Création de bibliothèques statiques

Pour **créer une bibliothèque statique** `X`, on suit les étapes suivantes :

1. **écriture des modules** `M1`, …, `Mn` qui vont constituer la bibliothèque ;
2. **compilation des modules** et création des fichiers objets `M1.o`, …, `Mn.o` ;
3. **création de l'archive** `libX.a` par

```
ar r libX.a M1.o ... Mn.o
```

4. **génération de l'index** de la bibliothèque par

```
ranlib libX.a
```

5. (étape facultative) **écriture d'un fichier d'en-tête global**

```
/* X.h */
#ifndef __X__
#define __X__

#include "M1.h"
...
#include "Mn.h"

#endif
```

Création de bibliothèques statiques — exemple

– Exemple (début) –

On suppose que l'on a créé trois modules :

1. `Liste` pour la représentation des listes chaînées ;
2. `Arbre` pour la représentation des arbres binaires de recherche ;
3. `Tri` pour l'implantation d'algorithmes de tris de tableaux.

On souhaite regrouper ces modules en une bibliothèque nommée `algo`.

Celle-ci sera constituée de deux fichiers ;

1. `libalgo.a`, l'implantation de la bibliothèque ;
2. `Algo.h`, l'en-tête de la bibliothèque.

Création de bibliothèques statiques — exemple

– Exemple (suite et fin) –

Pour créer de la bibliothèque `algo`, on saisit les commandes

```
gcc -c Liste.c ; gcc -c Arbre.c ; gcc -c Tri.c
ar r libalgo.a Liste.o Arbre.o Tri.o
ranlib libalgo.a
```

et on écrit l'en-tête global

```c
/* Algo.h */
#ifndef __ALGO__
#define __ALGO__

#include "Liste.h"
#include "Arbre.h"
#include "Tri.h"

#endif
```

Pour utiliser la bibliothèque `algo` dans un fichier `F` d'un projet `P`, il faut inclure dans `F` son en-tête, réaliser l'édition des liens de `P` avec l'option `-lalgo` et indiquer son chemin `chem` avec l'option `-Lchem`.

Index

Il est possible de **consulter l'index** d'une bibliothèque statique `LIB` par la commande `nm -s libLIB.a`

– Exemple –

```c
/* A.h */
#ifndef __A__
#define __A__
    char h(int a);
#endif
```

```c
/* A.c */
#include "A.h"
char h(int a) {
    return a % 256;
}
```

```c
/* B.h */
#ifndef __B__
#define __B__
    typedef int S;

    int f(S s);
    char g(int a);
#endif
```

```c
/* B.c */
#include "B.h"
int f(S s) {
    return s;
}
char g(int a) {
    return a % 64;
}
```

Création :

```
gcc -c A.c ; gcc -c B.c
ar r libAB.a A.o B.o
ranlib libAB.a
```

Affichage :

```
nm -s libAB.a
```

```
Archive index:
h in A.o
f in B.o
g in B.o

A.o:
0000000000000000 T h

B.o:
0000000000000000 T f
000000000000000c T g
```

Résumé de l'axe 1

Nous avons étudié plusieurs outils et posé des conventions pour le développement de projets.

1. Au niveau de la **présentation du code** :
   - indentation ;
   - choix des identificateurs ;
   - documentation.
2. Au niveau de l'**écriture de fonctions** :
   - pré-assertions ;
   - mécanismes de gestion d'erreurs.
3. Au niveau de la **conception de projets** :
   - analyse d'une spécification ;
   - découpage en modules ;
   - compilation séparée.

Tout ce qui a été étudié dans cet axe est à appliquer dans la pratique.

Axe 2 : comprendre les mécanismes de base

5. Allocation dynamique

6. Entrées et sorties

7. Types

8. Types structurés

5. Allocation dynamique

Plan

5. Allocation dynamique

La mémoire

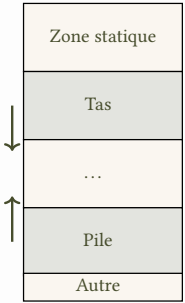
Lors de l'exécution d'un programme, une portion (de taille variable) de la mémoire lui est dédiée. Cette zone lui est exclusive. On l'appelle la mémoire.

On peut voir la mémoire comme un **très grand tableau d'octets**.

La mémoire est segmentée en plusieurs parties :

- la zone statique qui contient le code et les données statiques ;
- le tas, de taille variable au fil de l'exécution ;
- la pile, de taille variable au fil de l'exécution.

Il y a d'autres zones (non repr. ici).



Le **tas** contient les variables **allouées dynamiquement**.

La **pile** contient les **variables locales** lors des appels aux fonctions.

Pointeurs

Nous avons vu que chaque variable possède une **adresse**.

Connaître l'adresse d'une variable est **suffisant pour la manipuler** (c.-à-d., lire et modifier sa valeur).

L'objet qui permet de représenter et de manipuler des adresses est le pointeur.

Un pointeur est une entité constituée des deux éléments suivants :

1. une adresse vers une zone de la mémoire ;
2. un type.

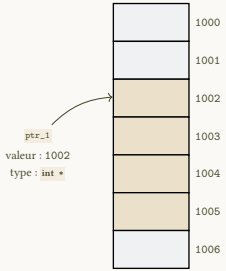
Pointeurs

Intuitivement, un pointeur est une **flèche** munie d'un mode d'emploi, le tout pointant vers une zone de la mémoire.

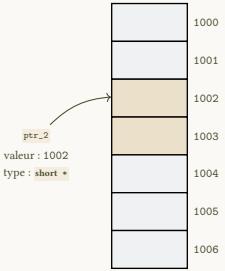
Le **mode d'emploi** décrit le type de la zone de la mémoire ainsi adressée en renseignant sur la **taille de la zone**.

– Exemples –

Si `ptr_1` est un pointeur sur une donnée de type **`int`** (4 octets) :



Si `ptr_2` est un pointeur sur une donnée de type **`short`** (2 octets) :



Déclaration de pointeurs

On **déclare** un pointeur sur une zone de la mémoire de type `T` par

```
T *ptr;
```

On **accède à la valeur** de la zone mémoire pointée par `ptr` par

```
*ptr
```

On **accède à l'adresse** de la zone mémoire pointée par `ptr` par

```
ptr
```

Attention : le même symbole `*` est utilisé pour la déclaration d'un pointeur et pour accéder à la valeur pointée. C'est une imperfection du langage C qui peut porter à confusion.

Manipulation de pointeurs

On suppose que `ptr` est un pointeur pointant sur une zone de la mémoire de type `T`.

Il est possible de **changer l'endroit** où `ptr` pointe par

```
ptr = ADR;
```

où `ADR` est une adresse de la mémoire de type `T`.

Il est possible d'**affecter une valeur** `VAL` de type `T` à `*ptr` par

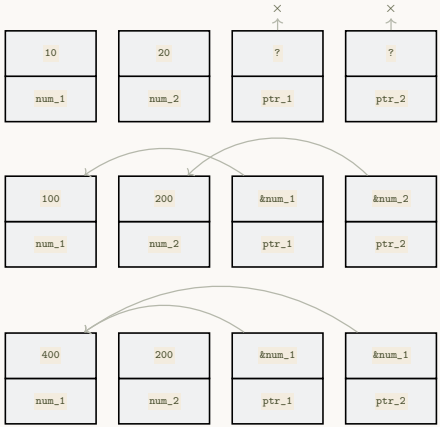
```
*ptr = VAL;
```

De cette manière, la zone mémoire d'adresse `ptr` est modifiée et devient de valeur `VAL`.

Manipulation de pointeurs — exemple

– Exemple –

```c
/* (1) */
int num1, num2;
int *ptr1, *ptr2;
num1 = 10;
num2 = 20;


/* (2) */
ptr1 = &num1;
ptr2 = &num2;
*ptr1 = 100;
*ptr2 = 200;


/* (3) */
ptr2 = ptr1;
*ptr1 = 300;
*ptr2 = 400;
```

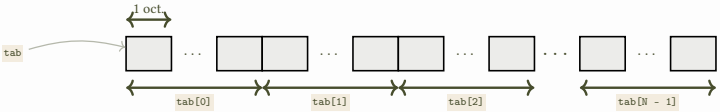


Pointeurs et tableaux statiques

Un tableau est un pointeur vers le 1er élément qui le constitue. Les autres éléments du tableau sont **contigus en mémoire** et situés à des adresses plus élevées.

On **déclare** un tableau statique de taille `N` de valeurs de type `T` par

```
T tab[N];
```



On **accède à la valeur** du `i`e élément de `tab` par

```
tab[i]
```

On **affecte une valeur** `VAL` de type `T` en `i`e position dans `tab` par

```
tab[i] = VAL;
```

Pointeurs et tableaux statiques

Sachant qu'un tableau `tab` est un pointeur et que ses éléments **sont contigus en mémoire**, la syntaxe

```
tab[i]
```

est équivalente à

```
*(tab + i)
```

Le **type du pointeur** `tab` permet de faire un **décalage correct** en fonction de la taille en mémoire de ses éléments.

En effet, si `ptr` est un pointeur pointant sur une zone mémoire de type `T`, la valeur de l'expression `ptr + i` dépend de la taille nécessaire pour représenter une valeur de type `T` (c.-à-d. de `sizeof(T)`).

Pointeurs et tableaux statiques

– Exemple –

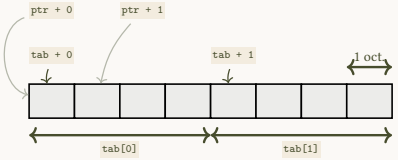
Considérons les instructions suivantes :

```
int tab[2];
char *ptr;

tab[0] = 300;
tab[1] = 60;

printf("%p %p\n", tab + 0, tab + 1);
printf("%d %d\n", tab[0], tab[1]);

ptr = (char *) tab;
printf("%p %p\n", ptr + 0, ptr + 1);
printf("%d %d\n", ptr[0], ptr[1]);
```

Leur exécution affiche

```
0x7fffc38b5d60 0x7fffc38b5d64
300 60
0x7fffc38b5d60 0x7fffc38b5d61
44 1
```

Le pointeur `ptr` interprète les éléments du tableau `tab` comme des valeurs de taille 1 octet (= **sizeof**`(char)`).



Plan

5. Allocation dynamique

Passage par valeur — rappel

Nous savons qu'il est impossible de modifier la valeur d'un argument passé à une fonction car celle-ci travaille sur une **copie de sa valeur** et non pas sur l'éventuelle variable figurant en argument.

– Exemple –

```
void incrementer(int nb) {
    nb = nb + 1;
}
...
num = 5;
incrementer(num);
printf("%d\n", num);
```

Ces instructions affichent `5`.

En ligne 6, c'est la **valeur** de `num` qui est transmise et non pas la variable elle-même.

Ceci est encore plus clair quand il s'agit de l'appel

```
incrementer(2 * num + 1)
```

qui est considéré. La fonction travaille en effet avec la valeur **issue de l'évaluation** de l'expression `2 * num + 1` qui est recopiée dans la variable locale (paramètre) `nb`. Cette expression n'étant **pas une valeur gauche**, il est clair que sa valeur ne peut donc pas être modifiée.

Passage par adresse

Cependant, si l'on donne en argument l'adresse et le type d'une zone mémoire (ce qui revient à donner un pointeur vers la zone mémoire considérée), il devient possible de la modifier.

Il s'agit d'un passage par adresse.

– Exemple –

```
void incrementer(int *ptr_nb) {
    *ptr_nb = *ptr_nb + 1;
}
...
num = 5;
incrementer(&num);
printf("%d\n", num);
```

Ces instructions affichent 6.

En ligne 6, c'est (la valeur de) l'**adresse** de `num` qui est transmise.

Celle-ci est **universelle** (visible partout).

Qualificateur `const`

Dans certains cas, un passage par adresse n'est pas fait pour modifier la valeur située à l'adresse spécifiée.

– Exemple –

Cette fonction affiche, **sans la modifier**, la chaîne de caractères `chaine` en substituant des étoiles `'*'` aux caractères `c`.

On souhaite **protéger** `chaine` de toute modification sur son contenu.

Pour cela, on place `const` devant la déclaration de `chaine`.

```
void etoiles(const char *chaine, char c) {
    int i;
    i = 0;
    while (chaine[i] != '\0') {
        if (chaine[i] == c)
            putchar('*');
        else
            putchar(chaine[i]);
        i += 1;
    }
}
```

Le mot-clé `const` est un qualificateur.

Qualificateur `const`

Ainsi, plus généralement,

```
const T *ID
```

déclare un paramètre `ID`, pointeur sur une zone mémoire de type `T`, dont le **contenu est protégé en écriture**.

– Exemples –

```
void exemple_1(const int *x) {
    *x = 40;
}
```

Cette tentative directe pour modifier la valeur à l'adresse `x` est sanctionnée par le compilateur par une erreur.

```
void exemple_2(const int *x) {
    int *tmp;
    tmp = x;
    *tmp = 40;
}
```

Cette tentative détournée pour modifier la valeur à l'adresse `x` est sanctionnée par le compilateur par un avertissement. Il est ainsi possible de modifier une valeur protégée.

Le qualificateur `const` est avant tout une **aide pour le développeur** : il informe d'un comportement à adopter.

Qualificateur `const`

Le qualificateur `const` permet quelques subtilités :

```
T *const ID
```

déclare un paramètre `ID`, **pointeur fixe** sur une zone mémoire de type `T` (il n'est pas possible de faire pointer `ID` ailleurs). De plus,

```
const T *const ID
```

déclare un paramètre `ID`, pointeur fixe sur une zone mémoire de type `T`, dont le contenu est protégé en écriture.

– Exemple –

Voici les quatre cas de figure :

```
void exemple_3(int *a,
        const int *b,
        int *const c,
        const int *const d) {

    int e;
```

```
    a = &e; /* Autorise */
    *a = 3; /* Autorise */

    b = &e; /* Autorise */
    *b = 3; /* Interdit */
```

```
    c = &e; /* Interdit */
    *c = 3; /* Autorise */

    d = &e; /* Interdit */
    *d = 3; /* Interdit */
}
```

Plan

5. Allocation dynamique

Données persistantes en mémoire

Nous savons que toutes les variables locales à une fonction n'existent plus après son appel.

Pour créer des **données persistantes** dans une fonction, il faut écrire dans la mémoire ailleurs que dans la pile.

On écrit pour cela dans le **tas**. Cela se fait en deux temps :

1. on demande au système d'**allouer** (de réserver) une certaine portion du tas ;
2. on écrit ensuite dans cette zone en lui affectant la valeur souhaitée.

Pour allouer une portion du tas, on utilise la fonction

```
void *malloc(int size);
```

Elle renvoie un **pointeur de type générique** `void *` sur une donnée en mémoire de taille `size` octets.

Utilisation de `malloc`

Pour **allouer** une zone de la mémoire pouvant accueillir `N` valeurs d'un type `T`, on utilise l'instruction

```
ptr = (T *) malloc(sizeof(T) * N);
```

où `ptr` est un pointeur pointant sur une zone de la mémoire de type `T`.

Explications :

- `(T *)` sert à préciser que la zone de la mémoire à allouer est de type `T`. Cette précision, qui est une coercition, est nécessaire car par défaut `malloc` renvoie un pointeur sur une zone non typée (`void *`) ;
- l'argument `sizeof(T) * N` permet de demander à allouer une zone mémoire de taille `sizeof(T) * N` octets. Celle-ci pourra donc accueillir `N` valeurs de type `T`.

Après exécution de cette instruction, `ptr` pointe sur le **début** d'une zone de la mémoire de taille `sizeof(T) * N` octets.

Tests d'erreurs et `malloc`

C'est le **système d'exploitation** qui, lors de l'appel à `malloc` se charge de fournir une adresse pour la zone mémoire à allouer.

Il se peut que pour une raison ou une autre, il ne soit pas possible d'allouer la zone mémoire demandée. Dans ce cas, `malloc` renvoie une valeur spéciale : `NULL`. Cette valeur vaut `0` et est une constante définie dans `stdlib.h`.

Il est **impératif de tester**, pour toute allocation dynamique réalisée, son **bon déroulement**. On procède de la manière suivante :

```
char *ptr;
ptr = (char *) malloc(sizeof(char) * 1);
if (NULL == ptr)
    ACTION
```

De cette manière, si l'allocation s'est mal passée, on prend en charge cette erreur par les instructions figurant dans `ACTION`.

`ACTION` peut être un `exit(EXIT_FAILURE);` si l'on se trouve dans le `main` ou un `return NULL;` voire un `return 0;` si l'on se trouve dans une fonction.

Libération de mémoire

Pour **désallouer** une zone de la mémoire, on utilise la fonction

```
void free(void *ptr);
```

On l'utilise de la manière suivante :

```
free(ptr);
ptr = NULL; Non obligatoire.
```

où `ptr` est un pointeur. La 2e ligne sert à ne plus conserver l'accès sur la zone libéré (non indispensable mais peut éviter des erreurs).

– Exemple –

```
1 short *ptr;
2 ptr = (short *) malloc(sizeof(short) * 15);
3 free(ptr);
4 ptr = NULL;
```

La l. 2 alloue une zone de la mémoire pouvant accueillir `15` valeurs de type `short`. En l. 3, cette zone est libérée. Il est d'ores impossible d'y accéder.

Important : pour éviter les **fuites mémoire**, il faut **désallouer** toute zone de la mémoire qui **n'est plus utilisée**.

La fonction `calloc`

La fonction

```
void *calloc(int length, int size);
```

alloue et renvoie un pointeur sur une zone de la mémoire de `length * size` octets, tous **initialisés** avec des `0`.

– Exemple –

```
long *ptr;
ptr = (long *) calloc(13, sizeof(long));
```

alloue une zone de la mémoire pouvant accueillir `13` valeurs de type `long`, initialisées à `0`.