PROGRAMMATION PAR OBJETS

LANGAGE C++

Notes de Cours

N. Castagné, M. Desvignes, F. Portet

Version 2015.01

PLAN DU COURS

1. Généralités sur le C++

2. C et C++

Les apports du C++ au C
ou le C++ sans les objets

3. Programmation Orientée Objets ?

Une introduction au paradigme de la POO

4. Classes et objets en C++

Classes et objets, attributs et méthodes,
constructeurs, destructeurs, opérateurs,...

5. Héritage, polymorphisme et virtualité en C++

Héritage simple, virtualité, liaison dynamique
Héritage multiple

6. Compléments sur le C++

Templates (Patrons), Flots et fichiers,
Exceptions, Standard Template Library

1. C++

==> **C++ = C** + typage fort + programmation objet <==

➢ Naissance du langage C++ : 1979/1983

Bjarne Stroustrup, ATT Bell Labs.

Inspirés de SIMULA, ALGOL...
Initialement : extension du langage C, traducteur « C++ vers C »
Ajout de concepts objet (classification, encapsulation, héritage, abstraction...)

➢ Versions

1983 : première version (à l'origine *via* un traducteur de C++ vers C)
1998 : premier standard ISO, C++98
2011 : dernier standard en date (C++11)

➢ Objectifs et intérêt du langage C++ :

L'un des langages les plus utilisés aujourd'hui.

Communs aux approches objets :

Ecrire facilement de bons programmes de grande taille.
- Protection et robustesse
- Structuration, Réutilisabilité
- Lisibilité et évolutivité
- ...

Propres au C++ :

Haute performance : compilé, possibilité d'écrire « proche de la machine »
Du bit (bas-niveau) au concept (haut-niveau)
Possibilité de mixer paradigmes objet et impératif (« comme en C »).

2. LES APPORTS DU C++ AU C

OU LE C++ SANS LES OBJETS

➢ Le C++ comme une extension du C

Tout programme écrit en C compile avec un compilateur C++. Ajout au C :

- nombreux compléments : *dans cette partie*
  Rq : Certains ont été ajoutés ensuite au standard C récents.

- notions propres à l'approche objets : *partie 3 du cours*

➢ Compilateur et environnement de travail

A l'origine (1979-83), le C++ était implanté avec un traducteur « C++ vers C ».
Désormais : compilateur C++.
Nous utiliserons le compilateur **g++** de la GNU Compiler Collection.

Fichier header main.cpp pour « hello world »

```
#include <string>
#include <iostream>
main() {
    std::string str = "Hello World" ; // un objet string !
    std::cout << str << std::endl ; // <=> printf(...)
}
```

Compilation, édition des liens et création de l'exécutable binaire dans le Terminal avec :

```
% g++ main.cpp -o main
```

Exécution avec :

```
% ./main
```

➢ IDE : netbeans

STRUCTURE D'UN PROGRAMME C++

➢ Struture d'un programme

Un programme est composé d'un ou plusieurs fichiers de la forme :

< Commandes du préprocesseur >
< Définition de types >
< Variables globales >
< Fonctions >

➢ Struture d'une fonction

<type de retour> nom_de_fonction(<Déclaration des arguments >)
{ <Déclaration locales>
<instructions>
}

Rq : il existe une seule et unique fonction int main(int argc, char** argv) dasn l'ensemble des fichiers forment le code

➢ Exemple

```
Using namespace std ;
#include <iostream>

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

main() { int i,j,k;
    i=10;
    j=20;
    k = max(i,j);
    cout<<"Max de "<<i<<" et de "<<j<<" est "<<k<<endl ;
}
```

VARIABLES ET ADRESSES

➢ Variable

Objet informatique permettant de conserver et de modifier sa valeur
Définie par :

- Un type : entier, réel, etc…
- Un nom
- Sa durée de vie (ou portée) : de l'endroit où elle est déclarée à la fin du bloc : }

➢ Adresse

Adresse d'un objet informatiqie : où est il situé en mémoire centrale ?
Définie par un entier : numéro de la case mémoire où ill se trouve
Obtenue à l'aide de l'opérateur **&**

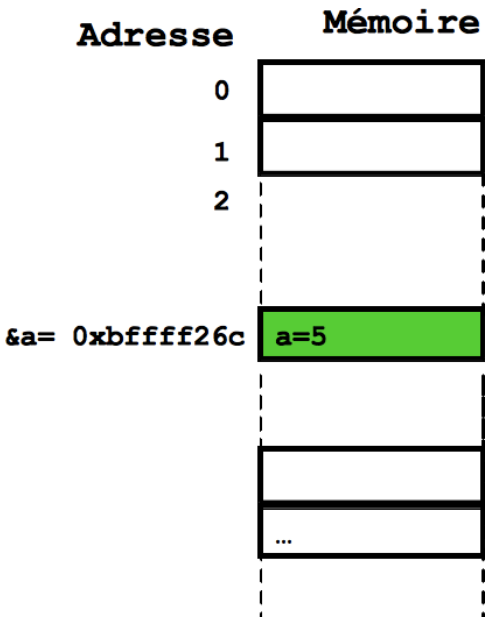
Sa durée de vie (ou portée) : de l'endroit où elle est déclarée à la fin du bloc : }

➢ Exemple

La variable a est créée à l'adresse 0xbffff26c en hexadécimal

```
Using namespace std ;
#include <iostream>

main(){ int a;
  a=5;
  cout <<"Bonjour"<<endl;
  cout << "La valeur de a est"  << a  <<endl;
  cout << "L'adresse de a est " << &a <<endl
}
```



VARIABLES (2)

➢ Déclaration libre des variables

- Une variable peut être déclarée n'importe où dans le programme
- Une variable ne peut être utilisée qu'entre la ligne où elle est déclarée et la fin du bloc (ou du fichier) dans lequel elle est déclarée.

➢ Variables de boucle

- une variable peut être déclarée dans une boucle.
- Elle existe alors jusqu'à la fin de la boucle.

➢ Exemples

```
main() {
    int i;      // Création de i
    cin >> i;  // Lecture de i;
    j = 9;      // ERREUR : j n'existe pas encore
    int j = 12;    // Création de j
    for (int k=0; k<i; k++) {  // Création de k
      int localVar = k*10*j ;// Création de localVar.
      cout << localVar;
    } // fin de k et de localVar
    k = 8;          // ERREUR ! k n'existe plus
    localVar= 9;    // ERREUR ! localVar n'existe plus
} // fin de i,j
```

RESUME DES E/S

➢ Notion de flot

Classes prédéfinies facilitant les E/S : gestion des flots (« stream »)
`#include <iostream> <fstream> <ostream> <sstream> ...`

➢ Lecture : Clavier, fichier, stringstream

flot >> variables

➢ Ecriture : Ecran, fichier, stringstream

flot << variables, constantes

Entrée / sortie standard
#include <iostream>
Donne accès aux flots `std::cin, std::cout, std::cerr`
`(l'équivalent de stdin, stdout, stderr en C)`

➢ Exemple

```
#include <iostream>
main() { int i; float x; double z;
    std::cout << "Entrez un entier et deux réels ";
    // Pas de retour chariot automatique
    std::cin >> i >> x >> z;
    // Lecture d'un entier, puis d'un float
    // puis d'un double
    std::cout << "i=" << i << " x=" << x
              << " z=" << z << std::endl;
    // Affiche i puis x puis z et retour chariot
}
```

➢ Base pour la mise en forme des sorties

Retour chariot, affichage hexadécimal / decimal, notation exponentielle / scientifique et precision pour les flottants…

```
#include <iomanip>
...
std::cout << std::hex << 16 << " "
          << std::dec << 16 << std::endl;
std::cout << std::scientific << 0.5 <<  " "
          << std::fixed << 0.5 << std::endl;
std::cout << std::setprecision(2) << 3.14159
          << std::endl;
```

NAMESPACE

Problème : mêmes symboles dans plusieurs bibliothèques utilisées
➔ Ambiguité sur les noms

Solution : les espaces de noms (namespace)
Nom complet des symboles:

```
nom_de_namespace::symbole
```

Exemple :

```
std::cin // la variable cin du namespace std
```

➢ Namespace global :

```
int i;
main(){ ::i = 6 ; }
```

➢ Utilisation d'un name space:

mot clé `using`

```
#include <iostream>
using namespace std;
main() {
    int i;
    cin >> i; // evite std::cin >>i;
}
```

➢ Définition d'un namespace

```
namespace nom_du_namespace{ …. };
```

Peut etre sur plusieurs fichiers. Le namespace est la reunion de tous les namespace de meme nom.

```
namespace monnamespace {
    int f() {…};
}
main() {int i;
    i=monnamespace::f();
}
```

PORTEE DES VARIABLES

** 5 types de portées : locale, fonction, boucle, fichier, classe.

** **locale** : un nom déclaré dans un bloc (`{..}`)est local à ce bloc. Il ne peut être utilisé qu'entre la déclaration et la fin du bloc.

** **fonction** : pour mémoire, les étiquettes peuvent être utilisées uniquement dans la fonction où elles sont déclarées.

** **variable de boucle** : du début à la fin de la boucle

** **fichier** : un nom déclaré hors de toute fonction, classe ou bloc a une portée de fichier. Il ne peut être utilisé qu'entre la déclaration et la fin du fichier. Attention à la directive `extern`.

** **classe** : un nom déclaré dans une classe est utilisable dans cette classe (fonction membres et amies).

** **Opérateur de résolution de portée : `::`**

accès à un membre de la classe nom_classe ou à la variable globale

Syntaxe : `<nom_classe>::membre`
`::nom`

Exemple:

```
int i;
void f(int j)
{
  int i;
  i=5;        // variable i locale
  ::i=10;     // variable i globale
}
```

STRUCTURES DE CONTROLE

➢ 2 types

1. Conditionelle: if … else, switch

   a. Conditionelle simple :
   réalise une ou plusieurs actions selon UN test sur UNE expression

```
if ( expression ){  instruction1};
[ else {instruction2;} ]
```

   b. Conditionelle multiple:
   réalise une ou plusieurs actions selon DES tests sur une expression

```
switch ( expression ){
case cste1 : instruction1;
case cste2 : instruction2;
[ default {instructionN;} ]
```

2. répétitives: while …, do {…} while, for

   a. do {…} while()
   réalise une action tant que le test est vérifié
   les actions sont faites au moins une fois

```
do {  instructions ;} while( expression ) ;
```

   b. while() {}
   tant que le test est vérifié, réalise une action

```
while(expression) {  instructions ;}
```

   c. boucle générique : for
   boucle while qui localise les différents éléments d'une boucle en les dissociant du reste du code :
   - initialisation (expr1),
   - test d'arrêt de la boucle (expr2)
   - passage à l'itération suivante (expr3)

```
for (expr1 ; expr2 ; expr3) {  instructions ;}
```

   equivalente à :

```
expr1 ;
while( expr2 ) {instructions ; expr3 ;}
```

STRUCTURES DE CONTROLE (2)

➢ Exemples

```
#include <iostream>
using namespace std;

main() {
    // conditionnelle
    int a,b;
    cout << "entrez deux entiers : " ;
    cin >> a >> b;
    if (a>b)
        cout << "a:"<<a<<" est plus grand que b:"<<b<< endl;
    else
        cout << "b:"<<b<<" est plus grand que a:"<<a<< endl;

    // un menu avec un switch
    int choix, prix;
    cout << "Au menu : " << endl;
    cout << "\t 1 : Vin rouge" << endl;
    cout << "\t 2 : Eau" << endl;
    cout << "Tapez votre choix : " ;
    cin >> choix ;
    cout << "Vous avez choisi : ";
    switch ( choix ) {
        case 1 : cout << "un rouge"; prix=100;          break;
        case 3 : cout << "de l'eau"; prix=10;           break;
        default : cout << "Choix non valide"; prix=0;   break;
    }
    cout << " pour un cout de " << prix << endl;

    int N;
    // exemple boucle do ... while : protéger une saisie
    do {
        cout<<"Produit des N premiers entiers - entrez N entre 1 et 100 : ";
        cin >> N;
    } while ( N < 1 || N > 100);

    // exemple boucle for
    int accumulateur = 1;
    for(int i = 1; i <= N ; i ++) {
        accumulateur = accumulateur * i;
    }
    cout << "Produit vaut : " << accumulateur << endl;

    // exemple boucle while
    int i = 1;      accumulateur = 1;
    while (i <= N) {
        accumulateur = accumulateur * i;
        i++;
    }
    cout << "Produit vaut : " << accumulateur << endl;
}
```

FONCTIONS

➢ Fonctions

Une fonction est une unité de traitement dans un programme

➢ Role

- Structurer le programme
  - Décomposer un problème en sous problèmes
  - Spécifier les dépendances entre ces sous problèmes
  - Meilleure lisibilité des étapes de résolution
- Réutiliser le code
  - une meme fonction peut etre utilisée plusieurs fois
- Séparer et rendre étanche différentes parties d'un programme.
  - limite les propagations d'erreurs
- Offrir une maintenance du code plus aisée

Une fonction prend en entrée des données et renvoie des résultats après avoir réalisé différentes actions

➢ Utilisation d'une fonction

- Déclarer le prototype : indique comment doit être utilisée une fonction
  - comporte le nom, la liste des paramètres et le type de la valeur de retour suivi d'un ";"
- Executer la fonction : appel avec les paramètres effectifs :
  - **`nom_fonction(paramètres);`**

➢ Exemple

```
int max( int a, int b);
/* Prototype d'une fonction appelée max,qui prend 2 entiers
en paramètres et qui retourne un entier. */

main() {int i,j,k,l,m,n;       /* 6 entiers */
  i=10; j=20; k=-8; l=12;
  m=max(i,j);
    /* On appelle max avec i et j, resultat dans m*/
  cout << "Le max de "<<i<<" et de "<<j<<" est "<<m<<endl;
  n=max(k,l);
    /* On appelle max avec k et l, resultat mis dans n*/
  cout << "Le max de "<<k<<" et de "<<l<<" est "<<n<<endl;
}
```

FONCTIONS(2)

➢ Ecrire une fonction

- Une fonction est constituée de
  - Son nom
  - Sa liste de paramètres s'ils existent
  - Son type de valeur de retour
  - Des instructions indiquant ce qu'elle doit faire
- Le code de la fonction s'écrit en utilisant les paramètres "formels"
- La valeur de retour :
  - Est unique et scalaire
    - une fonction ne peut pas renvoyer plusieurs valeurs
  - est renvoyée à l'aide de l'instruction " return "
    - on quitte la fonction IMMEDIATEMENT
    - on revient à la fonction qui a appelée

➢ Exemple

```
int max(int a, int b) { int c;
  if (a>b) c=a;
  else c = b;
  return c;
  cout << "Ce code n'est jamais attaint";
}
```

➢ Arguments par défaut

On peut donner une valeur par défaut aux paramètres d'une fonction

```
int circonference(int a, double x=3.1416) {
// si x est omis lors de l'appel, x vaudra 3.1416
  return 2*x*a ;
}

main() { int b ;
  double c,y,z,t ;
  cin >> b ;
  y=circonference(b) ;
  cout << "valeur du rayon "<<b<< ;
  cout <<" et de la circonfernce "<<y ;
  z=circonference(b+1,3.1415927) ;
}
```

Remarque les valeurs par défaut ne sont possibles que si les paramètres les plus à droite ont une valeur par défaut

```
int f(int a=9, float x) // Erreur
```

FONCTIONS(3)

➢ Executer une fonction : passage des paramètres par valeur

```
main() {int i,j,k,l,m,n;       /* 6 entiers */
  i=10; j=20; k=-8; l=12;
  m=max(i,j);
    /* On appelle max avec i et j, resultat dans m*/
  cout << "Le max de "<<i<<" et de "<<j<<" est "<<m<<endl;
  n=max(k,l);
    /* On appelle max avec k et l, resultat mis dans n*/
  cout << "Le max de "<<k<<" et de "<<l<<" est "<<n<<endl;
}
```

Lors de l'appel max(i,j), les parametres sont copiés sur la pile (zone mémoire particulière) en commencant par le plus à droite, soit j en premier.

les arguments de fonctions sont empilés (Stack) de la droite vers la gauche (C et Fortran) après évaluation et dépilés par la fonction appelée.

Pour executer max(i,j) :

1. j puis i sont recopiés dans b et a sur la pile
2. La fonction max utilise alors a et b et les instructions de max sont executées
   - la variable c est créée sur la pile
   - on teste si a est supérieur à b.
   - Si oui , a est mis dans c, sinon b est mis dans c ;
   - On execute le return : on quitte la fonction
3. a et b sont détruites quand la fonction se termine en executant return
4. On revient ensuite à la fonction appelante (ici main) avec la valeur de retour (ici, la valeur de c)

➢ Conséquence : une fonction ne peut jamais modifier ses paramètres effectifs (les variables qui lui sont passées en parametres)

Elle travaille avec une copie des parametres effectifs.

POINTEURS

Pointeurs

Variable contenant l'adresse d'un autre objet (variable ou fonction)

Rappel : Adresse : numéro d'une case mémoire

Declaration

type_pointé* identificateur;

Exemple

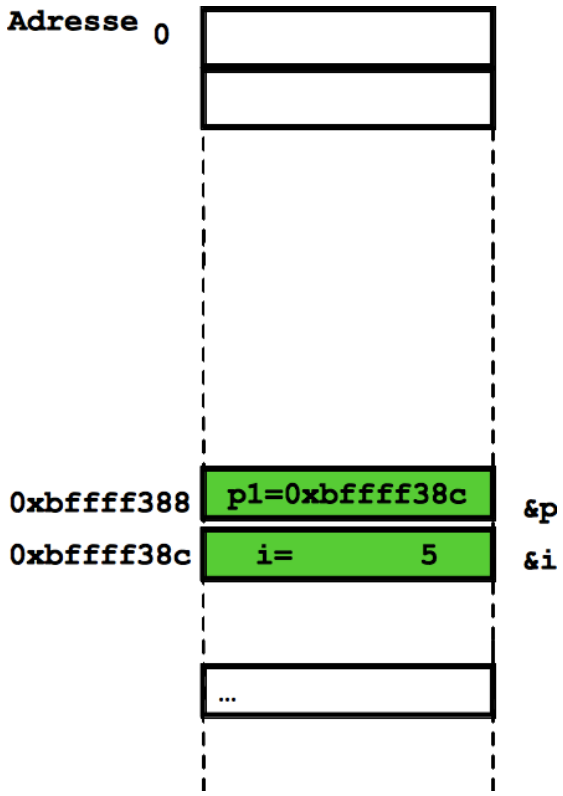
```
int* p1;       /* p1 peut contenir l'adresse d'un entier */
double* p2; /* p2 peut contenir l'adresse d'un réel */
```

Important

Un pointeur doit toujours être initialisé avant d'être utilisé = Il doit contenir une adresse légale :

- soit celle d'un objet existant
- soit celle obtenu par une demande d'allocation dynamique
- soit NULL, qui est la valeur 0. Il est interdit de lire et écrire à l'adresse 0

Exemple




```
using namespace std ;
#include <iostream>

main() {
  int i=0;
  int* p=NULL;
/* p: pointeur sur un entier */
  p = &i;
/*p pointe i,
 ie contient l'adresse de i*/
  *p = 5; /* identique à i=5;
*/
  cout<<"Valeur de i"<<i
      <<" Adresse de i"<<&i;
  cout<<"Valeur de p"<<p
      <<" Valeur pointée"<<*p);
  cout<<"Adresse de p:"<< &p1);
}
```

POINTEURS (2)

➢ Opérations sur pointeurs

- Affectation : donner une valeur au pointeur, celle d'une adresse légitime.

```
p1=&i; /* &i : l'adresse de i */
```

- Indirection : trouver la valeur pointée par p.

```
j = *p1;  /* *p1 : ce qu'il y a à l'adresse p1 */
```

- Comparaison :
  - == et != : p1==p2; p1 et p2 regardent ils la meme adresse ?
  - <, >, <=, >= : p1<p2 ; p1 est il avant p2 ?

- Arithmétique
  - Adresse + entier ==> adresse :
    p1 +1': adresse de l'élément suivant p1
  - Adresse - Adresse ==> entier :
    p2 -p1 : nombre d'éléments entre les adresses contenues dans p2 et p1.
    Valide uniquement si p1 et p2 sont de meme type.

➢ Attention

Les pointeurs étant typés, les opérations se font en nombre d'éléments et non en nombre d'octets

POINTEURS, ADRESSE ET FONCTION

➢ Rappel : parametres de fonctions en C, passage par valeur

- Il y a recopie de l'objet effectif (de x et y ci dessous) sur la pile
- La fonction travaille sur une copie des objets effectifs (a et b sont des copies de x et de y)
- Si la fonction modifie le paramètre formel (a et b), c'est en fait la copie de l'objet effectif qui est modifiée
- L'objet initial n'est pas modifié (x et y ne change pas)

➢ Exemple

```
void swap(int a, int b) { int c;
  cout<<"Debut de swap a:"<<a<<" b:"<<b<<endl;
  c=a; a=b; b=c;  /* On echange a et b  en utilisant c */
  cout<<"Fin de swap a:"<<a<<" b:"<<b<<endl;
}

main() { int x,y;
  x=1; y=2;
  cout<<"Avant le swap x:"<< x <<" y:"<< y <<endl;
  swap(x,y);
  cout<<"Apres le swap x:"<< x <<" y:"<< y <<endl;
}
```

Que fait ce programme ?

POINTEURS, ADRESSE ET FONCTION (2)

➢ Modification des parametres de fonctions en C++ : passage de l'adresse

- En passant son adresse
  - L'adresse n'est pas modifiée
  - Mais le contenu (obtenu par l'opérateur *) peut être modifié

➢ Exemple

```
void swap(int* , int* );

main() { int x,y;
  x=1; y=2;
  cout<<"Avant le swap x:"<< x <<" y:"<< y <<endl;
  swap(&x,&y);
  cout<<"Apres le swap x:"<< x <<" y:"<< y <<endl;
}

void swap(int* pa, int* pb) { int c;
  cout<<"Debut de swap a:"<< *pa <<" b:"<< *pb <<endl;
     /* On echange a et b  en utilisant c */
  c=*pa; pa=pb; *pb=c;
  cout<<"Fin de swap a:"<< *pa <<" b:"<< *pb <<endl;
}
```

Que fait ce programme ?

➢ Comment sait on qu'une fonction doit modifier ses paramètres

- Définissez et écrivez correctement le rôle de la fonction
  - La fonction échange les valeurs de 2 entiers
    - ==> la fonction a 2 paramètres et elle doit modifier la valeur de ces 2 paramètres
    - ==> ces 2 paramètres doivent être passés par adresse

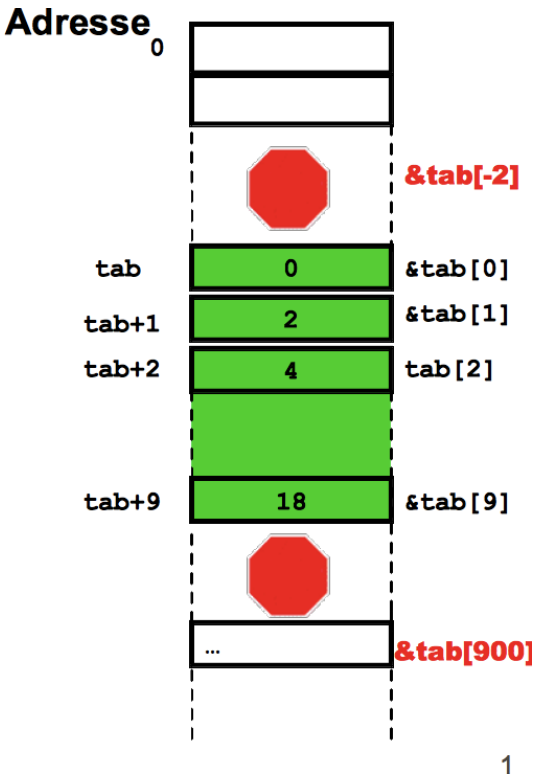
TABLEAUX

Tableaux

Collection de variables de même type, rangées continûment en mémoire

Déclaration : spécifier

- le type des éléments
- le nom du tableau
- le nombre des éléments



1

Exemple

```
main() { int i;
  int tab[10];
/* Tableau de 10 entiers */
  float c[20];
/* Tableau de 20 réels   */
  for (i=0; i<10; i++)
    tab[i]= 2*i;
/*Mettre 0,2,4,6..dans les
elements*/
  for (i=0; i<10; i++)
     cout<<tab[i]<<" ";
  cout << endl ;
/* afficher les éléments de t */
}
```

Important

- Nombre d'éléments constant, non modifiable
- Nom du tableau = son adresse
- Accès à un élément du tableau : nom_du_tableau[expression entiere]
  - Comment fait le compilateur : on part du début du tableau, on ajoute i : c'est l'endroit où se trouve notre élément
  - Pas de vérification sur les indices : erreur à l'execution du programme mais pas d'erreurs à la compilation
- AUCUNE opération globale sur un tableau
  - les opérations et les E/S doivent se faire élément par élément
  - En particulier
    - T1==T2 ne teste pas l'égalité de 2 tableaux
    - T1=T2 ne recopie pas les éléments de T1 dans T2

POINTEURS ET TABLEAUX

Lien entre pointeurs et tableau

- Tableau : une zone mémoire réservée par le compilateur pour contenir un nombre d'objets fixé à l'avance et constant
  - Nom du tableau : adresse du tableau : CONSTANTE
- Pointeur : une variable sur 64 bits qui contient une adresse :
  - VARIABLE
  - Ne définit pas une zone mémoire pour contenir des objets

Accès à un élément t[i]

1. on part de l'adresse de début du tableau,
2. on ajoute i et on obtient l'adresse du ième élément.
3. L'élément est obtenu par l'opérateur d'indirection *

- Conséquences
  - L'élément t[i] s 'écrit aussi *(t+i)
  - L'adresse de t[i] s'écrit &t[i] ou bien (t+i)

Utiliser un poiteur pour parcourir un tableau

On peut donc parcourir un tableau ou une zone mémoire en utilisant un indice pointeur et non un indice entier


```
main() {int* p=NULL;
   int i;
   int tab[5];

   for (i=0; i<5; i++)
      tab[i]=2*i+1;
   p=tab;
   while (p<tab+5) {
     cout<<" Pointeur:"<<p);
     cout<<" Valeur:"<<*p);
     *p=234;
     cout<<"Valeur modifiee:"<<*p);
     p++;
   }
```

POINTEURS DE FONCTIONS

➢ Lien entre pointeurs et fonctions

Les fonctions sont chargées en mémoire (zone TEXT), elles ont donc une adresse. On peut utiliser utilement cette propriété avec les fonctions.

- Nom de la fonction
  adresse de la fonction : on peut donc obtenir l'adresse d'une fonction
- Pointeur de fonction
  variable contenant l'adresse d'une fonction
- Déclarer un pointeur de fonction
  - type_de_retour (*id_fonct) (déclaration paramètres);
- Executer la fonction pointée : 2 écritures
  - (*id_fonct) (paramètres);
  - id_fonct (paramètres);

```
/* fonction qui retourne la somme de 2 entiers */
int somme (int a,int b){
  cout<<"Somme de "<<a <<" et "<<b<<end ;
  return(a+b);
}
/* fonction qui retourne le produit de 2 entiers */
int prod (int a,int b){
  cout<<"Produit de "<<a <<" et "<<b<<end ;
  return(a*b);
}
main() {
/* Le pointeur qui contiendra l'adresse des fonctions */
  int (*f)(int, int);
  int i,j,k; i=2; j=3;

/* On met l'adresse de la fonction somme dans f*/
  f=somme;
  /* Appel de somme par (*f) */
  k = (*f)(i,j); /* ou f(i,j); */
  cout << "Valeur de k "<<k ;
/* On met l'adresse de la fonction produit dans f*/
  f=prod;
  k = (*f)(i,j);
  cout << "Valeur de k "<<k ;
}
```

ALLOCATION DYNAMIQUE MEMOIRE

➢ Problématique

Que faire quand on ne connaît pas la dimension d'un tableau que l'on doit utiliser au moment où'on écrit le programme ?

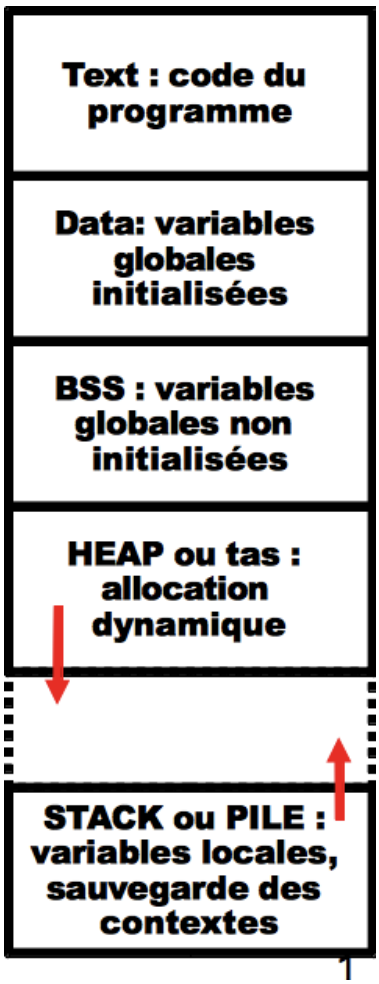
1. Solution 1 :
   - créer le plus grand tableau possible,
   - utiliser la partie dont on a besoin quand on exécute le programme
     - Le tableau est créé à la compilation,
     - il a toujours la meme taille : trop grande ou trop petite
2. Solution 2 ;
   - créer le tableau lorsque l'on connaît sa taille, c'est à dire pendant l'exécution du programme : c'est l'allocation dynamique
     - Le tableau a juste la bonne taille, pas de perte
     - cette taille est différente selon les exécutions

➢ Interet

- Dimension inconnue lors de la conception du programme
- Données de grande taille
- Données de taille variables : tableaux, listes, …

➢ Plan mémoire

- Allocation dans une zone mémoire gérée par l'utilisateur : le tas : heap
- Les variables locales sont gérées par le compilateur sur la pile : stack
- Le code est stocké dans une zone mémoire appelée text
- Les variables globales non initialisées sont gérées dans la zone « BSS »
- Les variables globales initialisées sont gérées dans la zone « data »



GESTION MEMOIRE – ALLOCATION

Gestion de la mémoire dynamique par les opérateurs new et delete
Plus efficace, meilleure cohérence (pas de fonctions spécifiques)
Même priorité que `++ ;-- ;+ ;- ;(type) ;* ;& ;sizeof ;`

➢ Allocation dynamique : opérateur new

Syntaxe : `new <type>`

- Alloue sizeof(type) octets.
- N'initialise pas la mémoire, retourne l'adresse allouée.
- Implémenté par une fonction `void * operator new (long)`
- Exemple :

```
int *pi= new int; // Création d'un entier. Sans init !
*pi = 4567;
```

Allocation avec initialization:

```
int *pi= new int(12); // int initialisé à la valeur 12
```

➢ Allocation dynamique de tableaux : opérateur new [ ]

Syntaxe : `new <type> [ <nb> ]`
Pas d'initialisation possible

```
// Création d'un tableau de 1000 char
char *pc = new char[1000]; // valeurs aléatoires !
int n;
cin >> n;
double *pd = new double[2*n+1]; // 2*n+1 double
```

➢ Allocation dynamique : instruction set_new_handler

Quand l'opérateur new ne peut allouer la mémoire, il peut faire appel à une fonction de l'utilisateur, précisée grâce à `set_new_handler`

```
#include <new>
void ma_fonction() {
    cout << "Erreur allocation mémoire : sortie";
    exit(1);
}
main() { set_new_handler(&ma_fonction);
     char *p=new char [1000000000];
}
```

GESTION MEMOIRE : LIBERATION

➢ Libération : opérateur delete

Syntaxe : `delete pointeur`
Libère la mémoire allouée par new. Implémenté par une fonction
`void * operator delete (void *)`

```
int *pi= new int; // Création d'un entier
*pi = 4567;
delete pi;
```

- delete(NULL) ne provoque pas d'erreur (sauf dans les premières versions)
- delete(p) est indéfini si p n'est pas correctement affecté

➢ Libération d'un tableau alloué dynamiquement : opérateur delete []

Syntaxe : `delete [ ] pointeur`
Libère la mémoire allouée par new [ ]

```
// Attention aux []
char *pc = new char[1000];
// Création d'une chaîne « C » de 1000 char
delete [] pc; // Attention aux [] !
int n;
cin >> n;
double *pd = new double[2*n+1];
…
delete [] pd; // Attention aux []
```

➢ Remarques

1. new et delete peuvent être redéfinis. `::new` et `::delete` sont les opérateurs standards
2. new invoque un constructeur d'un objet, delete le destructeur
3. Ne pas mélanger les pointeurs obtenus avec malloc et avec new
4. Ne pas confondre delete et delete[ ]

MATRICES : TABLEAUX MULTI-DIMENSIONS

Creation de matrices, de tableaux 3D, ….

Il suffit de spécifier toutes les dimensions :

```
type identificateur[dim1][dim2]….;
```

Exemple

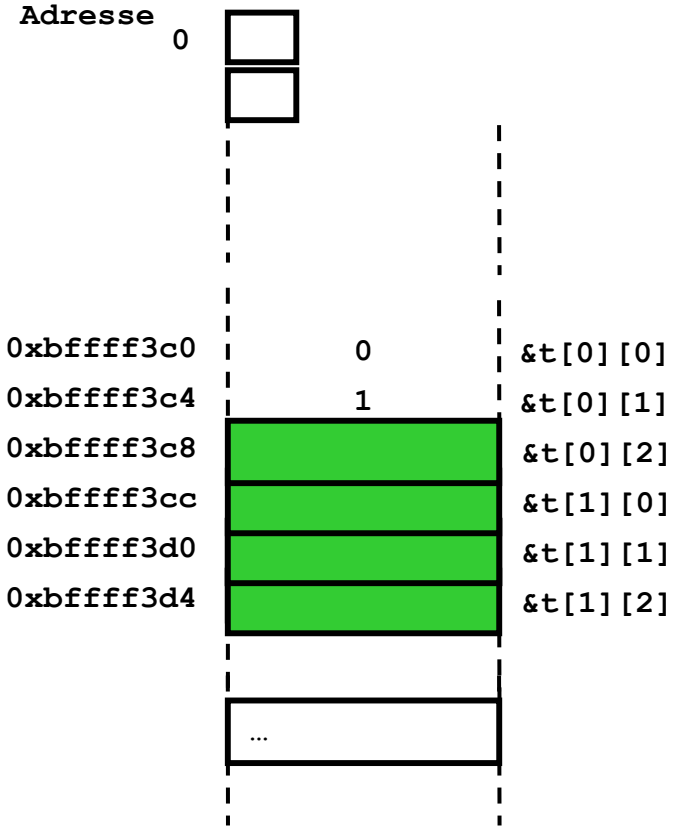
```
int t[2][3]; /* Tableau de 2 lignes et 3 colonnes */
```

Representation interne en mémoire

Les éléments sont stockés contiguement, ligne par ligne

Exemple

```
main() { int i,j;
 int t[2][3];
 for (i=0; i<2; i++)
   for (j=0; j<3; j++)
     t[i][j]=i+j;
 cout<<"Valeurs ";
 for (i=0; i<2; i++)
   for (j=0; j<3; j++)
    cout << t[i][j]<< " ";
 cout<<endl<<"Adresses ";
 for (i=0; i<2; i++)
   for (j=0; j<3; j++)
    cout <<&t[i][j]<< " ";
}
```



Acces aux elements

Comment le compilateur traduit t[i][j] d'un tableau int t[2][3];

- il prend l'adresse du premier élément) : c'est « t »
- il saute i lignes (chaque ligne contient 3 éléments), il a l'adresse de la ligne i : c'est « t+3*i »
- il saute les j premiers éléments de cette ligne : il a l'adresse de l'élément d'indice i et j : c'est « t+3*i +j»
- il utilise l'opérateur d'indirection : c'est *((int*)t+3*i+j) et c'est t[i][j]

Conclusion

Il faut connaître le nombre de colonnes pour trouver un élément du tableau

FONCTION ET TABLEAUX MULTI-DIMENSIONS

➢ Parametres de type tableaux

Les paramètres de fonctions qui sont des tableaux de N dimensions doivent préciser N-1 dimensions dans le prototype et l'entête de fonction

➢ Exemple

```
#include <iostream>
#define N_COL 5

void init_tab(int tab[][N_COL],int nl)
{ int i,j;
  for (i=0; i<nl; i++)
    for (j=0; j<N_COL; j++)
      tab[i][j]=i+j;
}

int main() { int ligne,colonne;
  int mon_tab[3][N_COL];
  int tab2[3][N_COL+1];

  init_tab(mon_tab,3);
  init_tab(tab2,3); // INTERDIT !!!!!!!!!!

  for (ligne=0; ligne<3; ligne++) {
    for (colonne=0; colonne<N_COL; colonne++)
       cout << mon_tab[ligne][colonne] << " ";
    cout << endl;
  }
}
```

➢ Conclusion

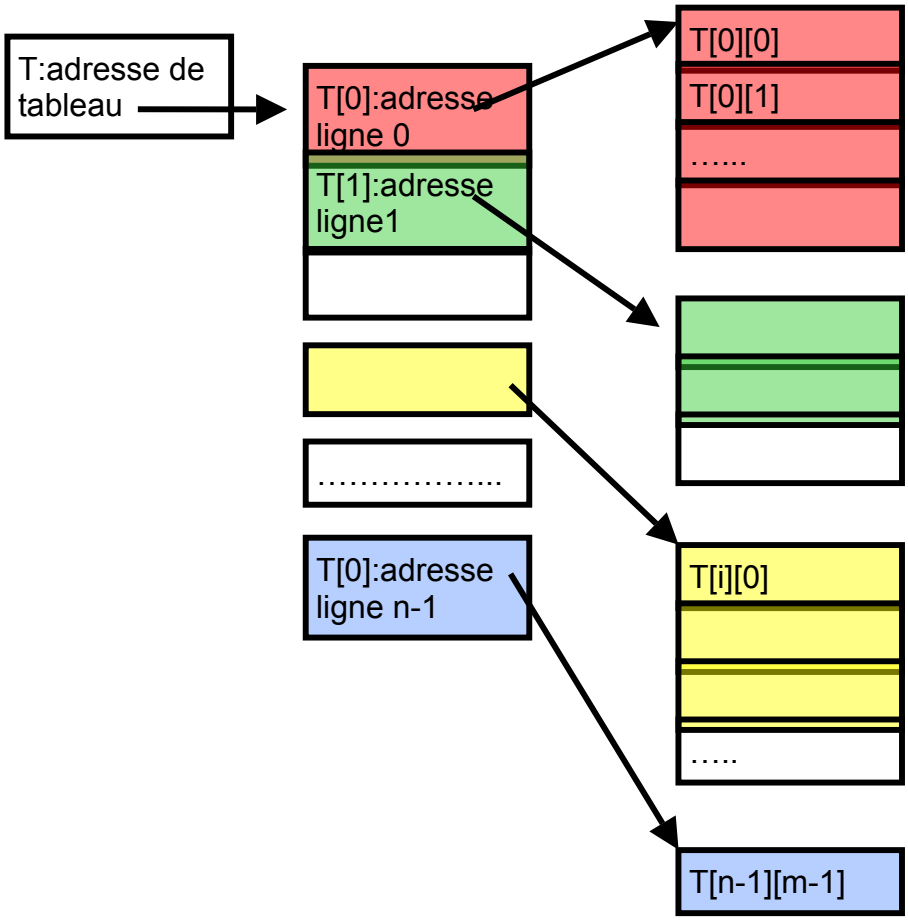
On utilise de préférence des tableaux 1D ou les pointeurs et l'allocation dynamique

MATRICES 2D DYNAMIQUES

Matrices 2D

Construction dynamique d'une matrice d'entiers de « n » lignes et « m » colonnes

1. Il faut déclarer une matrice sous la forme : **int** t;**
2. Il faut créer dynamiquement un tableau de « n » pointeurs qui contiendront chacun l'adresse d'une ligne
3. Il faut créer dynamiquement les « n » lignes qui contiendront chacune « m » elements



Trouver t[i][j]

t[i] ou bien *(t+i) contient l'adresse de la ligne i
t[i]+j ou bien (*(t+i)+j) est donc l'adresse de l'élément d'indice i,j
*(t[i]+j) ou bien *(*(t+i)+j) est donc l'élément t[i][j]

Conclusion

aucune dimension n'est nécessaire pour trouver la valeur de l'élément, les prototypes de fonctions ne doivent pas spécifier les dimensions dans ce cas

MATRICES 2D DYNAMIQUES (2)

➢ Matrices 2D

```
#include <iostream>
namespace std;

int ** alloue(int n, int m) {
  int **p ;
  p=new int* [n];
  if (p==NULL) return NULL;
  else
     for (int i=0 ; i<n ; i++) {
       p[i]=new int [m];
       if (p[i]== NULL) return NULL;
     }
  return p;
}

void destructeur(int **p, int n, int m){
  for (int i=0; i<n; i++) delete [] p[i];
  delete [] p;
}

void aff(int **m, int nl, int nc) {
  for (int i=0; i<nl; i++){
     for (int j=0;j<nc;j++)
       cout<< m[i][j]<< " ";
     cout << endl;
   }
}

main() { int a,b,i,j ;
  int** mat ;
  cout<< "dimensions ? ";
  cin >>a>>b;
  mat= alloue(a,b);
  for (i=0; i<a; i++)
     for (j=0;j<b;j++)
       mat[i][j]=i+j;
  aff(mat,a,b);
  destructeur(mat,a,b);
}
```

MATRICES 2D DYNAMIQUES (3)

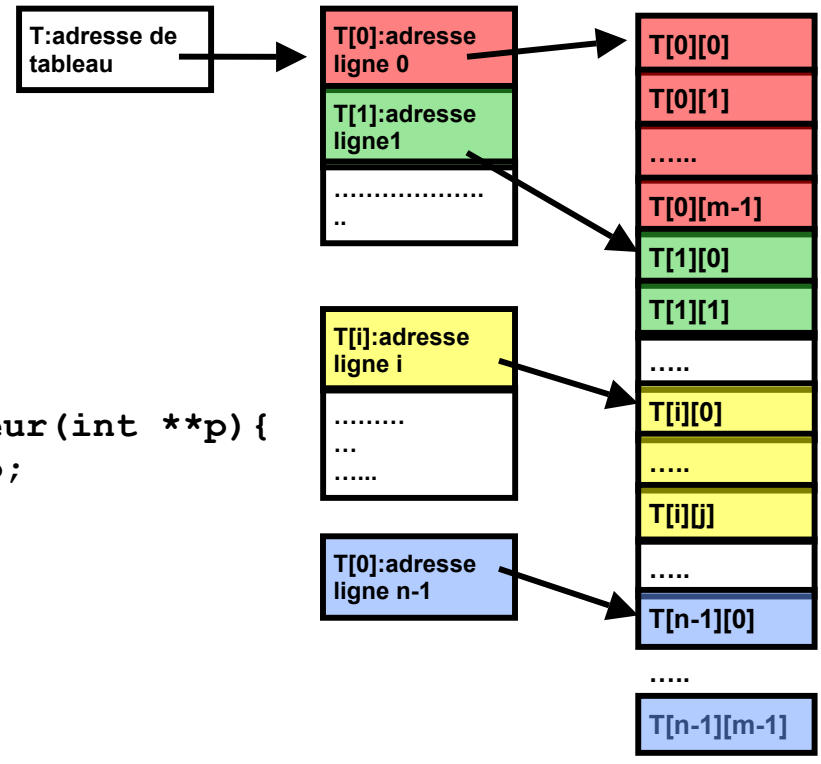
➢ Matrices 2D Version++

Toutes les lignes sont allouées en une fois et sont donc contigues
Il faut calculer les adresses des lignes

```
int ** alloue(int n, int m) {
  int **p ;
  p= new int* [n];
  *p = new int [n*m]; /* p[0]= new int [n*m];*/
  for (int i=1 ; i<n ; i++)
        p[i] = p[i-1]+m ;
  return p;
}

void copie1(int** m1, int** m2, int n, int m) {
  for (int i=0;i<n;i++)
    for(int j=0;j<m;j++)
      m1[i][j]=m2[i][j];
}

void copie2(int** m1, int** m2, int n, int m) {
  memcpy(*m1,*m2,n*m*sizeof(**m1));
}
```




```
void destructeur(int **p){
  delete [] *p;
  delete [] p;
}

}
```

SURCHARGE DE FONCTIONS

➢ Rappel : prototype d'une fonction

nom de la fonction, liste des types des paramètres, type de retour.
C++ : s'y ajoute pour les fonctions membres la classe et l'éventuel qualificateur const.

En C++, la déclaration des prototypes est obligatoire.

➢ Surcharge des fonctions

Un même nom de fonction peut être utilisé pour définir plusieurs fonctions si :

- nombre d'arguments différent
- ou au moins un des arguments est de types différents .

➢ Surcharge : déclaration

```
// Elévation à la puissance
int       pow(int x, int y);
double    pow(double x,  double y);
complex   pow(complex x, int y);
etc.
```

➢ Surcharge : appel

Lors de l'appel, une seule fonction est exécutée.
La surcharge est résolue à la compilation sur la base du type.
Choix en 3 étapes :

1. Recherche d'une correspondance exacte des types
2. Recherche d'une correspondance en employant les conversions prédéfinies du langage : `int` en `long`, `int` en `double`, `float` en `double` ...
3. Recherche d'une correspondance en employant les conversions définies par le développeur.

```
void affiche(double a)     { printf("%lf\n",a); }
void affiche(long a)       { printf("%ld\n",a); }
main(){  int i = 2; double x = 3.14;
   affiche(i);    // Conversion de i en long
                  // puis appel de     affiche(long)
   affiche(x);    // Appel direct de   affiche(double)
}
```

LA CLASSE STRING

➢ Les chaines, du C au C++

Rappel : en langage C ; pas de chaines mais des tableaux de char terminés par un caractère de fin de chaines : \0.

En C++, objets et encapsulation facilitent beaucoup les choses.
De nombreuses librairies définissent leur propre classe pour manipuler les chaines de caractères.

Nous utiliserons la classe std ::string du standard C++.

➢ La classe std::string

Premier exemple de classe dans ce cours ! page suivante.

Sortie du programme page suivante si l'utilisateur tape « Salut » :

```
taille de s1 : 13
entrez une chaine : Salut
s1: Hello World !
s2: Salut
s3 en chaine C : Hello World ! coucou Salut
s1 et s2 different
second caractere de s1: e
s4: World
Position de la chaine 'coucou' dans s3 : 14
 'plop' ne figure pas dans s3
s1 :
```

```
#include <string>
#include <iostream>
using namespace std; // évite « std:: » partout
main() {
    string s1;         // s1 : objet de type « string »
    string s2, s3, s4; // autres variables « string »

    s1 = "Hello World"; //conversion char* => string
    s1 += " !" ;        //opérateur de concaténation

    cout << "taille de s1 : " << s1.length() << endl ;

    // utilisation des string avec les flux standard c++
    cout << "entrez une chaine : ";
    cin >> s2 ;              // s2 est lue au clavier
    s3 = s1 + " coucou " + s2 ; // concaténation

    // utilisation des string avec les flux standard c++
    cout << "s1: " << s1 << endl << "s2:" << s2 << endl;

    // conversion string C++ vers chaine C
    printf("s3 en chaine C : %s\n",s3.c_str());

    // opérateur de comparaison ==.
    // Note : l'opérateur != existe aussi sur les string
    if(s1 == s2) {cout<< "s1 et s 2 sont identiques\n"; }
    else { cout << "s1 et s2 different\n" ; }

    // opérateur crochet [ ]. Ici, affiche 'e'
    cout << "second caractere de s1: " << s1[1] << endl;

    // extraction d'une sous-chaine
    s4 = s1.substr(6, 5) ;
    cout << "s4: " << s4 << endl ; // s4: World

    // recherche d'une chaine dans une chaine
    cout << "Position de la chaine 'coucou' dans s3 : "
         << s3.find("coucou") << endl ;
    if( s3.find("plop") == std::string::npos ) {
       //  std::string::npos est une constante qui vaut -1
       cout << " 'plop' ne figure pas dans s3" <<endl;
    }

    s1.clear() ; //vide s1
    cout << "s1 : " << s1 <<endl ; // s1 :
}
```

LA CLASSE STRINGSTREAM

➢ Classe stringstream : conversion string <=> valeur et variable

Comment convertir une valeur (eg : un int) en chaine de caractère ?
Comment lire une valeur (eg : un int) dans une chaine de caractère ?
Equivalent C++ des fonctions C sscanf() et sprintf() ?

On utilise une variable de type **stringstream** :
un flot (comme cin/cout) qui travaille sur / depuis la mémoire.

Ensuite, tout se passe comme avec n'importe quel flot !

➢ Exemple

```
#include <string>
#include <sstream>
using namespace std ;
main() {
    double x = 9.2;
    int i = 16;
    stringstream sStream1;
    sStream1 << "salut " << x << " " << std::hex << i;
    // conversion du stringstream en string
    string s = sStream1.str();
    cout << s <<endl ;

    string s1 = "bonjour 12 45.78";
    // nouveau stringstream
    // qui travaille sur le contenu de la chaine s1
    stringstream sStream2(s1);
    string s2 ;
    double x2;
    int i2;
    sStream2 >> s2 >> x2 >> i2;

    cout<<"s2 : "<<s2<<" x2 : "<< x2<<" i2 : "<<i2<<endl;
}
```

Ce programme affiche :

```
salut 9.2 10
s2 : bonjour x2 : 12 i2 : 45
```

REFERENCES (1)

➢ Définition

- **Synonyme pour un objet informatique** (une variable, une instance d'une classe...).
- Utilisation indifférente de la référence ou de l'objet informatique initial, quelque soit l'action réalisée.
- La référence et la chose ont même adresse, même valeur ...

➢ Syntaxe

```
type & identificateur = <variable> ;
// identificateur est une variable
// de type « reference vers type»
```

Une référence s'initialise toujours au moment de la déclaration.
On ne peut jamais changer ensuite cette initialisation.

➢ Exemple : variable

```
main() { int i, pi=NULL, j=2;
    int & ri = i;          // ri ⇔ i
    ri = 9 ;               // affecte 9 a ri... donc a i !
    ri = j ;               // affecte 2 a ri... donc a i !
    cin >> i;              // ⇔ cin >> ri;
    cout << ri;            // ⇔ cout << i;
    pi = &ri;              // ⇔ pi = &i;
    ri++;                  // Incrémente i
    &ri=j;                 // Interdit de référencer une
                           // autre variable !
}
```

➢ Exemple: Paramètre de fonction

```
// a et b sont des références à des objets
// extérieurs à la fonction, passés à la fonction !
swap( int &a, int &b){ int inter=a; a=b; b=inter; }
main() { int i,j;
    cin >> i >> j;         // Entrée de i et j
    cout << i << j;        // Affichage de i et j
    swap(i, j);
    // la fonction utilise des paramètres référence.
    // On l'exécute avec des références à i et j
    // La fonction travaille donc bien sur i et j !
    cout << i  << j;       // Affichage de i et j :
    // leurs valeurs sont inversées !
}
```

REFERENCES (2)

➢ Référence et pointeurs ? Référence, ou pointeur ?

Les notions de référence et de pointeur sont très proches.

La plupart des compilateurs C++ travaillent en fait dans votre dos avec des pointeurs lorsque votre code utilise des références.

« Quand une référence est utilisée, un pointeur pourrait l'être »

Exemple :

```
main() { int i;
int &ri = i; // ri est la même chose que i
ri =5 ;
/* Le code équivalent est :
     int *ri_assembleur = &i;
     *ri_assembleur = 5;   */
}
```

Pourquoi alors à la fois des références et des pointeurs en C++ ?

- Dans certains cas, les références permettent un code **beaucoup** plus « naturel » et « lisible ».
- Dans d'autres cas, toujours pour la clarté et la lisibilité du code, il faut au contraire éviter les références et préférer les pointeurs.

Conseils à venir plus tard dans ce cours.

REFERENCES (3)

➢ Remarques

On peut prendre l'adresse d'une référence : c'est l'adresse de la variable référencée :

```
int i = 9 ;
int &ri = i;
int *pi = &ri ;   // pi prend l'adresse... de i !
*pi = 10 ;        // i vaut 10
```

Attention : pas de références à des références, ni de références à des champs de bits, ni de tableaux de références (pb d'initialisation), ni de pointeurs sur des références (une référence n'est pas un objet « physique » : elle n'a pas d'adresse ou valeur au niveau assembleur).

➢ Référence et retour de fonction

Une référence à un objet peut être retournée par une fonction.

=> La fonction renvoie un synonyme de la variable retournée (au lieu d'une valeur)

Cela permet par exemple d'écrire :

```
int & f(int x) {...; return(aaa); }
// aaa retourné par reference !
f(i)=a+b; //=> modifie aaa via sa référence !
```

- **Exemple 1**

L'opérateur >> de cin est défini par :

```
istream & operator>> (istream &s, int &i)
{ // code lisant un entier
   return (s);
}

ce qui permet     cin >> i;
// Transcrit en istream::>>(cin,i)
cin >> i >> j;
// Transcrit en istream::>>(istream::>>(cin,j),i);
```

REFERENCES (4)

- **Exemple 2**

```
// f(int*, int) retourne une reference sur le premier
// élément nul du tableau supposé existé.
int & f (int *t, int n)
{ for (int i=0; i<n; i++)
    if (t[i]==0) return(t[i]);
}
// Le type de retour de la fonction est une
// référence, donc ce qui est retourné n'est pas
// la valeur de t[i] mais une référence à t[i] !

main() {
    int tab[10]={1,2,3,4,0,5,6,7,8,9};
    for (int i=0; i<10; ) cout << tab[i++]<< " ";
    // Affiche 1,2,3,4,0,5,6,7,8,9

    f(tab,10) = 15580; /* change la valeur du premier
                          élément du tableau qui est nul */
    for (int i=0; i<10; ) cout << tab[i++] << " ";
    // Affiche 1,2,3,4,15580,5,6,7,8,9
}
```

Attention : ne jamais retourner de référence à une variable locale à une fonction !

Ceci serait équivalent à retourner l'adresse d'une variable locale.

```
int & f (int n)  {
    int i;  // i est locale et automatique
    •••;
    return(i);// destruction de i. ERREUR de conception !
}

main()    { int a,b;
  a=f(10);
    // a est une référence à un objet qui
    // n'existe plus => plantage !
}
```

MOT CLE *CONST ET* VARIABLES *CONST* (1)

const : nouveau mot clé pour imposer la « constance » de la valeur d'un objet informatique.

➢ Constantes - Variable const

Deux syntaxes :

```
main(){
    const int i = 2;          // i est un entier constant
    double const x = 3.14;    // x est un double constant
    i = 3 ;                   // INTERDIT : i est const
    printf("%lf\n",x);        // OK
}
```

Intérêt : **garantit le typage fort** vérifié à la compilation.

➢ Const et type des variables

Le mot clé "const" fait partie du type d'une variable : une "variable entière const" n'est pas du meme type qu'une "variable entière".

Le caractère const ou non const est vérifié à la compilation.
Exemple de messages d'erreur du compilateur :

```
error: assignment of read-only location
error: invalid conversion from 'const int*' to 'int*'
```

➢ Pointeur const

Comme toute variable, un pointeur peut être constant :
<type> * **const** ptr = <adresse> ;
=> plus le droit de changer la valeur de ptr (l'endroit ou il pointe).

```
main(){
    int i = 2, j = 3;
    int * const p_i = &i ;  // p_i est un ptr constant
    printf("%d\n", *p_i);   // OK
    *p_i = 8 ;              // OK
    p_i = &j ;              // INTERDIT
}
```

MOT CLE *CONST ET* VARIABLES *CONST* (2)

➢ Pointeur et référence sur une variable const

Deux syntaxes équivalentes :

pointeurs	*références*
<type> **const** * ptr;	<type> **const** & ref = <adresse>;
const <type> * ptr;	**const** <type> & ref = <adresse>;

=> On a pas le droit d'utiliser le pointeur (ou la reference) pour *modifier* la variable

```
main(){
    int i = 2;
    const int * p_i = &i;
    // equivalent à int const * p_i = &i ;
    // p_i pointe sur int constant
    i = 9 ;                  // OK
    printf("%d\n", *p_i);    // OK
    *p_i = 8 ;               // INTERDIT

    const int & r_i = i;
    // equivalent à int const & r_i = i ;
    // r_i est une référence sur int constant
    printf("%d\n", r_i);     // OK
    r_i = 8 ;                // INTERDIT
}
```

➢ Pointeur, références et vérification de la *constness*

Un pointeur sur variable "const" peut pointer une variable "non const".
Un pointeur sur variable "non const" ne peut pas pointer une variable "const".

```
main(){
    int i = 2;
    const int * p_i = &i;    // OK
    const int & r_i = i;     // OK
    *p_i = 2 ;               // INTERDIT : *pi read_only
    r_i = 9 ;                // INTERDIT : ri read_only
    const int j = 9 ;
    const int * p_j = &j;    // OK
    int * p_jnonconst = &j; // INTERDIT
    const int & r_j = j;     // OK
    int & r_jnonconst = j;  // INTERDIT
}
```

MOT CLE *CONST ET* VARIABLES *CONST* (3)

➢ Paramètre « const pointeur » d'une fonction

La syntaxe *const ptr* pour un paramètre interdit la modification dans la fonction de la variable pointée.

Exemple : paramètre tableau "const"

```
// équivalent à void afficherTab(int const * t, int n)
void afficherTab(const int t[ ], int n){
    for (int i=0; i<10; ) cout << t[i++]<< " ";
    cout << endl ;
    t[5] = 9 ; // INTERDIT du fait du const
}
```

=> une fonction qui travaille sur un tableau ou un objet *sans le modifier* devrait toujours declarer ce parametre "const".

Exemple :

```
// signature de strcpy() :
char *strcpy(char* s1, const char* s2);
```

=> une fonction qui travaille sur un pointeur *sans modifier la variable pointée* devrait toujours declarer ce paramètre pointeur const.

➢ Paramètre « const référence » d'une fonction

La syntaxe *const reference* pour un paramètre interdit la modification dans la fonction de la variable référencée.

Exemple :

```
// équivalent à void afficherString(string const & str)
void afficherString(const string & str){
    cout << "str vaut : " << str << endl ; // OK
    str = 8 ; // INTERDIT du fait du const
}
```

Nous reviendrons sur cette notion dans la suite du cours.

MOT CLE *CONST ET* VARIABLES *CONST* (4)

➢ Exemple

Exemple :

```
 #1                    #2           #3
const int * function(const int * const p_i);
```

est équivalent à :

```
     #1                    #2       #3
int const * function(int const * const p_i);
```

#3 indique que le pointeur à gauche est *const* : **dans la fonction**, on ne pourra pas changer l'endroit ou il pointe.

```
// INTERDITS dans le code de la fonction:
p_i = <une autre adresse> ;
```

#2 indique que l'entier pointé est *const.* Permet de savoir que la fonction ne modifiera pas l'entier pointé. D**ans la fonction**, on ne pourra pas changer la valeur de cet entier

```
// INTERDITS dans le code de la fonction:
*p_i = 9 ;
p_i[6] = 2 ;
```

#1 indique que l'adresse retournée par la fonction pointe un entier const : après de l'appel de la fonction, le cone de pourra pas changer la valeur de cet entier.

```
main() {
    int k = 9 ;
    * ( function(&k) ) = 3;              // INTERDIT
      // car le ptr retourné pointe un entier const :
      // "assignment of read-only location"
    int * ptr1 = function(&k) ;          // INTERDIT
    // invalid conversion from 'const int* const' to 'int*'
    int * const ptr2 = function(&k) ;    // INTERDIT
    const int * ptr3 = function(&k) ;    // OK
    printf("%d\n", *ptr3);               // OK
    *ptr3 = 9;                           // INTERDIT
}
```

ECHANGE DE FONCTIONS ENTRE LE C ET LE C++

➢ Fonction C -> appel C++ :

Pour utiliser une librairie compilée avec un compilateur C dans un programme C++, il suffit de déclarer les prototypes des fonctions C avec la directive extern "C"

```
extern "C" { double mafct_C (double); }
main() {
    double rest = mafct_C(4) ; //appel depuis du code C++
}
```

➢ Handler C qui appelle du code C++

Pour qu'une fonction d'une librairie C puisse utiliser du code C++, : définir de nouvelles fonctions qui feront le lien avec les méthodes d'un objet particulier et imiteront ces méthodes.

Exemple : utiisation de l'algorithme qsort() de la librairie C.
Prototype de qsort (man qsort) :

```
void qsort(void *base, size_t nel, size_t width,
                  int (*compar)(const void *, const void *));
```

ou *compar()* est un *pointeur sur une fonction C* qui doit retourner la comparaison des deux arguments passés en paramètre.

```
#include <stdlib.h>
int monComparateur_C(const void * p1, const void * p2) {
    const string * s1 = static_cast<const string *> (p1);
    const string * s2 = static_cast<const string *> (p2);
    return s1->compare(*s2); //methode string::compare()
}

main() {
    string tab[3];
    tab[0] = "ef"; tab[1] = "ab" ; tab[2] = "cd";
    for(int i = 0 ; i < 3 ; i ++) cout << tab[i] << " " ;
    cout << endl;

    qsort( tab, 3, sizeof(string), monComparateur_C);
    for(int i = 0 ; i < 3 ; i ++) cout << tab[i] << " " ;
    cout << endl;
}
```

CONVERSION ; CAST

Nouveaux opérateurs de conversion, plus sécurisés et adaptés à la programmation objet.

➢ Conversion prédéfinie : `static_cast<T> (expr)`

- idem anciennes conversions
- conversions résolues à la compilation
- pas de vérifications à l'exécution
- doit être utilisé pour des conversions non-ambiguë

➢ Suppression de la constance : `const_cast<T> (expr)`

- résolu à la compilation
- Exemple :

```
void affStr(string & str ) {
    cout << "Chaine : " << str << endl;
}

void g(const string & s) {
    affStr(s);          // Erreur: s constant
                        // et affStr n'attend pas un const
    affStr(  const_cast< string & >(s)  );  // Ok
}

//Remarque : affStr(), en fait, devrait prendre un
//const string& en paramètre puisqu'elle ne modifie pas s
```

➢ Conversion dynamique : `dynamic_cast<T> (expr)`

- Utilisé sur des pointeurs ou des références dans une hiérarchie de classes

➢ Conversion de pointeur : reinterpret_cast<T> (expr)

- Permet de transformer n'importe quel pointeur en n'importe quel pointeur.
- Pas de vérification à l'exécution.
- Peut être utilisé sur des objets sans liens : dangereux

FONCTIONS INLINE

Une fonction (fonction simple ou fonction membre) déclarée **inline** est expansée par le compilateur : l'appel de la fonction est remplacé par son code.

Efficacité :

- Pas de passage de paramètres sur la pile, pas d'appel de fonction, pas de restauration de la pile.
- Très utile dans les méthodes d'accès aux membres privés.

Déclaration explicite :

```
inline int uneFctInline() { .... } ;
```

Déclaration implicite : les fonctions membres définies dans la déclaration de la classe (=> en général, dans le fichier header), sont inline .

Le corps doit être mis dans le fichier header (.h) et non pas dans le .cpp :

Fichier header MaClasse.h

```
// fonctions C
inline int fac(int n) {return i<2 ? i : i* fac(i-1) }

// fonctions membres
class A {
    //implicite dans la déclaration de la classe
    int f() { return 2 ; }
    int g() ;
} ;

// déclaration explicite de g en inline
inline int A::g() { return 3 ; }
```

Peut être ignorée par le compilateur (en particulier s'il y a des boucles à l'intérieur)
Implantation réelle dépend des compilateurs.
Exemple :

```
inline fac(int n) {return i<2 ? i : i* fac(i-1) }
```

L'appel se fait par `fac(5);`
Le compilateur générera :

- soit 720
- soit 6*5*4*3*2*1
- soit 6*fac(5)

COMPLEXITE (1)

➢ A quelles questions repond la complexité?

- Combien de temps peut prendre mon programme ?
- Se termine-t-il quelque soit le nombre de données traitées ?

➢ Complexité : représente le coût d'un algorithme

- taille mémoire
- temps d'exécution

- Regrouper dans une mesure intuitive ces informations sur le comportement d'un programme en fonction du volume des données traitées.
  - Complexité spatiale : taille mémoire
  - Complexité temporelle : définir ici les opérations coûteuses
  - Complexité pratique : programme
  - Complexité théorique : algorithme

- Complexité
  - moyenne
  - dans le pire des cas
  - dans le meilleur des cas

➢ Exemple sur un tri par insertion

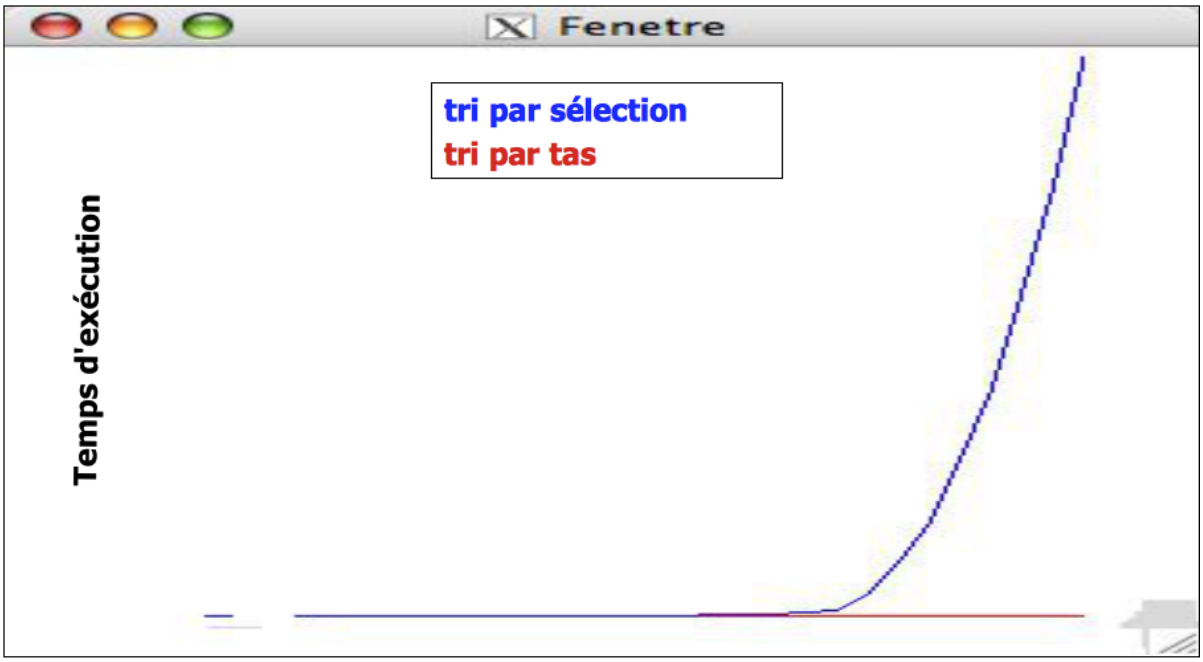
Principe : on insére le *i*-ème élément à sa place parmi ceux qui le précèdent et qui sont déjà triés.

```
void tri_insertion(int* T, int n) { int x, j ;
      for(int i=1 ; i<n ; i++)
          x = T[i]
          j = i
          while (j > 0 && T[j - 1] > x) {
              T[j] = T[j - 1] ; // On decale à droite
              j-- ;
          }
          T[j] = x
     }
}
```

Opérations couteuses : affectations et/ou comparaisons

- En moyenne et en pire cas, (n-1)*n/2 opérations
- Meilleur cas : n opeéations

COMPLEXITE (2)



➢ Mesure utilisée?

Comportement asymptotique : o(n), o(n^2), o(nlog(n))...

➢ Ordre de grandeurs

N	1	2	4	8	16	32
log	0	1	2	2	4	5
n	1	2	4	8	16	32
n log n	0	2	8	24	64	160
n^2	1	4	16	64	256	1024
n^3	1	8	64	512	4096	32768
2^n	2	4	16	256	65536	4294967296

➢ Conséquences

1. **avoir une idée de la complexité pratique**
2. **algorithme parfois impossible à utiliser**

RECURSIVITE

➢ Définition

- objet informatique autoréférent : fonction ou structure ou objet

➢ Exemple

Factorielle : n! = n x (n-1)!

```
int fac(int n) {
  return(n * fac(n-1));
}
```

Pourquoi n'est ce pas correct ?

➢ Programmation récursive:

1. Réduire le problème initial à un problème de taille inférieure
2. Trouver la formule de récurrence i.e le cas général
3. Préciser le cas connu

➢ Exemple de factorielle

1. Taille du problème :
   Paramètre n décroissant
2. Décomposition du cas général
   fac(n)=n*fac(n-1)
3. Cas particulier et arrêt
   fac(0) = 1

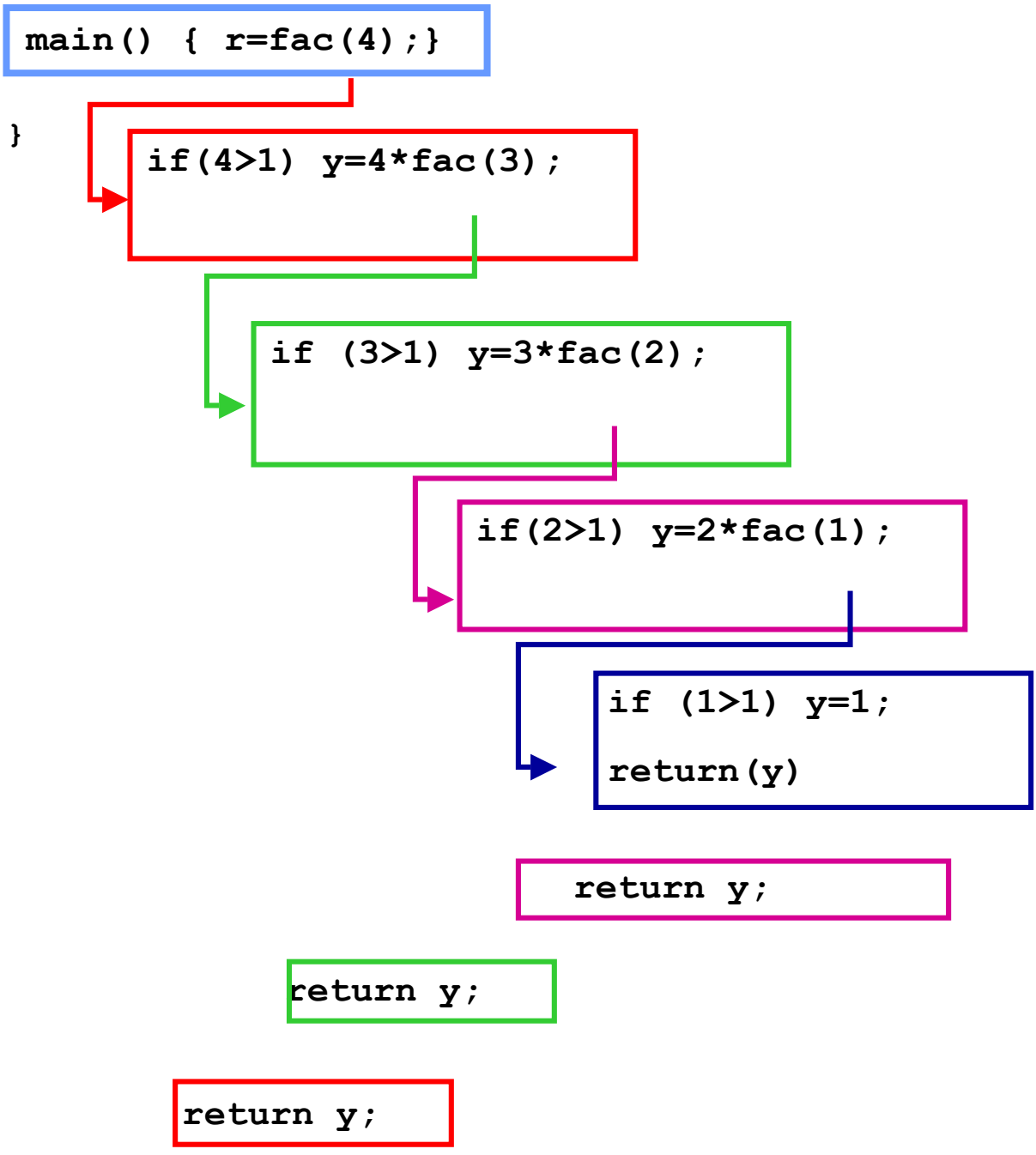
```
int fac(int n) { int y;
  cout<< "Entree dans fac avec n = "<< n<<endl;
  if(n>1) y=n*fac(n-1);
  else y=1;
  cout<< "Fin etape "<< n << "valeur :" << y << endl;
  return y;
}
main() { int a, res;
  cout << "Entrer un nombre";  cin >> a;
  res = fac(a);
  cout<< "Factorielle de "<<a<<" = " << res <<endl;
}
```

➢ Autre solution ?

RECURSIVITE (2)

➢ Que se passe t il ?

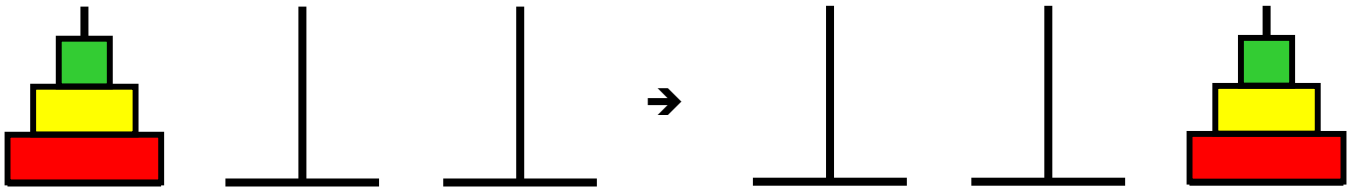
```
int fac(int n) { int y;
    if(n>1) y=n*fac(n-1);
    else y=1;
  return y;
}
```

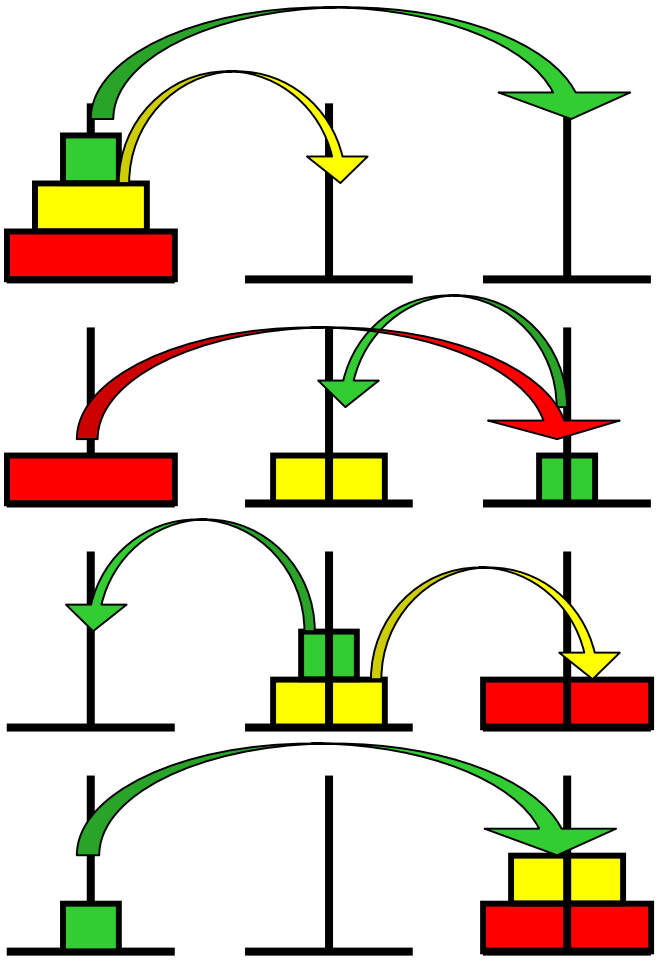


RECURSIVITE (3)

➢ Autre exemple : les tours de hanoi

- Transfert de N disques de tailles décroissantes entre 2 piquets A et B en respectant les tailles décroissantes et en utilisant un piquet intermédiaire





➢ Recursivité

1. Taille du problème :
   N : nombre de disques à transférer
2. Cas trivial :
   Transférer 1 disque
3. Cas général :
   Déplacer N disques de A vers B
   a. déplacer N-1 disques de A vers C
   b. transférer 1 disque de A vers B
   c. déplacer N-1 disques de C vers B
4. Paramètres
   a. nombres de disques
   b. piquet de depart
   c. piquet d'arrivée
   d. piquet intermédiaire

```
void hanoi(int n, int dep, int arr, int inter) {
  if (n==1)
    cout<<"deplace 1 disque de" <<dep<<" vers "<<arr<<endl;
  else {
    hanoi(n-1,dep,inter,arr);
    cout<<"deplace 1 disque de" <<dep<<" vers "<<arr<<endl;
    hanoi(n-1,inter,arr,dep);
  }
}
```

RECURSIVITE (4)

➢ Conclusion

- Simplicité de programmation
  - rapide à écrire
  -
- Coût
  - mémoire : pile
    - passage des paramètres
    - création variables locales
  - CPU
    - appel de fonctions
    - sauvegarde de « l'état précédent »

- Utilisation à bon escient
  - à éviter pour les problèmes non récursifs

➢ Récursivité croisée

Sous programme SP1 appelant SP2
Sous programme SP2 appelant SP1

➢ Exemple de Récursivité croisée : le baguenaudier

n pions à placer sur les n cases consécutives d'une tablette : ce sont les codes de Gray

Règles : "jouer un coup" signifie :
poser ou enlever un pion
soit sur la première case
soit sur la case suivant la première case occupée

Exemples

o	o			o	o		
1	2	3	4	5	6	7	8

enlever le pion de la case 1
enlever le pion de la case 2

		o		o	o		
1	2	3	4	5	6	7	8

poser le pion dans la case 1
poser le pion dans la case 4

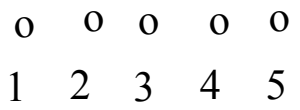
o		o	o	o			
1	2	3	4	5	6	7	8

enlever le pion dans la case 1
poser le pion dans la case 2

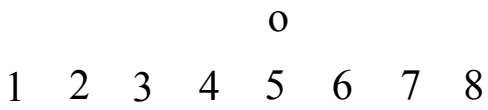
RECURSIVITE (5)

➢ Comment place n points ?

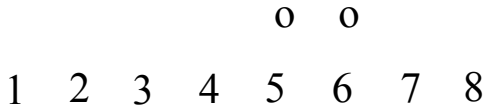
Comment placer le point 6 ?



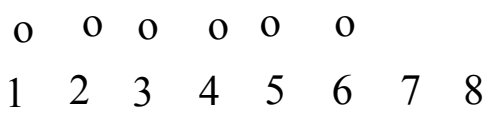
- Vider les 4 premiers
  - On ne sait pas comment, mais tant pis



- Poser le 6ieme
  - On sait comment, c'est la règle



- Remettre les 4 premiers
  - On ne sait pas comment, mais tant pis



- Conclusion
  - On a transformé un problème de rang n en 2 problèmes de rang n-2
  - On sait faire le cas n=0 ou n=1
  - Donc on sait que le problème a une solution

➢ Algorithme

Pour poser n pions, si n-1 pions sont posés :

- vider la tablette des n-2 premiers pions
- poser le $n^{ième}$
- remplir à nouveau la tablette avec les n-2 premiers pions,

```
void emplir(int* tab,int n){
  if (n>1){emplir(tab, n-1);
     vider (tab, n-2) ;
     tab[n-1]=1 ;
     emplir (tab, n-2) ;
  }
else if (n==1) tab[0]=1 ;
}
```

```
void vider(int* tab,int n) {
  if (n>1){vider (tab, n-2);
     tab[n-1]=0 ;
     emplir (tab,n-2) ;
     vider(tab, n-1) ;
}
  else if (n==1) tab[0]=0 ;
}
```