DAUPHINE
UNIVERSITÉ PARIS

Scribes ***M1-MIAGE App. Promotion 2011-2012***

Programmation C++

Notes de cours

Denis CORNAZ

1 Introduction

1.1 Bibliographie

Bjarne Stroustrup (créateur du C++) : *Le langage C++* (Pearson Education)

1.2 Généralités

- Pointeurs
- Références
- Références constantes

1.3 Mécanismes de base

1.3.1 Arithmétique et variables

- Types fondamentaux : (bool, char, int, double …)
- Opérateurs arithmétiques : (+, -, *, /, % …)
- Opérateurs comparaison : (==, !=, <, >, <=, >=)
- Tests et boucles : (while, switch, if, for)

Booléen :

```
void f(int a, int b) {
     bool b = (a == b) ;// il va d'abord résoudre (a == b)
     bool c ;  // c = true
     int i = true ; // i = 1
}
```

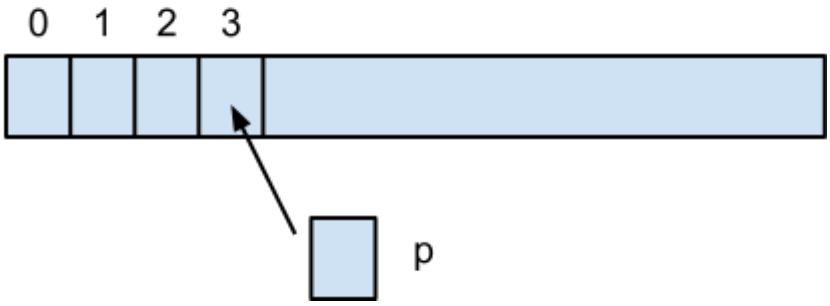
1.3.2 Résolution de portée

Portée du bloc supérieur

```
int x ;
void f() {
     int x = 1 // masque le x global
     ::x = 2 // affectation du x global
     x = 3 // affectation du x local
}
```

1.3.3 Pointeurs et tableaux

```
char v[10]; // tableau de 10 caractères

char* p; // pointeur de caractère
// char* contient l'adresse de quelque chose qui est censé contenir un caractère

p = &v[3]; // adresse, permet d'initialiser un pointeur
// p pointe sur le 4ème caractère de v
```



« * » : Opérateur unaire préfixe d'indirection

« *&* » : Opérateur unaire préfixe d'adressage

1.3.4 Constantes

```
const int model = 90;           // model est constant
const int v[]={1,2,3,4};        // v[i] est constant
const int x;                    // Erreur : pas d'initialisation

void f(){
        model = 200;      // Erreur
        v[2]++;           // Erreur
}

void g(const X* p){// Pour éviter de copier l'objet
        // On ne peut pas modifier *p
}

void h(){
       X val;       // val peut être modifiée
       g(&val);
}
```

1.3.5 Fonction membre constante

```
class A {
    int x;
    public:
    int f() const;
};

int A::f() const {
    return x++;    // Erreur : tente de modifier la valeur d'un membre
    // Fonction constante ne peut pas modifier la valeur d'un membre
}
```

Il y a toujours intérêt à déclarer ce qui est constant avec *const* pour toutes les fonctions membres qui ne modifient rien.

```
class Date{
    int d,m,y;

    public:
```

```
        int day() const { return d;}
        int month() const { return m; }
        int year() const { return y; }

       //...
       void add_year(int n) { y += m; }; // S'applique uniquement sur des objets
                                            non constants
};

void f(Date & d, const Date & cd){
    int i = d.year();       // ok
    d.add_year(1);          // ok
    int j = cd.year();      // ok

    cd.add_year(1);         // Erreur
    // Car add_year() est une fonction non constante
}
```

1.3.6 Pointeurs et constantes

```
void f1(char* p) {

    char s[]= “Truc” ;
    const char * pc = s ;  // Pointeur de constante
    pc[3]=’g’;             // Erreur : pc pointe sur const
    pc=p;                  // ok

    char * const cp = s;   // Pointeur constant
    cp[3]=’b’;             // ok
    cp = p;               // Erreur : cp est const

    const char *const cpc=s ;   // Pointeur constant de constante
    cpc[3]='a' ;                // Erreur
    cpc=p ;                     // Erreur

}
```

Une variable peut devenir une constante lorsqu'on y accède par un pointeur.

Exemple :

```
char* strcpy(char* p, const char* q) ;
//*q ne peut pas être modifié
```

```
void f2(){

    int a = 1;
    const int c = 2;
    const int* p1 = &c;   // Pointeur const sur int const
    const int* p=&a;      // Pointeur const sur int non-const

    int* p3=&c;     // Erreur. p3 ne prend que du non const
    *p3=7;          // Tente de modifier la valeur de c

}
```

1.3.7 Références et constantes

```
void f(){

    int i = 1;
    int &r = i;     // r et i font maintenant référence au même int
    int x = r;      // x=1
    r = 2;          // i=2

}
```

- Référence = pointeur constant, on n'a pas le droit de modifier l'endroit sur lequel il pointe
- Référence constante = référence normale ou référence sur quelque chose de constant

L'initialisation est indispensable :

```
int & r3;    //erreur

const int x = 3;

int & r3 = x;       //erreur

const int& r3 = x;  //ok

int x;

double &y = x;      // erreur à cause du type
```

Aucun opérateur ne s'applique à une référence.

```
void g(){

      int ii = 0;
      int & rr = ii;
      rr++;               // i++
      int * pp = &rr;     // pp pointe sur ii

}
```

***const**	Pointeur constant
const T&	Référence constante
T*	Pointeur
T&	Référence

L'initialisateur d'une référence constante constT& n'a pas besoin d'être une "left value" ou même de type T :

- Conversion de type implicite si nécessaire;
- Valeur obtenue placée dans une variable temporaire de type T;
- Cette variable temporaire utilisée pour l'initialisation.

Exemple :

```
double & dr = 1;          // erreur : lvalue requise

const double& cdr = 1;     //ok (double.temp=double(1);  const double& cdr=temp;)
```

```
class Matrix {
      double m[4][4];
      public:
            Matrix();
            friend Matrix operator+ (const Matrix&, const Matrix &);
            friend Matrix operator* (const Matrix&, const Matrix &);
};

Matrix operator+ (const Matrix& arg1, const Matrix& arg2) {
      Matrix sum;
      for(int i =0; i < 4; i++) {
            for(int j = 0; j < 4; j++) {
                  sum.m[i][j] = arg1.m[i][j] + arg2.m[i][j];
            }
      }
      return sum;
}
```

Puisque les conversions sont autorisées sur *const T&* (mais pas sur *T&*) on peut attribuer à *const T&* le même ensemble de valeur qu'à *T*;

Exemple :

```
float fsqrt(const float &);

void g(double d) {
      float r = fsqrt(2.0f);
      r = fsqrt(r);
      r = fsqrt(d);
}
```

On évite les erreurs de mise à jour de variable temporaire.

Exemple :

```
void update(float &i);

void g(double d, float r) {
      update(2.0f);             // erreur : argument constant
      update(r);
      update(d);                // erreur : conversion de type requise
}
```

2 Paradigmes supportés

Le langage C++ supporte plusieurs types de programmation :

- Programmation procédurale
- Programmation modulaire
- Programmation orientée objet
- Programmation générique
- (Abstraction des données)

2.1 Programmation procédurale

Un programme C++ c'est des fonctions + une fonction *main()*. S'il n'y a pas de fonction *main()* le programme ne compile pas.

Paradigmes :

- Choisir les procédures dont on a besoin
- Utiliser les meilleurs algorithmes

Fonction racine carré :

```
double sqrt(double arg) {
     //Code calcul de la racine carrée
}
void f() {
     double root2 = sqrt(2);
}
```

```
T1 f(T2 x) {
     // …
     return y;
 }
```

- T1 : argument retour (*return*)
- T2 : argument donné
- T1, T2 sont des types tels que *int*, *double*, *const int*, *int**, objet créé par l'utilisateur

2.2 Programmation modulaire

Un module est un ensemble de procédures connexes avec les données qu'elles manipulent.

Paradigmes :

- Choisir le module
- Découper le programme de telle sorte que les données soient masquées par ces modules
- Cloisonnement, rendre indépendant

Exemple de module : une pile LIFO

- Fournir une interface à l'utilisateur
  - *push()* : fonction qui empile
  - *pop()* : fonction qui dépile
- S'assurer que l'accès à la représentation se fait uniquement par l'interface : donner à l'utilisateur ce dont il a besoin **uniquement.**
- Initialisation

Problème : on ne gère pas les dépassements
Solution : mettre les exceptions dans l'interface

Gestion des exceptions :
Quelles sont les erreurs gérées par chaque module ?

- Ce n'est pas le module qui envoie l'exception qui sait comment la gérer.
- On créé une classe *Overflow* (dépassement)

```
Class Overflow {} ; // type représentant les exceptions de dépassements de capacité
```

Dans la représentation :

```
void Stack:push(char c) {
      if (top == MAX_SIZE) {
            throw Overflow();
      }
      // ...
}
```

On utilise un mécanisme de "try … catch" pour attraper l'erreur :

```
void f() {
      // …
      try {
            // ici les erreurs sont gérées par le gestionnaire défini après
            while (true) {
                  Stack::push('c');
            }
      } catch (Stack::Overflow) {
            // Dépassement détecté
            // Gérer l'erreur ici …
      }
}
```

2.3 Abstraction des données

Forme de type créé (ex : utilisation de plusieurs piles)
Représentation de système complexe

2.3.1 Modules définissant des types (truqués)

Gestionnaire de pile :

Stack.hpp

```
namespace Stack {       // Interface
  struct Rep;           // La définition ou représentation de la pile est ailleurs
                        // Rep est le nom d'un type qu'il reste à définir
  typedef Rep & stack; // Attribue le nom stack à une référence de Rep
                        // Une pile sera identifiée par son Stack::stack
  stack create();            // création d'une nouvelle pile
  void destroy (stack s);    // supprimer
  void push(stack s, char c); // empile c sur s
  char pop(stack s);          // récupère le sommet de s;
}
```

Tant que l'interface reste inchangée l'utilisateur n'est pas concerné.

User.cpp

```
struct Bad_pop{ };

void f() {
    Stack::stack s1 = Stack::create();
    Stack::stack s2 = Stack::create();

    Stack::push(s1, 'c');
    Stack::push(s2, 'd');


    if(Stack::pop(s1) != 'c')
        throw Bad_pop();
    if(Stack::pop(s2) != 'd')
        throw Bad_pop();

    Stack::destroy(s1);
    Stack::destroy(s2);
}
```

Stack.cpp

```
#include "stack.hpp"

namespace Stack {
   const int max_size = 200;
   Struct Rep {
       char v[max_size];
       int top;
```

```
    }

    const int max = 16; // nb max de piles
    Rep stacks[max];    // représentation des piles préallouées
    bool used[max];     // used[i] est vrai si stacks[i] est en cours d'utilisation
}
```

2.3.2 Types définis par l'utilisateur

Se comportant comme des types intégrés

Paradigmes :

- Choisir les types dont vous avez besoin
- Fournir un ensemble complet d'opérations pour chaque type

Rappel : Multiplication de deux complexes

```
z = a + i*b
i * i = -1
(a+ib)(c+id) = (ac-bd)+i(ad+bc) = z1*z2
```

Complex

La représentation *im* et *re* est privée, c'est à dire accessible uniquement par les fonctions membres ou amies (*friend*) spécifiées dans la déclaration de la classe.

Fonctions membres de nom identique à celui de sa classe = constructeurs

```
class complex {
    double re,im; // représentations privée

public :

//Construction d'un complexe à partir de deux scalaires
complex (double r, double i){
    re=r;
    im=i;
}

//Construction d'un complexe à partir d'un scalaire
complex(double r){
    re =r;
    im =0;
}

//Constructeur par défaut
complex(){
    re=im=0;
}
```

```
friend complex operator+(complex,complex);
friend complex operator-(complex,complex);
friend complex operator==(complex,complex);
friend complex operator!=(complex,complex);

} ;
```

```
friend complex operator+(complex,complex) {
   return complex (a1.re + a2.re, a1.im + a2.im);
}
friend complex operator-(complex,complex) {
   return complex (a1.re - a2.re, a1.im - a2.im);
}
friend bool operator==(complex,complex) {
   return ((a1.re == a2.re) && (a1.im == a2.im));
}
friend bool operator!=(complex,complex) {
   return ((a1.re != a2.re) || (a1.im != a2.im));
}
```

Utilisation

```
void f(complex z){
    complex a=2.3;
    complex b=1/a;
    complex c= a+b * complex(1,2.3);
    if(c!=b)
    c=-(b/a)+2*b;
}
```

2.3.2.1 Types concrets

La représentation est privée mais présente dans la classe. Si elle est modifiée (si on décide de changer la représentation), un utilisateur devra recompiler (et c'est à dire avoir le code).

Stacck.hpp

```
class Stack { //Interface

      // Représentation
      char *v;
      int top;
      int max_size;

      public:

            class Underflow{};
            class Overflow{}; // si on dépasse le max_size
            class Badsize{};

            Stack(int s); // le constructeur
```

```
            ~Stack(); // le destructeur

            void push(char c);
            char pop();

};
```

Stack obéit aux mêmes règles qu'au "int" ou "char".

```
#ifndef STACK_H
#define STACK_H

namespace Stack { //interface

char* v;
int top;
int max_size;

public:
   class Underflow{};    //
   class overflow{};    // Exceptions
   class bad_size{}:    //

   Stack(int s);         // Constructeur
   ~void Stack();        // Destructeur

   void push(char a);
   char pop();
};

#endif
```

Le constructeur *Stack(int s)* sera appelé à chaque création d'un objet de la classe. Si un nettoyage est nécessaire lorsqu'un objet ne se trouve plus dans la portée courante le destructeur est appelé. Le mécanisme de création et destruction se fait automatiquement maintenant.

2.3.2.2 Types Abstraits (virtuel)

Isolation complète des utilisateurs de la représentation et des modifications de son implémentation

```
class Stack{
    public:
         class Underflow{};
         class Overflow{};

         virtual void push(char c)  = 0;  // Classe virtuelle pure
         virtual char pop() = 0;

         // Stack ne peut pas être instanciée
         // « =0 » : Doit être impérativement implémentée
         // Sans « =0 » : Peut être définit plus loin mais pas obligatoire
}
```

Le mot clé ***virtual*** signifie : « Peut être redéfini plus tard dans une classe dérivée de celle-ci »

« = 0 » implique qu'une classe dérivée doit définir la fonction.

Utilisation

```
void f(Stack & s_ref){
    s_ref.push('c');
    if(s_ref.pop() != 'c')
        throw Bad_pop();
}
```

f() utilise l'interface de Stack indépendamment de son implémentation, une telle classe fournissant l'interface à d'autre classes est de type polymorphique.

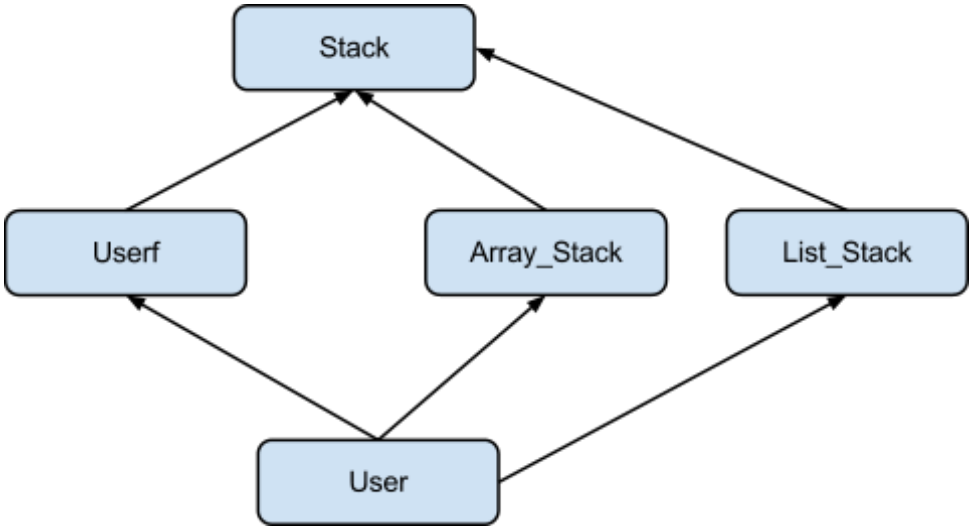
```
class Array_Stack : public Stack { //Array_Stack implémente Stack

    char * p;
    int max_size;
    int top;

    public:
      Array_Stack(int s);
      ~Array_Stack();

      void push(char c);
      char pop();
}
```

« *: public* » signifie : « est dérivée de », « implémente », « est de sous-type ».

Pour que la fonction *f()* utilise une pile *Stack* sans connaitre les détails de l'implémentation, une fonction *g()* devra créer un objet sur lequel *f()* peut opérer.

```
void g(){
    Array_Stack as(200);
    f(as);
}
```

f() ne sait qu'une chose : as est de type *Stack*. Pas besoin de recompiler *f()* si on modifie *Array_Stack*. La fonction *f()* ne connait pas *Array_Stack* mais seulement l'interface de *Stack*.




```
#include "Stack.hpp"

class List_Stack:public Stack{
    list<char> lc;

    public:

      List_Stack() {};

      void push (char c){
        lc.push_front(c);
      }

    char pop(){
       char x = lc.front();
       lc.pop_front();
       return x;
    }

}
```

2.4 Programmation orientée objet

Les types concrets ne sont pas modifiables sauf en touchant à leur définition (désavantage).

Paradigmes :

- Choisir les classes
- Fournir un ensemble complet d'opérations pour chaque classe
- Rendre toute similitude explicite à l'aide de l'héritage

Exemple :

```
class Print{ /*...*/};
class Color {/*...*/};

enum Kind {circle, triangle, square};

class Shape {

	Kind k;
	Point center;
	Color col;
	// …

	public:
		void draw();
		void rotate(int);
		// …

};

void Shape::draw {

	switch(k) {

		case circle:
			// dessine un cercle
			break;

		case triangle:
			// dessine un triangle
			break;

		case square:
			// dessine un carre
			break;

	}

}
```

drawr() doit connaître les types de formes auxquelles elle s'applique : chaque opération concernant l'ensemble des formes (shape) est lourde.

Hiérarchies de classes : Distinguer les propriétés générales des propriétés spécifiques.

```
class Shap {

	Point center;
	Color col;
	//...

	public:

		Point where() {
			return center;
```

```
            }

            void move(Point to) {
                  center = to;
                  // …
                  draw();
            }

            virtual void draw() = 0;
            virtual void rotate(int angle) =0;
            // ou virtual void rotate(int) =0; >> même chose
            //...

};
```

draw() et *rotate()* ne peuvent être définies que pour des formes spécifiques.

On peut écrire des fonctionnes générales en manipulant des vecteurs de pointeurs de forme :

```
// Fait tourner les éléments de "v" de "angle" donné

void rotate_all(vector<Shape *> & v, int angle) {

      for(int i = 0; i < v.size(); i++) {
            v[i] -> rotate(angle);
            // (*v[i]).rotate(angle);
      }
}
```

Pour définir une forme particulière, il faut le désigner comme étant une forme, puis spécifier ses propriétés.

```
class Circle: public Shape {

      int radius;



public:

      void draw() {

            // …

      }

      void rotate(int) {}  //  la fonction nulle : une rotation sur un cercle ne
sert à rien

};
```

Héritage

- Circle est dérivée (sous-classe) de shape
- Shape est une base (Super classe) de Circle → Héritage

2.5 Programmation générique

Paradigmes :

- Choisir les algorithmes
- Paramétrez-les de façon à ce qu'ils soient en mesure de fonctionner avec une variété de types et structures de données appropriées.

Conteneurs
Généralisons la « pile de caractère » à la « pile de quelque chose ».
Mot clé : *template*

Le préfixe *template<class T>* signale que *T* est un paramètre de la déclaration.

Les algorithmes génériques
Ex : trier sur des *vector*, *list*, *array* sans réécrire *sort()* ni convertir la structure de données.

Notion de séquence
C'est tout simplement des objets reliés à d'autres. Là, ce serait l'emplacement d'un conteneur relié à un autre avec un début et une fin. On va pouvoir les incrémenter et faire des vecteurs des tableaux avec.

Itérateurs (manipulation)
Un itérateur fait référence à un objet et fournit une opération lui permettant de se référer à l'élément suivant.

* (indirection) et ++ (incrémentation)

Si notion de séquence →on peut faire du générique.

```
template<class In, class Out>
void copy(In from, In too_far, Out to){
	while(from != too_far){
		*to = *from		//copie l'élément pointé
		++to;
		++from;
	}
}
```

Supporté par le pointeur et tableaux et tous les conteneur de la STL.

```
char wc1[200]             //tableau de 200 char
char wc2[500]             //tableau de 500 char
void f(){
      copy(&wc1[0], &wc2[200], &wc2[0]);
}
```

Copie wc1 de son 1er à son dernier élément dans wc2 à partir du 1er élément de wc2.

```
complex ac[200];
void g(vector<complex> & wc, list<complex> & lc){
      copy(&ac[0], &ac[200], lc.begin());
      copy(lc.begin(), lc.end(), wc.begin());
}
```

In et Out, permettent la copie d'un type de conteneur vers un autre type.
En l'occurrence, copie tableau → liste, puis liste → *vector*

Exemple de la pile

```
#include <iostream>
#include "complex.hpp" // définit l'interface
#include <list>

using namespace std;

class Bad_pop{};

template<class T> class Stack {

    T* v;
    int top;
    int max_size;

    public:

         class Underflow{};
         class Overflow{};
         class Bad_size{};

         Stack(int s);     // Taille max
         ~Stack();         // Destructeur

         void push(T c); // empile c sur s
         T pop(); // récupère le sommet de s

};

/*
 On voit les différentes utilisations des Stack (variable globale, locale, en
parametre...)
```

```
*/

            Stack<char> sc(200);                  // pile de 200 caractères
            Stack<complex> scplex(30);            // pile de 30 nombres complexes
            Stack<list<int> sli(45);              // pile de 45 listes d'entrees

void f() {

    sc.push('c');

    if(sc.pop()!='c';
            throw Bad_pop();

    scplex.push(complex(1,2));

    if(scplex.pop()!=complex(1,2))
            throw Bad_pop();

}

main() {
    f();
}

template<class T> Stack<T>::Stack(int s) {

    top=0;

    if(s<0 || s>1000)
      throw Bad_size();

    max_size=s;
    v = new T[s];                 //allocation

}

template<class T> Stack<T>::~Stack() {

    delete[] v;                   // libère de la mémoire

}

template<class T> void Stack<T>::push(T c) {

    if(top==max_size)
           throw Overflow();

    v[top]=c;
    top++;

}

template<class T> T Stack<T>::pop() {

    if(top==0)
          throw Underflow();

    top--;
    return v[top];

}
```

3 Mécanisme de l'abstraction

3.1 Objets

3.1.1 Destructeurs

```
class Name {
	const char* s;
	// ..
};

class Table {
	Name* p;
	int sz;

	public:
		// Constructeur
		Table(int s = 15) {
			p = new Name[sz = s];
		}

		// Destructeur
		~Table() {
			delete[] p;
		}

		Name* lookup (const char*);
		bool insert (Name*);
};
```

3.1.2 Constructeurs par défaut

```
Struct Tables{

	int i;
	int vi[10];
	Table t1;
	Table vt[10];

};

Tables tt;
```

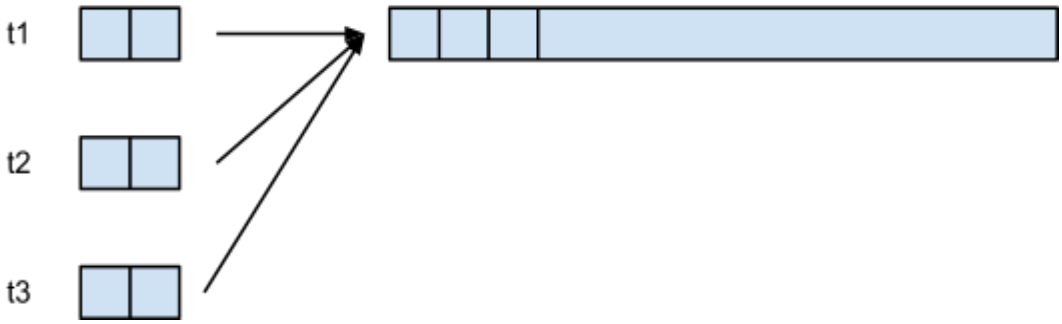
Le constructeur par défaut appelle Table(15) pour tt.t1 et tt.vt[i] pour tout i tt.i et tt.vi ne sont pas initialisés.

```
struct X {
	const int a;
	const int& r;
};
```

```
X x; // erreur : pas de constructeur par défaut de X


Table t1, t2;
t2 = t1;     // copie membres à membres de t1 dans t2
```

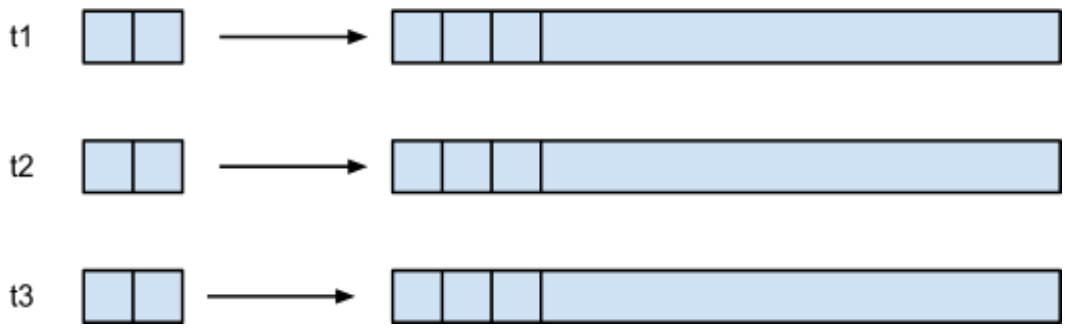
ATTENTION :

```
void h() {
        Table t1;
        Table t2 = t1;  // !
        Table t3;
        t3 = t2;        // !
}
```



Définition de la copie pour Table

```
class Table {
      // ..
      Table (const Table&);                  // Constructeur de copie
      Table& operator =  (const Table&);     // Affectation
};

Table :: Table(const Table& t) {
      p = new Name[sz = t.sz];
      for(int i = 0; i < sz; i++) {
            p[i] = t.p[i];
      }
}
```



this = Adresse de l'objet appelant

```
Table & Table::operator = (const Table& t) {
	if(this != &t) {
		delete[] p;
		p = new Name[sz = t.sz];
		for(int i = 0; i < sz; i++) {
			p[i] = t.p[i];
		}
	}
	return *this;
}
```

.

4 Remarques

TD 1

```
&x → adresse de x
T & x → reférence de T

int x=1 ;
int &r=x ;

x++ ;
r++ ;
```

Formalisme pointeur

int tab[10] : tableau de 10 élements
tab est un pointeur ! Tab <==> &tab[0]

tab++ : signifie incrémenter le pointeur
Ou bien tab+i qui incrémente i fois le pointeur

Allocation dynamique de la mémoire : *new*

```
int size;
T * x = new T[size];
ou T * x = new T;
```

Désallocation :

```
delete[] x ;        // pour la première solution d'allocation
delete x ;          // pour la seconde
```

L'utilisateur n'a pas besoin de gérer la mémoire.

Exemple :

```
Stack s_vec1(10);    // pile globale de 10 char

void f(Stack & s_ref, int i){
    Stack s_vec2(i);    //pile locale de i éléments
    Stack* s_ptr = new Stack(20);    // pointeur de Stack alloué dans le tas
(mémoire dynamique  disponible pour le programme).
    s_var1.push(''a'');
    s_var2.push(''b'');
    s_ref.push(''c'');
    s_ptr->push(''d'');
}
```

Pour un pointeur p :

p →f() ; p →atrib ; <=> (*p).f() ; (*p).atrib ;

Pour éviter les inclusions multiples, dans les fichiers « *hpp* » :

```
#ifndef _STACK_H_

#define _STACk_H_

// code...

#endif
```

5 Annexe

Arborescence répertoire de travail :

```
C++
|- APPLI
     |_______ Test
     |_______TD
                |____________ TD1
                |____________ TD2
                |____________ …..
|- Cours
        |
        Paradigmes
                |___Procedural
                                |___sqrt
                                |___.....
                |___Modulaire
                                |___ sqrt
                                |___ stack.cpp
                                |___user.cpp
                |___Abstraction
                                |___ModuleDéfinissantTypes
                                                |___stack.cpp
                                                |___ stack.hpp
                                                |___ user.cpp

                                |___ TypesDéfinisParUtilisateur
                                                |___ Complex
                                                |___ Frac
                                                |___ Type concret
                                                                |___ stack.cpp
                                                                 |___ stack.hpp
                                                                 |___ user.cpp
                                                |___ Type Abstrait
                |___ POO

                        |___ GENERIC

        MécanismesParadigmes
                |___Procedural
                        |___Base
                        |___AbstraitProcedural
        STL
```