

Plan

Compilation

Étapes de compilation

Compilation séparée

Makefile simples

Makefile avancés

Bibliothèques

Variables dans les Makefile

Il est possible de **définir des variables** dans un Makefile par

```
ID=VAL
```

Ceci définit une variable identifiée par `ID`. Sa valeur est la **chaîne de caractères** `VAL`.

On accède à la valeur d'une variable identifiée par `ID` par

```
$(ID)
```

Ceci substitue à l'occurrence de `$(ID)` la chaîne de caractères qui lui a été attribuée lors de sa définition.

Variables dans les Makefile — exemple

Les variables permettent de factoriser les règles d'un Makefile :

```
1 Main: Main.o A.o
2     gcc -o Main Main.o A.o -ansi -pedantic -Wall
3
4 Main.o: Main.c
5     gcc -c Main.c -ansi -pedantic -Wall
6
7 A.o: A.c A.h
8     gcc -c A.c -ansi -pedantic -Wall
```

s'écrit plus simplement par

```
1 CFLAGS=-ansi -pedantic -Wall
2
3 Main: Main.o A.o
4     gcc -o Main Main.o A.o $(CFLAGS)
5
6 Main.o: Main.c
7     gcc -c Main.c $(CFLAGS)
8
9 A.o: A.c A.h
10    gcc -c A.c $(CFLAGS)
```

Variables dans les Makefile

On utilise en général les noms de variable suivants :

- ▶ `CFLAGS` pour les options de compilation, p.ex.,

```
CFLAGS=-ansi -pedantic -Wall
```

- ▶ `LDFLAGS` pour l'inclusion de bibliothèques, p.ex.,

```
LDFLAGS=-lm -lMLV
```

- ▶ `CC` pour le compilateur utilisé, p.ex.,

```
CC=gcc      ou bien      CC=colorgcc
```

- ▶ `OPT` pour les option d'optimisation de code

```
OPT=-O1     ou bien      OPT=-O2     ou encore      OPT=-O3
```

Règles courantes

Observation : la plupart des règles des `Makefile` sont sous l'une de ces deux formes :

1. `M.o: M.c DEP2... DEPn`
→ `gcc -c M.c`
2. `EXEC: DEP1 ... DEPn`
→ `gcc -o EXEC DEP1 ... DEPn`

La 1^{re} forme de règle a pour but de construire le fichier objet d'un module `M`. Dans ce cas, `DEP2, ..., DEPn` sont les `.h` dont le module `M` dépend.

La 2^e forme de règle a pour but de construire l'exécutable `EXEC` du projet. Les dépendances `DEP1, ..., DEPn` sont dans ce cas les fichiers `.o` du projet.

Variables internes dans les Makefile

Il est possible de simplifier l'écriture de ces règles courantes au moyen des **variables internes**. Il y en a trois principales et deux secondaires :

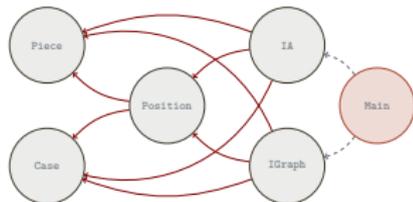
Symbole	Ce qu'il désigne
<code>\$@</code>	Nom de la cible
<code>\$<</code>	Nom de la 1 ^{re} dép.
<code>\$^</code>	Noms de toutes les dép.
<code>\$?</code>	Noms des dép. plus récentes que la cible
<code>\$*</code>	Nom de la cible sans extension

En les utilisant, les deux règles précédentes deviennent

`M.o: M.c DEP2 ... DEPn`
`gcc -c M.c` → `M.o: M.c DEP2 ... DEPn`
`gcc -c $<`

`EXEC: DEP1 ... DEPn`
`gcc -o EXEC DEP1 ... DEPn` → `EXEC: DEP1 ... DEPn`
`gcc -o $@ $^`

Variables internes dans les Makefile – exemple



L'utilisation des variables et variables internes permet de simplifier les Makefile.

```
1 CC=colorgcc
2 CFLAGS=-ansi -pedantic -Wall
3 OBJ=Main.o Piece.o Case.o
4     Position.o IA.o IGraph.o
5
6 Echecs: $(OBJ)
7     $(CC) -o $@ $^ $(CFLAGS)
8
9
10 Main.o: Main.c IA.h IGraph.h
11     $(CC) -c $< $(CFLAGS)
12
13 Piece.o: Piece.c Piece.h
14     $(CC) -c $< $(CFLAGS)
15
```

```
16 Case.o: Case.c Case.h
17     $(CC) -c $< $(CFLAGS)
18
19 Position.o: Position.c Position.h
20     Piece.h Case.h
21     $(CC) -c $< $(CFLAGS)
22
23 IA.o: IA.c IA.h
24     Piece.h Case.h Position.h
25     $(CC) -c $< $(CFLAGS)
26
27 IGraph.o: IGraph.c IGraph.h
28     Piece.h Case.h Position.h
29     $(CC) -c $< $(CFLAGS)
```

Règles génériques

Il est possible de simplifier encore d'avantage l'écriture des `Makefile` au moyen des **règles génériques**.

Ce sont des règles de la forme

```
%.o: %.c
```

→ COMMANDE

où COMMANDE est une commande.

Cette syntaxe simule une règle

```
M.o: M.c
```

→ COMMANDE

pour chaque fichier `M.c` du projet.

Intérêt principal : l'unique règle

```
%.o: %.c
```

```
gcc -c $<
```

permet de construire chaque fichier objet du projet.

Règles génériques

Attention : la règle

```
%.o: %.c  
    gcc -c $<
```

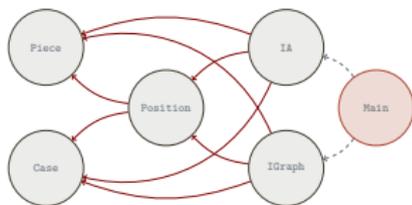
ne prend pas en compte des dépendances des modules aux fichiers `.h` concernés.

Il faut les mentionner explicitement de la manière suivante :

```
M.o: M.c M.h DEP1 ... DEPn
```

pour chaque module `M` du projet. `DEP1, ..., DEPn` sont les `.h` dont le module `M` dépend.

Règles génériques — exemple



L'utilisation des règles génériques permet de simplifier encore les Makefile.

```
1 CC=colgccc
2 CFLAGS=-ansi -pedantic -Wall
3 OBJ=Main.o Piece.o Case.o
4   Position.o IA.o IGraph.o
5
6 Echecs: $(OBJ)
7   $(CC) -o $@ $^ $(CFLAGS)
8
9
10 Main.o: Main.c IA.h IGraph.h
11
12 Piece.o: Piece.c Piece.h
13
14 Case.o: Case.c Case.h
```

```
15
16 Position.o: Position.c Position.h
17   Piece.h Case.h
18
19 IA.o: IA.c IA.h
20   Piece.h Case.h Position.h
21
22 IGraph.o: IGraph.c IGraph.h
23   Piece.h Case.h Position.h
24
25 %.o: %.c
26   $(CC) -c $< $(CFLAGS)
```

Règles de nettoyage

Règle de nettoyage :

```
1 clean:
2   rm -f *.o
```

Cette règle permet, lorsque l'on

saisit la commande `make clean`, de supprimer les fichiers `.o` du répertoire courant.

Règle de nettoyage total :

```
1 mrproper: clean
2   rm -f EXEC
```

Cette règle, où `EXEC` est le nom de l'exécutable du projet, permet de

supprimer tous les **fichiers régénérables** (c.-à-d. les fichiers `.o` et l'exécutable) à partir des fichiers `.c` et `.h` du projet.

Règles d'installation / désinstallation

Règle d'installation :

```
1 install: EXEC
2     mkdir ../bin
3     mv EXEC ../bin/EXEC
4     make mrproper
```

Cette règle permet de compiler le projet, de placer son exécutable `EXEC` dans un répertoire `bin` et de supprimer les fichiers régénérables.

Règle de désinstallation :

```
1 uninstall: mrproper
2     rm -f ../bin/EXEC
3     rm -rf ../bin
```

fichiers régénérables, l'exécutable du projet, ainsi que le répertoire `bin` le contenant.

Cette règle permet de supprimer les

Plan

Compilation

Étapes de compilation

Compilation séparée

Makefile simples

Makefile avancés

Bibliothèques

Principes généraux

Une **bibliothèque** est un **regroupement de modules** offrant des fonctionnalités allant vers un même objectif.

Principes généraux

Une **bibliothèque** est un **regroupement de modules** offrant des fonctionnalités allant vers un même objectif.

Par exemple, la bibliothèque graphique MLV est un ensemble de plusieurs modules (`MLV_audio`, `MLV_keyboard`, `MLV_image`, *etc.*) réunis dans le but d'offrir des fonctions d'affichage graphique et de gérer les entrées / sorties (son, clavier, souris, *etc.*).

Principes généraux

Une **bibliothèque** est un **regroupement de modules** offrant des fonctionnalités allant vers un même objectif.

Par exemple, la bibliothèque graphique MLV est un ensemble de plusieurs modules (`MLV_audio`, `MLV_keyboard`, `MLV_image`, *etc.*) réunis dans le but d'offrir des fonctions d'affichage graphique et de gérer les entrées / sorties (son, clavier, souris, *etc.*).

L'intérêt des bibliothèques est double :

1. il suffit de réaliser une inclusion d'**un seul en-tête** pour bénéficier des fonctionnalités d'une bibliothèque, plutôt que des inclusions de plusieurs en-têtes sans être sûr du fichier précis à inclure ;

Principes généraux

Une **bibliothèque** est un **regroupement de modules** offrant des fonctionnalités allant vers un même objectif.

Par exemple, la bibliothèque graphique MLV est un ensemble de plusieurs modules (`MLV_audio`, `MLV_keyboard`, `MLV_image`, *etc.*) réunis dans le but d'offrir des fonctions d'affichage graphique et de gérer les entrées / sorties (son, clavier, souris, *etc.*).

L'intérêt des bibliothèques est double :

1. il suffit de réaliser une inclusion d'**un seul en-tête** pour bénéficier des fonctionnalités d'une bibliothèque, plutôt que des inclusions de plusieurs en-têtes sans être sûr du fichier précis à inclure ;
2. sous réserve de savoir comment créer des bibliothèques, il est possible de **partager et réutiliser** son propre code entre plusieurs de ses projets, sans avoir à le recompiler.

Bibliothèques statiques

Une **bibliothèque statique** est un fichier d'extension `.a`.

Bibliothèques statiques

Une **bibliothèque statique** est un fichier d'extension `.a`.

Lors de son utilisation, son code est inclus dans l'exécutable pendant l'édition des liens.

Bibliothèques statiques

Une **bibliothèque statique** est un fichier d'extension `.a`.

Lors de son utilisation, son code est inclus dans l'exécutable pendant l'édition des liens.

- ▶ **Avantage** : tout projet qui utilise une bibliothèque statique peut être exécuté sur une machine où la bibliothèque n'est pas installée.

Bibliothèques statiques

Une **bibliothèque statique** est un fichier d'extension `.a`.

Lors de son utilisation, son code est inclus dans l'exécutable pendant l'édition des liens.

- ▶ **Avantage** : tout projet qui utilise une bibliothèque statique peut être exécuté sur une machine où la bibliothèque n'est pas installée.
- ▶ **Inconvénient** : l'exécutable est plus volumineux.

Bibliothèques dynamiques

Une **bibliothèque dynamique** est un fichier d'extension `.so`.

Bibliothèques dynamiques

Une **bibliothèque dynamique** est un fichier d'extension `.so`.

Lors de son utilisation, son code n'est pas inclus dans l'exécutable. C'est lors de l'exécution que les symboles provenant de la bibliothèque sont résolus au moyen de l'éditeur de liens dynamique.

Bibliothèques dynamiques

Une **bibliothèque dynamique** est un fichier d'extension `.so`.

Lors de son utilisation, son code n'est pas inclus dans l'exécutable. C'est lors de l'exécution que les symboles provenant de la bibliothèque sont résolus au moyen de l'éditeur de liens dynamique.

- ▶ **Avantages** : l'exécutable est moins volumineux. Il n'y a pas de duplication du code de la bibliothèque sur un même système si plusieurs projets l'utilisent.

Bibliothèques dynamiques

Une **bibliothèque dynamique** est un fichier d'extension `.so`.

Lors de son utilisation, son code n'est pas inclus dans l'exécutable. C'est lors de l'exécution que les symboles provenant de la bibliothèque sont résolus au moyen de l'éditeur de liens dynamique.

- ▶ **Avantages** : l'exécutable est moins volumineux. Il n'y a pas de duplication du code de la bibliothèque sur un même système si plusieurs projets l'utilisent.
- ▶ **Inconvénient** : tout projet qui utilise une bibliothèque dynamique ne peut être exécuté que sur un système où cette dernière est installée.

Compilation d'un projet utilisant des bibliothèques

On suppose que l'on travaille sur un projet constitué de deux modules A et B et d'un fichier principal `Main.c`. Ce projet utilise deux bibliothèques statiques : `libm.a` et `libMLV.a`.

Compilation d'un projet utilisant des bibliothèques

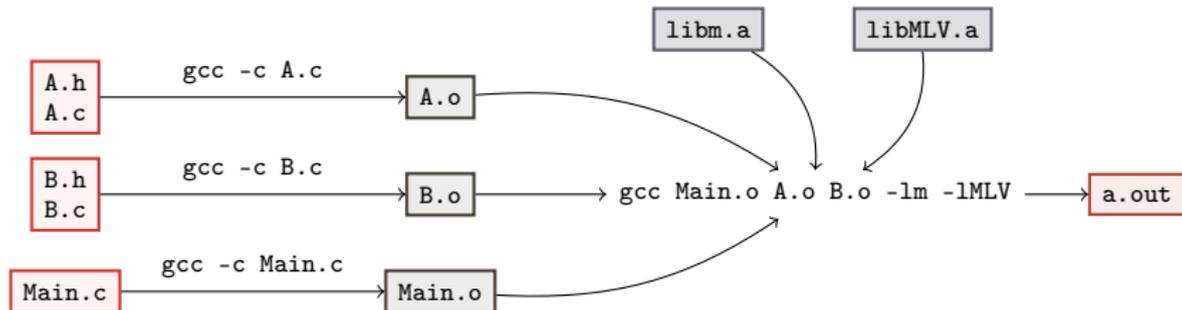
On suppose que l'on travaille sur un projet constitué de deux modules A et B et d'un fichier principal `Main.c`. Ce projet utilise deux bibliothèques statiques : `libm.a` et `libMLV.a`.

La compilation de ce projet se réalise de manière habituelle. La différence porte sur l'étape d'**édition des liens** dans laquelle il est nécessaire de **signaler les bibliothèques utilisées**.

Compilation d'un projet utilisant des bibliothèques

On suppose que l'on travaille sur un projet constitué de deux modules A et B et d'un fichier principal `Main.c`. Ce projet utilise deux bibliothèques statiques : `libm.a` et `libMLV.a`.

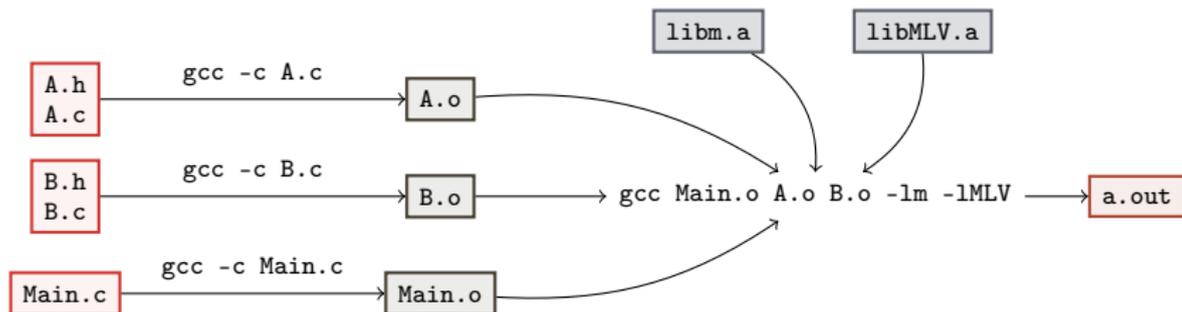
La compilation de ce projet se réalise de manière habituelle. La différence porte sur l'étape d'**édition des liens** dans laquelle il est nécessaire de **signaler les bibliothèques utilisées**.



Compilation d'un projet utilisant des bibliothèques

On suppose que l'on travaille sur un projet constitué de deux modules A et B et d'un fichier principal `Main.c`. Ce projet utilise deux bibliothèques statiques : `libm.a` et `libMLV.a`.

La compilation de ce projet se réalise de manière habituelle. La différence porte sur l'étape d'**édition des liens** dans laquelle il est nécessaire de **signaler les bibliothèques utilisées**.

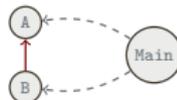


Pour utiliser des bibliothèques qui ne sont pas situées dans le répertoire standard des bibliothèques, on précisera leur chemin `chem` lors de l'édition des liens au moyen de l'option `-Lchem`.

Compilation d'un projet utilisant des bibliothèques

Le nom d'une bibliothèque commence par `lib`. On signale l'utilisation d'une bibliothèque à l'éditeur de liens par `-lNOM` où `libNOM` est le nom de la bibliothèque.

En supposant que le graphe d'inclusions (étendues) du projet précédent est celui ci-contre, un `Makefile` possible est



```
1  CC=colorgcc
2  CFLAGS=-ansi -pedantic -Wall
3  LDFLAGS=-lm -lMLV
4  OBJ=Main.o A.o B.o
5
6  Projet: $(OBJ)
7          $(CC) -o $@ $^ $(CFLAGS) $(LDFLAGS)
8
9  Main.o: Main.c A.h B.h
10
11 A.o: A.c A.h
12
13 B.o: B.c B.h A.h
14
15 %.o: %.c
16     $(CC) -c $^ $(CFLAGS)
17
18  clean:
19      rm -f *.o
20
21  mrproper: clean
22      rm -f Projet
23
24  install: Projet
25      mkdir ../bin
26      mv Projet ../bin/Projet
27      make mrproper
28
29  uninstall: mrproper
30      rm -f ../bin/Projet
31      rm -rf ../bin
```

La bibliothèque standard

La **bibliothèque standard** `libc.a` (`libc.so`) regroupe les vingt-quatre modules

<code>assert,</code>	<code>complex,</code>	<code>ctype,</code>	<code>errno,</code>
<code>fenv,</code>	<code>float,</code>	<code>inttypes,</code>	<code>iso646,</code>
<code>limits,</code>	<code>locale,</code>	<code>math,</code>	<code>setjmp,</code>
<code>signal,</code>	<code>stdarg,</code>	<code>stdbool,</code>	<code>stddef,</code>
<code>stdint,</code>	<code>stdio,</code>	<code>stdlib,</code>	<code>string,</code>
<code>tgmath,</code>	<code>time,</code>	<code>wchar,</code>	<code>wctype.</code>

La bibliothèque standard

La **bibliothèque standard** `libc.a` (`libc.so`) regroupe les vingt-quatre modules

```
assert,    complex,    ctype,      errno,  
fenv,      float,      inttypes,   iso646,  
limits,    locale,     math,       setjmp,  
signal,    stdarg,     stdbool,    stddef,  
stdint,    stdio,      stdlib,     string,  
tgmath,    time,       wchar,      wctype.
```

Cette bibliothèque est **liée implicitement** lors de toute édition des liens.

La bibliothèque standard

La **bibliothèque standard** `libc.a` (`libc.so`) regroupe les vingt-quatre modules

<code>assert,</code>	<code>complex,</code>	<code>ctype,</code>	<code>errno,</code>
<code>fenv,</code>	<code>float,</code>	<code>inttypes,</code>	<code>iso646,</code>
<code>limits,</code>	<code>locale,</code>	<code>math,</code>	<code>setjmp,</code>
<code>signal,</code>	<code>stdarg,</code>	<code>stdbool,</code>	<code>stddef,</code>
<code>stdint,</code>	<code>stdio,</code>	<code>stdlib,</code>	<code>string,</code>
<code>tgmath,</code>	<code>time,</code>	<code>wchar,</code>	<code>wctype.</code>

Cette bibliothèque est **liée implicitement** lors de toute édition des liens.

Il est donc possible d'utiliser certains des modules de la bibliothèque standard simplement en les incluant (`#include <NOM.h>`), sans avoir à utiliser l'option `-lc`.

La bibliothèque standard

La **bibliothèque standard** `libc.a` (`libc.so`) regroupe les vingt-quatre modules

<code>assert,</code>	<code>complex,</code>	<code>ctype,</code>	<code>errno,</code>
<code>fenv,</code>	<code>float,</code>	<code>inttypes,</code>	<code>iso646,</code>
<code>limits,</code>	<code>locale,</code>	<code>math,</code>	<code>setjmp,</code>
<code>signal,</code>	<code>stdarg,</code>	<code>stdbool,</code>	<code>stddef,</code>
<code>stdint,</code>	<code>stdio,</code>	<code>stdlib,</code>	<code>string,</code>
<code>tgmath,</code>	<code>time,</code>	<code>wchar,</code>	<code>wctype.</code>

Cette bibliothèque est **liée implicitement** lors de toute édition des liens.

Il est donc possible d'utiliser certains des modules de la bibliothèque standard simplement en les incluant (`#include <NOM.h>`), sans avoir à utiliser l'option `-lc`.

L'utilisation de certains modules doit cependant s'accompagner de l'option `-lX` (comme `-lm` pour utiliser `math.h`).

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

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.

Création de bibliothèques statiques — exemple

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`.

Création de bibliothèques statiques — exemple

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

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

```
/* 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
```

<pre>/* A.h */ #ifndef __A__ #define __A__ char h(int a); #endif</pre>	<pre>/* A.c */ #include "A.h" char h(int a) { return a % 256; }</pre>
--	---

<pre>/* B.h */ #ifndef __B__ #define __B__ typedef int S; int f(S s); char g(int a); #endif</pre>	<pre>/* B.c */ #include "B.h" int f(S s) { return s; } char g(int a) { return a % 64; }</pre>
--	---

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
```

Indexe de l'archive:

```
h in A.o
f in B.o
g in B.o
```

```
A.o:
0000000000000000 T h
```

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

Résumé de l'axe 1

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

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

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.

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

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.

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

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

Allocation dynamique

Entrées et sorties

Types

Types structurés

Plan

Allocation dynamique

Pointeurs

Passage par adresse

Allocation dynamique

Tableaux dynamiques

Plan

Allocation dynamique

Pointeurs

Passage par adresse

Allocation dynamique

Tableaux dynamiques

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**.

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

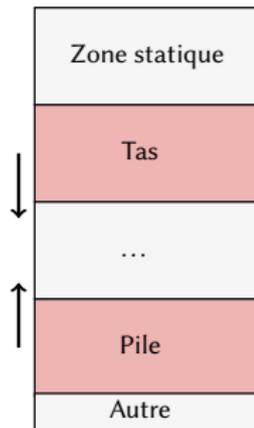
Lors de l'exécution d'un programme, une portion (de taille variable) de la mémoire lui est dédiée. Cette zone lui est exclusive. On l'appelle la **mémoire**.

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).



La mémoire

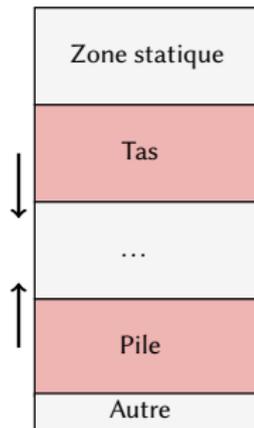
Lors de l'exécution d'un programme, une portion (de taille variable) de la mémoire lui est dédiée. Cette zone lui est exclusive. On l'appelle la **mémoire**.

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**.

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.

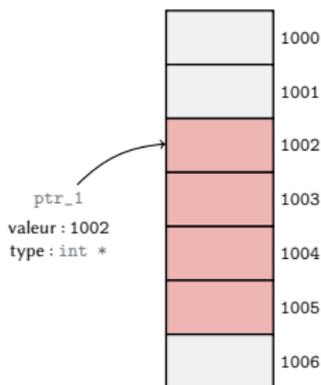
Le **mode d'emploi** décrit le type de la zone de la mémoire ainsi adressée en renseignant sur la taille de la zone.

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.

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

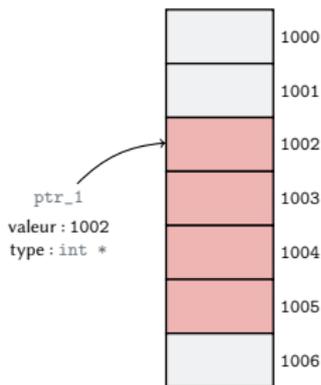


Pointeurs

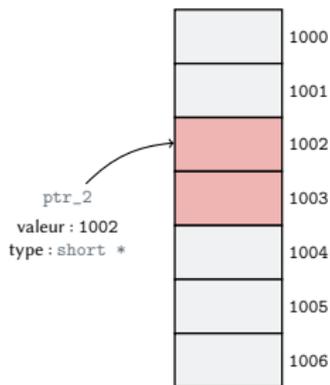
Intuitivement, un pointeur est une **flèche** munie d'un mode d'emploi, le tout pointant vers une zone de la mémoire.

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

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;
```

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
```

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
```

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`.

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`.

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`.

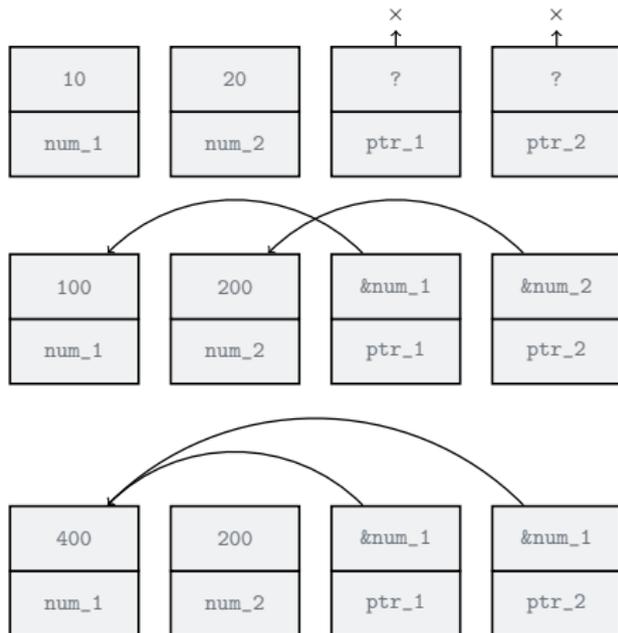
Il est possible d'**affecter une valeur** `VAL` de type `T` à `*ptr` par

```
*ptr = VAL;
```

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

Manipulation de pointeurs – exemple

```
1  /* (1) */
2  int num1, num2;
3  int *ptr1, *ptr2;
4  num1 = 10;
5  num2 = 20;
6
7  /* (2) */
8  ptr1 = &num1;
9  ptr2 = &num2;
10 *ptr1 = 100;
11 *ptr2 = 200;
12
13 /* (3) */
14 ptr2 = ptr1;
15 *ptr1 = 300;
16 *ptr2 = 400;
```



Pointeurs et tableaux statiques

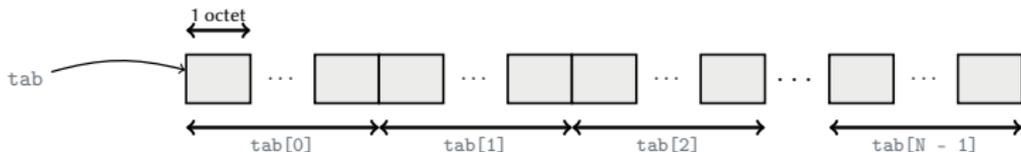
Un **tableau** est un pointeur vers le 1^{er} élément qui le constitue. Les autres éléments du tableau sont contigus en mémoire et situés à des adresses plus élevées.

Pointeurs et tableaux statiques

Un **tableau** est un pointeur vers le 1^{er} 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];
```

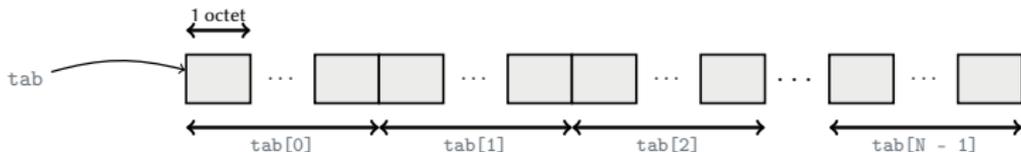


Pointeurs et tableaux statiques

Un **tableau** est un pointeur vers le 1^{er} 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

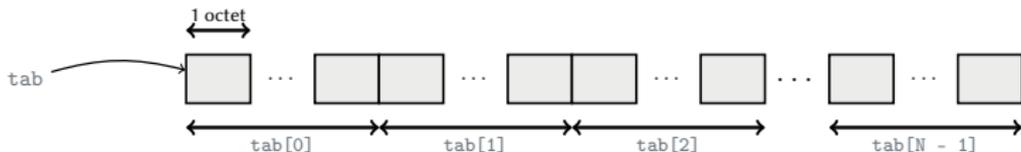
```
tab[i]
```

Pointeurs et tableaux statiques

Un **tableau** est un pointeur vers le 1^{er} 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;
```