Introduction à la programmation en Langage C

L'objectif de ce chapitre est de présenter la chaîne allant de l'écriture d'un programme à son exécution. D'introduire la notion de module avec les concepts d'interface et d'implémentation. De montrer la compilation séparée. D'illustrer les qualités d'un module/programme et de montrer comment on peut tester une fonction.

1. Braquelaire (J.-P.). – Méthodologie de la programmation en C. – Dunod, 2000.
2. Delannoy (C.). – Programmer en langage C. – Eyrolles, 1992.
3. Faber (F.). – Introduction à la programmation en ANSI-C- http://www.ltam.lu/Tutoriel_Ansi_C/.
4. Kernighan (B.W.) et Ritchie (D.M.). – The C programming language. – Prentice Hall, 1988, seconde édition.
5. Loukides (M.) et Oram (A.). – Programming with GNU software. – O'Reilly, 1997.

Le Langage C.

Le langage C est un langage de programmation qui appartient au paradigme de programmation impérative. Inventé au début des 1970 dans les Laboratoires Bell pour aider la programmation du système Unix,
C est devenu un des langages les plus utilisés. Il n'est pas consacré qu'à la programmation système.

1- Langage compilé
2- Langage typé
3- Langage avec des instructions bas-niveau (au plus près de la machine)

D'un programme à son exécution

a. Il faut vérifier que le programme est syntaxiquement correct
b. Il faut générer le code machine qui correspond aux instructions du langage.
c. Il faut assembler les différents codes pour faire un exécutable

1.1 Un exemple de programme

Exemple : Le fichier ProgSomme.c

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

int somme(int);

int
main(int argc, char **arg){
  int i = 10;
  printf("La somme des %d entiers est %d \n", i, somme(i));
return EXIT_SUCCESS ;
}
```

```
int
somme(int i){
   int resultat = 0;
   for (int k = 0; k <= i; k++)
      resultat += k;
   return resultat;
}
```

1.2 Commentaires sur le code.

1- La directive #include permet l'inclusion de fichiers stdlib.h et stdio.h
2- Il y a une différence entre définition et déclaration.
- La déclaration de la fonction somme
- La définition de la fonction somme

3- Toutes les instructions sont terminées par un « ; »
4- On définit un programme principal la fonction « main »
5- On utilise la notion de « bloc » d'instructions.
6- On utilise le type « int »
7- On définit une variable et on l'initialise en même temps
8- On utilise la fonction « printf »
9- On utilise une boucle d'instruction for (… ;…. ;…)
10- On passe des paramètres et on récupère une valeur en retour de fonction.

1.3 Illustration de la chaîne programme -> exécutable

Pour compiler le programme on utilise le compilateur gcc :
gcc –std=c99 –c ProgSomme.c
→Génération d'un fichier ProgSomme.o

a. Où est la fonction printf ???
b. Comment le compilateur vérifie-t-il que cette fonction est correctement utilisée ??
c. On ne peut pas l'exécuter.

Pour pouvoir le rendre exécutable on utilise l'éditeur de lien
gcc ProgSomme.o –o ProgSomme
→ Génération d'un fichier Somme qui est exécutable.

a. Où l'éditeur de lien a-t-il trouvé la fonction printf ?
Dans la bibliothèque standard (libstdc.a ou libc.a).
b. Définition d'une bibliothèque
c. Quel est le résultat de l'exécution
d. Le programme principal : Quelle est la fonction qui est appelée au moment du lancement de l'exécutable
int main(int argc, char **argv)

Le premier module

On découpe le même programme en plusieurs fichiers :

- **Somme.h** ; Ce fichier constitue l'interface du module « Somme ». Il contient les déclarations nécessaires à l'utilisation du module « Somme »par un client.
- **Somme.c** ; Ce fichier constitue l'implémentation du module « Somme ». Il contient les définitions (codes) nécessaires au fonctionnement du module « Somme ».
- **Client1Somme.c ;** Ce fichier est l'utilisation par un client des fonctionnalités présentées par le module « Somme ».

1.4 Le code du fichier Somme.h :

```
#ifndef _SOMME_H_
#define _SOMME_H_
extern int somme(int);
#endif
```

1.5 Le code du fichier Somme.c

```
#include "Somme.h"

int
somme(int i){
  int resultat = 0;
  for (int k = 0; k <= i; k++){
    resultat += k;
  }
  return resultat;
}
```

1.6 Le code du fichier Client1Somme.c

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

int
main(int argc, char **arg){
  int i = 10;
  printf("La somme des %d entiers est %d \n", i, somme(i));
}
```

On dit que le module Client1Somme est un client du module fournisseur Somme. Un module fournit un ensemble d'informations qui peuvent être utilisées par un module client. *Cette notion sera précisée plus tard. Mais on peut remarquer que le fichier Client1Somme inclut le fichier Somme.h pour vérifier la bonne utilisation du module Somme.*

Un module en langage C est composé de deux fichiers :
1. Le fichier « .h » représente l'interface d'un module. Il contient l'ensemble des déclarations (fonctions, variables) qui peuvent être utilisés par les clients du module. Il peut également contenir des définitions de types ainsi que des pseudo-constantes ou des macros.

De manière conceptuelle l'interface d'un module présente l'ensemble des services/variables du module qui peut-être utilisé par un des clients du module. Elle représente la perception par l'extérieur des fonctionnalités d'un module. L'interface d'un module peut évoluer, mais elle doit le faire

de manière compatible. C'est-à-dire que la manière dont un client percoit un module à un instant donné ne peut diminuer, elle ne peut que croître.

« Pourquoi ??? Est-il facile de trouver TOUS les clients d'un module ?»

2. Le fichier « .c » représente l'implémentation d'un module. Il doit fournir une implémentation (du code) à ce qui est présenté par l'interface (services, types, variables). Il s'agit donc d'une solution informatique choisie pour réaliser l'interface. Cette solution informatique peut donc évoluer pour être plus efficace, plus lisible, plus sécuritaire…. L'implémentation doit donc donner du code à tous les services décrits par l'interface et il peut y avoir aussi du code pour des services internes à l'implémentation. On peut remarquer que le fichier Somme.c inclut l'interface du module Somme à savoir le fichier Somme.h

La compilation séparée.

On doit dans un premier temps compiler séparément le module « Somme », pour cela on exécute la commande

gcc –std=c99 –c Somme.c

Puis on compile le fichier Client1Somme

gcc –std=c99 –c Client1Somme.c

On a donc obtenu deux fichiers « .o » qui sont Somme.o et Client1Somme.o. Ces deux fichiers doivent donc maintenant être assemblés pour faire un exécutable.

gcc Client1Somme.o Somme.o –o Client1Somme

On peut maintenant exécuter le programme.

1.7 Un changement d'implémentation.

On change maintenant l'implémentation du module « Somme »

```
#include "Somme.h"

int
somme(int i){
  int resultat = 0;
  while(i >=0){
    resultat += i ;
    i-- ;
  }
  return resultat;
}
```

Que doit-on refaire pour que le module Client1Somme puisse fonctionner avec la nouvelle implémentation ?

- Il faut refaire la compilation du module Somme.
- Il faut refaire l'édition de lien.

1.8 Un nouveau client du module Somme.

Que doit-on faire pour que le nouveau client puisse utiliser le module « Somme » ?

- Il faut inclure « Somme.h » dans le code du nouveau client
- Il faut compiler le module « Client2Somme ».
- Il faut faire l'édition de lien avec le module « Somme ».

1.9 Résumé.

Le fichier « .h » contient les déclarations
Le fichier « .c » contient les définitions
Le client d'un module contient les appels reflétant l'utilisation du module.
Il faut que les définitions soient en accord avec les déclarations. On inclut toujours les déclarations dans l'implémentation. C'est à dire que dans un module « Module », la première ligne du fichier « Module.c » est toujours
#include « Module.h »
Il faut que les utilisations soient en adéquation avec les déclarations. Dans un module « CLIENT » qui utilise un module « FOURNISSEUR » on met toujours l'inclusion de « FOURNISSEUR.h » dans « CLIENT.c »
Sur notre exemple les dépendances sont :
Somme.o: Somme.h Somme.c
gcc -c -std=c99 Somme.c
Client1Somme.o:Somme.h Client1Somme.c
gcc -c std=c99 Client1Somme.c
Client1Somme: Somme.o ClientSomme.o
gcc std=c99 Client1Somme.o Somme.o -o ClientSomme.
Si Somme.h change, on refait tout.
Si Somme.c change, on refait la compilation de Somme.c et l'édition de lien, on ne recompile pas Client1Somme.
Si Client1Somme.c change on refait la compilation de Client1Somme.c et on refait l'édition de lien.

Les qualités d'un programme, d'un module

Un programme doit être:
- Fiable: On peut avoir confiance dans ces résultats, on peut dire aussi « conforme à ces spécifications fonctionnelles »
- Robuste: Il peux fonctionner dans des conditions anormales sans s'arrêter.
- Extensible: On peut ajouter de nouvelles fonctionnalités, étendre le périmètre des données facilement.
- Maintenable: Facilement corrigeable, proche de l'extensibilité
- Sécurisé: Il ne peut compromettre les ressources sur lesquelles il s'exécute.

Un module doit être :
- Lisible : Facile à comprendre dès la première lecture.
- Autonome : Faiblement couplé, c'est à dire dépendre le moins possible d'autres modules.
- Maintenable: les modifications d'une partie de l'implémentation doivent impliquer un nombre minimal de modifications de code.
- (NON DUPLICATION DE CODE, séparation entre interface et inplémentation)
- Robuste et fiable (même notion que pour un programme, mais au niveau du module).

Un exemple de test.

On veut tester le module Somme que l'on vient d'écrire, comme le module ne rend qu'un seul service, il suffit de tester la fonction somme.
Les test unitaires portent sur le test d'un module, on peut :

- Tester les fonctions une à une ;
- Tester des suites d'appel de fonction.

Le test du module « Somme » (le module « TestSomme »)

Il faut écrire l'interface « TestSomme.h » et l'implémentation « TestSomme.c ».

On écrit l'interface « TestSomme.h »

```
#ifndef _Test_Somme_h_
#define _Test_Somme_h_
#include <stdbool.h>

extern bool testSomme(int);

#endif
```

On peut déjà remarquer la dépendance entre le module "TestSomme" et le module "stdbool"
On écrit maintenant l'implémentation du module « TestSomme » à savoir le fichier « TestSomme.c »

```
#include "TestSomme.h"
#include "Somme.h"

static int jeuDeTest [] = {1, 4, 9, 13, 35};

bool
testSomme(int valATester){
  int resultatTheorique = valATester*(valATester+1)/2;
  return resultatTheorique == somme(valATester);
}
```

On écrit maintenant le programme « ClientTestSomme »

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

int
main(int argc, char **argv){
  printf("debut du Test");
  int tailleJeuDeTest = sizeof(jeuDeTest)/sizeof(int)- 1;
  for(int i=0; i <= tailleJeuDeTest; i++){
    printf("Test de la valeur %d \n", jeuDeTest[i]);
    if(!testSomme(jeuDeTest[i])){
      printf("Code Faux pour la valeur %d \n",jeuDeTest[i]);
      return EXIT_FAILURE;
    }
  }
```

```
  printf("Test Reussi \n") ;
  return EXIT_SUCCESS;
}
```

Attention, il faut regarder plus précisément le jeu de Test.
Qu'est ce qui se passe avec la valeur -2 ?
Qu'est ce qui se passe avec la valeur 1000000000000 ?

Syntaxe de Base en Langage C.

Les types

Les types de bases sont : char, int, long, float, double, on peut aussi étendre les types de bases en utilisant les mots clefs "union, struct, typedef"

Un type définit une taille mémoire et un codage de cet espace. Le mot sizeof permet d'avoir le nombre d'octets occupés par un type.

```
int
main(int argc, char **argv){
  printf(" La taille d'un short %d \n", sizeof(short));
  printf(" La taille d'un int %d \n", sizeof(int));
  printf(" La taille d'un long  %d \n", sizeof(long));
  printf(" La taille d'un float %d \n", sizeof(float));
  printf(" La taille d'un double %d \n", sizeof(double));
  printf(" La taille d'un char  %d \n", sizeof(char));
  return EXIT_SUCCESS ;
}
```

On peu aussi utiliser le mot ***unsigned*** qui oblige à n'utiliser que les valeurs positives mais qui change aussi le codage et son interprétation.

Expression, instruction.

Une expression est définie comme toute combinaison qui possède une valeur, cette valeur est nécessairement associée à un type;

Exemple d'expressions :

Une constante : 'a', 1, 1.0, 1 E 23, « TOTO »
L'appel à un opérateur : 1 + 2 ; i > 2 ;
L'appel à une fonction : f(3)

Une instruction se termine par un « ; » point virgule.

Elle contient : une expression, une expression et une affectation , une déclaration de variable ou une instruction de contrôle (test, boucle sélecteur de cas ...)

Un bloc d'instructions est un ensemble d'instructions englobées par une paire d'accolades ; Cette paire d'accolade définit un contexte dans lequel on peut définir des variables.

```
{ x += 1 ; int i ; i = x – 3 ; x = i * x}
```

Les variables

Une variable est un quadruplet :

Type : représente le type de la variable et donc il définit la taille mémoire occupée par la variable et aussi le codage de cette taille mémoire.

Nom : représente l'identifiant de la variable

Valeur : représente la valeur de la variable

Adresse mémoire : représente l'emplacement de la variable ;

Exemple :

```
int i = 1 ;
```

La variable de nom i est de type int, elle a pour valeur 1 et se situe (par exemple) à l'adresse 0x2333

Pour évoquer la valeur de la variable il suffit de l'appeler;
Exemple:

```
f(i);
printf("La valeur est %d ", i);
```

Les variables ont une portée, c'est-à-dire qu'elles n'existent qu'à partir de leur définition et que dans le bloc où elles sont définies.

On appelle variable globale, une variable qui est définie en dehors de tout bloc.

Une variable peut masquer la définition d'une autre variable;

Les variables locales (variables dans un bloc, variable passée en paramètre) sont toutes stockées dans la pile.

```
int var_globale = 1; // une variable globale

int
main(int argc, char **argv){
  // argc, et argv sont des variables locales qui sont valables
  // pour toute la fonction main;

  int var_loc = var_globale;
  // var_loc est une variable locale
  // on peut l'affecter avec var_globale qui est aussi présente.

  { // un nouveau bloc
    char var_loc = 'c'; // on redéfinit la variable locale
    // on ne peut plus atteindre int var_loc
  }

  // char var_loc n'existe pas.
  // Par contre int var_loc est de nouveau accessible
}
```

L’affectation

L’affectation consiste à changer la valeur d’une variable ;

Variable = Expression ;

```
i = 1 ;
```

Si le type de l’expression est compatible avec le type de la variable alors les bits de l’expression vont devenir les bits de la variable avec possiblement une conversion.

Si les types ne sont pas compatibles on peut forcer la conversion mais c’est aux risques et périls de l’utilisateur.

Pour qu'un type soit compatible avec un autre type il faut qu'il ait une taille de codage supérieure (mais ce n'est pas suffisant).

Par exemple, short est compatible avec int et long.

Quand on convertit un short en un int ou en un long, on parle de promotion.

```
short k = 1;
long j = k; // promotion de k en long int pour affectation.
```

Quand on force la conversion d'un type de codage plus grand vers un type de codage plus petit, par exemple; un float vers un short on parle de coercition.

```
double k= 12343443,34553;
short i = (short) k;
```

Le test

if (expression booléenne) instruction ;

Si l’expression booléenne est vraie alors l’instruction est évaluée ;

En C la valeur 0 est fausse toutes les autres valeurs sont vraies.

Néanmoins, il est préférable d'utiliser des expressions booléennes ou des booléens pour montrer explicitement les conditions pour des raisons de lisibilité.

Par exemple:

```
if(i%2) // est déconseillé car on ne voit pas apparaître l'expression
booléenne
  printf("le nombre i est impair \n");
```

Il vaut mieux écrire

```
if(i%2 != 0) // plus clair
  printf("le nombre i est impair \n");
```

On peut aussi avoir une clause à exécuter si la condition booléenne est fausse.

if (condition)

intruction;

else

instruction ;

Un exemple plus complexe.

```
int
main(int argc, char **argv){
  int i = 3;
  if(i %2 == 0)
    printf(" La valeur de i est pair \n");
  if(i % 3 == 0)
    printf(" La valeur de i est un multiple de 3 \n");
  else {
    printf(" La valeur de i n'est pas un multiple de 3");
    printf(" \n");
  }
}
```

1.10

Les boucles

Trois boucles en C :

1.11 La boucle for

for(INIT ; CONDITION ; EVOLUTION)

Instruction ;

```
for(int i = 0, int j = 10; i < j; i++,j--)
  printf(" je suis dans une boucle for \n");
```

LA CLAUSE INIT EST EFFECTUÉE AVANT DE RENTRER DANS LA BOUCLE;

LA CLAUSE CONDITION EST VÉRIFIÉE AVANT DE COMMENCER LA BOUCLE;

LA CLAUSE ÉVOLUTION EST EFFECTUÉE APRÈS L'INSTRUCTION DE BOUCLE ET AVANT L'ÉVALUATION DE LA CONDITION.

1.12 La boucle while

while(CONDITION)

Instruction ;

```
int i = 0;
int j = 10;
while(i < j){
  printf(" je suis dans une boucle while \n");
  i++;
  j--;
}
```

SEULE LA CLAUSE CONDITION EST OBLIGATOIRE POUR UNE BOUCLE WHILE

MAIS L'ON VOIT QU'IL Y A UNE PARTIE INITIALISATION ET AUSSI UNE PARTIE ÉVOLUTION À L'INTÉRIEUR DE LA BOUCLE. LA CONDITION EST ÉVALUÉE SI ELLE EST VRAIE ALORS LES INSTRUCTIONS DU WHILE SONT EXÉCUTÉES, SINON ON ARRÈTE LA BOUCLE. SI LA CONDITION N'EST PAS VRAIE ON NE RENTRE PAS DANS LA BOUCLE.

1.13 La boucle do … while

do

Instruction ;

while(CONDITION) ;

```
int i = 0; int j = 10;
do {
   printf(" je suis dans une boucle while \n");
   i++;
   j--;
}while(i < j);
```

LES INSTRUCTIONS DU DO/WHILE SONT QUOIQU'IL ARRIVE EXÉCUTÉES UNE FOIS AU MOINS, ENSUITE LA BOUCLE RECOMMENCE SI LA CONDITION DU WHILE EST VÉRIFIÉE.

1.14 Les instructions d'échappement break; continue;

L'INSTRUCTION continue ARRÊTE L'ITÉRATION COURANTE QUAND ELLE RENCONTRE CETTE INSTRUCTION ET L'ITÉRATION RECOMMENCE EN DÉBUT DE BOUCLE. DANS LE CAS, D'UNE BOUCLE for LES INSTRUCTIONS CORRESPONDANT À L'ÉVOLUTION SONT EXÉCUTÉES ET L'ON RECOMMENCE À ÉVALUER LA CONDITION.

L'INSTRUCTION break FAIT PUREMENT ET SIMPLEMENT SORTIR DE LA BOUCLE.

```
int
main(int argc, char **argv){
   for(int i=0; i < 50; i++){
      if(i%2 == 0)
         continue;
      if(i == 21)
         break;
      printf(" Valeur de l'indice %d \n", i);
   }

   int i = 0;
   while(i <= 50){
      i++;
      if(i%2 == 0)
         continue;
      if(i == 21)
         break;
      printf(" Valeur de l'indice %d \n", i);
   }
}
```

Les Fonctions

Une fonction est constituée:

Un nom ;
Un type de retour ;
Une suite de paramètres ;
La définition de la fonction c'est à dire son code.

On parle de déclaration de fonction, lorsqu'on ne donne pas le code de la fonction. On emploie alors l'un des mots signature, prototype ou déclaration.

Exemple:

```
int min (int a, int b);
```

Cette fonction a pour nom min, elle prend deux paramètres: le type du premier paramètre est int et le type du deuxième paramètre est aussi un int. Cette fontion retourne un int. La définition de la fonction est alors la suivante:

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

Pour appeler la fonction min il faut lui communiquer deux paramètres. Voici, deux exemples pour l'appel de la fonction min.

```
a = min (5, 6);
int i = 1; int j = 3; b = min(i,j);
```

Les variables a et b de la fonction min sont des variables locales à cette fonction. Lors de l'appel de la fonction min, avant l'appel les données qui sont communiquées à la fonction sont placées dans la pile d'appel.
Les paramètres sont passés par valeur, mais l'on verra plus tard que l'on peut passer des références, c'est à dire des valeurs qui représentent des adresses mémoire. Cela sera abordé avec les pointeurs.
Le passage par valeur consiste à recopier les bits qui correspondent au codage du paramètre dans la pile.

Le retour d'une information par une fonction, se fait en utilisant le mot clef « return » qui est une instruction d'échappement (c'est à dire qu'elle arrête l'exécution de la fonction qui la contient).

Les tableaux

Les tableaux sont un type particulier. On peut donc définir des variables de ce type.

Par exemple:

int tab[10] ;

déclare la variable de nom tab, dont le type est int [] .

Pour un tableau, il existe une propriété particulière, à savoir que la valeur d'une variable tableau est son adresse. On ne peut changer la valeur d'un tableau. Lors de la déclaration d'un tableau, la taille mémoire nécessaire est égale à la taille mémoire d'un élément multipliée par le nombre d'éléments, l'ensemble de la mémoire réservée à un tableau est constitué d'octets consécutifs.

Sur l'exemple précédent, la taille de tab est égale à 10 * sizeof(int).

Pour un tableau de 10 éléments, les indices pour accéder aux éléments vont de 0 à 9.

Pour évoquer la valeur d'un élément d'un tableau on utilise tab[3], tab [i],....

Attention le compilateur ne vérifie pas la validité de l'indice d'un tableau, c'est à vous de le faire.. ATTENTION AU DEBORDEMENT ce qui peut entraîner des "bugs" dont les conséquences peuvent n'apparaître que beaucoup plus tard.

On peut directement initialiser un tableau sans avoir à définir sa taille.

Exemple:

```
int tab [] = {1,2,3,4,5};
```

On peut aussi créer des tableaux bidimensionnels.

```
int tab[2][5] = on crée un tableau de 2 lignes, chaque ligne à 5 colonnes.
```

Les éléments du tableau bi-dimensionnel sont consécutifs. On verra plus tard que cette implémentation ne correspond pas à tableau de tableau.

Exemple :

```
void
afficheTableau(int tab[], int taille) {
  for(int i = 0; i < taille; i++)
    printf("L'élément %d du tableau a pour valeur %d \n", i, tab[i]);
}
```

```
int
main(int argc, char **argv){
   int tab1[] = {1,2,3,4};
   printf("L'adresse du tableau est %p
             le contenu de cette adresse est%d\n", tab1,*tab1);
   afficheTableau(tab1, 4);
   for(int i = 0; i < sizeof(tab1)/sizeof(int); i++)
     tab1[i] = 0;

// attention l'expression sizeof(tab1) ne peut être utilisée
// que si la taille du tableau est connue à la compilation

   afficheTableau(tab1, 4);
}
```

1.15 Les tableaux et le passage de paramètres.

Lorsqu'on passe un tableau en paramètre à une fonction, c'est la valeur du tableau qui passe. Comme on l'a vu dans la section précédente la valeur d'un tableau est l'adresse du tableau. Donc, on passe l'adresse du tableau c'est à dire une référence aux éléments du tableau. Dans ce cas, si on change la valeur d'un élément du tableau cela aura des conséquences sur la variable qui a été passée en paramètre.

```
void
videTableau(int tab[], int taille){
   for(int i = 0; i < taille; i++)
     tab[i] = 0;
}
```

La fonction videTableau prend en paramètre un tableau, c'est à dire une référence à une adresse mémoire. Par exemple, l'appel à cette fonction avec

videTableau(tab1, 4);

fera passer les quatre premiers éléments de tab1 à 0

1.16

Les structures

Définition et utilisation d'une structure:

La déclaration de structure permet de créer un nouveau type qui est composé de plusieurs champs. Par exemple, la déclaration suivante :

```
struct complex {
  double reel;
  double imaginaire;
};
```

crée un nouveau type qui s'appelle struct complex. Une fois le nouveau type déclaré, on peut déclarer des variables de ce type, par exemple :

```
struct complex c1 = {0.0, 1.0};
```

Cette définition crée une nouvelle variable c1, qui est de type struct complex. La taille mémoire de cette variable est au moins de 2*sizeof(double).

Pour initialiser une structure on peut utiliser la même syntaxe que pour l'initialisation des tableaux.

Sur cet exemple, le champ reel de la variable c1 sera égal à 0.0 et le champ imaginaire de la variable c1 sera égal à 1.0.

Pour accéder au champ x d'une variable v qui est de type structure on utilise la notation suivante.

```
v.x;
```

Par exemple, pour la variable struct complex c1 on peut écrire:

```
c1.imaginaire = c1.reel = 1;
```

1.17 Affectation de structure

Pour l'affectation de structure, les bits de la première structure sont copiés dans la seconde structure. Si on déclare struct complex c1, c2; et si l'on écrit :

```
c1 = c2;
```

ce code est équivalent à

c1.imaginaire = c2.imaginaire; et c1.reel = c2.reel;

Il en est de même pour le passage de structure qui se fait par valeur et aussi pour la structure correspondant à un retour de fonction.

Par exemple,

```
struct complex
additionComplexe(struct complex c1, struct complex c2){
  struct complex tmp = {0,0};
  tmp.reel = c1.reel + c2.reel;
  tmp.imaginaire = c1.imaginaire + c2.imaginaire;
  return tmp;
}
```

On peut écrire le code suivant:

```
struct complex c1 = {1,1}
struct complex c2 = {2,2};
struct complex resultat = additionComplexe(c1,c2);
```

1.18 Comparaison de structures

Si on peut affecter une structure en une seule fois, ce qui correspond à une affectation champ par champ (bit à bit), on ne peut pas comparer deux structures directement.

Par exemple, l'opération c1==c2 n'est pas correcte syntaxiquement. Il faut faire alors une comparaison champ par champ.

Il faut donc écrire le code suivant :

```
(c1.reel == c2.reel ) && (c1.imaginaire == c2.imaginaire)
```

Le mot clef typedef.

Le mot clef typedef permet de définir des types synonymes.

Par exemple, la déclaration suivante.

typedef int ValeurDiscreteTemperature;

crée un nouveau type qui s'appelle ValeurDiscreteTemperature et qui est équivalent au type int.

Par exemple, le code suivant illustre la synonimie de type. On peut affecter des int à des ValeursDiscreteTemperature et réciproquement.

```
int i =1;
ValeurDiscreteTemperature v =1;
i = v;
v = i;
```

L'intérêt de l'instruction typedef est d'augmenter la lisibilité d'un programme en précisant le type d'une variable. En effet, il est plus lisible de savoir que l'on attend une variable de type ValeurDiscreteTemperature qu'une variable de type

int. De plus il sera plus facile plus tard de changer le codage de ValeurDiscreteTemperature en un long par exemple. Par le code suivant:

```
typedef long ValeurDiscreteTemperature;
```

Syntaxiquement:

La définition d'un nouveau type correspond à la déclaration d'une variable précédée du mot clef typedef, où la variable représente le nom du nouveau type.

Par exemple,

typedef struct complex Complex;

crée un nouveau type Complex qui est un synonyme de struct complex.

Les paramètres de la fonction main.

En principe le prototype de la fonction main est le suivant:

int main (int argc, char **argv)

En principe le type de retour est un int pour communiquer au système d'exploitation le bon ou le mauvais fonctionnement du programme.

Les paramètres argc et argv permettent de représenter les arguments de la ligne de commande au moment de l'appel.

Le paramètre argc est un entier qui représente le nombre de paramètres de la ligne de commande y compris le nom de l'exécutable.

Le deuxième paramètre argv est un tableau de chaîne de caractères et chaque chaîne correspond à un paramètre.

```
Int
main(int argc, char **argv){
  for (int i = 0; i < argc; i++)
    printf(" L'argument %d est %s \n", i, argv[i]);
}
```

Le programme précédent permet de vérifier le fonctionnement des arguments argc et argv.

La ligne de commande principal toto 1 tutu 3 titi produit le résultat suivant :

L'argument 0 est principal
L'argument 1 est toto
L'argument 2 est 1

L'argument 3 est tutu
L'argument 4 est 3
L'argument 5 est titi

Les informations passées à un programme permettent de paramétrer son comportement. La plupart du temps, les informations passées sont des modes de fonctionnement du programme. En principe, il existe une fonction

int usage (int argc, char **argv)

qui est en charge de vérifier que les paramètres fournis sont corrects et qui doit aussi fixer le comportement du programme en fonction des paramètres passés.

Le switch … case

L'instruction switch .... case permet d'associer une suite d'instructions en fonction de la valeur d'une constante.

On ne peut associer aux clauses case que des constantes. Par exemple, sur l'exemple ci-dessous, en fonction de la valeur de la variable i, on peut passer en mode "*francais* ou en mode "*anglais*".

```
int
main(int argc, char **argv){
  int i = atoi(argv[1]);
  char mode = 'f';
  switch(i) {
    case 1:
      printf(" Le mode francais est activé \n");
      mode = 'f';
      break;
    case 2:
      printf("Le mode anglais est activé \n");
      mode = 'a';
      break;
    default :
      printf("Le mode n'est pas prévu");
  }
}
```

Le fonctionnement du **switch/case** est le suivant:

l'ensemble des clauses *case* est parcouru selon l'ordre de leur déclaration (du premier case au dernier case).
La première clause *case* dont la constante correspond à la valeur du switch est exécutée.
Le programme continue, à moins qu'il ne soit interrompu par une instruction *break*. Si il n'y a pas d'instruction break entre les clauses "*case*", toutes les instructions des clauses case suivant la première exécution qui convient sont exécutées.
Si aucune des constantes associées aux clauses "case" ne convient, la clause *default* est exécutée si elle est présente, sinon rien ne sera exécuté.

L'utilisation du mot clef static.

Le mot clef static permet soit :

1. de créer des variables rémanentes.
2. de réduire la portée d'une variable ou d'une fonction.

Définition de variable rémanente.

Une variable rémanente est une variable locale d'une fonction qui garde sa valeur d'un appel sur l'autre.

Par exemple, soit le code suivant :

```
void
compteAppel(){
  static int nbAppel = 0;
  printf("Le nombre d'appel à la fonction est %d \n", nbAppel++);
}
```

La variable nbAppel est initialisée avec la valeur 0. Mais elle gardera sa valeur d'un appel sur l'autre. A la fin du premier appel de la fonction compteAppel, la valeur de la variable nbAppel vaudra 1. Au début du deuxième appel elle vaudra toujours 1 et à la fin du deuxième appel elle aura la valeur 2. Et ainsi de suite...

Une variable rémanente est une variable qui ne peut être stockée dans la pile car elle doit conserver sa valeur d'un appel sur l'autre. Il faut que cette variable soit stockée dans le code même de la fonction. Elle a donc une adresse qui se trouve attaché au code compilé ("fichier.o".)

L'utilisation du mot clef static pour les variables et fonctions globales dans un fichier ".c".

L'utilisation du mot clef static change la visibilité d'une variable en la réduisant au fichier où elle est déclarée.

Par exemple, la définition suivante dans le fichier "static.c"

```
static int i = 0;
```

réduit la visibilité de la variable i au seul fichier "static.c", elle ne peut être utilisée à l'extérieur de se fichier. On parle dans ce cas, de "variable globale locale".

On peut maintenant distinguer les variables :

- globales qui sont déclarées dans l'interface, définies dans un seul fichier ".c" et utilisables depuis n'importe quel autre fichier.
- locales qui sont déclarées/définies dans un bloc appartenant à une définition de fonction.
- globales/locales qui sont définies dans un fichier ".c" qui sont précédées du mot clef "static" et qui peuvent être accédées seulement dans le fichier où elles sont définies.

Le **même concept est applicable pour les fonctions**. On peut faire précéder la déclaration/définition d'une fonction du mot clef static.

Par exemple, la déclaration de la fonction "usage" qui vérifie que les paramètres de la ligne de commande sont corrects n'a pas de raison d'être appelée en dehors du fichier ".c" dans lequel la fonction main est définie. Pour ce style de fonction, on utilisera le même qualificatif que pour les variables et l'on parlera alors de fonction globale/locale.

On a vu dans le cours précédent que l'interface contenait les déclarations d'informations qui sont utilisables depuis l'extérieur du module. Principalement l'interface d'un module contient un ensemble de déclaration de fonctions.

Le rôle de l'implémentation est de fournir du code pour les fonctions déclarées dans l'interface.

Il est parfois utile de définir des variables qui servent à l'implémentation de plusieurs fonctions déclarées par l'interface. Ces variables n'ont pas à être déclarées dans l'interface et comme elles ne sont utiles qu'à l'implémentation elles doivent être déclarées globales/locales.
De même, il est aussi utile de factoriser le code entre plusieurs fonctions déclarées dans l'interface. Ces fonctions de factorisation de code n'ont de raison d'être que dans l'implémentation car elles n'ont pas à faire partie de l'interface. Elles aussi doivent être globales/locales. Ces notions seront reprises largement dans le prochain chapitre.

Pointeurs
Allocation dynamique de mémoire

Les pointeurs.

Nous avons vu au début de ce chapitre qu'une variable est un quadruplet, dont un champ est une adresse. L'idée des pointeurs est d'utiliser une adresse pour accéder au contenu de la mémoire plutôt que de passer par une variable. On va ainsi pouvoir considérer qu'une adresse mémoire peut -être la valeur d'une variable.

Un pointeur est un nouveau type. La notation d'un pointeur est alors la suivante:

```
T *pointeur;
```

Dans ce cas, la variable de nom pointeur est de type T*. avec T un type existant (int, char, struct complexe, ...).

Par exemple,

```
int * tmp;
```

est la déclaration d'une variable tmp de type int *.

Une manière d'initialiser un pointeur est d'utiliser l'adresse d'une variable existante. Pour obtenir l'adresse d'une variable on utilise l'opérateur &.

Par exemple, soit la déclaration:

int i =1;

Supposons que l'adresse de la variable i soit *0x123*, le contenu de l'adresse *0x123* contient les bits nécessaire au codage de l'entier 1. La valeur de l'expression &i est *0x123* c'est à dire l'adresse de la variable i.

On peut donc maintenant initialiser un pointeur de la façon suivante.

```
int * tmp = &i;
```

Maintenant la variable tmp est définie comme le quadruplet

< nom = tmp, valeur = *0x123*, adresse = *0x500*, taille = "la taille d'un mot mémoire">

Tous les pointeurs occuperont dont la même taille mémoire pour coder une adresse. Cette propriété est importante comme on le verra par la suite. En effet le compilateur à besoin de la taille d'un type pour pouvoir compiler le code.

Si on évoque la valeur de tmp, la valeur retournée sera *0x123.* La valeur

référencée par tmp sera le contenu de l'adresse du pointeur à savoir. la valeur contenue à l'adresse *0x123*

```
printf("La valeur du pointeur est %p \n", tmp);
```

Quand on utilise le symbole de l'affectation avec comme membre gauche un pointeur on change la référence du pointeur puisqu'on change la valeur du pointeur.

Par exemple, si on considère la variable

int j = 2; Si la variable j est à l'adresse *0x200* alors l'instruction

tmp = &j; change la valeur de tmp et maintenant le contenu de l'adresse *0x500* est *0x200*.

Donc, affecter un pointeur implique le changement de la variable qu'il référence.

Maintenant, il existe un opérateur supplémentaire pour les pointeurs. Il s'agit de l'opérateur *.

*tmp se traduit comme ''accéder au contenu du contenu de tmp''.

On a vu que écrire *tmp consiste à accéder au contenu de tmp. Comme le contenu de tmp est une adresse, écrire *tmp consiste à accéder au contenu de cette adresse. Donc, si le contenu de tmp est *0x200* alors *tmp fait référence au contenu de *0x200* c'est à dire la valeur 2.

L'opérateur * peut être utilisé soit :

- en lecture comme pour l'instruction i = *tmp; Dans ce cas, on met le contenu de j dans la variable i.
- en écriture comme pour l'instruction *tmp = i; Dans ce cas, on met le contenu de la variable i dans le contenu de l'adresse *0x200*. C'est à dire que sur cet exemple, on met le contenu de i dans la variable j;

L'avantage des pointeurs c'est que l'on peut manipuler le contenu de la mémoire comme une valeur. Ce qui permet de changer des valeurs de variables sans connaître leur nom.

Les passages de références dans les appels de fonctions.

Une première utilisation des pointeurs est, pour des fonctions, de simuler un passage de paramètre par référence. On a vu que le passage des paramètres en C se fait par valeur. Mais quand on passe une valeur qui représente une référence, on a un passage par référence.

Par exemple, si on considère la fonction suivante.

```
void
remiseAZero(int *tmp){
   *tmp = 0;
}
```

Cette fonction prend le contenu du contenu de tmp. Comme tmp represente une adresse, on met le contenu de cette adresse à zéro.

Si on considère la variable i précédente.

int i =1;

Supposons que l'adresse de la variable i soit *0x123*, le contenu de l'adresse *0x123* contient les bits nécessaire au codage de l'entier 1.

remiseAZero(&i);

Consiste à appeler la fonction remiseAZero avec la valeur *0x123* qui représente l'adresse de la variable i. Le contenu de la variable tmp de la fonction remiseAZero est donc *0x123*, c'est à dire la valeur passée en paramètre à savoir l'adresse de la variable i.

Comme précédemment en écrivant, *tmp = 0, on met le contenu de l'adresse *0x123* à zéro. Donc, la fonction remiseAzero modifie la valeur d'une variable existant dans l'appelant. On parle dans ce cas d'effet de bord.

Le programme complet est alors le suivant.

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

static void
remiseAZero(int *tmp){
   *tmp = 0;
}

int
main(int argc, char **argv){
   int i = 1;
   printf("La valeur de i avant l'appel %d \n", i);
   remiseAZero(&i);
   printf("La valeur de i après l'appel %d \n", i);
}
```

Une autre illustration des effets de bords est la fonction swap

```
void
swap(int *x, int *y){
   int tmp = *x;
   *x = *y;
   *y = tmp;
}
```

Cette fonction commence par sauvegarder le contenu de l'adresse référencée par le pointeur x, puis elle change le contenu de l'adresse référencée par le pointeur

x en lui mettant le contenu de l'adresse référencée par le pointeur y. Et enfin, elle change le contenu de l'adresse référencée par le pointeur y en l'affectant à tmp. Le programme suivant est une illustration de l'utilisation de la fonction swap.

```
int
main(int argc, char **argv){
   int a = 1;
   int b = 2;
   printf ("la valeur de a est %d la valeur de b est %d \n", a, b);
   swap(&a,&b);
   printf ("la valeur de a est %d la valeur de b est %d \n", a, b);
}
```

Les pointeurs et les tableaux

Un tableau est un pointeur particulier. En effet, la valeur d'une variable de type tableau est égale à l'adresse de la variable elle même. Pour une déclaration int tab[3]; l'expression tab == &tab est donc toujours vraie.

Par contre, on ne peut changer le contenu de la variable tab car il faut que la propriété précédente soit toujours vraie pour un tableau. On ne peut donc écrire une instruction comme tab = &x ou bien comme tab++;

Par contre on peut très bien utiliser un pointeur pour référencer un tableau. Une instruction comme int *tmp = tab est tout à fait correcte. Dans ce cas, l'adresse de tab qui est aussi la valeur de tab est maintenant référencée par le pointeur tmp.

Le fait qu'un tableau soit un pointeur particulier explique pourquoi les effets de bord sont possibles lorsqu'une fonction prend en paramètre un tableau. C'est la cas par exemple de la fonction viderTableau.

Enfin, on peut manipuler les tableau en utilisant la notation * et on peut également manipuler un pointeur en utilisant la notation []. Tous ces éléments sont résumés sur le programme suivant.

```
#include <stdio.h>
void
videTableau(int tab[], int taille){
   for(int i = 0; i < taille; i++)
     tab[i] = 0;
}

void
afficheTableau(int *tab, int taille){
   for(int i = 0; i < taille; i++)
     printf("L'élément%d du tableau a pour valeur %d \n", i, tab[i]);
}
```

```
int
main (int argc, char **argv){
  int tab[] = {1,2,3,4,5,6,7};
  printf("l'adresse de la variable tab est %p sa valeur est %p \n", &tab,
          tab);
  printf(" Le contenu du premier élément du tableau tab %d \n",  tab);
  int *tmp = tab;
  for(int i = 0; i < 7; i++)
    tmp[i] = tmp[i] * 2;
  afficheTableau(tab, 7);
  videTableau (tmp, 7);
  afficheTableau(tab, 7);
}
```

L'arithmétique des pointeurs.

La valeur d'un pointeur est une adresse, par contre le contenu de l'adresse pointée dépend du type du pointeur et donc de la taille et du codage du type pointé.

Pour un pointeur de type T, il faut sizeof(T) octets pour coder le type.

Ainsi les opérateur d'addition et de soustraction pour les pointeurs sont spécifiques au pointeur. Soit un pointeur déclaré de la manière suivante:

T *tmp;

Si on écrit tmp + 1 l'interprétation est la suivante :
il ne s'agit pas d'avancer d'un mot mémoire mais de passer à l'adresse qui code l'élément pointé suivant.. La valeur de tmp + 1 est donc égale à la valeur de tmp + sizeof(T).

Comme on peut adresser un pointeur en utilisant les crochets on a l'équivalence suivante :
tmp[i] est équivalente *(tmp + i) .
L'expression tmp + i donne comme résultat l'adresse contenue dans tmp + i*sizeof(T).

L'opération de soustraction avec un entier produit le même résultat.
L'expression tmp - i donne comme résultat l'adresse contenue dans tmp - i*sizeof(T).

Les opérateurs ++ et -- donnent exactement les mêmes résultats.

La soustraction entre pointeurs de même type est elle aussi définie.
Soient les déclarations suivantes.

```
   int tab[10];
   int *tmpDebut = tab + 1;

   int *tmpFin = tab +5;
```

La valeur tmpFin - tmpDebut doit représenter le nombre d'éléments entre tmpFin et tmpDebut. Le résultat de cette opération est donc le suivant:

tmpFin - tmpDebut = (tab + 5*sizeof(int) - (tab + sizeof(int)) / sizeof(int) = 4* sizeof(int) / sizeof(int) = 4.

```
int
main(int argc, char **argv){
   int *tmpInt = (void *) 0;
   printf(" SIZEOF(int) = %ld \n", sizeof(int));
   printf("La valeur de tmpInt est %p la valeur de tmpInt+1 est %p \n",
          tmpInt, tmpInt+1);
   printf ("La valeur de ++tmpInt est %p \n", ++tmpInt);
   char *tmpChar = (void *) 0;
   printf(" SIZEOF(char) = %ld \n", sizeof(char));
   printf("La valeur de tmpChar est %p la valeur de tmpChar+1 est %p \n",
          tmpChar, tmpChar+1);
   printf ("La valeur de tmpChar++ est %p \n", ++tmpChar);
   int *tmpIntDebut = (void *) 0;
   int *tmpIntFin   = tmpIntDebut + 4;
   printf("Debut %p Fin %p nbElements %ld \n ",tmpIntDebut, tmpIntFin,
          tmpIntFin – tmpIntDebut);
   return EXIT_SUCCESS ;

}
```

Le pointeur générique void *.

On a vu dans la section précédente que tous les pointeurs occupaient la même place mémoire. A savoir le nombre d'octets nécessaire pour adresser la mémoire. On a vu également que pour effectuer des opérations comme l'addition, la soustraction avec un entier ou bien la différence entre deux pointeurs, il fallait connaître la taille du type pour effectuer correctement ces opérations.

Il existe un type de pointeur particulier qui est le type void *.

Ce type de pointeur est appelé pointeur générique.

Ce type de pointeur permet d'adresser la mémoire comme tout autre pointeur mais on ne peut pas utiliser ce pointeur pour faire les opérations de déplacement dans la mémoire. Car si on déplace d'octet en octet cela n'a plus aucun sens. On ne peut même pas évoquer son contenu puisqu'on ne connaît pas la taille du type qui est associé à l'adresse mémoire.

Par exemple,

void *tmp;

On ne peut écrire tmp++, *tmp, tmp +1, ce sont des instructions dangereuses voir interdites.

L'intérêt du type **void *** est de pouvoir unifier tous les types de pointeurs. En effet, on peut convertir tous les pointeurs vers le type **void *** et réciproquement.

```
int
main(int argc, char **argv){
   void *tmp = (void *) 0;
   int *tmpInt = NULL;
   tmp = tmpInt;
   tmpInt = tmp;
}
```

On verra l'intérêt de ce type de pointeur pour la factorisation de code.

La mémoire dynamique.

Pour l'instant, pour utiliser de la mémoire il est nécessaire de déclarer des variables. Ce qui implique que l'on connaît au moment de la compilation toutes les variables nécessaires ou alors il existe un risque que le programme ne puisse plus continuer faute de mémoire ou alors qu'il consomme trop de mémoire ce qui peut le pénaliser en terme de performance.

Supposons par exemple, qu'un programme stocke des formulaires de connexions journalières. Le nombre de connexion varie d'un jour sur l'autre et il est difficile de savoir à l'avance combien de connexions sont possibles, deux solutions sont alors possibles pour définir la variable JournalConnexion.

- Soit Connexion journalConnexion [1000] et il se peut que le programme s'arrête (si il y a plus de 1000 Connexion).
- Soit Connexion journalConnexion [100000000000] et ce programme risque de passer son temps à swaper sur le disque, ce qui le rendra inutilisable.

La solution consiste à allouer de la mémoire dynamiquement c'est à dire pendant que le programme s'exécute.

Il existe trois types de mémoire différente pour un processus:

1. La mémoire statique. C'est la mémoire qui est utilisée par les variables globales ou les variables rémanentes d'un programme. Cette mémoire est attachée au code, elle est contenue dans le code compilé.
2. La mémoire de pile, qui est utilisée, entre autre, pour les variables locales au fur et à mesure de l'exécution du programme. Cette mémoire n'est valable que tant que la fonction qui la définit n'est pas terminée.
3. La mémoire dynamique qui est créée au fur et à mesure des besoins du programme, une fois cette mémoire allouée elle continue à être utilisable tant qu'elle n'est pas libérée. Son contenu continue donc à exister et à être

accessible même si la fonction qui a alloué cette mémoire est terminée.

Les primitives de bases pour la gestion de la mémoire dynamique sont :

- void *malloc(size_t nbOctets). Cette fonction alloue un espace mémoire qui peut contenir au moins nbOctets consécutifs de mémoire utilisable. La valeur retournée par cette fonction est l'adresse du premier octet utilisable. La mémoire n'est pas du tout initialisée par cette fonction.
  exemple: malloc(100*sizeof(int)) alloue dynamiquement un tableau de 100 entiers.

- void *calloc(size_t nbElements, size_t tailleElement), même chose que malloc mais la mémoire est initialisée à zéro.
  exemple: calloc(100,sizeof(int)) alloue dynamiquement un tableau de 100 entiers.

- void *realloc(void *origine, size_t nbOctets). cette fonction peut agrandir ou rétrécir la zone mémoire origine qui a déjà été allouée dynamiquement. La variable nbOctets représente alors la taille de la nouvelle zone. Comme pour malloc la valeur retournée est l'adresse du premier octet utilisable. De plus, le contenu pointé par origine est copié à l'adresse retournée. On garde ainsi les informations qui étaient présentes dans l'ancien espace mémoire. Par contre, l'ancienne mémoire qui est pointée par origine a été libérée, elle n'est donc plus valide et ne peut plus être utilisée.
  exemple:

```
int *tmp = malloc(100*sizeof(int));
…
tmp = realloc(tmp, 200*sizeof(int));
```

- void free(void *memoireDynamique). Cette fonction sert à libérer la mémoire allouée dynamiquement.

Un exemple récapitulatif : Le module VecteurUnique.

Il s'agit de faire un vecteur extensible unique qui permet d'écrire et de lire dans un tableau sans limitation de taille. Cet exemple permet de revoir la plus part des notions présentées dans ce chapitre. On verra dans le chapitre suivant comment le faire évoluer.

VECTEURUNIQUE.H

```
#ifndef _VECTEUR_
#define _VECTEUR_

extern void reInitaliseVecteur(void);
/* Elle remet le vecteur dans un état d'origine */

extern int  lireElement(int indice);
/* La fonction LireElement ne marche que si indice est positif ou nul
  et que indice représente un indice valide
  (indice inférieur ou égal a max indice).
*/

extern void ecrireElement(int indice, int element);
/*Cette fonction  ne marche que si indice est positif ou nul */

extern int indiceMax(void);

/*
extern int donneIndice(int element);

Cette fonction est redondante, elle ne respecte pas la minimalité de
l'interface.On peut l'implémenter en utilisant les fonctions lireElement et
indiceMax. Néanmoins, il peut-être intéressant de l'écrire pour des raisons
d'efficacité et pour éviter de la duplication de code.
*/

#endif
```

VECTEURUNIQUE.C

```
#include "VecteurUnique.h"
#include <assert.h>
#include <malloc.h>

/* DEFINIR LES DONNEES : TYPES, CONSTANTES, VARIABLES....*/

static int *tab = (void *) 0; //NULL
// globale locale....

static const int tailleDefault = 10;
static int maxIndice = -1;
```

```
allongerVecteur(int indice){
  tab = realloc(tab, (indice + tailleDefault) * sizeof(int));
  for(int i = maxIndice + 1; i < indice + tailleDefault; i++)
    tab[i] = 0;
  maxIndice = (indice + tailleDefault) – 1;
}

void
reInitaliseVecteur(void){
  if(tab != (void *) 0)
    free(tab);
  tab = malloc(sizeof(int)* tailleDefault);
  maxIndice = tailleDefault – 1;
  for(int i = 0; i <= maxIndice; i++)
    tab[i] = 0;
}

int
lireElement(int indice){
  assert(!((indice < 0 || indice > maxIndice ) || tab == NULL
  return tab[indice];
}

void
ecrireElement(int indice, int element){
  assert( ! (indice < 0 || tab == (void *) 0));
  if(indice >= maxIndice)
    allongerVecteur(indice);
  *(tab + indice) = element;
}

int
indiceMax(void){
  return maxIndice;
}
```

Programmation Modulaire
Partie 1.

Module Bibliothèque et Module instance.

Un module bibliothèque est un module qui regroupe plusieurs fonctions qui ont une même sémantique. Il s'agit par exemple de la bibliothèque mathématique, de la bibliothèque arithmétique etc. Ce type de module ne déclare pas de nouveau type.

Un module instance est un module qui déclare un nouveau type et les fonctions qui permettent de manipuler ce type. Dans un module instance, il y aura au moins deux fonctions qui sont :

1. La fonction de création d'une instance du type.
2. La fonction de destruction d'une instance du type.

1.19 Schéma d'un module d'instance.

Tous les modules d'instance ont un schéma commun. Par exemple le module « NouveauType » est composé d'une interface et d'une implémentation. L'interface contient au moins la déclaration du nouveau type et les fonctions de création et de destruction.

1 L'interface d'un module instance

```
#ifndef _NOUVEAUTYPE_H
#define _NOUVEAUTYPE_H

typedef struct nouveauType *NouveauType;
extern NouveauType creerNouveauType();
extern void libererNouveauType(NouveauType self);

#endif
```

2 L'implémentation d'un module instance.

```
#include « NouveauType.h »
struct nouveauType {
….
} ;



NouveauType
creerNouveauType(){
  NouveauType self = malloc(sizeof(struct nouveauType));

  .....

  return self;
}
```

```
void
libererNouveauType(NouveauType self){
  free(self);
}
```

3 L'abstraction de type par pointeur.

L'instruction t**ypedef struct nouveauType *NouveauType;** sert à déclarer le type NouveauType qui est un pointeur sur la structure **struct nouveauType *.** Cette structure sera définie dans l'implémentation.

Cette technique s'appelle l'abstraction de type par pointeur, car on abstrait le type en utilisant un pointeur. Cette solution est possible car en C tous les pointeurs ont la même taille, la taille est la seule information dont a besoin le compilateur pour générer le code.

L'abstraction de pointeur permet de créer un nouveau type qui sera manipulable par les clients du module. Mais il permet aussi d'assurer la séparation entre l'interface et l'implémentation. En effet, le client ne connaît que le pointeur sur la structure, il ne connaît donc pas les champs de la structure et donc il ne connaît pas l'implémentation de la structure. On pourra changer les champs de la structure sans devoir reprendre le code des clients. On remarquera que la structure n'est définie que dans l'implémentation du module (cf. section précédente). Toutes les instances d'un module d'instance sont des adresses mémoires. Une variable de ce type représente alors une référence à une adresse mémoire. On dira que les instances d'un module d'instance sont référencées par des variables.

4 Les fonctions creer et liberer.

L'implémentation d'un module d'instance contient la définition de la structure représentant le type ainsi que les fonctions creerNouveauType et libererNouveauType.

La fonction **creerNouveauType** doit construire une instance, c'est à dire allouer la place nécessaire pour la structure qui représente une instance du nouveau type. L'allocation mémoire se fait grâce à l'instruction **malloc(sizeof(struct nouveauType)).** Bien sûr si des champs de la structure réclament eux aussi de la place mémoire allouée dynamiquement c'est dans cette fonction que se fera l'allocation dynamique.

La fonction **libererNouveauType(NouveauType self)** doit libérer la place mémoire occupée par l'instance. Cette libération se fait avec la fonction **free(self).** Si l'instance avait alloué de la place mémoire pour ces propres besoins cette mémoire sera aussi libérée dans cette fonction.

1.20 Un client du module.

```
#include "NouveauType.h"
int
main(int argc, char **argv){
  NouveauType instance1 = creerNouveauType();

  ......

  libererNouveauType(instance1);
}
```

Sur cet exemple, on inclut l'interface de nouveau type pour pouvoir le manipuler. Ensuite on peut créer

des instances du nouveau type en utilisant la fonction de création. La variable instance1 est donc une variable qui référence (pointe sur) une instance du nouveau type. Il faut ensuite libérer la place mémoire occupée lorsqu'on n'en a plus besoin.

1.21 Un exemple complet de module d'instance « Complexe ».

On veut maintenant dans une application, manipuler des complexes comme l'on manipule des entiers, des réels, etc.... On veut donc ajouter un nouveau type au langage C. Pour cela, on va définir un module instance qui représentera le type Complexe.

Le module « Complexe » est donc un module type. On va donc définir l'interface de ce nouveau type de telle manière qu'elle soit complète et qu'elle assure la séparation avec son implémentation . On reviendra plus tard sur la minimalité.

```
#ifndef _COMPLEXE_H
#define _COMPLEXE_H


#include <stdbool.h>

typedef struct complexe * Complexe;

extern Complexe creerComplexe(double reel, double img);
extern Complexe copierComplexe(Complexe);
extern void detruireComplexe(Complexe);

extern double imaginaireComplexe(Complexe);
extern double reelleComplexe(Complexe);
extern double moduleComplexe(Complexe);
extern double argumentComplexe(Complexe);

extern Complexe additionComplexe(Complexe,Complexe);
extern Complexe soustractionComplexe(Complexe,Complexe );
extern Complexe multiplicationComplexe(Complexe, Complexe);
extern Complexe divisionComplexe(Complexe, Complexe);
extern bool egaliteComplexe(Complexe, Complexe);

extern Complexe opposeComplexe(Complexe);
extern Complexe inverseComplexe(Complexe);
extern Complexe conjugeComplexe(Complexe);
extern void affectationComplexe(Complexe, Complexe);

#endif
```

Abstraction de type :

Comme « Complexe » est un module type, on doit commencer par définir l'abstraction de pointeur pour déclarer le type Complexe.

```
typedef struct complexe * Complexe;
```

On a bien déclaré Complexe comme un pointeur sur struct complexe. On a bien la séparation vis à vis de l'implémentation car on ne connaît pas la définition de la structure et on ne sait pas si le complexe est implémenté en utilisant :

1. la partie réelle et la partie imaginaire
2. le module est l'argument.

Le choix sera fait dans l'implémentation.

Fonction de création et de libération:

Comme Complexe est un module type, il doit avoir les deux fonctions :

```
extern Complexe creerComplexe(double reel, double img);

extern void detruireComplexe(Complexe c);
```

La fonction **creerComplexe(double reel, double img);** crée un nouveau complexe. Mais elle a besoin de paramètres pour créer un nouveau complexe. On a décidé que l'information dont on avait besoin était la partie réelle et la partie imaginaire. Mais ce n'est qu'un choix d'interface, cela ne vaut pas dire que c'est le choix retenu pour l'implémentation.

La fonction detruireComplexe(Complexe c); doit, elle, libérer la place mémoire occupée par le complexe.

La fonction **Complexe copierComplexe(Complexe c);** est analogue à la fonction creerComplexe. En effet, elle doit produire elle aussi au nouveau complexe mais au lieu de prendre comme paramètre la partie imaginaire et la partie réelle, elle prend comme paramètre un complexe existant. Elle crée un nouveau complexe qui a les mêmes valeurs que le complexe passé en paramètre.

Les accesseurs:

Les fonctions accesseurs

```
extern double imaginaireComplexe(Complexe);
extern double reelleComplexe(Complexe );
extern double moduleComplexe(Complexe);
extern double argumentComplexe(Complexe);
```

permettent d'accéder aux différentes informations qui constituent une instance du module complexe. Dans ces fonctions, il y a deux familles, celle qui considèrent qu'un complexe possède une partie réelle et une partie imaginaire et celle qui considère qu'un complexe possède un module et un argument. Afin de pouvoir assurer l'indépendance entre l'implémentation et l'interface il est nécessaire de représenter ces deux familles, car sinon l'implémentation pourrait transparaître dans l'interface.

Pour ces quatre fonctions, on demande à une instance de fournir certaines de ses propres valeurs. Dans ce cas, l'instance est active car elle est concernée par l'action à effectuer. C'est elle qui est active car elle doit fournir ou faire quelque chose. Ce n'est pas la cas, pour les fonctions creerComplexe ou libererComplexe car dans ce cas, c'est le module complexe qui doit faire une action. Nous verrons par la suite, la distinction entre :

- Fonction de module: Fonction qui concerne le module et qui ne dépend pas d'une instance particulière.
- Fonction d'instance: Fonction qui concerne une instance particulière.

Les accesseurs sont toujours des fonctions d'instance.

Autres fonctions d'instance.

Il y a dans cette interface d'autres fonctions d'instance. Les quatre autres fonctions d'instance sont :

1. extern Complexe opposeComplexe(Complexe);
2. extern Complexe inverseComplexe(Complexe);
3. extern Complexe conjugeComplexe(Complexe);
4. extern void affectationComplexe(Complexe, Complexe);

En effet on demande à un complexe particulier, au complexe précis, de produire un nouveau complexe qui est soit l'opposé, soit le conjugué, soit l'inverse.

Par exemple, le code suivant:

```
Complexe c1 = creerComplexe(2,3);

Complexe c2= inverseComplexe(c1);

Complexe c3 = creerComplexe(3,4);

affectationComplexe(c1,c3);
```

S'interprète comme demander à l'instance c1 de créer un nouveau complexe qui est l'inverse de l'instance c1. Il en est de même avec la fonction affectationComplexe. Sur cet exemple, on demande à l'instance c1 de changer de valeur et de prendre la même valeur que celle du complexe c3. En aucun cas, on ne demande au complexe c1 de devenir le complexe c3. Pour les types complexe, la fonction affectationComplexe ne produit pas un changement de référence mais demande un changement de valeur. L'instance c1 reste la même instance, c'est simplement sa valeur qui change.

Les Fonctions de modules.

Par contre, il y a d'autres fonctions de module, c'est à dire qui ne dépendent pas d'une instance particulière. C'est le cas des fonctions suivantes:

1. Complexe additionComplexe(Complexe,Complexe);
2. Complexe soustractionComplexe(Complexe,Complexe);
3. Complexe multiplicationComplexe(Complexe, Complexe);
4. Complexe divisionComplexe(Complexe, Complexe);
5. bool egaliteComplexe(Complexe, Complexe);

Toutes ces fonctions ne concernent pas une instance particulière. Elles font toutes intervenir le module. On demande par exemple au module « Complexe » de produire un nouveau complexe qui est

égal à la somme de deux complexes passés en paramètre. On demande au module complexe de dire si deux complexes sont égaux (contiennent la même valeur).

Comme l'interface est complète, on peut accéder à la partie réelle et à la partie imaginaire d'un complexe et donc programmer toutes ces fonctions à l'extérieur du module complexe car on a pas besoin de l'implémentation du module complexe. Dans ce cas précis, les fonctions précédentes sont là pour proposer de la factorisation de code.

L'interface du module Complexe n'est pas minimale car toutes les fonctions de module peuvent s'obtenir à partir des accesseurs et de la fonction creerComplexe. Simplement pour éviter de la duplication de code au sein des différents clients du module, le code est factorisé dans le module. Il est de même pour la fonction divisionComplexe. En effet, il existe la fonction multiplicationComplexe et la fonction inverseComplexe. Or, la division consiste à multiplier par l'inverse. Donc on peut obtenir la division en utilisant les deux fonctions précédentes. Par contre, si les deux fonctions sont présentes dans l'interface, il ne faut pas créer de duplication de code dans l'application et donc pour faciliter la maintenance il faut tenir compte de cette propriété dans l'implémentation.

1.22 L'implémentation du module Complexe.

```
#include "Complexe.h"
#include <assert.h>
#include <math.h>
#include <stdlib.h>

struct complexe{
  double img;
  double reel;
};

void
detruireComplexe(Complexe c1){
  free(c1);
}

Complexe
creerComplexe(double reel, double img){
  Complexe res = malloc(sizeof(struct complexe));

  res->img  = img;
  res->reel = reel;

  return res;
}

Complexe
copierComplexe(Complexe c){
  return creerComplexe(reelleComplexe(c), imaginaireComplexe(c));
}

double
imaginaireComplexe(Complexe c){
  return c->img;
}
```

```
double
reelleComplexe(Complexe c){
  return c->reel;
}

double
moduleComplexe(Complexe c){
  return sqrt(c->reel*c->reel + c->img*c->img);
}

double
argumentComplexe(Complexe c){
  double tg = c->reel / c->img;
  double arg = atan(tg);
  return arg;
}

Complexe
additionComplexe(Complexe c1,Complexe c2){
  Complexe res = creerComplexe(0,0);
  res->img =  c1->img + c2->img;
  res->reel = c1->reel + c2->reel;
  return res;
}

Complexe
soustractionComplexe(Complexe c1,Complexe c2){
  Complexe oppC2 = opposeComplexe(c2);
  Complexe res = additionComplexe(c1,oppC2 );
  detruireComplexe(oppC2);
  return res;
}

Complexe
multiplicationComplexe(Complexe c1,Complexe c2){
  Complexe res = creerComplexe(0,0);
  res->reel = c1->reel * c2->reel - c1->img * c2->img;
  res->img  = c1->reel * c2->img + c2->reel * c1->img;
  return res;
}

Complexe
divisionComplexe(Complexe c1,Complexe c2){
  Complexe invC2 = inverseComplexe(c2);
  Complexe res =  multiplicationComplexe(c1,invC2);
  detruireComplexe(invC2);
  return res;
}

Complexe
opposeComplexe(Complexe c1){
  Complexe res = copierComplexe(c1);
  res->img *= -1;
  return res;
}
```

```
Complexe
inverseComplexe(Complexe c1){
  double moduleC1 = moduleComplexe(c1) * moduleComplexe(c1);
  assert(moduleC1 != 0.0);
  Complexe inverseC1 = conjugeComplexe(c1);
  inverseC1->reel /= moduleC1;
  inverseC1->img  /= moduleC1;
  return inverseC1;
}

Complexe
conjugeComplexe(Complexe c1){
  return creerComplexe(c1->reel,  - c1->img);
}

bool
egaliteComplexe(Complexe c1, Complexe c2){
  return c1->img == c2->img && c1->reel == c2->reel;
}
```

L'implémentation du module « Complexe » commence par la définition du type « struct complexe ». Le choix d'implémentation retenu pour ce module consiste à représenter un complexe par sa partie réelle et sa partie imaginaire. La fonction de création consiste seulement à allouer la place mémoire nécessaire à un complexe puis à l'affecter. La fonction de libération libère l'espace mémoire précédemment alloué. La fonction copierComplexe utilise la fonction créerComplexe.

1.23 Un autre exemple de module, le module « VecteurComplexe ».

On a vu sur le cours précédent que le module « VecteurExtensible » possédait comme limitation de ne pouvoir avoir qu'une seule instance de « VecteurExtensible ». On nous inspirant du module VecteurInt, nous allons maintenant définir un nouveau module qui est le module « **VecteurComplexe ».** Pour ne pas avoir les limitations précédentes, nous allons définir le module VecteurComplexe comme un module instance. Nous pourrons ainsi créer plusieurs instances de ce module et donc pouvoir utiliser plusieurs VecteurComplexe en même temps.

L'interface de « VecteurComplexe ».

```
#ifndef _VECTEURCOMPLEXE_
#define _VECTEURCOMPLEXE_
#include <stdbool.h>
#include "Complexe.h"

typedef struct vecteurComplexe *VecteurComplexe;

extern VecteurComplexe  creerVecteurComplexe(void);
extern void detruireVecteurComplexe(VecteurComplexe self);
extern Complexe lireElementVecteurComplexe(VecteurComplexe self, int
                                           indice);
extern void ecrireElementVecteurComplexe(VecteurComplexe self, int indice,
                                           Complexe element);

extern int indiceMaxVecteurComplexe(VecteurComplexe self);

#endif
```

On retrouve dans ce module toutes les caractéristiques d'un module instance. L'abstraction de type avec l'instruction **typedef struct vecteurComplexe *VecteurComplexe** qui définit le nouveau type VecteurComplexe. Puis les deux fonctions de création et de libération qui sont respectivement, **creerVecteurComplexe()** et **detruireVecteurComplexe(VecteurComplexe self).** Ces deux fonctions sont des fonctions de module.

Il y a de plus trois autres fonctions qui servent à lire et écrire les éléments dans le vecteur et enfin de connaître l'indice maximum qui contient un élément. Ces trois fonctions sont des fonctions d'instance de VecteurComplexe.

1.24 Dépendance d'interface entre modules.

On peut remarquer également la dépendance entre le module VecteurComplexe et le module complexe. En effet, le module VecteurComplexe ne peut exister sans le module Complexe car les fonctions d'interface de vecteurComplexe utilisent le type Complexe. Elles ne peuvent exister sans que le type Complexe soit déclaré. Et donc il s'agit d'une dépendance forte entre le module VecteurComplexe et le module Complexe. On parle dans ce cas d'un couplage fort interface. Car un module ne peut exister sans un autre.

1.25 VecteurComplexe comme un conteneur homogène de références.

On appellera **conteneur**, un module qui contient une collection d'éléments. Des exemples sont les listes, les vecteurs, les ensembles, les suites etc.

Un conteneur de références contient des références à des instances. Cela veut dire que si la valeur de l'instance change, elle change aussi dans le conteneur. C'est à dire que les effets de bords ont des répercussions à l'intérieur du conteneur. En effet, le conteneur ne contient que des pointeurs dont le contenu sont des adresses mémoires définies par le client. Dans ce cas, le conteneur ne fait que stocker les adresses données par le client, et il restitue les adresses qu'il a conservées. Le conteneur est **homogène** car il ne peut contenir que des instances d'un même module.

Concernant le stockage par référence on étudie l'exemple suivant:

```
Complexe c1 = creerComplexe(1,2);
VecteurComplexe v = creerVecteurComplexe();
ecrireElementVecteurComplexe(v,c1,0);
Complexe c2 = creerComplexe(3,2);
affecterComplexe(c1,c2);
Complexe c3 = lireElementVecteurComplexe(v,0);
```

Sur cet exemple, la variable c1 référence une instance du module Complexe. La variable v stocke cette référence. Lorsqu'on appelle la fonction affecterComplexe on change la valeur de l'instance qui est référencée par la variable c1 et qui est aussi stocké dans le conteneur v. Lorsqu'on récupère le contenu de la référence stockée à l'indice 0 on récupère l'adresse de l'instance qui est contenue dans c1. Après l'appel à lireElementVecteurComplexe(v,0), la variable c3 contient la même valeur que la variable c1. Ces deux variables identifient une même instance du module Complexe. Et donc la valeur référencée par c3 est un complexe qui contient la valeur (3,2) alors qu'au moment de son insertion dans le vecteur v il contenait la valeur (1,2). Sur cet exemple, on voit bien que les modifications depuis le client ont des répercussions sur le conteneur. Puisque l'on peut changer les valeurs qui sont référencées par le conteneur.

L'implémentation du module.

```
#include "VecteurComplexe.h"
#include "Complexe.c"
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include <malloc.h>

static const int tailleDefault = 10;

struct vecteurComplexe {
  Complexe *tab;
  int maxIndice;
};

static void
allongerVecteur(VecteurComplexe self, int indice){
  self->tab = realloc(self->tab, (indice + tailleDefault) *
sizeof(Complexe));
  for (int i = self->maxIndice + 1; i < indice + tailleDefault; i++)
    self->tab[i] = 0;
  self->maxIndice = (indice + tailleDefault) – 1;
}

VecteurComplexe
creerVecteurComplexe(void){
  VecteurComplexe self = malloc(sizeof(struct vecteurComplexe));
  self->tab = malloc(sizeof(Complexe)* tailleDefault);
  self->maxIndice = tailleDefault – 1;
  for(int i = 0; i <= self->maxIndice; i++)
    self->tab[i] = 0;
  return self ;
}

void
detruireVecteurComplexe(VecteurComplexe self){
  free(self->tab);
free(self);
}

Complexe
lireElementVecteurComplexe(VecteurComplexe self, int indice){
  assert(!(indice < 0 || indice > self->maxIndice));
  return self->tab[indice];
}

void
ecrireElementVecteurComplexe(VecteurComplexe self, int indice,Complexe
element){
  assert(! (indice < 0));
  if(indice >= self->maxIndice)
    allongerVecteur(self, indice);
  self->tab[indice] = element;
}
```

```
int
indiceMaxVecteurComplexe(VecteurComplexe self){
  return self->maxIndice;
}
```

La fonction **creerVecteurComplexe(void)** doit créer un nouveau vecteur. Elle doit d'abord réserver la mémoire pour la structure. Mais comme cette structure contient aussi un tableau, ce tableau doit lui aussi être alloué au moment de la construction. Le rôle de la fonction créer est de réserver la place mémoire pour une instance et aussi de lui donner une valeur par défaut.

La fonction **detruireVecteurComplexe** doit libérer la mémoire qui a été allouée par le module et pas une autre. C'est pour cette raison que la fonction libérer ne libère pas les éléments contenus dans le conteneur car ce n'est pas le vecteur qui a créé les éléments, il ne fait que les référencer de manière temporaire. Un grand principe de programmation est :

On ne peut libérer que la mémoire que l'on a crée ou demandée pour ses besoins propres.

Les autres fonctions d'instance servent à stocker ou à retourner les références stockées par le VecteurComplexe.

On peut remarquer dans cette implémentation la fonction locale

static void allongerVecteur(VecteurComplexe self, int indice)

qui est une fonction d'instance locale qui sert à rallonger une instance particulière de VecteurComplexe. Il existe donc des fonctions locales d'instance.

Programmation Modulaire

Partie 2.

VecteurInt comme un conteneur homogène de pointeur d'entiers.

De la même manière que l'on a défini un vecteur de complexe, on peut définir un vecteur de références d'entier. Le code du module serait à peu près le même que celui du vecteur de complexes.

```
/* VECTEUR.H */

#ifndef _VECTEUR_
#define _VECTEUR_

#include <stdbool.h>

typedef struct vectExt *VectExt;
extern VectExt  creerVecteur(void);
extern void detruireVecteur(VectExt self);
extern int* lireElement(VectExt self, int indice);
extern void ecrireElement(VectExt self, int indice, int* element);
extern int indiceMax(VectExt self);

#endif
```

```
/* VECTEUR.C */

#include "Vecteur.h"
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include <malloc.h>

static const int tailleDefault = 10;

struct vectExt {
  int **tab;
  int maxIndice;
};

static void
allongerVecteur(VectExt self, int indice){
  self->tab = realloc(self->tab, (indice + tailleDefault) * sizeof(int *));
```

```
  for(int i = self->maxIndice + 1; i < indice + tailleDefault; i++)
    self->tab[i] = 0;
  self->maxIndice = (indice + tailleDefault) – 1;
}

VectExt
creerVecteur(void){
  VectExt self = malloc(sizeof(struct vectExt));
  self->tab = malloc(sizeof(int *)* tailleDefault);
  self->maxIndice = tailleDefault – 1;
  for(int i = 0; i <= self->maxIndice; i++)
  self->tab[i] = 0 ;
  return self;
}

void
detruireVecteur(VectExt self){
  free(self->tab);
  free(self);
}

int*
lireElement(VectExt self, int indice) {
  assert(!(indice < 0 || indice > self->maxIndice));
  return self->tab[indice];
}

void
ecrireElement(VectExt self,int indice, int *element){
  assert(! (indice < 0));
  if(indice >= self->maxIndice)
    allongerVecteur(self,indice);
  self->tab[indice] = element;
}

int
indiceMax(VectExt self){
  return self->maxIndice;
}
```

Premier pas vers la généricité.

On peut remarquer que le code est exactement le même entre Vecteur.c et VecteurComplexe.c, il suffit de remplacer le type Complexe par le type int *. On ne peut pas considérer qu'il y a une réelle duplication de code puisque les types sont différents mais on retrouve le même code au type près. Nous verrons par la suite la généricité peut apporter une solution à ce problème. Malheureusement la généricité n'existe pas en langage C. Mais l'idée est la suivante, il faut faire un patron (un fichier qui est paramétré par le type). On note <T> toutes les informations qui dépendent du type.

```
/* PATRON Générique de Vecteur.h */

#ifndef _VECTEUR_
#define _VECTEUR_

#include <stdbool.h>

typedef struct vect<T> *Vect<T>;
extern Vect<T> creerVecteur<T>(void);
extern void detruireVecteur<T>(Vect<T> self);
extern <T> lireElement<T>(Vect<T> self, int indice);
extern void ecrireElement<T>(Vect<T> self, int indice, <T> element);
extern int indiceMax<T>(VectExt self);

#endif



/* PATRON générique de Vecteur.c */

#include "Vecteur.h"
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include <malloc.h>

static const int tailleDefault = 10;

struct vect<T> {
  <T> *tab;
  int maxIndice;
};

static void
allongerVecteur(Vect<T> self, int indice){
  self->tab = realloc(self->tab, (indice + tailleDefault) * sizeof(<T>));
  for(int i = self->maxIndice + 1; i < indice + tailleDefault; i++)
    self->tab[i] = 0;
  self->maxIndice = (indice + tailleDefault) – 1;
}

Vect<T>
creerVecteur<T>(void){
  Vect<T> self = malloc(sizeof(struct vectExt));
  self->tab = malloc(sizeof(<T>)* tailleDefault);
  self->maxIndice = tailleDefault – 1;
  for(int i = 0; i <= self->maxIndice; i++)
    self->tab[i] = 0;
  return self;
}

void
detruireVecteur<T>(Vect<T> self){
  free(self->tab);
  free(self);
}
```

```
<T>
lireElement<T>(Vect<T> self, int indice){
  assert(!(indice < 0 || indice > self->maxIndice));
  return self->tab[indice];
}

void
ecrireElement<T>(Vect<T> self,int indice, <T>  element){
  assert(! (indice < 0));
  if(indice >= self->maxIndice)
    allongerVecteur(self,indice);
  self->tab[indice] = element;
}

int
indiceMax<T>(Vect<T> self){
  return self->maxIndice;
}
```

Si on veut faire un vecteur de complexe, il suffit de faire sous emacs un query replace de <T> par Complexe, si on veut faire un vecteur d'entier, il suffit de faire un query replace de <T> par int * (attention tout de meme à l'* à la fin du nom de fonction). Comme il n'existe pas en C de mécanisme pour gérer la généricité, nous allons maintenant aborder une autre notion qui est basé sur le pointeur générique.

Vecteur de pointeur Générique (void *)

Les deux conteneurs précédents stockent tous les deux des références c'est à dire des pointeurs. Comme tous les pointeurs sont unifiables sur le type «void *» alors on va écrire un vecteur générique qui est constitué de «void *». Le code sera exactement le même que pour les deux autres modules précédents, il suffit de remplacer <T> par void * est de renommer certaines fonctions.

```
/*  VecteurGenerique.h */

#ifndef _VECTEURGENERIQUE_
#define _VECTEURGENERIQUE_

typedef struct vecteurGenerique *VecteurGenerique;

extern VecteurGenerique creerVecteurGenerique(void);
extern void detruireVecteurGenerique(VecteurGenerique self);
extern void* lireElementVecteurGenerique(VecteurGenerique self, int
indice);
extern void ecrireElementVecteurGenerique(VecteurGenerique self, int
indice,                                          void* element);
extern int indiceMaxVecteurGenerique(VecteurGenerique self);
#endif
```

```
/* VecteurGenerique.c */

#include "VecteurGenerique.h"
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include <malloc.h>

static const int tailleDefault = 10;

struct vecteurGenerique{
  void **tab;
  int maxIndice;
};

static void
allongerVecteur(VecteurGenerique self, int indice){
  self->tab = realloc(self->tab, (indice + tailleDefault) * sizeof(void
*)); for(int i = self->maxIndice + 1; i < indice + tailleDefault; i++)
    self->tab[i] = 0;
  self->maxIndice = (indice + tailleDefault) – 1;
}



VecteurGenerique
creerVecteurGenerique(void){
  VecteurGenerique self = malloc(sizeof(struct vecteurGenerique));
  self->tab = malloc(sizeof(void *) * tailleDefault);
  self->maxIndice = tailleDefault – 1;
  for(int i = 0; i <= self->maxIndice; i++)
    self->tab[i] = (void *) 0;
  return self;
}

void
detruireVecteurGenerique(VecteurGenerique self){
  free(self->tab);
  free(self);
}

void*
lireElementVecteurGenerique(VecteurGenerique self, int indice){
  assert(!(indice < 0 || indice > self->maxIndice));
  return self->tab[indice];
}

void
ecrireElementVecteurGenerique(VecteurGenerique self,int indice, void
*element){
  assert(! (indice < 0));
  if(indice >= self->maxIndice)
    allongerVecteur(self, indice);
  self->tab[indice] = element;
}
```

```
int
indiceMaxVecteurGenerique(VecteurGenerique self){
  return self->maxIndice;
}
```

On a maintenant un code générique qui permet de représenter un conteneur de références. Par contre ce conteneur n'est pas homogène car l'on peut insérer dans une instance de vecteur générique n'importe quel type de pointeur. On ne peut pas contrôler le type des éléments insérés dans le vecteur. En cela, on peut dire que le vecteur générique est un conteneur hétérogène. Dans le cas de vecteurComplexe ou de vecteurInt, l'homogénéité du conteneur était assurée par l'interface des modules. En effet, comme les déclaration de fonctions faisaient explicitement référence au type Complexe, ou au type int *, on ne pouvait pas mettre n'importe quoi dans le conteneur. C'est l'interface qui assurait la vérification du type. On va donc conserver cette caractéristique, c'est à dire que l'interface va permettre la vérification de l'homogénéité et par contre l'implémentation va utiliser les services de « VecteurGenerique ».

Une nouvelle implémentation de VecteurComplexe basée sur la délégation de code vers VecteurGenerique.

Dans cette section, nous allons maintenant éviter de la duplication dans l'application en implémentant le module « VecteurComplexe » en utilisant la délégation de code vers VecteurGénérique. La **délégation** consiste à utiliser un autre module pour implémenter tout au partie du code que l'on doit réaliser. C'est à dire qu'une instance du module a implémenter contient une instance du module délégué. Sur cet exemple, une instance de VecteurComplexe contiendra une instance de vecteurGénérique et elle lui délèguera tout le code à implémenter. Ce mécanisme est d'abord présent dans la déclaration de la structure.

```
struct vecteurComplexe {
  VecteurGenerique deleg;
};
```

L'implémentation d'un VecteurComplexe se résume à un vecteurGénérique. Si on regarde maintenant l'implémentation du VecteurComplexe.

```
struct vecteurComplexe {
  VecteurGenerique deleg;
};

VecteurComplexe
creerVecteurComplexe(void){
  VecteurComplexe self = malloc(sizeof(struct vecteurComplexe));
  self->deleg = creerVecteurGenerique();
}

void
detruireVecteurComplexe(VecteurComplexe self){
```

```
  detruireVecteurGenerique(self->deleg);
  free(self);
}

Complexe
lireElementVecteurComplexe(VecteurComplexe self, int indice){
  return lireElementVecteurGenerique(self->deleg, indice);
}

void
ecrireElementVecteurComplexe(VecteurComplexe self,int indice, Complexe
element){
  ecrireElementVecteurGenerique(self->deleg, indice, element);
}

int
indiceMaxVecteurComplexe(VecteurComplexe self){
  return indiceMaxVecteurGenerique(self->deleg);
}
```

Créer un VecteurComplexe consiste à créer la place pour la structure d'une instance. C'est à dire créer de la place pour le champs deleg. Et surtout cela revient à créer un VecteurGénérique. En résumé créer une instance de vecteurComplexe revient à créer son délégué.

Libérer un VecteurComplexe, consiste à libérer son délégué et à libérer la place pour la structure.

Écrire un élément dans Vecteur complexe, consiste à demander au délégué d'écrire un élément. Le délégué stocke l'élément ajouté. On assure la conversion de type par la définition du type du paramètre dans la fonction EcrireElementVecteurComplexe.

Lire un élément dans VecteurComplexe, consiste à demander au délégué de lire un élément. Le délégué restitue l'élément qu'il a stocké. On assure la conversion par le type de retour de la définition de la fonction LireElementVecteurComplexe.

Retourner l'indice max pour VecteurComplexe consiste à retourner l'indice max de son délégué.

REMARQUE: Il ne s'agit que d'un changement de l'implémentation du module VecteurComplexe, l'interface est identique. Comme le module VecteurComplexe assurait la séparation entre l'interface est l'implémentation, les clients du module n'ont pas besoin d'être recompilé, il suffirat simplement de refaire l'édition de lien. Cette solution présente l'avantage :

- **d'assurer l'homogénéité du conteneur VecteurComplexe car on ne change pas l'interface qui assure la cohérence de type.**
- **De faciliter la maintenance, car il suffit simplement de maintenir le module VecteurGenérique car on lui délègue l'implémentation de VecteurComplexe.**

Pour implémenter le module VecteurInt, il suffit de faire la même chose, c'est à dire ne pas changer l'interface, mais de modifier l'implémentation en utilisant la délégation. Il faut redéfinir la structure.

```
struct vecteurInt {
        VecteurGenerique deleg
};
```

Et d'implémenter toutes les autres fonctions en utilisant la délégation.

Programmation Modulaire

Partie 3.

Nous avons vu dans la partie précédente comment factoriser le code à travers une application en utilisant les pointeurs générique et la délégation. Nous allons maintenant reprendre cette démarche pour implémenter un conteneur par Valeur.

VecteurComplexe comme un conteneur homogène de Valeurs.

Jusqu'à présent le client connaissait des instances d'un complexe et le vecteur stockait ces références. Maintenant le conteneur ne va pas stocker des références à des instances mais va stocker des copies de ces références. C'est à dire que le conteneur va conserver ses propres copies des instances et il va retourner des copies des instances qu'il stocke. Avec cette technique, les effets de bord du client ne peuvent plus maintenant modifier le contenu du conteneur. On qualifie ce type de conteneur de **conteneur par valeur car il ne contient que des valeurs du client.**

Si on reprend l'exemple précédent:

```
Complexe c1 = creerComplexe(1,2);
VecteurComplexe v = creerVecteurComplexe();
ecrireElementVecteurComplexe(v,c1,0);
Complexe c2 = creerComplexe(3,2);
affecterComplexe(c1,c2);
Complexe c3 = lireElementVecteurComplexe(v,0);
affecterComplexe(c3,c2);
```

Lorsqu'on appelle ecrireElementVecteurComplexe(v,c1,0); le conteneur v va stocker une copie de c1. Donc si on modifie c1 on ne peut pas modifier la valeur stockée dans le conteneur car on ne la connaît pas. Quand on évoque l'appel c3 = lireElementVecteurComplexe(v,0), le conteneur va de nouveau retourner une copie, donc la variable c3 ne référence aucune information du conteneur v. Si on modifie cette valeur cela n'aura aucune conséquence sur le contenu de v.

L'interface du module VecteurComplexe n'est modifiée, on lui rajoute une nouvelle fonction donneIndiceElementVecteurComplexe qui rend l'indice qui contient un élément. L'interface est alors la suivante:

```
#ifndef _VECTEURCOMPLEXE_
#define _VECTEURCOMPLEXE_

#include <stdbool.h>
#include "Complexe.h"

typedef struct vecteurComplexe *VecteurComplexe;

extern VecteurComplexe creerVecteurComplexe(void);
extern void detruireVecteurComplexe(VecteurComplexe self);
extern Complexe lireElementVecteurComplexe(VecteurComplexe self,
             int indice);
extern void ecrireElementVecteurComplexe(VecteurComplexe self, int indice,
            Complexe element);
extern int indiceMaxVecteurComplexe(VecteurComplexe self);
extern int donneIndiceElementVecteurComplexe(VecteurComplexe self,
            Complexe element);

#endif
```

On va maintenant regarder les changements d'implémentation entre le module VecteurComplexe par référence et le module VecteurComplexe par Valeur.

```
static const int tailleDefault = 10;

struct vecteurComplexe {
  Complexe *tab;
  int maxIndice;
};

static void
allongerVecteur(VecteurComplexe self, int indice){
  self->tab = realloc(self->tab, (indice + tailleDefault) *
sizeof(Complexe));
  for(int i = self->maxIndice + 1; i < indice + tailleDefault; i++)
    self->tab[i] = (void *) 0;
  self->maxIndice = (indice + tailleDefault) – 1;
}

VecteurComplexe
creerVecteurComplexe(void){
  VecteurComplexe self = malloc(sizeof(struct vecteurComplexe));
  self->tab = malloc(sizeof(Complexe)* tailleDefault);
  self->maxIndice = tailleDefault – 1;
  for(int i = 0; i <= self->maxIndice; i++)
    self->tab[i] = (void *) 0;
}
```

```
void
detruireVecteurComplexe(VecteurComplexe self){
  for(int i = 0; i <= self->maxIndice; i++)
    detruireComplexe(self->tab[i]);
  free(self->tab);
  free(self);
}

Complexe
lireElementVecteurComplexe(VecteurComplexe self, int indice){
  assert(!(indice < 0 || indice > self->maxIndice));
  return copierComplexe(self->tab[indice]);
}

void
ecrireElementVecteurComplexe(VecteurComplexe self, int indice,Complexe
element){
  assert(! (indice < 0));
  if(indice >= self->maxIndice)
    allongerVecteur(self, indice);
  if(self->tab[indice] != (void *) 0)
    libererComplexe(self->tab[indice]);
  self->tab[indice] = copierComplexe(element);
}

int
indiceMaxVecteurComplexe(VecteurComplexe self){
  return self->maxIndice;
}

int
donneIndiceElementComplexe(VecteurComplexe self, Complexe element){
  for(int i = 0; i <= self->maxIndice; i++)
    if(egaliteComplexe(self->tab[i], element))
      return i;
  return -1;
}
```

La déclaration de structure, les fonctions allongerVecteurComplexe, creerVecteurComplexe et donneIndiceMaxVecteurComplexe sont identiques que ce soit pour un conteneur par référence ou pour un conteneur par valeur.

La première différence est dans la fonction detruireVecteurComplexe. Comme une instance de vecteur complexe par valeur stocke des copies de complexe qu'il a lui même effectuée pour ses **propres besoins, c'est à lui de les libérer lorsqu'il est libéré. Pour éviter les fuites mémoires, lorsqu'on libère une instance de vecteur complexe par valeur, on doit aussi libérer les instances de complexe qu'il contient.**

La deuxième différence est dans les fonctions lire et ecrire à chaque fois que l'on doit stocker ou retourner une valeur, il faut en faire une copie. C'est la seule

façon de protéger le conteneur des effets de bord du client. Cela peut-être couteux mais c'est la seule manière d'assurer la sécurité du module. On remarquera que dans le cas de la fonction lireElementVecteurComplexe, ce sera au client de libérer l'instance de complexe quand il n'en aura plus besoin.

La dernière différence concerne la fonction donneIndiceElementComplexe, ou l'on ne peut par utiliser la comparaison de référence == mais l'on doit utiliser la comparaison de contenu c'est à dire la fonction égalité complexe.

Pour conclure, la différence d'implémentation entre le conteneur par valeur et le conteneur par référence se situe au niveau du comportement de 3 fonctions. Le tableau suivant résume les différences de comportement entre le comportement d'un conteneur par référence et d'un conteneur par valeur. Remarque : en cours le problème était traité avec 4 fonctions, pour simplifier nous n'en considèrerons que 3 dans ces notes de cours.

	Réference	Valeur
création/copie	identité	copierComplexe
libération	rien	detruireComplexe
comparaison	Comparaison de référence(==)	EgaliteComplexe.

En conclusion, les fonctions précédentes permettent d'adapter le comportement d'un module.

VecteurGénérique comme un conteneur de Valeurs.

Nous allons essayer maintenant d'adapter la même architecture que pour le conteneur par référence. Il s'agit maintenant de définir un module « VecteurGénérique » qui fonctionne par valeur. **Pour cela nous allons utiliser les pointeurs de fonctions pour paramétrer le comportement du module. Jusqu'à présent les pointeurs de fonctions étaient utilisés pour paramétrer le comportement d'un algorithme**. Maintenant nous allons les utiliser pour paramétrer le comportement d'une instance. Comme nous l'avons vu dans la section précédente la différence entre un conteneur par valeur et un conteneur par référence dépend des fonctions de création, libération et comparaison. Nous allons donc abstraire ce comportement en ajoutant des pointeurs de fonctions dans la structure d'une instance. Nous allons déclarer trois types de pointeurs de fonctions.

1. typedef void* (*funCopier) (void *); sera utilisé pour copier le contenu d'une référence en créant une nouvelle instance.
2. typedef void (*funDetruire) (void *); sera utilisé pour libérer une instance quand elle devient inutile

3. typedef bool (*funEgalite) (void *, void *); sera utilisé pour comparer le contenu de deux instances.

Le comportement d'une instance de Vecteur Générique doit être défini au moment de sa création. Il est donc nécessaire de modifier l'interface de vecteur générique en ajoutant des paramètre à la fonction creerVecteurGenerique.

Voici la nouvelle interface du Vecteur Générique.

```
#ifndef _VECTEURGENERIQUE_
#define _VECTEURGENERIQUE_

#include <stdbool.h>

typedef struct vecteurGenerique *VecteurGenerique;
typedef void* (*funCopier) (void *);
typedef void (*funDetruire) (void *);
typedef bool (*funEgalite) (void *, void *);

extern VecteurGenerique  creerVecteurGenerique(funCopier, funDetruire,
  funEgalite);
extern void detruireVecteurGenerique(VecteurGenerique self);
extern void* lireElementVecteurGenerique(VecteurGenerique self,
              int indice);
extern void ecrireElementVecteurGenerique(VecteurGenerique self,
              int indice, void* element);
extern int indiceMaxVecteurGenerique(VecteurGenerique self);
extern int donneIndiceElementVecteurGenerique(VecteurGenerique    self,
              void *element);

#endif

Voici maintenant l'implémentation du Vecteur Générique par Valeur.

static const int tailleDefault = 10;

struct vecteurGenerique{
  void **tab;
  int maxIndice;
  funCopier copier;
  funDetruire detruire;
  funEgalite  egalite;
};

static void
allongerVecteur(VecteurGenerique self, int indice){
  self->tab = realloc(self->tab, (indice + tailleDefault) * sizeof(void
*));
  for(int i = self->maxIndice + 1; i < indice + tailleDefault; i++)
    self->tab[i] = (void *) 0;
  self->maxIndice = (indice + tailleDefault) – 1;
}

VecteurGenerique
creerVecteurGenerique(funCopier copier, funDetruire detruire, funEgalite
egalite){
```

```
  VecteurGenerique self = malloc(sizeof(struct vecteurGenerique));
  self->tab = malloc(sizeof(void *) * tailleDefault);
  self->maxIndice = tailleDefault - 1;
  self->copier   = copier;
  self->detruire = detruire;
  self->egalite  = egalite;
  for(int i = 0; i <= self->maxIndice; i++)
    self->tab[i] = (void *) 0;
  return self;
}

void
detruireVecteurGenerique(VecteurGenerique self){
  for(int i = 0; i <= self->maxIndice; i++)*
    self->detruire(self->tab[i]);
  free(self->tab);
  free(self);
}

void*
lireElementVecteurGenerique(VecteurGenerique self, int indice){
  assert(!(indice < 0 || indice > self->maxIndice));
  return self->copier(self->tab[indice]);
}

void
ecrireElementVecteurGenerique(VecteurGenerique self,int indice, void
*element){
  assert(! (indice < 0));
  if(indice >= self->maxIndice)
    allongerVecteur(self, indice);
  if(self->tab[indice] != (void *) 0)
    self->detruire(self->tab[indice]);
  self->tab[indice] = self->copier(element);
}

int
indiceMaxVecteurGenerique(VecteurGenerique self){
  return self->maxIndice;
}

int
donneIndiceElementVecteurGenerique(VecteurGenerique self, void * element){
  for(int i = 0; i <= self->maxIndice; i++)
    if(self->egalite(self->tab[i], element))
      return i;
  return -1;
}
```

On augmente maintenant la structure VecteurGenerique en adjoignant les champs qui représentent les pointeurs de fonctions.

```
struct vecteurGenerique{
  void **tab;
  int maxIndice;
```

```
  funCopier copier;
  funDetruire detruire;
  funEgalite  egalite;
};
```

Chaque instance de VecteurGenerique stockera les pointeurs de fonctions qui lui sont donnés au moment de sa création en passant les paramètres. Comme ces pointeurs de fonctions représentent le comportement du module générique on peut dire que le comportement d'une instance de vecteur générique est fixé au moment de sa création. A chaque fois que l'on aura besoin de copier, detruire ou comparer des instances on utilisera ces pointeurs de fonctions. Par exemple,

```
void*
lireElementVecteurGenerique(VecteurGenerique self, int indice){
  assert(!(indice < 0 || indice > self->maxIndice));
  return self->copier(self->tab[indice]);
}
```

Cette fonction doit retourner une copie de l'instance qu'elle contient. Pour faire la copie, on utilise alors la fonction qui permet de faire une copie. On voit sur cet exemple, l'intérêt des pointeurs de fonctions. En effet, si on a une instance de vecteur générique qui contient des complexes, à chaque fois on appellera la fonction copier des complexes. Si une autre instance de vecteur générique contient des rationnels on appellera alors la fonction de copie des rationnels.

Une nouvelle implémentation de VecteurComplexe par valeur basée sur la délégation de code vers le nouveau module VecteurGenerique.

Sur le même principe que pour l'implémentation du vecteur complexe par référence on va implémenter le vecteur complexe par valeur en utilisant la délégation vers le vecteur générique. L'interface de VecteurComplexe par valeur reste exactement la même que pour vecteur complexe par référence.

L'implémentation est elle aussi pratiquement la même, le seul changement concerne la création d'un vecteurComplexe. En effet, cette fonction doit créer un vecteur générique pour pouvoir lui déléguer le travail. Or, l'interface de vecteur générique a changé pour la fonction de création car celle-ci a changé. Il faut maintenant passer en paramètres les pointeurs de fonctions qui correspondent à un comportement par valeur.

1. Il faut que la copier fasse appel à la fonction copierComplexe.
2. Il faut que la libération fasse appel à la fonction detruireComplexe.
3. Il faut que la comparaison fasse appel à la fonction egaliteComplexe.

Pour assurer la cohérence entre les déclarations de fonctions et les types de pointeurs de fonctions attendus, on va créer dans l'implémentation du module Vecteur Complexe des fonctions enveloppes qui assurent la cohérence de type.

La fonction static void * copier(void *element) encapsule copierComplexe.

La fonction static void detruire(void *element) encapsule detruireComplexe.

La fonction static bool egalite(void *element1, void *element2) encapsule egalite complexe.

On peut maintenant regarder le code de l'implémentation de Vecteur Complexe par valeur.

```
struct vecteurComplexe {
  VecteurGenerique deleg;
};

static void *
copier(void *element){
  Complexe c = (Complexe) element;
  return copierComplexe(c);
}

static void
detruire(void *element){
  detruireComplexe((Complexe) element);
}

static bool
egalite(void *element1, void *element2){
  return egaliteComplexe((Complexe) element1, (Complexe) element2);
}

VecteurComplexe
creerVecteurComplexe(void){
  VecteurComplexe self = malloc(sizeof(struct vecteurComplexe));
  self->deleg = creerVecteurGenerique(copier,  detruire, egalite);
}

void
detruireVecteurComplexe(VecteurComplexe self){
  detruireVecteurGenerique(self->deleg);
  free(self);
}

Complexe
lireElementVecteurComplexe(VecteurComplexe self, int indice){
  return lireElementVecteurGenerique(self->deleg, indice);
}

void
ecrireElementVecteurComplexe(VecteurComplexe self,int indice, Complexe
element){
  ecrireElementVecteurGenerique(self->deleg, indice, element);
}
```

```
int
indiceMaxVecteurComplexe(VecteurComplexe self){
  return indiceMaxVecteurGenerique(self->deleg);
}

int
donneIndiceElementVecteurComplexe(VecteurComplexe self, void * element){
  return donneIndiceElementVecteurGenerique(self->deleg, element);
}
```

On remarque que à l'exception de la fonction de création, le code est identique à celui de vecteur complexe par référence. C'est tout à fait normal, car la seule différence entre les deux modules dépends du comportement de l'instance à qui ils délègue le travail. Comme pour le vecteur complexe par référence, l'interface ne sert qu'à assurer l'homogénéité du conteneur.

Une nouvelle implémentation de VecteurComplexe par référence basée sur la délégation de code vers le nouveau module VecteurGenerique.

On a vu que le vecteur générique encapsule son comportement au moment de sa création, on a vu aussi que le vecteur complexe par valeur définissait des enveloppes pour les fonctions copier, détruire et comparer. On a vu également que la différence entre un vecteur complexe par valeur et un vecteur par référence dépendait de ces mêmes fonctions. Alors on peut également utiliser le même module de vecteur générique mais en passant un comportement différent.

Il faut maintenant passer en paramètres les pointeurs de fonctions qui correspondent à un comportement par valeur.

1. Il faut que la copie fasse appel à la fonction « identité ».
2. Il faut que la libération fasse appel à la fonction « ne rien.faire »
3. Il faut que la comparaison fasse appel à la fonction « == ».

Pour assurer la cohérence entre les déclarations de fonctions et les types de pointeurs de fonctions attendus, on va créer dans l'implémentation du module Vecteur Complexe des fonctions enveloppes qui assurent la cohérence de type.

La fonction static void * copier(void *element) encapsule l'identité.

La fonction static void detruire(void *element) encapsule « rien »

La fonction static bool egalite(void *element1, void *element2) encapsule la comparaison de références.

La différence est donc dans les enveloppes, le reste du code demeure identique.

```
static void *
copier(void *element){
  return element;
}

static void
detruire(void *element){
}

static bool
egalite(void *element1, void *element2){
  return  element1 == element2;
}
```