations (modules objets) afin de constituer l'application totale.*
- *Il est donc nécessaire d'établir un protocole de communication entre les divers modules. Il permet de spécifier dans le programme source, les liens à établir entre ces modules séparément.*

Communication entre modules

- *On dit qu'une variable ou une fonction est partagée entre deux modules lorsqu'elle est utilisée dans ces deux modules.*
- *En C seules les variables globales peuvent être partagées entre deux modules.*
- *Par contre, toute fonction peut être partagée entre deux modules.*

Communication entre modules

- *Dans le module où la variable est définie, une définition introduit l'identificateur de la variable, son type et éventuellement une valeur initiale.*
- *La variable est visible dans tout le module, à partir du point de sa déclaration.*
- *Dans un autre module où la variable est utilisée, une déclaration introduit l'identificateur de type de la variable.*
- *Cette déclaration doit être précédée de l'attribut extern. La variable est alors visible dans tout le module à partir du point de cette déclaration.*

Communication entre modules

- ***Exemple :***

Module 1	Module 2

int v = 23;
 	
int main()
{
  extern int val;
  …
}
	int val = 100;
	
	int fonc()
	{
	  extern int v;
	  …
	}

Communication entre modules

- ***Une fonction définie dans un module peut être manipulée dans un autre module, à condition d'avoir effectué une déclaration d'importation dans le second module.***

Module 1	Module 2

int v = 23;
extern int val;
extern int fonc();
 	
int main()
{
  int i = fonc();
}
	int val = 100;
	
	int fonc()
	{
	  extern int v;
	  ...
	}

L'autre utilisation de static

- *Nous avons vu que static permet de rendre statique une variable locale.*
- *Il sert également à cacher à l'éditeur de liens les variables globales.*
- *Si on ne souhaite pas que les variables globales d'un module soient accessibles dans un autre module, il faut utiliser le préfixe static.*

L'autre utilisation de static

Module 1
static int v = 0; /* variable locale au module 1*/

Module 2
int v = 0; /* variable exportable*/

Module 3
extern int v = 0; /* variable v définie dans le module 2*/

Huitième partie

Les structures

Notion de structure

- *Il est habituel en programmation d'avoir besoin d'un mécanisme permettant de regrouper un certain nombre de variables.*
- *On a déjà vu que les tableaux permettaient de regrouper plusieurs éléments. Toutefois, ceux-ci devaient être tous du même type.*
- *Il est parfois nécessaire de regrouper des variables de types différents.*

Notion de structure

- ***Par exemple, pour créer un fichier de personnes on aurait besoin d'une structure de la forme :***
  - ***NOM*** *: chaîne de caractères*
  - ***AGE*** *: entier*
  - ***TAILLE*** *: réel*
- ***La solution est le concept de structure qui est un ensemble d'éléments de types différents repérés par un nom.***
- ***Les éléments d'une structure sont appelés membres ou champs de la structure.***

Déclaration des structures

- *Syntaxe :*

```
struct{
   liste des membres
}
```

- *Exemple :*

```
struct{
    char *Nom;
    int Age;
    float Taille;
} p1,p2;
```

Déclaration des structures

- ***La déclaration précédente déclare deux variables de noms p1 et p2 comme deux structures contenant trois membres.***
- ***L'espace mémoire nécessaire a été réservé pour que les deux variables contiennent chacune trois membres.***

p1	
Nom	char *
Age	int
Taille	float

p2	
Nom	char *
Age	int
Taille	float

Déclaration des structures

- *L'inconvénient de cette méthode est qu'il ne sera plus possible de déclarer d'autres variables du même type.*
- *Si nous déclarons ensuite :*

```
 struct{
    char *Nom;
    int Age;
    float Taille;
} p3;
```

- *p3 ne sera pas considérée du même type, il sera impossible en particulier d'écrire p1 = p3;.*

Déclaration des structures

- *Heureusement, il est possible de définir des modèle de structure.*
- *Syntaxe :*

```
struct identificateur{
  liste des membres
}
```

- *Exemple :*

```
struct personne{
   char *Nom;
   int Age;
   float Taille;
};
```

Déclaration des structures

- *Une telle déclaration n'engendre pas de réservation mémoire.*
- *Le modèle ainsi défini peut être utilisé dans une déclaration de variable.*
- *Exemple :*
  struct personne Michel, Anne;
  / Déclare deux variables de type "struct personne" */*

Déclaration des structures

- ***Dans une structure, tous les noms de champs doivent être distincts.***
- ***Par contre rien n'empêche d'avoir 2 structures avec des noms de champs en commun : l'ambiguïté sera levée par la présence du nom de la structure concernée.***

```
struct personne{
   char *Nom;
   int Age;
   float Taille;
}
```

```
struct pays{
   char *Nom;
   int nb_habitants;
   int superficie;
}
```

Initialisation de structures

- *Une déclaration de structure peut contenir une initialisation par une liste d'expressions constantes à la manière des initialisations de tableau.*
- *La valeur initiale est constituée d'une liste d'éléments initiaux (1 par champ) placée entre accolades.*
- *Exemple :*
  struct personne p = {"Dupond", 25, 1.80};

Accès à un champ

- *L'opérateur d'accès est le symbole "." (point) placé entre l'identificateur de la structure et l'identificateur du champ désigné.*
- *Syntaxe :*

  <ident_objet_struct>.<ident_champ>
- *Exemples :*

```
p1.Age = 25;
if(p2.Nom[0]=='D')
...
```

Affectation des structures

- *On peut affecter l'intégralité des valeurs d'une structure à une seconde structure ayant impérativement le même type.*
- *Exemple :*

```
struct personne p1,p2;
p1 = p2;
```

- *L'affectation d'une structure recopie tous les champs un à un.*
- *Attention la recopie est superficielle.*

Tableau de structures

- *Une déclaration de tableau de structures se fait selon le même modèle que la déclaration d'un tableau dont les éléments sont de type simple.*
- *Pour déclarer un tableau de 100 structures personne, on écrira :*
  ```
  struct personne t[100];
  ```
- *Pour référencer le nom de la personne qui a l'index i dans t, on écrira :*
  ```
  t[i].Nom = "Dupond";
  ```

Structures composées

- *Puisqu'une structure définit un type, ce type peut être utilisé dans une déclaration de variable mais aussi dans la déclaration d'une autre structure comme type de l'un de ses champs.*
- *Exemple :*

```
struct Date
{
   int Jour, Mois, Annee;
};
struct personne
{
   char *Nom;
   struct Date Naissance;
};
```

Structures composées

- *Exemple d'initialisation :*

```
struct personne p = {"Dupond",
{21,05,1980}};
```

- *Exemple d'accès aux membres :*

```
if (p.Naissance.Annee < 1985)
    if(p.Naissance.Mois == 5)
```

Pointeurs vers une structure

- *Supposons la structure personne déclarée, nous pouvons déclarer une variable de type pointeur vers cette structure de la façon suivante :*
  ```
  struct personne *p;
  ```
- *Nous pouvons alors affecter à p des adresses de struct personne.*

Pointeurs vers une structure

- *Exemple :*

```
struct personne
{
    char *Nom;
    int Age;
};

int main()
{
  struct personne pers;
  struct personne p*;

  p = &pers;
}
```

Accès aux champs d'une structure pointée

- *Soit p un pointeur vers une structure personne, *p désigne la structure pointée par p.*
- *Mais *p.Nom ne référence pas le champ Nom de la structure car l'opérateur d'accès aux champs a une priorité supérieure à l'opérateur d'indirection.*
- **p.Nom est équivalent à *(p.Nom)*
- *Il faut donc écrire (*p).Nom pour forcer l'indirection à se faire avant l'accès au champ.*

Accès aux champs d'une structure pointée

- *Pour alléger la notation, le langage C a prévu un nouvel opérateur noté -> qui réalise à la fois l'indirection et la sélection.*
- *p->Nom est équivalent à (*p).Nom*
- *Exemple :*

```
struct personne pers;
struct personne *p;

p = &pers;
p->Nom="Dupont";
```

Champ pointant vers une structure

Exemple :

```
struct Date
{
    int Jour, Mois, Annee;
};

struct personne
{
    char *Nom;
    struct Date *Naissance;
};
```

Champ pointant vers une structure

- *Accès aux champs :*

```
pers.Nom = "Dupont";
pers.Naissance->Jour = 21;
pers.Naissance->Mois = 05;
pers.Naissance->Annee = 1985;
```

- *Si p est un pointeur sur pers :*

```
p->Nom = "Dupont";
p->Naissance->Jour = 21;
p->Naissance->Mois = 05;
p->Naissance->Annee = 1985;
```

Structures récursives

- *Ce genre de structures est fondamental en programmation car il permet d'implémenter la plupart des structures de données employées en informatique (listes, files, arbres, etc...).*
- *Ces structures contiennent un ou plusieurs champs du type pointeur vers une structure du même type et non de simples champs du type structure.*

Structures récursives

- *Exemple :*

```
struct personne
{
  char *Nom;
  int Age;
  struct personne *Pere, *Mere;
};
```

- *Déclaration incorrecte car la taille de Pere et Mere n'est pas connue :*

```
struct personne
{
  char *Nom;
  int Age;
  struct personne Pere, Mere;
};
```

Passage comme argument

- *La norme ANSI a introduit le transfert direct d'une structure entière comme argument d'une fonction, ainsi que la remontée de cette structure depuis une fonction appelée via une instruction return.*
- *Le passage de paramètres de type structure se fait par valeur.*
- *Pour les compilateurs antérieur à la norme ANSI, comme le C K&R, il n'est pas possible de passer en paramètre une structure, il faut utiliser un pointeur sur cette structure.*

Passage comme argument

```
struct personne
{
  int jour, mois, annee;
}

int cmp_date(struct date d1, struct date d2)
{
  if (d1.annee > d2.annee)
            return (APRES);
  if(d1.annee < d2.annee)
            return (AVANT);
… /* comparaison portant sur mois et jour */

}
int main()
{
  struct date1, date2;

  if(cmp_date(date1,date2)==AVANT)
     …
}
```

typedef

- *Pour simplifier les écritures et éviter de répéter systématiquement struct personne, on utilise typedef pour définir les structure ou union comme des nouveaux types.*
- ***Exemple :***

```
typedef struct personne
{
    char * Nom;
    int Age;
} Personne;

int main()
{
  Personne p;
  …
}
```

Neuvième partie

Le préprocesseur

Arguments de la ligne de commande

Fonction avec un nombre variable de paramètres

Introduction

- *Le préprocesseur est un programme standard qui effectue des modifications sur un texte source.*
- *Il modifie le source d'après les directives données par le programmeur, et introduites par le caractère #. Ces directives peuvent apparaître n'importe où dans le fichier, pas nécessairement au début.*
- *La directive du préprocesseur commence par un # et se termine par la fin de ligne. Si la directive ne tient pas sur une seule ligne, on peut l'écrire sur plusieurs en terminant les premières lignes par \ qui annule le retour chariot.*

Introduction

- ***Le préprocesseur peut rendre plusieurs services :***
  - *l'inclusion de fichier*
  - *la substitution de symboles*
  - *le traitement de macro-instructions*
  - *la compilation conditionnelle*

L'inclusion de fichier source

- ***La commande #include du préprocesseur permet d'inclure des fichiers source.***
- ***Lorsque le préprocesseur rencontre la commande #include, il remplace cette ligne par le contenu du fichier.***
- ***Syntaxe***
  - *#include <nom_de_fichier>*
    - inclusion d'un fichier système, recherché dans les répertoires systèmes connus du compilateur
  - *#include "nom_de_fichier"*
    - inclusion d'un fichier utilisateur, recherché dans l'espace des fichiers de l'utilisateur.

Substitution de symboles

- ***#define nom reste-de-la-ligne***
  - *le préprocesseur remplace toute nouvelle occurrence de nom par reste-de-la-ligne dans toute la suite du source.*
- ***Exemple :***
  - *#define NB_COLONNES 80*
    #define NB_LIGNES 24
    *#define TAILLE_TAB NB_LIGNES * NB_COLONNES*
- ***Exemple de mauvaise utilisation (mais correct !) :***
  - *#define IF if(*
    #define THEN){
    #define ELSE }else{
    ...

Constantes prédéfinies

- **`__LINE__`**
  - *numéro de la ligne courante du programme*
- **`__FILE__`**
  - *nom du fichier source en cours de compilation*
- **`__DATE__`**
  - *date de compilation*
- **`__TIME__`**
  - *heure de compilation*
- **`__STDC__`**
  - *1 si le compilateur est ISO, 0 sinon*

Macro-instructions

- ***Une définition de macro-instruction permet une substitution de texte paramétrée par des arguments.***
- ***Syntaxe :***
  - *#define nom(liste-parametres-formels) reste-de-la-ligne*
- ***La liste de paramètres formels est une liste d'identificateurs séparés par des virgules.***
- ***Il n'y a pas d'espace entre le nom et (***
- ***Toute occurrence ultérieure de nom sera un appel de la macro et devra avoir la forme :***
  - *nom(liste-parametres-effectifs)*

Macro-instructions

- *Lors de l'appel, le préprocesseur remplace l'ensemble nom de la macro et liste des paramètres effectifs parenthésés par la chaîne reste-de-la-ligne dans laquelle toutes les occurrences des paramètres formels sont remplacées par les paramètres effectifs.*

Macro-instructions

- *Exemple :*

```
#define min(a,b) ((a)<(b) ? (a):(b))
#define max(a,b) ((a)>(b) ? (a):(b))

int main()
{
  int i,j,borne_inferieure,borne_superieure;
  …
  borne_inferieure = min(i,j);
  borne_superieure = max(i,j);
}
```

- *Remplacé par :*

```
  borne inferieure = ((i)<(j) ? (i):(j))
  borne superieure = ((i)>(j) ? (i):(j))
```

Macro-instructions

- ***Si on ne met pas de parenthèses sur les paramètres formels, il peut y avoir des erreurs.***
- ***Exemple :***

```
#define carre(a) (a*a)
…
x = carre(y+1) sera transformé en x =
(y+1*y+1)
soit x = 2*y+1 !!!
```

- ***On recommande donc de toujours utiliser deux règles dans la définition d'une macro-instruction :***
  - *parenthéser les occurrences des paramètres formels;*
  - *parenthéser le corps complet de la macro.*

Macro-instructions

- *Attention aux effets de bord inattendus sur les paramètres effectifs.*
- *Exemple :*
  #define carre(a) ((a)(a))*
- *L'utilisation de carre(x++) sera transformé en ((x++)*(x++)), l'opérateur ++ sera donc appliqué deux fois à x.*

Macro-instructions

- *Macros contenant des instructions :*

```
#define F(x) if(x>10){printf("erreur\n");}
```

- *Appel dans un if :*

```
if (…)
  F(i)
else
  …
```

- *Équivaut à :*

```
if (…)
 if(i>10){printf("erreur\n");}
else
  …
```

- *Problème : le else va être raccroché au if de la macro*

Compilation conditionnelle

- *Les mécanismes de compilation conditionnelle ont pour but de compiler ou d'ignorer des ensembles de lignes, le choix étant basé sur un test exécuté à la compilation.*

#if

- *Syntaxe :*

```
#if expression
  ensemble-de-lignes
#endif
```

- *ensemble-de-lignes est une suite de lignes quelconques.*
- *L'évaluation de expression à lieu au moment de la compilation.*
- *Si expression est vraie ensemble-de-lignes est compilé.*

#if #elif #else

```
#if expression
  ensemble-de-lignes1
#else
  ensemble-de-lignes2
#endif
```

```
#if expression1
  ensemble-de-lignes1
#elif expression2
  ensemble-de-lignes2
#else
 ensemble-de-lignes
#endif
```

#ifdef #ifndef

- *Ces commandes permettent de tester si un symbole a été défini par la commande #define*
- *Exemple :*

```
#define TEST
...
#ifdef TEST
#define VALEUR 1
#else
#define VALEUR 2
#endif
```

#ifdef #ifndef

- *Ces commandes sont souvent utilisées pour éviter les inclusions multiples d'un fichier d'en-tête.*
- *Exemple :*

```
#ifndef TOTO
#define TOTO
#include "toto.h"
#endif
```

L'opérateur defined

- *L'opérateur defined est un opérateur spécial : il ne peut être utilisé que dans le contexte d'une commande #if ou #elif.*
- *Il permet d'écrire des test portant sur la définition de plusieurs symboles alors que #ifdef ne peut en tester qu'une.*
- *Syntaxe :*
  defined nom ou defined(nom)
- *Exemple :*
  #if defined(SYMBOLE1) || defined(SYMBOLE2)

L'opérateur

- *L'opérateur # ne peut s'employer que dans la définition d'une macro.*
- *Il permet de remplacer un argument formel par un argument effectif transformé automatiquement en chaîne de caractères constante.*
- *Exemple :*
  #define str(s) printf(s)
  str(bonjour) sera remplacé par printf(bonjour)

  #define str(s) printf(#s)
  str(bonjour) sera remplacé par printf("bonjour")

L'opérateur

- *Cet opérateur ne peut s'employer que dans un #define (macro ou substitution de symbole).*
- *Lorsque deux éléments sont séparés par ## ils sont concaténés pour former un nouveau symbole.*
- *Exemple :*
  #define DESPRES "tres fort"
  #define colle(a,b) a##b

  colle(DES,PRES) donnera DESPRES
  donc printf(colle(DES,PRES)) affichera tres fort !

L'opérateur

- *Exemple plus intéressant :*

```
#define trace(s,t) printf("x"#s"=%d,
x"#t"=%s",x##s,x##t)
...
trace(1,2);
sera transformé en
printf("x1=%d, x2=%s", x1,x2);
```

Arguments de la ligne de commande

- ***Il est possible en C de récupérer les arguments passés sur la ligne de commande.***
- ***Quand on appelle la fonction main, celle-ci reçoit deux paramètres :***
  - *Le premier est un entier qui représente le nombre d'arguments présents sur la ligne de commande. On l'appelle généralement* ***argc****.*
  - *Le second est un pointeur vers un tableau de chaînes de caractères qui contiennent les arguments proprement dits. On l'appelle généralement* ***argv****.*
- ***Par convention, argv[0] est le nom du programme exécutable et donc argc vaut au moins 1.***

Arguments de la ligne de commande

- *Exemple :*

```
echo     Bonjour    Grand            Maître
argv[0]  argv[1]    argv[2]    argv[3]
et argc = 4
```

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    while (--argc>0)
        printf("%s%s", *++argv, (argc>1) ? " " : "");
    printf("\n");
    return 0;
}
```

Fonction à nombre variable de paramètres

- ***Il arrive que le nombre de paramètres d'une fonction puisse varier d'un appel à l'autre.***
- ***C'est le cas pour printf et scanf :***
  - *printf("Bonjour\n");*
  - *printf("Bonjour %s\n",nom);*
  - *printf("Bonjour %s et %s\n",nom1,nom2);*
  - *Le premier paramètre de **printf** est obligatoire, les suivants sont optionnels :*

    *int printf(const char * format, identificateurs ....)*

Fonction à nombre variable de paramètres

- *Une fonction peut avoir en plus de ces paramètres obligatoires, des paramètres optionnels.*
- *Pour déclarer une telle fonction, les paramètres obligatoires apparaissent en premier et l'unité lexicale … (3 points à la suite) représente les paramètres optionnels.*
- *Exemple :*

```
int printf(const char * format, ...)
```

Fonction à nombre variable de paramètres

- ***Dans le corps de la fonction on ne dispose pas de nom pour désigner les paramètres optionnels.***
- ***L'accès aux paramètres optionnels se fait en utilisant un ensemble de macros définies dans la bibliothèque standard (inclure stdarg.h) :***
  - *va_list*
  - *va_start*
  - *va_arg*
  - *va_end*

Fonction à nombre variable de paramètres

- *va_list permet de déclarer une variable opaque au programmeur, à passer en paramètre aux autres macros.*
- *Cette variable s'appelle traditionnellement ap (pour argument pointer), et a pour but de repérer le paramètre effectif courant.*
- *va_list(ap) déclare la variable ap comme pointeur sur argument.*

Fonction à nombre variable de paramètres

- *va_start initialise le pointeur sur argument, elle doit être appelée avant toute utilisation de la variable ap et de la macro va_arg.*
- *va_start a deux paramètres : la variable ap et le nom du dernier paramètre obligatoire de la fonction.*

Fonction à nombre variable de paramètres

- *va_arg délivre le paramètre effectif courant.*
- *Le premier appel à va_arg délivre le premier paramètre, puis chaque nouvel appel délivre le paramètres suivant.*
- *La macro va_arg admet deux paramètres : la variable ap et le type du paramètre courant.*

Fonction à nombre variable de paramètres

- *va_end doit être appelée après toutes les utilisations de va_arg.*
- *La macro va_end admet un seul paramètre : la variable ap.*
- *Si va_end n'est pas appelée, le comportement du programme sera imprévisible.*

Fonction à nombre variable de paramètres

- *Rien n'est prévu pour communiquer à la fonction le nombre et le type des paramètres effectivement passés : c'est un problème à la charge du programmeur.*

Fonction à nombre variable de paramètres

Exemple :

```
int add(int nb_param, …)
{
  int i,s=0;

  va_list(ap);
  va_start(ap,nb_param);
  for(i=0; i<nb_param; i++)
    s = s + va_arg(ap,int);
  va_end(ap);
  return s;
}
```

Dixième partie

Les Entrée/Sorties

Introduction

- ***En C les E/S relèvent de la bibliothèque standard et pas du langage lui-même qui est totalement découplé de ce sujet.***
- ***`Il existe deux types de fonctions de manipulation de fichiers en C :`***
  - *Les fonctions de bas niveau*
  - *Les fonctions de haut niveau*

Les fonctions de bas niveau

- *Les fonctions de bas niveau sont des fonctions proches du système d'exploitation qui permettent un accès direct aux informations car elles ne sont pas bufferisées.*
- *Elles manipulent les informations sous forme binaire sans possibilités de formatage, et le fichier est identifié par un numéro (de type entier) fixé lors de l'ouverture du fichier.*

Les fonctions de bas niveau

- ***Ouverture d'un fichier :***
  int open (char * nomfichier , int mode, int permissions)
  - ➢ *mode est une combinaison de*
    - O_CREAT : Le fichier est créé s'il n'existe pas
    - O_RDONLY : Ouverture en lecture seule
    - O_WRONLY : Ouverture en écriture seule
    - O_RDWR : Ouverture en lecture/écriture
    - O_APPEND : Ajout en fin
    - O_TRUNC : Si le fichier existe son contenu est détruit.

Les fonctions de bas niveau

- ***open renvoie un entier qui sera utilisé pour accéder au fichier par la suite (read, write et close).***
- ***open renvoie -1 en cas d'erreur.***
- ***La permission définie les droits d'accès au fichier, par ex. : 0777***
- ***Fermeture du fichier : prototype***
  close (int numerofichier)
  - *renvoie 0 si succès*

Les fonctions de bas niveau

- *Ecriture dans le fichier :*
  *int write (int num_fic, void *adre, int nombre_octets)*
- *write écrit nombre_octets octets du tableau de caractères adre dans le fichier désigné par num_fic.*
- *write renvoie le nombre d'octets effectivement écris.*
- *renvoie 0 lorsqu'on atteint une fin de fichier et -1 pour toute autre erreur.*

Les fonctions de bas niveau

- *lecture dans le fichier :*
  *int read (int num_fic, void * adre_stock, int nombre_octets)*
- *read lit nombre_octets octets dans le fichier désigné par num_fic et les range dans le tableau de caractères adre_stock.*
- *read renvoie le nombre d'octet effectivement lus.*
- *renvoie 0 lorsqu'on atteint une fin de fichier et -1 pour toute autre erreur.*

Les fonctions de bas niveau

- ***Fin de fichier :***
  eof (numero_fichier)
  - *renvoie -1 dès que la fin de fichier a été atteinte, 0 sinon.*
- ***Accès direct :***
  lseek (numero_fichier, déplacement, mode)
  - *mode peut prendre les valeurs 0, 1, 2 selon que le déplacement doit être :*
    - 0 : par rapport au début du fichier
    - 1 : par rapport à la position courante
    - 2 : par rapport à la fin du fichier.

Les fonctions de bas niveau

- *Exemple : recopier l'entrée sur la sortie*

```
#include <stdio.h>

main()
{     char tamp[10] ;
      int n;
      while ((n=read(0, tamp, 10))>0)
            write(1, tamp, n);
      return 0;
}
```

Les fonctions de haut niveau

- *Les fonctions de haut niveau sont basées sur les fonctions de bas niveau.*
- *Elles effectuent des E/S bufferisées : utilisation d'un tampon mémoire (accès plus rapide).*
- *Elles permettent une manipulation binaire ou formatée des informations.*

Les fonctions de haut niveau

- *FILE est l'alias d'une structure dont l'utilisateur n'a pas besoin de connaître l'organisation.*
- *FILE contient des informations élaborées relatives au fichier : adresse du buffer, pointeur sur le buffer, numéro du fichier pour les fonctions de bas niveau, etc....*
- *Ce type n'est jamais utilisé en tant que tel, on utilise toujours le type FILE * que l'on appelle un flux.*

Les fonctions de haut niveau

- ***Les trois flux standard stdin, stdout, stderr sont automatiquement ouverts au lancement de tout programme. Ils sont du type FILE * :***
  - ***stdin*** *: entrée standard,*
  - ***stdout*** *: sortie standard,*
  - ***stderr*** *: sortie standard d'erreur.*
- ***Ils peuvent être explicitement redirigés vers un fichier ou un périphérique.***

Les fonctions de haut niveau

- ***Ouverture d'un fichier :***

  FILE * fopen(char * nomfichier, char * mode)

- ***mode :***
  - *"r" : ouverture en lecture au début du fichier (erreur si le fichier n'existe pas)*
  - *"w" : ouverture en écriture au début du fichier (création du fichier s'il n'existe pas, écrasement s'il existe)*
  - *"a" : ouverture en ajout à la fin du fichier (création du fichier s'il n'existe pas)*
  - *"r+" : ouverture en lecture et écriture (erreur si le fichier n'existe pas)*
  - *"w+" : ouverture en lecture et écriture (création du fichier s'il n'existe pas, écrasement s'il existe)*
  - *"a+" : ouverture en lecture et en ajout à la fin (création du fichier s'il n'existe pas)*

Les fonctions de haut niveau

- *Fermeture d'un fichier :*
  *int fclose (FILE * flux)*
- *flux est un pointeur sur FILE et détermine le flux (ou fichier) à fermer. Cette valeur a été obtenu en retour d'un appel à la fonction open.*
- *fclose renvoie 0 si tout c'est bien passé et EOF sinon.*

Les fonctions de haut niveau

- *fflush :*
  *int fflush(FILE *flux);*
- *fflush provoque l'écriture physique immédiate du tampon à la demande et non lorsque le système le décide.*
- *fflush renvoie 0 si tout c'est bien passé et EOF sinon.*

Les fonctions de haut niveau

- *Ecriture "brutale" :*
  *size_t fwrite (const void *ptr, size-t taille, size_t nobj, FILE * flux) ;*
- *fwrite écrit les nobj objets de taille taille octets se trouvant à l'adresse ptr dans le flux flux.*
- *fwrite renvoie le nombre d'objets écrits si tout c'est bien passé.*

Les fonctions de haut niveau

- *Lecture "brutale" :*
  *size_t fread (void * ptr, size-t taille, size_t nobj, FILE * flux);*
- *fread lit les nobj objets de taille taille octets se trouvant dans le flux flux à l'adresse ptr.*
- *fread renvoie le nombre d'objets lus.*

Les fonctions de haut niveau

- ***Accès direct :***
  int fseek (FILE *flux, long offset , int mode)
- ***fseek fait pointer le pointeur de fichier associé au flux flux offset octets plus loin.***
- ***mode peut prendre les valeurs 0, 1, 2 selon que le déplacement doit être :***
  - *0 : par rapport au début du fichier*
  - *1 : par rapport à la position courante*
  - *2 : par rapport à la fin du fichier.*

Les E/S non-formatées

- ***Lecture/écriture d'un caractère :***
  - *int fgetc(FILE *flux);*
  - *int fputc(int c, FILE *flux);*
- ***Idem en macros :***
  - *int getc(FILE *flux)*
  - *int putc(int c, FILE *flux);*
- ***fgetc et getc renvoient un caractère provenant de flux et incrémente le pointeur de fichier de flux afin qu'il pointe sur le caractère suivant.***
- ***fputc et putc écrivent le caractère c dans le flux flux.***

Les E/S non-formatées

- ***`int getchar();`***
  - *macro définie comme getc(stdin)*
- ***`int putchar(int ch);`***
  - *macro définie comme putc(c,stdout)*

Les E/S non-formatées

- ***Lecture/Ecriture d'une chaîne de caractères***
  - *char *fgets(char *s, int n, FILE *flux);*
    - **fgets** lit les caractères depuis le flux *flux* et les place dans la chaîne *s*. La fonction cesse soit à la lecture d'un \n soit quand n-1 caractères ont été lus. La fin de la chaîne de caractères est marquée par l'ajout d'un caractère nul (\0).
    - **fgets** renvoie s si tout c'est bien passé, null sinon.
  - *int fputs(const char *s, FILE *flux);*
    - **fputs** écrit la chaîne s dans *flux* le caractère \0 n'est pas copié.
    - **fputs** renvoie le dernier caractère écrit si tout c'est bien passé, EOF sinon.
- ***char *gets(char *s) et int puts(const char *s) travaillent respectivement avec stdin et stdout.***

Les E/S formatées : écriture

- *Prototypes :*
  ```
  int fprintf(FILE * flux, const char * format,
  identificateurs ....)
  int printf(const char * format,
  identificateurs ....)
  ```
- *fprintf convertit ses arguments d'après les spécifications de son format et écrit le résultat dans le flot de sortie flux.*
- *printf convertit ses arguments d'après les spécifications de son format et écrit le résultat dans le flot de sortie standard stdout.*
- *fprintf et printf retournent le nombre de caractères écrits ou une valeur négative en cas d'erreur.*

Les E/S formatées : écriture

- ***La chaîne de caractères format contient deux types d'informations :***
  - *des caractères ordinaires qui sont recopiés sur le flot de sortie (flux ou stdout)*
  - *des spécifications de conversions dont chacune provoque la conversion et l'impression des arguments suivants de **fprintf** et **printf**.*

Les E/S formatées : écriture

- ***Chaque spécification de conversion commence par % et se termine par un caractère de conversion. Entre les deux on peut placer dans l'ordre :***
  - ***des drapeaux*** *qui modifient la spécification :*
    - \- cadre l'argument à gauche
    - \+ imprime systématiquement le signe du nombre
    - espace : si le premier caractère n'est pas un signe, place un espace au début.
    - 0 : pour les conversion numériques, complète le début du champ par des 0.

Les E/S formatées : écriture

- ***Un nombre*** *qui précise la largeur minimum du champ d'impression.*
- ***Un point*** *qui sépare la largeur du nombre de décimales.*
- ***Un nombre*** *indiquant la précision*
- ***Une lettre*** *h (short ou unsigned short), l (long ou unsigned long) ou L (long double) qui modifie la largeur du champ*

- ***La largeur et la précision peuvent être données en paramètre à printf en mettant * à leur place. Les paramètres suivant le format seront alors convertis en entiers et utilisés.***

Les E/S formatées : écriture

- ***Les spécifications de conversions sont :***

Caractères	Type de l'argument
d, i	int ; notation décimale signée
o	int ; notation octale non signée
x, X	int ; notation hexadécimale non signée
u	int ; notation décimale non signée
c	int ; un seul caractère après conversion en unsigned char
s	char * ; les caractères sont imprimés jusqu'à \0

Les E/S formatées : écriture

Caractères	Type de l'argument
f	double ; notation décimale de la forme [-]mmm.ddd la précision par défaut est 6
e, E	double ; notation exponentielle
g, G	double ; l'impression se fait suivant le format %e si l'exposant est inférieur à –4 ou supérieur à la précision, sinon %f
p	void * ; écrit l'argument sous forme de pointeurs
n	int *; le nombre de caractères écrits jusqu'à présent par cet appel à printf est écrit dans l'argument
%	affiche un %

Les E/S formatées : lecture

- ***prototype :***

```
int fscanf (FILE * flux, const char *
format, adresses ...)
int scanf (const char * format,
adresses ...)
```

- ***le format contient :***
  - *des espaces ou des caractères de tabulations qui sont ignorés*
  - *des caractères ordinaires (différents de %), dont chacun est supposé s'identifier au caractère suivant du flot d'entrée.*
  - *des spécifications de conversion du même type que pour **printf***

Onzième partie

La compilation séparée

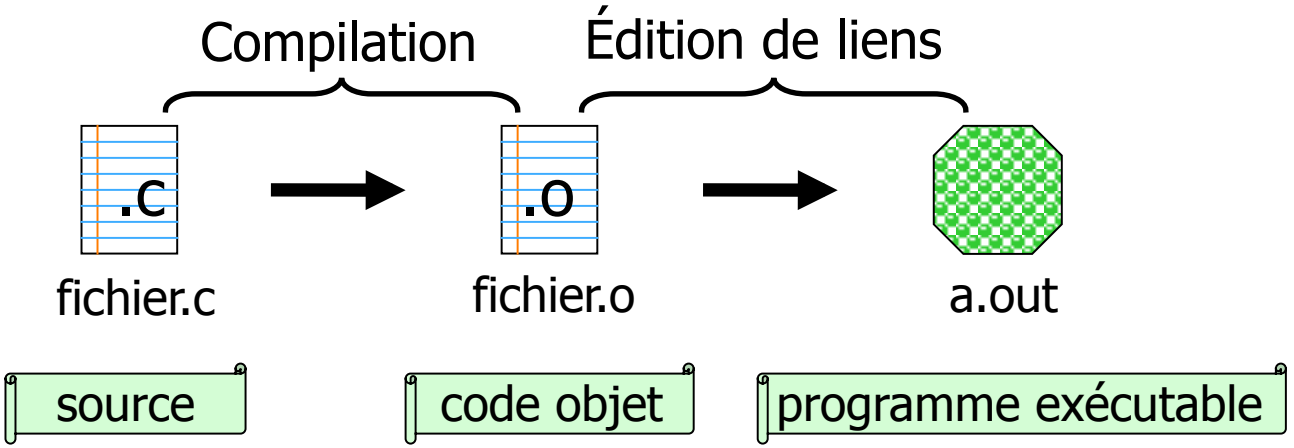
Un programme simple

```
#include <stdio.h>


int main()
{
  printf("Bonjour Monde\n");
  return 0;
}
```

Compilation simple

- ***Phase de compilation :***
  - *entrée : fichier source (.c)*
  - *sortie : fichier objet (.o)*
- ***Phase d'édition de liens :***
  - *entrée : fichier objet (.o)*
  - *sortie : fichier exécutable*



La compilation séparée

- *Lorsqu'un programme devient grand, il devient intéressant de diviser le code source en plusieurs fichiers .c, il n'est plus nécessaire de recompiler l'ensemble du source à chaque fois mais uniquement ce qui a été modifié.*
- *De plus, vous pouvez regrouper un ensemble de fonctions et l'isolé dans un fichier source ce qui permet de pouvoir les réutiliser dans d'autres projets.*

La compilation séparée

- ***En C un module doit être implanté à l'aide de deux fichiers :***
  - *les définitions dans un fichiers source (.c)*
  - *les déclarations des variables et fonctions exportées (utilisables dans d'autres modules) dans un fichier d'en-tête (.h)*
- ***Un module utilisant les fonctions et/ou les variables d'un autre module doit "inclure" le fichier d'en-tête de celui-ci (pour avoir les déclarations).***

Exemple

```
void f1();
void f2(int,int);
```

fichier.h

```
void f1()
{
  ...
}

void f2(int a, int b)
{
  ...
}
```

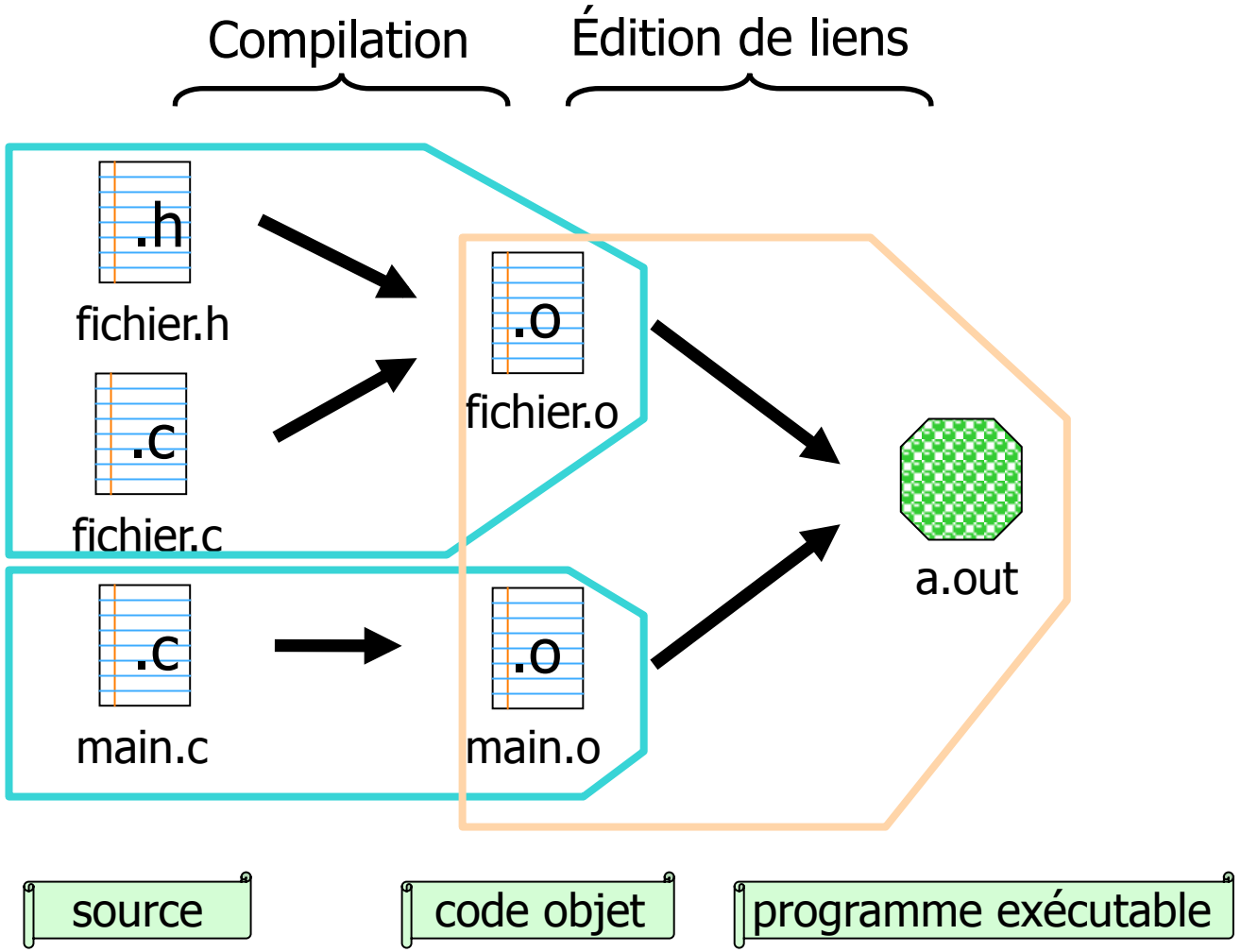
fichier.c

```
#include "fichier.h"

int main()
{
  f1();
  f2(2,3);
  return 0;
}
```

main.c

La compilation séparée



La compilation

- ***La phase de compilation transforme le code source (texte en langage C) en code objet (code non encore exécutable).***
- ***Un fichier objet (.o) est un fichier contenant :***
  - *le code compilé du source correspondant*
  - *une table des variables et des fonctions exportées définies dans le source*
  - *une table des variables et des fonctions qui sont utilisées mais non définies*

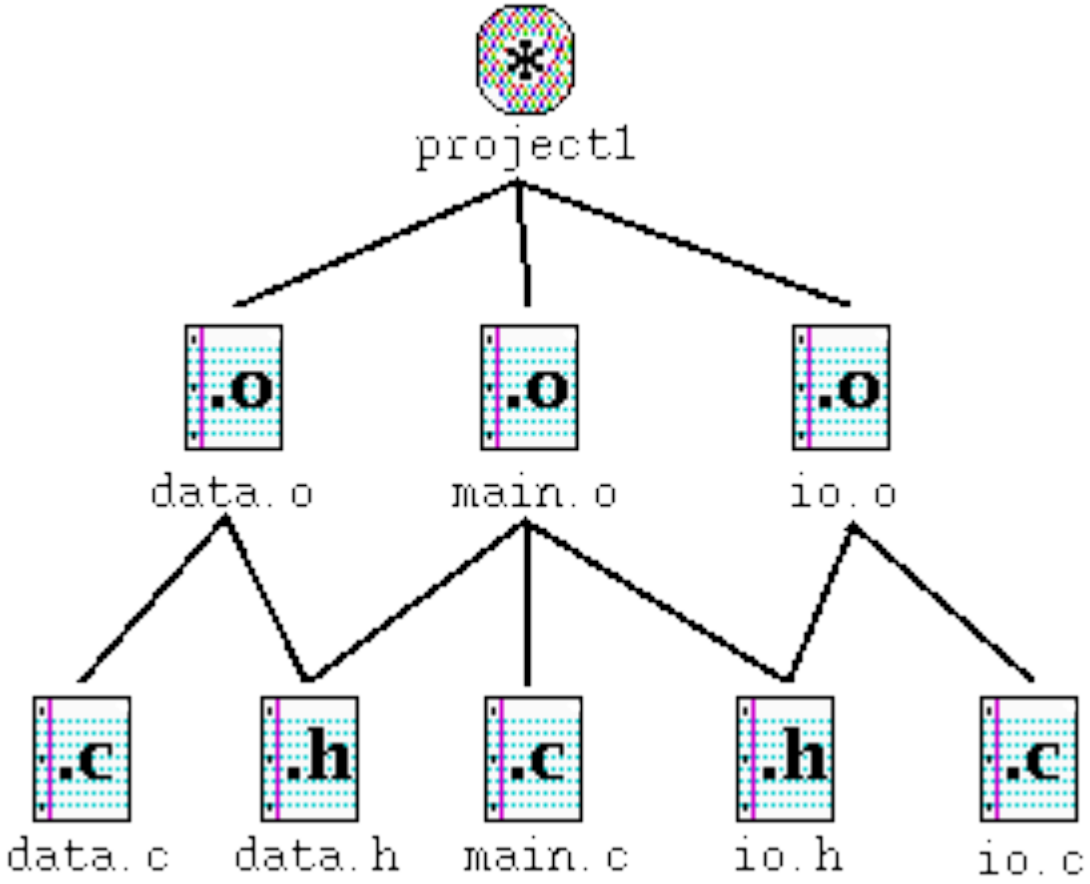
L'édition de liens

- *Cette phase finale crée le programme exécutable.*
- *Elle fait le lien entre les déclarations et appels des variables et fonctions non définies dans un module avec les variables et fonctions définies dans les modules inclus.*

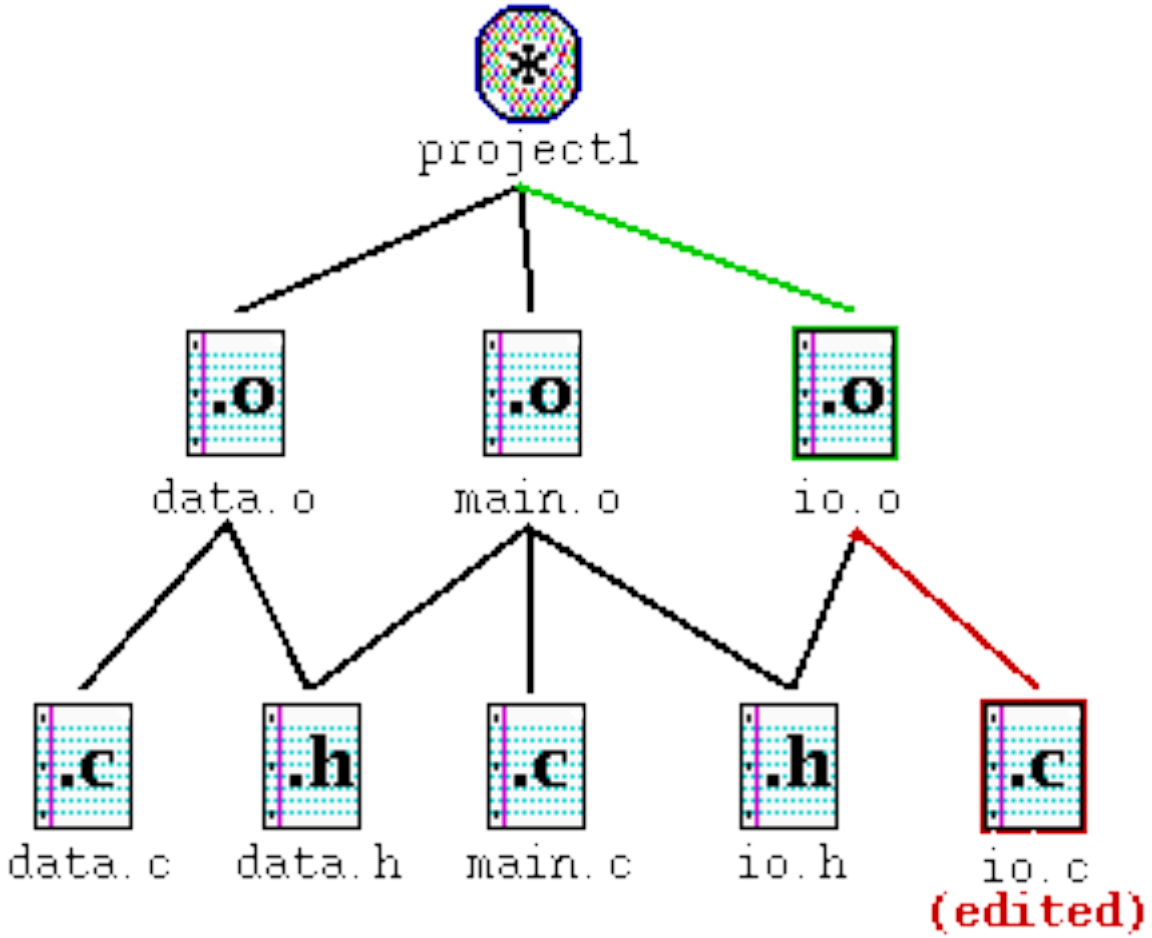
Exemple

- ***Compilation séparée "à la main" avec le compilateur gcc :***
  - *gcc -c fichier.c*
    gcc -c main.c
  - *l'option -c permet de compiler et de créer le fichier objet mais ne fait pas l'édition de lien.*
- ***Édition de liens et création du programme exécutable :***
  - *gcc fichier.o main.o*

Le graphe de dépendances



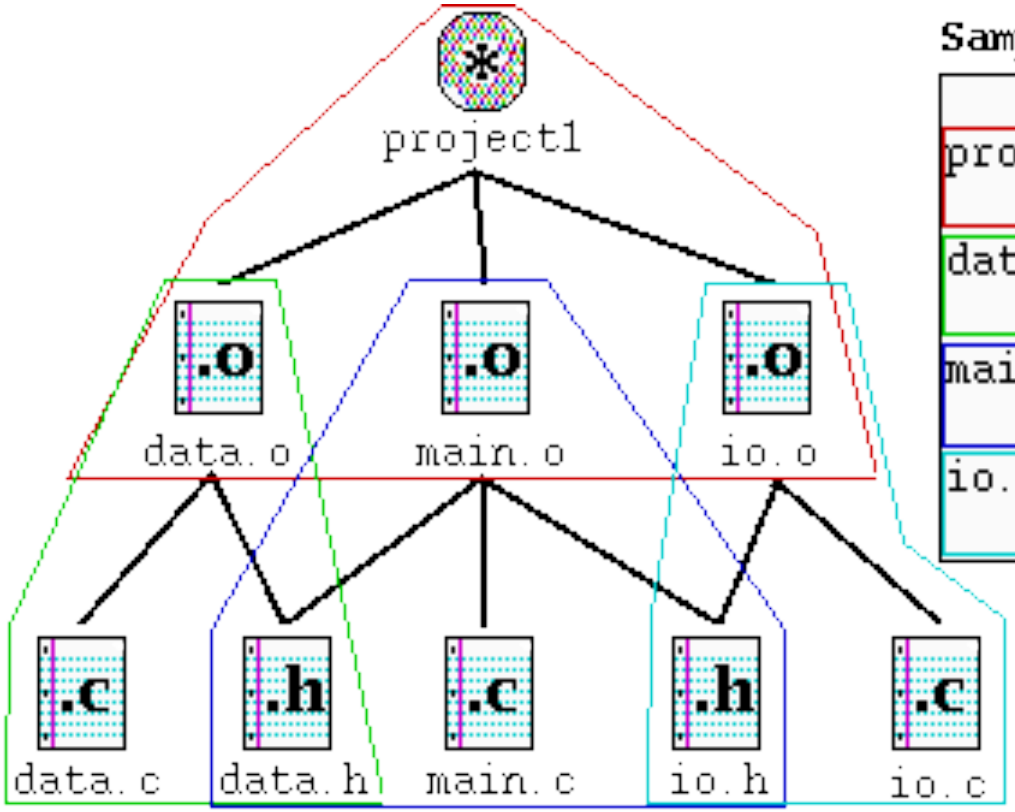
Le graphe de dépendances



Make

- Garder trace des dépendances des fichiers entre eux et réaliser manuellement les phases de compilation serait fastidieux.
- Un utilitaire générique pour résoudre ce genre de problèmes de dépendances existe : **make**.
- La commande **make** fonctionne en suivant des règles qui définissent les actions à effectuer.
- Il cherche par défaut ces règles dans un fichier texte nommé **Makefile**.

Makefile



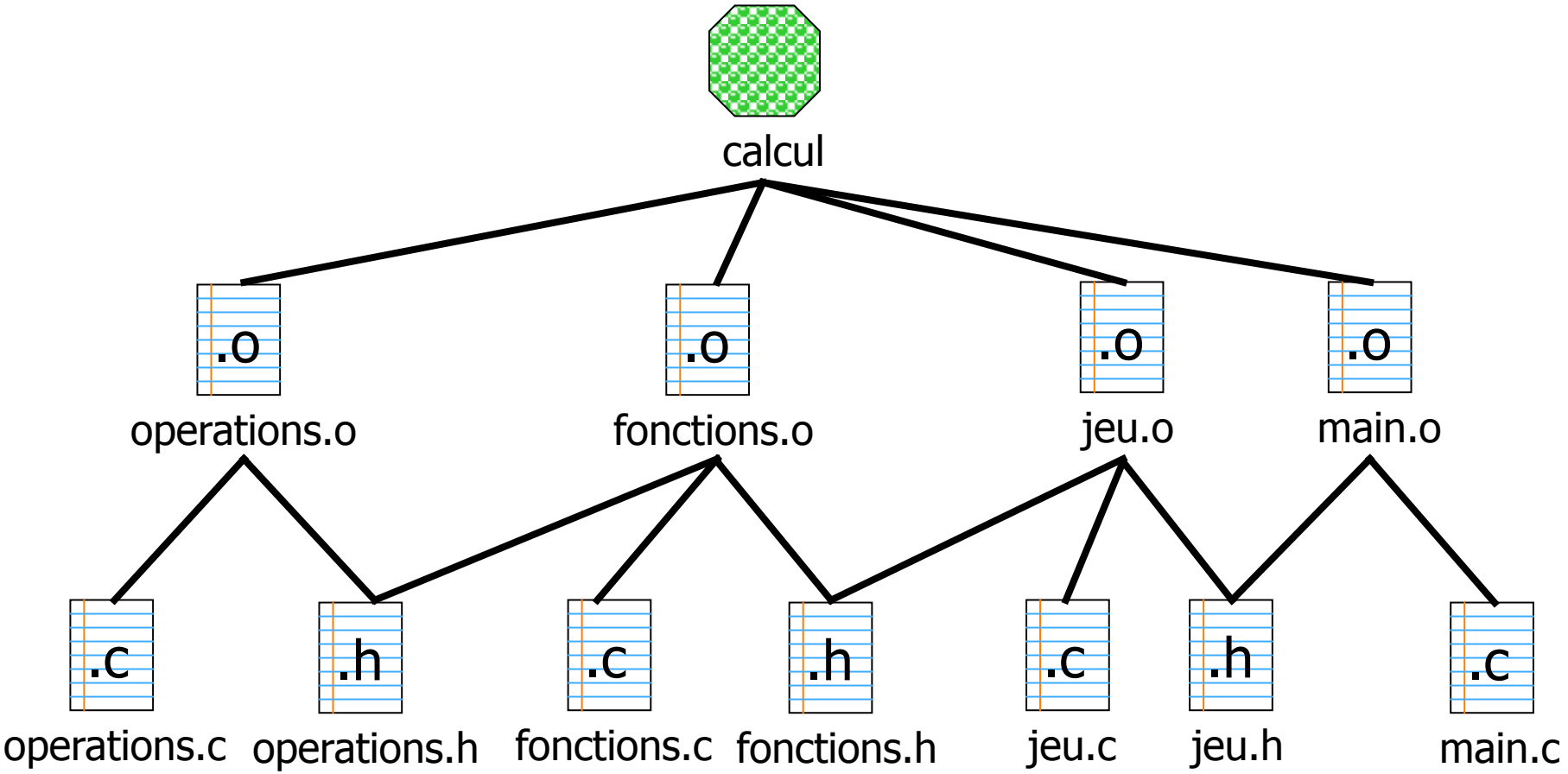
Sample Makefile

```
project1: data.o main.o io.o
    cc data.o main.o io.o -o project1
data.o: data.c data.h
    cc -c data.c
main.o: data.h io.h main.c
    cc -c main.c
io.o: io.h io.c
    cc -c io.c
```

Syntaxe du makefile

- ***Syntaxe d'une règle d'un fichier makefile :***
  - *1ère ligne =nom_de_la_cible : liste_des_fichiers*
  - *2ème ligne = [TAB] commande_pour_fabriquer_la_cible*
    Attention ! Le premier caractère de cette ligne dans un ***Makefile*** *est une tabulation et non des espaces.*

Exemple


calcul
.o
operations.o
.o
fonctions.o
.o
jeu.o
.o
main.o
.c
operations.c
.h
operations.h
.c
fonctions.c
.h
fonctions.h
.c
jeu.c
.h
jeu.h
.c
main.c

Douzième partie

Complexité des programmes

Introduction

- ***Lorsque des programmes doivent être exécutés sur de nombreuses données, le temps d'exécution devient un critère vital.***
- ***Celui-ci dépend de différents paramètres, comme***
  - *la puissance du microprocesseur de la machine ;*
  - *la vitesse de lecture/écriture du disque dur ;*
  - *la taille données ;*
  - *etc.*

Introduction

- *La taille des données peut contrairement aux autres critères, être étudié sur le programme (ou l'algorithme) lui-même, indépendamment de l'implantation.*
- *On essaie alors d'évaluer le temps d'exécution par exemple en nombre d'instructions exécutées.*
- *Ce temps sera en général exprimé sous la forme d'une fonction, qui dépendra notamment de n, nombre des données.*

Introduction

- *Le paramètre intéressant dans cette fonction est son ordre : il permet d'estimer comment va évoluer le temps de calcul avec la taille des données.*
- *Ceci permet de tester le programme à petite échelle, tout en ayant une idée du temps que cela mettra sur le problème en taille réelle.*
- *C'est cet ordre que l'on appel complexité de l'algorithme*

Ordre d'une fonction

- ***Lorsque l'on cherche à évaluer l'ordre d'une fonction, on "supprime'' tout ce qui devient négligeable à l'infini. On donne alors l'ordre sous la forme o(f), où f est une fonction "simple''.***
- ***Exemple :***
  - *la fonction 3n(log(n) + 2n-1) est en o(3n log(n))*

Différentes mesures d'une complexité

- ***La complexité d'un algorithme est en général exprimée en fonction de la taille des données.***
- ***On distingue cependant là encore plusieurs types. Les principaux sont :***
  - *Complexité moyenne*
  - *Complexité au pire*

Différentes mesures d'une complexité

- *Un programme peut contenir des instructions de test, fonction des valeurs des données (inconnues a priori)*
- *L'évaluation de ces tests induira l'exécution de séquences d'instructions de complexité variable.*
- *La "complexité à l'exécution'' dépend donc des données elles-mêmes.*

Différentes mesures d'une complexité

- *On évalue donc en général ce que l'on appelle la complexité moyenne.*
- *Cependant, pour vérifier que dans certains cas particuliers, on n'atteint pas des temps rédhibitoires, il peut arriver que l'on calcule aussi la complexité maximale.*

Que mesurer ?

- ***Chaque instruction a son propre temps d'exécution.***
  - *l'affectation est quasi-instantanée ;*
  - *l'addition et la soustraction sont tres rapides ;*
  - *la multiplication un peu moins ;*
  - *la division est relativement lente ;*
  - *etc.*

Que mesurer ?

- ***On ne peut donc pas considérer chaque instruction de la même façon.***
- ***Donc là aussi, on aura tendance à ne pas compter ce qui est négligeable :***
  - *Par exemple, dans un problème qui fait beaucoup de calculs, on négligera les affectations.*
  - *Par contre, dans un algorithme de tri, qui ne fait que des tests et affectations, cela serait ridicule !*

Que mesurer ?

- *On peut ne pas forcément faire un décompte instruction par instruction.*
- *Ainsi, on s'intéresse souvent à des blocs d'instructions toujours exécutées d'un bout à l'autre.*
- *Par exemple, dans des algorithmes donnés sous forme d'une boucle, on pourra évaluer le nombre de fois où le corps de la boucle est effectué.*

Exemple

```
for(i=0;i<N;i++)
  for(j=0;j<N;i++)
    tab[i][j]=i*j;
```

- ***Dans ce cas, il y aura à coup sûr N*N instructions de réaliser.***
- ***La complexité est donc en o(n^2)***

Exemple

```
int max, i, temp, trie = 0;
for (max = taille; max > 1 && !trie ; max--)
{
  trie = 1;
  for( i = 0 ; i < max-1 ; i++)
    if (tableau[i+1] < tableau[i])
   {
      temp = tableau[i];
      tableau[i] = tableau[i+1];
      tableau[i+1] = temp;
      trie = 0;
   }
}
```

Exemple

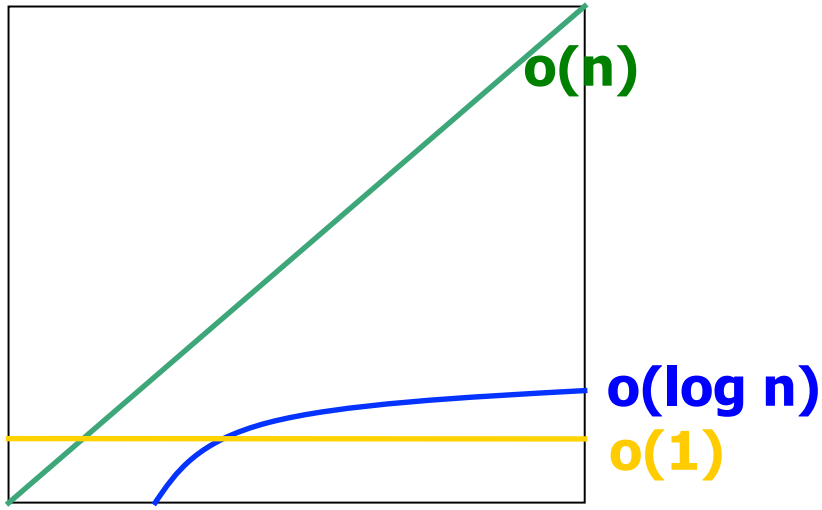
- *Dans ce programme, on prendra comme mesure de complexité le nombre de fois où le corps de la boucle interne est exécuté, et on évalue la complexité dans le pire des cas.*
- *Au pire, le tableau ne se révèle trié qu'à la dernière étape.*
- *Or à la première étape, la boucle interne est exécutée n-1 fois. À l'étape suivante, n-2 fois, ... Et finalement, 1 fois à la dernière étape.*

Exemple

- **D'où la complexité au pire :**
  - $f(n) = \sum_{[i=1,i=n-1]} i$
- **Soit :**
  - $f(n) = (n(n-1))/2$
- **Soit :**
  - $f(n) \approx n^2/2$

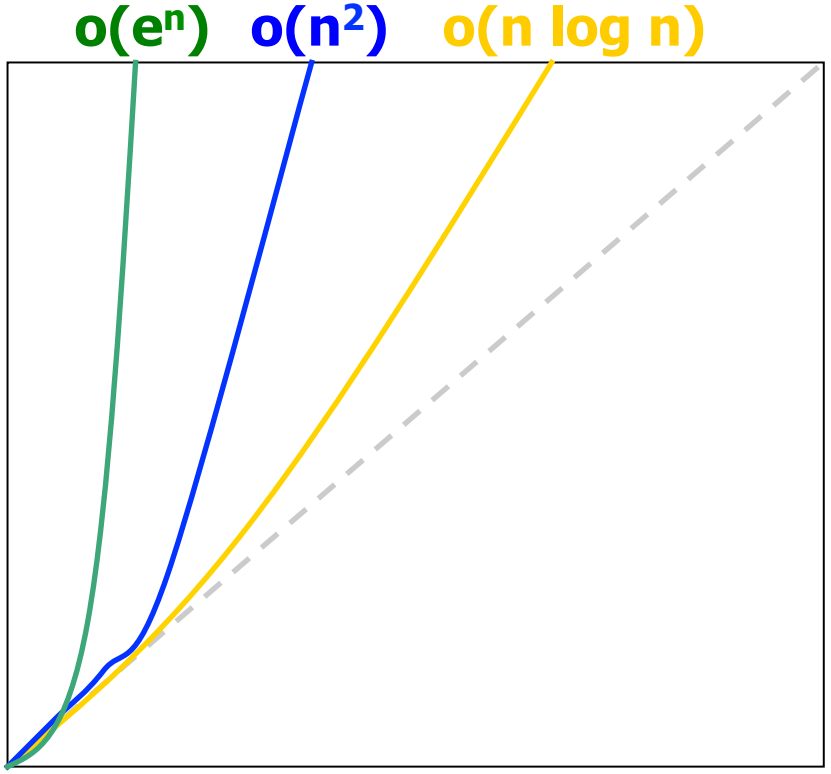
Différentes complexités

- *o(1) : temps d'exécution constant (indépendant du nombre d'éléments traités)*
- *o(log n) : complexité logarithmique (temps d'exécution proportionnel au log du nombre d'éléments traités)*
- *o(n) : complexité linéaire (temps d'exécution proportionnel au nombre d'éléments traités)*



Différentes complexités

- *o(n log n) : complexité de l'ordre de n log n*
- *o(n^2) : complexité quadratique*
- *o(e^n) : complexité exponentielle*



Treizième partie

Les Tris

Méthodes de tris

1- Introduction

- *spécification, clés structurées,*
- *organisation de la mémoire, critères d'efficacité*

2- Tris simples

- *par sélection, par insertion , tri shell*

3- Tri rapide

4- Tri fusion

5- Tri par tas

6- Conclusion

Spécification du tri

- ***données :***
  - *une liste de n éléments*
  - *à chaque élément est associée une clé*
  - *l'ensemble des clés est totalement ordonné*
- ***résultat :***
  - *une liste obtenue par permutation des éléments de la précédente*
  - *où les clés sont croissantes quand on parcourt la liste séquentiellement*
- ***clés structurées***
  - *une clé est un ensemble de valeurs*
  - *une clé est décomposée en clé principale, clé (s) secondaire(s)*

Organisation de la mémoire pour faciliter le tri

- ***éléments volumineux :***
  pour ne pas avoir à déplacer les éléments
  - *trier une liste où le stockage est indirect (tri indirect) on trie une listes de pointeurs sur les éléments*
- ***tri interne : tous les éléments tiennent en mémoire centrale***
  - *2 solutions*
    - A coté : dans un autre tableau
    - Sur place : dans le tableau
- ***tri externe : le fichier est chargé en mémoire par morceaux***
  - *pb : minimiser les entrées sorties*

Critères d'analyse (pour le tri interne)

- ***deux critères de base :***
  - *nombre de comparaisons entre clés*
  - *nombre de transferts (ou affectations) d'éléments*
- ***place mémoire nécessaire***
  - *tri sur place (o(1))*
  - *tri à côté (en o(n))*
  - *tri récursif : utilise une pile*
- ***stabilité***
  - *conserve l'ordre originel pour éléments de clés égales*
- ***progressivité***
  - *à l'étape k trie les k premiers éléments de la liste*
- ***configuration particulière des données : aléatoires, triées, inversées, égales, presque triées***

Routines de tri

- ***tri tout terrain prêt à l'emploi codé en langage machine***
- ***exemple. en C qsort de sdtlib.h***
  - *trie par ordre croissant un tableau tab de n objets de taille t donnée et de fonction de comparaison cmp donnée*
  - *la fonction de comparaison comporte deux arguments et retourne*
    - un nombre négatif si le premier argument est inférieur au premier
    - 0 si les 2 arguments sont égaux
    - un nombre positif sinon

Méthodes de tris

1- Introduction

- *spécification, clés structurées,*
- *organisation de la mémoire, critères d'efficacité*

✓ ***2- Tris simples***

- *par sélection*
- *par insertion*
- *tri shell*

3- Tri rapide

4- Tri fusion

5- Tri par tas

6- Conclusion

Méthodes simples de tri

intérêts :

- ***pédagogiques (terminologie et mécanismes fondamentaux)***
- ***tris simples plus efficaces***
  - *sur des petites listes (moins de 50 éléments)*
  - *sur des listes presque triées ou contenant un grand nombre de clés égales*
- ***certains tris sophistiqués***
  - *prolongent ces tris simples*
  - *sont améliorés en faisant appel à des tris simples sur une partie des données*

Méthodes simples de tri

- ***complexité :***
  - *en temps : en o(n^2) en moyenne et au pire des cas*
  - *en espace : en o(1) (tris sur place)*
- ***sauf tri shell, ne conviennent pas pour un nombre important de données***
- ***faciles à comprendre et à mettre en œuvre***

Tris par sélection

- ***principe :***
  - *sélectionner le minimum dans la liste à trier et l'échanger avec l'élément à la première place*
  - *trier la liste des éléments restants suivant le même principe*
- ***progressivité :***
  à l'étape i les i plus petits éléments de la liste sont à leur place définitive
- ***recherche du minimum : 2 méthodes***
  - *sélection ordinaire*
  - *tri à bulles*
- ***complexité en moyenne et dans le pire des cas***
  - *nb de comparaisons : pour les 2 tris en $o(n^2)$*
  - *nb d'échanges : pour les 2 tris en $o(n^2)$*

Tri par sélection

- ***0 <= i <= n - 1 ,***
  - *le tableau étant trié jusqu'à i - 1 et comprenant les i plus petits éléments*
  - *sélectionner le minimum min entre i et n - 1*
  - *échanger tab[i] et min*
- ***algorithme récursif***
  tri-select-récursif(tab, i, n, cmp)
  si i < n - 1
  - *indmin = i*
  - *pour j allant de i + 1 à n - 1*
    si cmp(tab[j], tab[indmin]) < 0, indmin = j
  - *échanger (tab [i], tab[indmin])*
  - *tri-select-récursif(tab, i + 1, n)*

  fin

Tri par sélection

algorithme itératif
tri-select-itératif(tab, n, cmp)
pour i allant de 0 à n - 2

- *indmin = i*
- *pour j allant de i + 1 à n - 1*
  si cmp(tab[j], tab[indmin]) < 0, indmin = j
- *échanger (tab [i], tab[indmin])*

fin

Tri par sélection

Donnée		101	115	30	63	47	20
Sélection	20						
Placement		**20**	115	30	63	47	101
Sélection	30						
Placement		**20**	**30**	115	63	47	101
Sélection	47						
Placement		**20**	**30**	**47**	63	115	101
Sélection	63						
Placement		**20**	**30**	**47**	**63**	115	101
Sélection	101						
Placement		**20**	**30**	**47**	**63**	**101**	115

Tri bulles

- ***consiste à faire remonter petit à petit les éléments les plus légers vers le haut du tableau***
- ***commencer par la fin et effectuer des échanges chaque fois que 2 éléments sont mal placés***
- ***à la fin du i ième parcours, les i premiers éléments du tableau sont triés***
- ***algorithme itératif***
  tri-bulles-itératif(tab, n, cmp)
  pour i allant de 0 à n - 2
  - *pour j allant de n -1 à i*
    - si cmp(tab[j], tab[j -1]) < 0),
      échanger (tab [j], tab[j-1])

  fin

Tri bulles

- ***amélioration :***
  si à une étape pas d'échange : arrêt le tableau est trié
- ***complexité :***
  - *meilleur cas : tableau trié*
    (n comparaisons, 0 échanges)
  - *pire des cas : ordre inverse*
    (n(n+1) / 2 comparaisons et échanges)
  - *en moyenne Cf. pire des cas*
  - *en général : plus de transferts que la sélection ordinaire*

Tri bulles

Donnée	101	115	30	63	47	20

Échanges

20 <->47

20 <->63

20 <->30

20 <->115

20 <->101

	20	101	115	30	63	47

Échanges

47 <->63

30<->115

30<->101

	20	**30**	101	115	47	63

Échanges

47 <->115

47 <->101

	20	**30**	**47**	101	115	63

Échanges

63 <->115

63 <->101

	20	**30**	**47**	**63**	101	115

Pas d'échanges

Résultat :

	20	**30**	**47**	**63**	**101**	115

Tris par insertion (méthode du joueur de cartes)

- ***trier les i premiers éléments de la liste***
  puis insérer l'élément i + 1 au bon endroit dans les éléments déjà triés
  - ➢ *pour rechercher l'endroit où insérer :*
    - recherche séquentielle
    - recherche dichotomique
  - ➢ *pour insérer : décaler (représentation par tableau)*
- ***tri non progressif***

Tri par insertion

Donnée	**101**	115	30	63	47	20
1ière insertion :	**101**	**115**	30	63	47	20
2ième insertion :	**30**	**101**	**115**	63	47	20
3ième insertion :	**30**	**63**	**101**	**115**	47	20
4ième insertion :	**30**	**47**	**63**	**101**	**115**	20
5ième insertion :	**20**	**30**	**47**	**63**	**101**	**115**

Les éléments en gras sont déjà triés ; l'élément à insérer est souligné

Tri par insertion séquentielle

- ***pour i allant de 0 à n-1***
  - *j = i + 1*
  - *x = Tab [i+1]*
  - *tant que Tab[j-1] > x et (j > 0)*

    échanger(Tab[j], Tab[j-1])

    j --

- ***complexité***
  - *en général : plus coûteux que la sélection ordinaire*
  - *meilleure des cas : liste est presque triée (n comparaisons, 0 échanges)*
  - *pire des cas : fichier en ordre inverse $o(n^2)$*
  - *en moyenne aussi en $o(n^2)$*

Tri par insertion dichotomique

- ***au lieu de parcourir les i premiers élément triés pour faire l'insertion du i + 1 ième, rechercher la place où insérer par dichotomie***
- ***complexité***
  - *nb de transferts est identique à celui de l'insertion séquentielle o(n^2) dans le pire des cas et en moyenne*
  - *nb de comparaisons en o(n log n) (au pire et en moyenne)*
- ***en pratique : n'améliore pas vraiment les performances***

Tri shell

- *lenteur du tri par insertion :*
  les échanges ne portent que sur des voisins
  si le plus petit élément est à la fin il faut n étapes pour le placer définitivement
- *principe du tri shell :*
  éliminer les plus grands désordres pour abréger le travail aux étapes suivantes
- *idée : ordonner des séries d'éléments éloignés de h, h prenant des valeurs de plus en plus petites (jusqu'à 1)*

Tri shell

On prend n = 8 puis la suite des distances $(h_i) = (4, 2, 1)$

À la première étape : on trie les éléments **distants de 4** :

on trie 4 sous-listes de 2 éléments

à l'étape suivante les éléments **distants de 2**

on trie 2 sous-listes de 4 éléments

à la fin les éléments **distants de 1**

on trie 1 liste de 8 éléments

0	1	2	3	4	5	6	7
26	36	6	79	26	45	75	13

increment = 4

26				26			
	36				45		
		6				75	
			13				**79**

résultat :

26	36	6	13	26	45	75	79

increment = 2

6		**26**		26		75	
	13		**36**		45		79

résultat :

6	13	26	36	26	45	75	79

incrément 1

6	13	26	**26**	**36**	45	75	79

Tri Shell

- *Une variante du tri par insertion (D. L. Shell, 1959)*

- *Son principe est d'éviter d'avoir à faire de longues chaînes de déplacements si l'élément à insérer est très petit.*

- *Au lieu de comparer les éléments adjacents pour l'insertion, on les compare tous les ..., 1093, 364, 121, 40, 13, 4, et 1 éléments. (On utilise la suite $u_{n+1} = 3\ u_n + 1$).*

Tri Shell

- ***Quand on finit par comparer des éléments consécutifs, ils ont de bonnes chances d'être déjà dans l'ordre.***
  - *On peut montrer que le tri Shell ne fait pas plus que $O(N^{3/2})$ comparaisons, et se comporte donc bien sur des fichiers de taille raisonnable (5000 éléments).*
- ***On peut prendre tout autre générateur que 3 pour générer les séquences à explorer.***
  - *Pratt a montré que pour des séquences de la forme 2^p 3^q, le coût est $O(n \log^2 n)$ dans le pire cas (mais il est coûteux de mettre cette séquence des 2^p 3^q dans l'ordre).*

Simulation Shell

Idée* : *faire un tri par insertion

pour les sous-suites

t[0], t[0+h], t[0+ 2h], t[0+3h], …

t[1], t[1+h], t[1+ 2h], t[1+3h], …

t[2], t[2+h], t[2+ 2h], t[2+3h], …

où h = …, 121, 40, 13, 4, 1

Simulation Shell

1	20	15	19	21	10	14	7	5	24	1	13	16	12	5
1													12	
	5													20
1	5	15	19	21	10	14	7	5	24	1	13	16	12	20
1				5				16				21		
	5				10				12				24	
		1				14				15				20
			7				13				19			
1	5	1	7	5	10	14	13	16	12	15	19	21	24	20
1	1	5	5	7	10	12	13	14	15	16	19	20	21	24

Cadeau 1 : le Tri Shell en CAML

```
let tri_shell a =
  let h = ref 1 in
   while !h <= vect_length a - 1 do
    h := 3 * !h + 1
   done;
   while !h > 1 do
     h := !h / 3;
     for i = !h to vect_length a - 1 do
       if a.(i) < a.(i - !h) then begin
         let v = a.(i) in
         let j = ref i in
         while !j >= !h && a.(!j - !h) > v do
           a.(!j) <- a.(!j - !h);
           j := !j - !h
         done;
         a.(!j) <- v
       end
     done
   done;;
```

Cadeau 2 : Tri Shell en Java

```
static void triShell() {

  int h = 1; do
     h = 3*h + 1;
  while ( h <= N );
  do {
      h = h / 3;
      for (int i = h; i < N; ++i)
          if (a[i] < a[i-h]) {
              int   v = a[i], j = i;
              do {
                  a[j] = a[j-h];
                  j = j - h;
              } while (j >= h  &&  a[j-h] > v);
              a[j] = v;
          }
  } while ( h > 1);
}
```

Tri shell : efficacité

- *on démontre que la complexité dans le pire des cas est $o(n^{3/2})$*
- *en choisissant pour suite d'incréments 1, 3, 7, 15, 31 ($h_{i+1} = 2h_i + 1$) on obtient expérimentalement une complexité en $o(n^{1,2})$*
- *très efficace jusqu'à 5000 éléments que les données soient aléatoires, triées ou en ordre inverse, code facile et compact, conseillé quand on n'a aucune raison d'en choisir un autre*

Méthodes de tris

1- Introduction

- *spécification, clés structurées,*
- *organisation de la mémoire, critères d'efficacité*

2- Tris simples

- *par sélection,*
- *par insertion ,*
- *tri shell*

✓ ***3- Tri rapide (quick-sort)***

4- Tri fusion

5- Tri par tas

6- Conclusion

Tris Sophistiqués

- *Les méthodes de tri dites simples précédentes ont des complexités; en nombre de transfert ou en nombre de comparaison en* $O(n^2)$.
- *Des tris plus sophistiqués tels que le tri rapide, le tri par tas, ou le tri fusion permettent d 'atteindre des compléxité en* $O(n \log n)$ *en nombre de comparaison*
- *la complexité en nombre de transfert étant du même ordre ou inférieure*

Tri rapide (quick-sort)

- ***c 'est un tri dichotomique appelé aussi tri des bijoutiers :***
  trier les perles avec des tamis plus ou moins fins
  pour chaque tamis
  - *on sépare les perles en deux sous-ensembles (celles qui passent et celles qui restent dans le tamis)*
  - *puis on trie celles qui sont passées avec un tamis plus fin*
  - *et les autres avec un tamis plus gros*

Tri rapide (quick-sort)

- ***principe (Hoare 1960) : algorithme récursif de type diviser pour résoudre***
  - *choisir dans le tableau un élément "pivot" :*
  - *mettre*
    - à gauche du pivot tous les éléments plus petits que lui,
    - à droite tous les éléments plus grands que le pivot
    - et entre les 2 le pivot est à sa place définitive (partition du tableau en 2)
  - *trier chacune des deux sous-listes indépendamment*

Tri rapide (quick-sort)

- ***algorithme récursif :***
  Tri-rapide (tab, deb, fin, cmp)
  - *si deb < fin*
    - k = Partition (tab, deb, fin , cmp)
    - tri-rapide(tab, deb, k - 1, cmp)
    - tri-rapide(tab, k + 1,fin, cmp)

Tri rapide (quick-sort)

Partition (principe)

- ***choisir le pivot***
- ***parcourir le tableau***
  - *en partant de la gauche (indice g) et s'arrêter dès que l'on rencontre un élément > au pivot*
  - *en partant de la droite(indice d) et... <= au pivot*
  - *ces 2 éléments étant mal placés on les échange*

on continue le parcours jusqu'à ce que les indices g et d se croisent

- ***placer le pivot à la frontière entre les deux en d***
- ***retourner d***

Tri rapide (quick-sort)

101	213	20	123	47	79	195
	79				213	195
	79	20	47	123	213	195
47	79	20	**101**	123	213	195
47	79	20	**101**	123	213	195
	20	79				
20	**47**	79				
20	**47**	**79**	**101**	123	213	195
				123	213	195
					195	**213**
20	**47**	**79**	**101**	**123**	**195**	**213**

Tri rapide (quick-sort)

- ***choix du pivot :***
  - *stratégies simples:*
    premier élément ou dernier élément
  - *efficace : pivot médian de 3 entre*
    - tab[deb]
    - tab[fin]
    - tab[(deb+ fin) / 2)]
- ***complexité :***
  - *en moyenne : o(n log n)*
  - *pire des cas : o(n^2)*
  - *en espace : une pile de taille maximum o(log n)*

Tri rapide (quick-sort)

avantages :

- ***très bien compris, étudié tant au niveau théorique qu'expérimental***
- ***très bonnes performances s'il est optimisé***
  - *dérécursivé*
  - *tri par insertion sur les petites listes (fin-deb > 25)*
  - *en langage assembleur*

inconvénients :

- ***tri fragile : très sensible aux erreurs de programmation***
- ***mauvaises performances sur tableaux déjà triés***

Méthodes de tris

1- Introduction

- *spécification, clés structurées,*
- *organisation de la mémoire, critères d'efficacité*

2- Tris simples

- *par sélection,*
- *par insertion ,*
- *tri shell*

3- Tri rapide (quick-sort)

✓ ***4- Tri fusion***

5- Tri par tas

6- Conclusion

Tri fusion

- ***principe : diviser pour résoudre 2 options***
  - *le tableau complet est dans l'ordre après le tri des 2 sous-ensembles (tri-rapide)*
  - *après le tri des 2 sous-ensembles on les combine pour obtenir le tableau trié (tri-fusion)*
- ***avantages du tri fusion***
  - *complexité en moyenne et dans le pire des cas en o(n log n)*
  - *efficace sur les listes chaînées*
  - *utilisé pour les tris externes*
- ***inconvénients du tri fusion***
  - *tri à côté : utilise une place mémoire en o (n)*

Tri par tas (heapsort)

- ***ce qui est inefficace dans les tris simples par sélection c'est que l'on refait plusieurs fois les mêmes comparaisons***
- ***idée de mémoriser les comparaisons dans une structure de données appelée tas (arbre parfait partiellement ordonné)***
- ***complexité :***
  - *en moyenne et dans le pire des cas en o(n log n) pour le nombre de comparaisons et d'échanges*
  - *tri sur place : pas de mémoire*
- ***en moyenne cet algorithme est deux fois plus lent que le tri rapide et aussi plus lent que le tri fusion***

Conclusion sur les tris

- *on démontre que la complexité optimale en nombre de comparaisons aussi bien qu'en nombre d'échanges d'un algorithme de tri par comparaison d'une clé est (n log n) (en moyenne et au pire)*
- *pas de tri miracle dépend du nombre de données, de leur structure (coût relatif des transferts et des comparaisons)*
- *dépend aussi de la qualité de la programmation (éviter les tests de dépassement d'indice en plaçant des sentinelles)*

Conclusion sur les tris : complexité

- ***complexités théoriques : résumé (Cf. tableau)***

Algorithme	Nombre de comparaisons		Nombres de transferts		
	Moyenne	Pire des cas	Moyenne	Pire des cas	Mémoire
Sélection ordinaire	$o(n^2)$	$o(n^2)$	o(n)	o(n)	o(1)
Tri à bulles	$o(n^2)$	$o(n^2)$	$o(n^2)$	$o(n^2)$	o(1)
Insertion Séquentielle	$o(n^2)$	$o(n^2)$	$o(n^2)$	$o(n^2)$	o(1)
Insertion dichotomique	o(n log n)	o(n log n)	$o(n^2)$	$o(n^2)$	o(1)
tri shell		$o(n^{1,2})$			o(1)
tri rapide	o(n log n)	$o(n^2)$			o(log n)
tri par tas	o(n log n)	o(n log n)	o(n log n)	o(n log n)	o(1)
tri fusion	o(n log n)	o(n log n)	o(n log n)	o(n log n)	o(n)

adapté de Froidevaux

Conclusion sur les tris : complexité

- ***comparaisons expérimentales***
  - ➢ *pour n < 100 : tri simple*
    - tri sélection meilleur : si gros éléments (moins de transferts)
    - tri insertion meilleur : listes presque triées
  - ➢ *le tri rapide (optimisé) toujours plus rapide que le tri par tas (même si la liste est triée)*
  - ➢ *le tri fusion moins efficace en moyenne mais*
    - moins risqué dans le pire des cas que le tri rapide
    - prend de la place mémoire

Conclusion sur les tris : autres critères

- ***stabilité : conservation de l'ordre d'origine des éléments qui ont des clés égales***
  - *non stables : tris rapide, par tas et par sélection, shell*
  - *stables : les autres*
- ***progressivité : à l'étape k les k premiers éléments sont triés***
  - *les tris par sélection sont progressifs : sélection, bulles, tas*

Sites Internet : Animation

- http://lwh.free.fr/pages/algo/tri/tri.htm
- http://www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html
- http://www.cs.rit.edu/~atk/Java/Sorting/sorting.html
- http://www.cs.brockport.edu/cs/java/apps/sorters/race_sorters/sortchoiceinp.html
- http://www.cs.brockport.edu/cs/java/apps/sorters/race_sorters_ind/sortchoiceinp.html
- ***Tri Shell :***
  http://www.cs.princeton.edu/~rs/shell/animate.html

Quatorzième partie

La récursivité

Introduction

- *La notion de récursivité est une notion essentielle en informatique car c'est à la fois une manière d'appréhender certains problèmes et une méthode de programmation.*
- *On dit qu'on définit une notion nouvelle de manière récursive lorsque cette notion fait partie de sa propre définition.*
- *Cette manière de procéder heurte le sens commun car on a l'habitude de définir un objet en utilisant des objets connus.*

Introduction

- ***Exemple : la fonction factorielle***
  - *Dans le domaine numérique, on définit la fonction factorielle sous la forme suivante :* ***n! = (n-1)! * n***
  - *Ainsi 5! est définie par la connaissance éventuelle de 4!*
  - *4! est définie par la connaissance éventuelle de 3!*
  - *Pour pouvoir calculer 5!, il est impératif de disposer, à une étape, d'un résultat explicite qu'on appellera point d'appui.*
  - *Ce point d'appui pour la fonction factorielle est 1! = 1*
- ***D'une manière générale, une définition récursive doit être complétée par la connaissance d'un cas particulier au moins (le point d'appui) pour lequel on connaît explicitement le résultat.***

Définition

- *« Le fait pour un programme ou une procédure de s'appeler au moins une fois lui-même. »*
- *Le principe de la récursivité est essentiel en programmation, il permet de résoudre de façon élégante la plupart des problèmes, soit par l'implémentation, soit par le simple fait de penser le problème en terme de récursivité.*

La récursivité fondée sur une relation de récurrence

- ***Exemple 1 : la fonction factorielle***
  - *L'évaluation de la fonction Fac telle que Fac(n) = n! peut s'obtenir à l'aide de la suite récurrente suivante :*
    - 1! = 1
    - 2! = 1! * 1
    - ...
    - n! = (n-1)! * n
  - *C'est-à-dire la suite (u_i) telle que :*
    - $u_i = u_{i-1} * i$ pour $1 < i \leq n$
    - $u_1 = 1$

La récursivité fondée sur une relation de récurrence

- ***La fonction Fac_I exprime de manière itérative le calcul de n! :***

```
int Fac_I(int n)
{
  int i, res;

  res = 1;
  for(i=1;i<=n;i++)
    res*=i;
  return res;
}
```

La récursivité fondée sur une relation de récurrence

- ***La fonction Fac_R exprime de manière récursive le calcul de n! :***

```
int Fac_R(int n)
{
  if(n==1)
    return 1;
  else
    return n * Fac_R(n-1);
}
```

La récursivité fondée sur une relation de récurrence

- ***Exemple 2 : la fonction Fibonacci***
  - *L'évaluation de la fonction Fib(n) donne la suite suivante :*
    - 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, etc.
  - *Elle est issue de la suite (u_n) définie telle que :*
    - $u_0 = u_1 = 1$
    - $u_n = u_{n-1} + u_{n-2}$ pour $n > 1$

La récursivité fondée sur une relation de récurrence

- ***La fonction Fib_I(n) exprime de manière itérative le calcul de l'élément u_n de la suite de Fibonacci :***

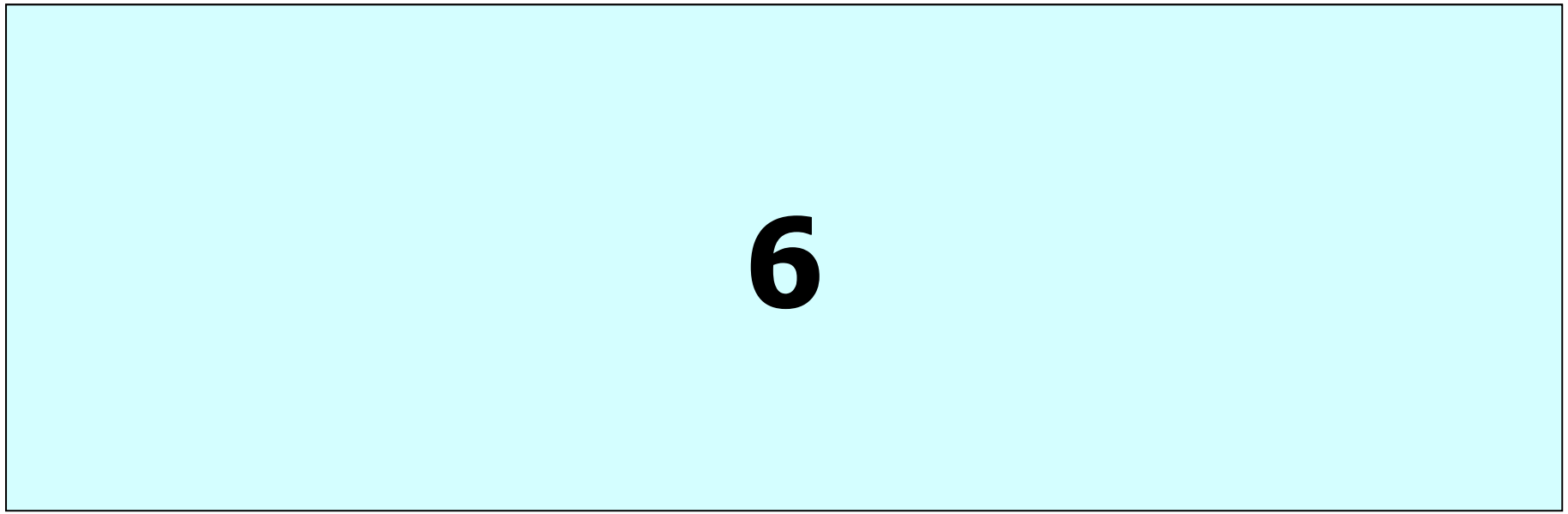
```
int Fib_I(int n)
{
  int u, v, u0, v0, i;
  u = 1; v = 1;
  for(i=2;i<=n;i++)
  {
    u0 = u; v0 = v;
    u = u0 + v0;
    v = u0;
  }
  return u;
}
```

La récursivité fondée sur une relation de récurrence

- ***La fonction Fib_R exprime de manière récursive le calcul de l'élément u_n de la suite de Fibonacci :***

```
int Fib_R(int n)
{
  if(n<=1)
    return 1;
  else
    return Fib_R(n-1) * Fib_R(n-2);
}
```

Mécanisme de fonctionnement de la récursivité

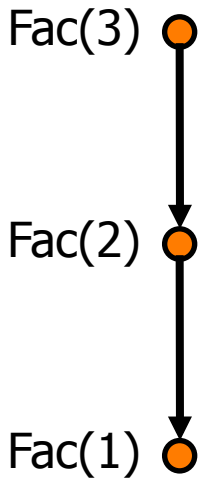


Mécanisme de fonctionnement de la récursivité

- *Le premier calcul qui abouti est celui de Fac(1), c'est-à-dire celui du troisième appel.*
- *Le calcul effectif ne commence qu'à partir du moment où on a atteint le point d'appui*
- *Lorsque l'exécution d'un appel i se termine, on « remonte » à l'appel précédent (i-1) en changeant de contexte.*
- *L'ordre chronologique d'entrée dans les fonctions est A1 -> A2 -> A3*
- *L'ordre chronologique de sortie des fonctions est A3 -> A2 -> A1*

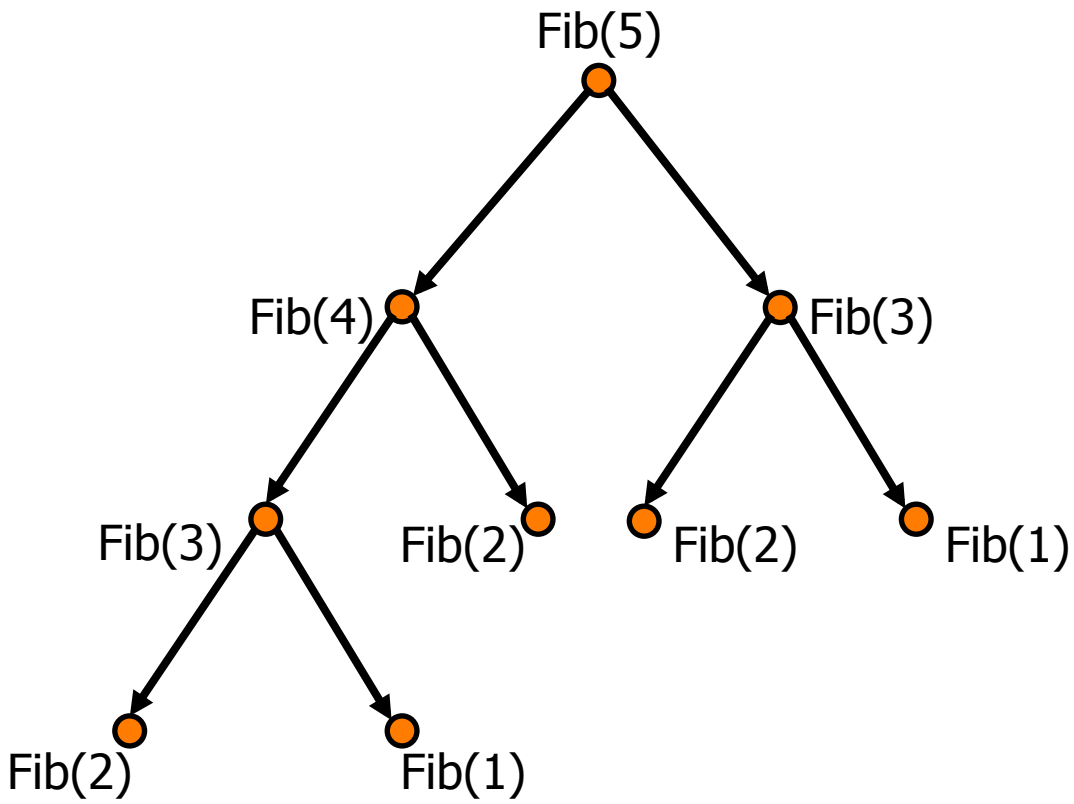
Graphe des appels récursifs

- *Chaque sommet du graphe correspond à un appel de la fonction.*
- *Graphe d'appels pour 3! :*



Graphe des appels récursifs

- ***Graphe d'appels pour Fib(5) :***



Règles de fonctionnement de la récursivité

- ***L'appel à une fonction est suivi de l'exécution proprement dite de la fonction puis du retour à l'endroit d'où l'appel a eu lieu :***
  - *Appel d'une fonction*
    - Les valeurs des paramètres effectifs, les variables locales et l'adresse de retour **sont empilés**.
  - *Exécution*
    - Au cours de l'exécution, s'il y a récursivité, la fonction peut s'appeler elle-même, auquel cas les empilements décrits à l'étape précédente sont réitérés.
  - *Sortie de la fonction*
    - Après l'exécution de la fonction, on retourne à l'adresse qui est au sommet de la pile et on désempile.

Empilement des contextes

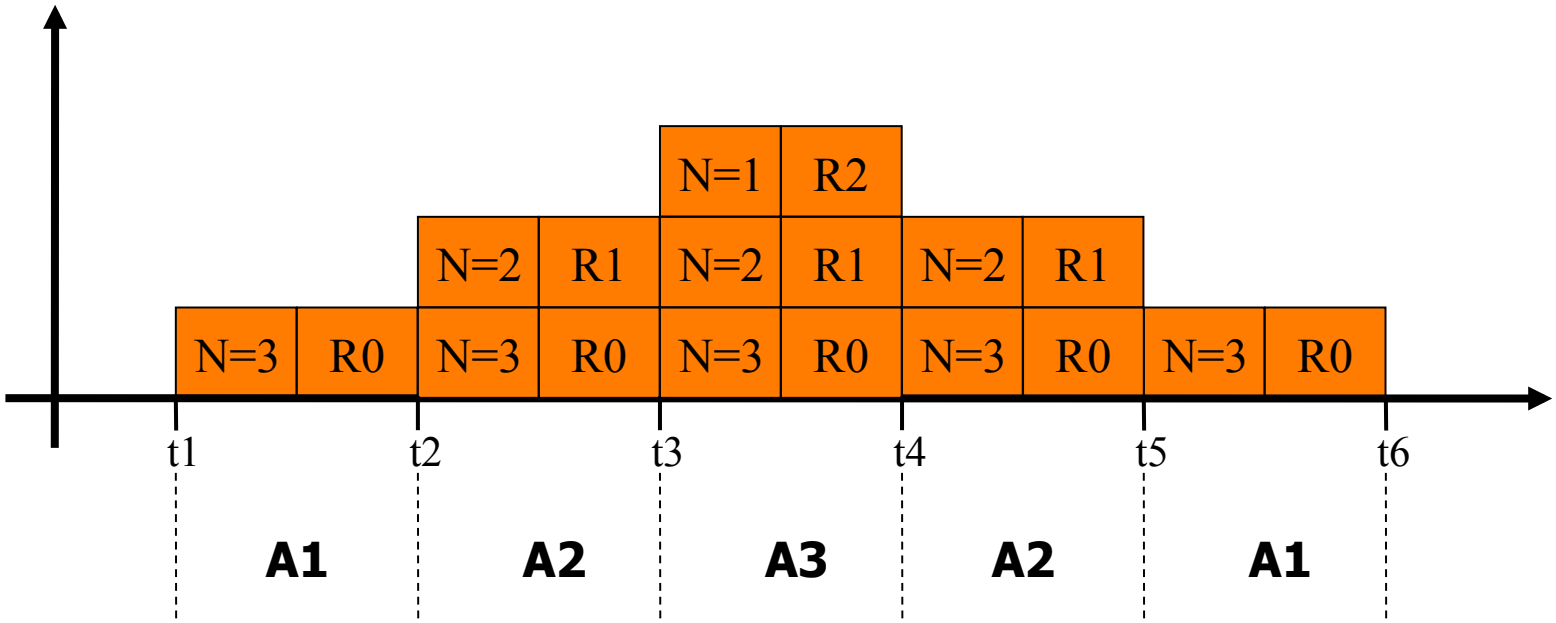


Schéma général d'un algorithme récursif

- ***Dans un algorithme récursif A, on distingue essentiellement deux parties :***
  - *Celle qui correspond au point d'appui*
  - *Celle qui contient au moins un appel à A*
- ***On peut donc représenter A sous la forme :***

```
Procedure A(p1,…,pn)
{
  Si cond alors
     I1
  sinon
  {
    I2
    A(f(p1,…,pn))
    I3
    …
    A(g(p1,…,pn))
    Im
  }
}
```

p_1,…, p_n désignent les paramètres

$A(f(p_1,\ldots,p_n)) = A(f1(p_1,\ldots, p_n),\ldots, fn(p_1,\ldots, p_n))$ où $fi(p_1,\ldots, p_n)$ est une transformation du paramètre d'entrée pi

I1, I2, Im représentent chacune une suite d'instructions ~~(éventuellement vide) qui ne comporte aucun appel à A~~

La condition qui permet de distinguer le traitement du point d'appui du traitement courant est une expression booléenne contenant au moins un pk parmi les paramètres p1,…, pn

De plus, pour le rappel de A, la fonction fk(p1,…,pn) doit faire évoluer le paramètre pk vers le point d'appui

Algorithme simplement récursif

- ***Un algorithme est dit simplement récursif lorsqu'il ne contient qu'un seul appel récursif.***
- ***Dans ce cas particulier, le schéma est de la forme suivante :***

```
Procedure A(p1,…,pn)
{
  Si cond alors
     I1
  sinon
  {
    I2
    A(f(p1,…,pn))
    I3
  }
}
```

Algorithme simplement récursif : Factorielle

- ***La fonction factorielle est simplement récursive.***
  - *cond* -> *n = 1*
  - *I1* -> *return 1;*
  - *I2* -> *instruction vide*
  - *A(f(p1,…,pn))* -> *res = fact(n-1);*
  - *I3* -> *return n * res;*

```
int Fac_R(int n)
{
  if(n==1)
    return 1;
  else
    return n * Fac_R(n-1);
}
```

Récursivité terminale

- *Une invocation récursive d'une fonction f est dite terminale si elle est de la forme return f(...)*
- *Autrement dit, la valeur retournée est directement la valeur obtenue par l'invocation récursive, sans qu'il n'y ait d'opération sur cette valeur.*
- *Dans le cas de la fonction Fac_R, la récursivité est non-terminale puisqu'il y a multiplication par n avant de retourner :*

```
return n*f(n-1);
```

Récursivité terminale

- *On dit qu'un appel récursif à l'algorithme A est un appel terminal, si et seulement si cet appel n'est jamais suivi par l'exécution d'une ou plusieurs instructions de A.*
- *Sur le schéma général d'un algorithme récursif, cela signifie que Im est vide.*
- *Dans le cas particulier d'un seul appel, c'est I3 qui est vide.*

Récursivité terminale

- *Voici une fonction récursive terminale pour le calcul de factorielle :*

```
int f(int n, int a)
{
  if (n<=1)
       return a;
  else
       return f(n-1,n*a);
}
```

- *Dans cette version, le paramètre a joue le rôle d'un accumulateur ;*
- *L'évaluation de f(5,1) conduit à la suite d'invocations :*
  f(5,1) → f(4,5) → f(3,20) → f(2,60) → f(1,120) →
- *Et donc à la suite de retour :*
  f(1,120)←120←f(2,60)←120←f(3,20)←120←f(4,5)←120←f(5,1)
- *Une fonction est récursive terminale quand toute invocation récursive est terminale.*

Dérécursivation (cas de la récursivité terminale)

- *Si les fonctions $f_i(p1,\ldots,pn)$ sont de la forme gi(pi), c'est-à-dire ne dépendent que de pi, alors on démontre que l'algorithme itératif équivalent a pour schéma :*

```
procedure A(p1,…,pn)
  TantQue non(cond) Faire
      I2
      pk=gk(pk) pour tout pk transformé
  FinTantQue
  I1
Fin
```

- *L'ordre d'exécution des instructions est : I2, I2, …, I2, I1*

Dérécursivation (cas de la récursivité terminale)

- ***L'exécution des instructions I2 est suivie à chaque fois d'un changement de contexte qui correspond aux transformations : pk=gk(pk)***
- ***Ces instructions s'exécutent tant que cond = faux***
- ***I1 s'exécute ensuite.***

```
int f(int n, int a)
{
  if (n==1)
    return a;
  else
    return f(n-1,n*a);
}
```

```
int f(int n)
{
  int a;

  while(n!=1)
    a = a * n--;
  return a;
}
```

Dérécursivation (cas de la récursivité terminale)

```
Procedure A(p1,…,pn)
  Si cond Alors
     I1
  Sinon
    I2
    A(f(p1,…,pn))
 FinSi
FinProc
```

```
Procedure A(p1,…,pn)
  TantQue non(cond) Faire
    I2
    pk=gk(pk)
  FinTantQue
  I1
FinProc
```

```
int f(int n, int a)
{
  if (n==1)
    return a;
  else
  {  a = a*n; n = n-1;
     return f(n,a);
  }
}
```

```
int f(int n)
{
  int a;

  while(n!=1)
  {  a = a * n; n = n-1;
  }
  return a;
}
```

Exercice : nombre en chiffres romains

- *M =1000*
  D = 500
  C = 100
  L = 50
  X = 10
  V = 5
  I = 1
- *Si le premier chiffre d'un nombre romain a une valeur inférieure au deuxième, alors on le soustrait de la valeur de tout le reste, sinon on l'additionne à la valeur de tout le reste.*
- *Si le nombre romain a un seul chiffre, alors on prend simplement la correspondance (M=1000,…)*

Exercice : nombre en chiffres romains

- ***Soit Eval la fonction qui évalue un nombre romain et valeur la fonction qui retourne la valeur d'un chiffre romain***
- ***Point d'appui :***
  - *Si r est un caractère romain -> Eval(r) = valeur(r)*
- ***Cas général :***
  - *Si la valeur du premier chiffre romain est supérieure à la valeur du deuxième ->*
    Eval(r) = valeur(premier chiffre) + Eval(reste)
  - *Si la valeur du premier chiffre romain est plus petite que la valeur du deuxième ->*
    Eval(r) = Eval(reste) - valeur(premier chiffre)

Exercice : le triangle de Pascal

- *Le triangle de Pascal est le tableau des coefficients qui sont utilisés pour le développement de certaines expressions comme : $(a+b)^2$ ou $(a+b)^n$*
- *Ce triangle est le suivant :*

  0 : 1
  1 : 1 1
  2 : 1 2 1
  3 : 1 3 3 1
  4 : 1 4 6 4 1
- *On obtient chaque coefficient en additionnant le nombre qui lui est situé au-dessus ainsi que celui qui lui est situé au-dessus à gauche.*

Exercice : le triangle de Pascal

- *Le numéro qui est en tête de chaque ligne de ce triangle est la puissance à laquelle « a+b » est élevé :*

  $(a+b)^0 = 1$

  $(a+b)^1 = 1*a + 1*b$

  $(a+b)^2 = 1*a^2 + 2*ab + 1*b^2$

  $(a+b)^3 = 1*a^3 + 3*a^2b + 3*ab^2 + 1*b^3$

  $(a+b)^4 = 1*a^4 + 4*a^3b + 6*a^2b^2 + 4*ab^3 + 1*b^4$

- *On remarque que la diagonale est toujours à 1 -> point d'appui*
- *On remarque que la première colonne est toujours à 1 -> point d'appui*
- *Pour tout autre élément qui se trouve à la ligne y et à la colonne x, le résultat est la somme de l'élément de coordonnées y-1,x*

Exercice : calcul de la racine carrée

- ***Soit N le nombre dont on recherche la racine carrée et R1 une valeur approchée de celle-ci.***
- ***La formule suivante nous fournit une valeur beaucoup plus précise que nous nommerons R2 :***

$$R2 = \frac{\left(R1 + \frac{N}{R1}\right)}{2}$$

Exercice : calcul de la racine carrée

- *Appliquons la formule pour N = 5 et R1 = 2 :*

$$R2 = \frac{\left(2 + \frac{5}{2}\right)}{2} = \frac{(2 + 2{,}5)}{2} = \frac{4{,}5}{2} = 2{,}25$$

- *Cette valeur approche mieux la réalité puisque 2,25² = 5,0625.*
- *Reprenons la formule avec R1 = 2,25*

$$R2 = \frac{\left(2{,}25 + \frac{5}{2{,}25}\right)}{2} = \frac{(2{,}25 + 2{,}2222)}{2} = \frac{4{,}4722}{2} = 2{,}2361$$

- *Si nous élevons au carré cette nouvelle valeur, nous obtenons*

Exercice : calcul de la racine carrée

- *Cette précision est déjà honorable et elle peut être suffisante. Toutefois, si nous souhaitons un plus petit écart avec la réalité nous devons appliquer la formule avec R1 = 2,2361 et nous obtiendrons :*

$$R2 = \frac{\left(2,2361 + \frac{5}{2,2361}\right)}{2} = \frac{(2,2361 + 2,2360359555)}{2} = \frac{4,4721359555}{2} = 2,23606797775$$

- *L'élévation au carré de cette nouvelle valeur de R2 fournit 5,0000000011189.*