

# Programmation impérative

---

## Programmation avancée en langage C

F. Pellegrini  
ENSEIRB

Ce document est copiable et distribuable librement et gratuitement à la condition expresse que son contenu ne soit modifié en aucune façon, et en particulier que le nom de son auteur et de son institution d'origine continuent à y figurer, de même que le présent texte.

# Débogage (1)

---

- Plus des quatre cinquièmes du temps de programmation ne concernent pas l'écriture de nouvelles lignes mais le débogage des lignes déjà écrites
- Rendre plus efficace le débogage est donc essentiel d'un point de vue économique
- Des outils adaptés existent, mais s'ils ne sont pas disponibles sur la machine sur laquelle le programme est lancé, il faut recourir à des techniques plus rustiques

# Débogage (2)

---

- Parmi les symptômes de bogues les plus courants, on trouve :
  - Débordements de tableaux
    - Chaînes de caractères y compris
    - Dans la pile ou dans le tas (erreurs en retour de fonction ou lors des appels de fonctions gérant la mémoire)
    - Problèmes supplémentaires avec l'allocation dynamique
  - Lectures de zones mémoire non initialisées
    - Y compris les pointeurs
  - Utilisation d'espace mémoire dynamique déjà libéré
    - Déjà réutilisé ou en voie de réutilisation

# Débogage « au printf » (1)

---

- Pour avoir une idée précise du moment où un programme plante, il est possible de l'instrumenter au moyen de fonctions à effets de bord
- Le « printf » est le moyen le plus classique
  - Simple et facile à mettre en œuvre
  - Flexible
    - On peut faire afficher ce que l'on veut
  - Sorties à l'écran ou dans un fichier (redirection)

# Débogage « au printf » (2)

- Il existe deux flots de sortie standard
  - `stdout` : sortie standard
    - Associé au descripteur de fichiers 0
    - Bufferisée : les données sont écrites dans une zone tampon, et sont effectivement transmises au système :
      - A la fin de chaque ligne si la destination est un terminal
      - Lorsque le tampon est plein si la destination est un fichier
    - Améliore l'efficacité des programmes
  - `stderr` : sortie standard d'erreur
    - Associé au descripteur de fichiers 2
    - Non bufferisée : les données écrites sont immédiatement transmises au système pour affichage immédiat

# Débogage « au printf » (3)

---

- Il ne faut donc pas utiliser « `printf` » mais « `fprintf (stderr, ...)` »
  - Sinon, des messages écrits avec `printf` peuvent ne pas encore avoir été effectivement envoyés au système lorsque le programme se plante
  - On peut croire que le programme s'est planté en amont de l'endroit où il s'est effectivement planté
  - La commande de redirection shell est différente
- On peut également utiliser la fonction « `fflush (stdout)` » pour forcer la vidange du tampon de sortie standard

# Débogage symbolique

---

- Le débogage « au `printf` » est un processus souvent long et fastidieux
  - Comme on ne sait pas vraiment ce que l'on cherche, on génère souvent de grosses masses de traces avant d'y partir à la pêche aux indices
- L'idéal est de pouvoir regarder s'exécuter le programme, au niveau du code source, comme si on l'exécutait « à la main », et de pouvoir contrôler pas à pas la mise à jour des variables
  - Débogage « symbolique », au niveau du code source

# Gdb (1)

---

- Débogueur symbolique
- Permet à l'utilisateur :
  - D'examiner de façon symbolique, au niveau du code source, les valeurs des variables et du pointeur d'instructions d'un processus ayant terminé anormalement
    - Données extraites du fichier `core` du processus
  - D'exécuter pas à pas, de façon interactive, un processus actif, et d'agir sur le contenu de ses variables



## Gdb (2)

- Pour que le débogueur symbolique puisse connaître de façon explicite le nom et le type de toutes les entités manipulées par le programme, il faut compiler les programmes avec l'option `-g`, qui ajoute une table des symboles à la fin de l'exécutable
- Pour que les numéros de lignes affichés correspondent bien aux lignes du code source, il faut désactiver les optimisations de code, avec l'option `-O0` (ou au plus `-O1`)

```
% gcc -g -O0 bro1.c -o bro1
```

# Gdb (3)

```
#include <stdlib.h>
int
f ()
{
    int i;
    i = getuid ();
    return (i);
}
int
main (void)
{
    int    i;
    int    ids[2];
    char   buf[10];

    ids[0] = f();
    for (i = 0; i < 10; i ++ )
        printf (": ");
    scanf ("%s", buf);
    printf ("Votre uid est %d\n", ids[0]);
}
```

bro1.c

# Gdb (4)

- Pour lancer le débogueur, on préfixe le nom du programme avec le nom de la commande `gdb`

```
% gdb bro1
```

- Pour analyser le fichier core produit par le plantage d'un programme, on donne le nom du fichier core en deuxième argument

```
% gdb bro1 core.1558
```

- Pour autoriser la création de fichiers core, il faut parfois paramétrer le shell

```
% unlimit coredumpsize
```

# Gdb (5)

- Principales commandes :
  - `run [arguments]` : (re)lance le programme
  - `^C` : rend la main à Gdb
  - `c` (« continue ») : reprend l'exécution du programme
  - `s` (« *step* ») : exécute la ligne courante, en rentrant dans les appels de fonctions
  - `n` (« *next* ») : exécute la ligne courante, sans entrer dans les appels de fonctions
  - `u` (« *unroll* ») : exécute jusqu'à la sortie de la boucle
  - `f` (« *finish* ») : exécute jusqu'au retour de la fonction

# Gdb (6)

---

- `break [nom | ligne]` : positionne un point d'arrêt au début de la fonction ou du numéro de ligne donnés
- `cond numéro condition` : définit une condition associée au point d'arrêt de numéro donné
- `dis | ena numéro` : désactive ou réactive le point d'arrêt de numéro donné
- `watch zone` : définit un point de surveillance sur la zone de mémoire donnée
- `del numéro` : supprime le point d'arrêt ou de surveillance de numéro donné

# Gdb (7)

- `where` : affiche la pile des contextes d'appel
- `up` | `down` [nombre] : remonte ou descend d'un ou de plusieurs contextes dans la pile d'appel
- `p` entité [@nombre] : affiche l'entité donnée, ou le nombre donné d'entités consécutives en commençant à partir de l'adresse de la première entité
- `set` entité = valeur : remplace la valeur courante de l'entité par la valeur donnée
- `call` fonction (paramètres) : exécute l'appel de la fonction donnée et renvoie le résultat. Si la fonction provoque une erreur, le programme ne peut continuer

# Gdb (8)

---

- `dir` répertoire : ajoute le répertoire à la liste des répertoires dans lesquels Gdb cherchera les fichiers sources
- `help` : affiche une aide succincte
  - Sous-rubriques par type de fonction
- `quit` : termine l'exécution de Gdb

# Erreurs avec l'allocation dynamique (1)

---

- L'allocation dynamique est un mécanisme fragile
  - Blocs libres et occupés chaînés par des pointeurs situés avant et après chaque bloc, facilement corrompus en cas d'écriture en dehors des bornes
  - Pas de vérifications de cohérence afin de conserver l'efficacité des routines
- Les erreurs peuvent n'être détectées que longtemps après l'instruction qui les cause
  - *Segmentation fault* dans un `malloc` parce que le `free` précédent s'est fait sur un bloc corrompu



# Erreurs avec l'allocation dynamique (2)

---

- Existence de bibliothèques et d'outils destinés à détecter les erreurs d'accès à la mémoire
  - Bibliothèques de gestion mémoire instrumentées pour détecter les écritures en dehors des bornes
    - Nécessité de compiler spécifiquement avec ces bibliothèques ou positionnement de variables d'environnement pour certaines autres (variable `MALLOC_CHECK` sous Linux)
  - Outils de vérification dynamique de chaque accès mémoire lors de l'exécution
    - Nécessitent le plus souvent une compilation particulière
    - Exemples : Valgrind, Purify ...

# Valgrind (1)

---

- Banc de test d'exécution de programmes
- Permet de détecter les erreurs d'exécution et/ou de réaliser un profilage de code
- Basé sur un émulateur de langage machine, qui interprète pas à pas chaque instruction du programme binaire et peut effectuer un certain nombre de vérifications
  - Lent à exécuter car émulation de chaque instruction
  - Besoin de rechercher des jeux de test les plus petits possibles

# Valgrind (2)

- Différentes fonctions de test prédéfinies (appelées « outils ») peuvent être appliquées
- Outils disponibles :
  - memcheck : vérification mémoire poussée
    - Utilisation de mémoire non initialisée, accès en dehors des blocs mémoires alloués ou de la pile, accès à des zones déjà libérées, fuites mémoires, passage en paramètre de pointeurs non valides, recouvrement de zones dans les fonctions de copie mémoire, mauvaise utilisation de certaines routines de gestion de *threads*
  - addrcheck : version allégée de memcheck, sans le test d'accès aux zones mémoires non initialisées

# Valgrind (3)

---

- **cachegrind** : simulateur de comportement de la hiérarchie de caches pour l'analyse poussée de performances
- **massif** : analyse de l'occupation du tas
- **helgrind** : analyse du comportement de programmes multi-threads, pour détecter les zones accédées par plusieurs tâches et non protégées par des mécanismes d'exclusion mutuelle
- **Possibilité de créer ses propres fonctions outils**

# Valgrind (4)

- Pas besoin d'instrumentation particulière du code machine, mais, comme avec Gdb, nécessité de compiler avec les options `-g` et `-O0` pour que les numéros de lignes donnés correspondent aux lignes du code source

```
#include <stdlib.h>
int
main (void)
{
    int * x = malloc (10 * sizeof (int));
    x[10] = 0;
}
```

bro1.c

```
% gcc -g -O0 bro1.c -o bro1
```

# Valgrind (5)

- Lors de l'exécution sous Valgrind, des messages d'avertissement sont générés à la volée pour chaque erreur potentielle détectée, puis à la fin pour les fuites mémoire

```
% valgrind --leak-check=yes bro1
...
==4739== Invalid write of size 4
==4739==      at 0x80483BA: main (bro1.c:6)
==4739==  Address 0x1BA46050 is 0 bytes after a block of size
40 alloc'd
==4739==      at 0x1B8FF896: malloc (vg_replace_malloc.c:149)
==4739==  by 0x80483AD: main (bro1.c:5)
...
40 bytes in 1 blocks are possibly lost in loss record 1 of 1
==4739==      at 0x1B8FF896: malloc (vg_replace_malloc.c:149)
==4739==  by 0x80483AD: main (bro1.c:5)
```

# Purify (1)

---

- Logiciel de vérification des accès mémoire
- Permet de détecter les erreurs d'exécution
- Basé sur l'instrumentation du code machine généré par le compilateur afin d'y insérer des instructions de vérification des accès mémoire
  - Nécessité de disposer d'une version compatible avec le compilateur utilisé pour le développement
  - Compilation et édition de liens très lentes
  - Moyennement lent à exécuter car seules les instructions d'accès mémoire sont ralenties

# Purify (2)

- La compilation s'effectue en préfixant le nom du compilateur avec la commande « `purify` »
- Nécessité de compiler avec les options `-g` et `-O0` pour que les numéros de lignes donnés correspondent effectivement aux lignes du code source

```
% purify gcc -g -O0 bro1.c -o bro1
Purify 2002a.06.00 Solaris 2 (32-bit) Copyright (C) 1992-2002
Rational Software Corp. All rights reserved.
Instrumenting: crt1.o crti.o crtbegin.o ccNJJa0Ew.o
libgcc.a.....
libgcc_eh.a... crtend.o crtn.o Linking
```



# Purify (3)

- Le programme compilé se lance naturellement
- Possibilité de lancer le programme sous débogueur
  - Fonction de point d'arrêt « `purify_stop_here` » appelée juste après chaque détection d'erreur

```
% gdb ./brol
GNU gdb 4.17
Copyright 1998 Free Software Foundation, Inc.
...
(gdb) break purify_stop_here
Breakpoint 1 at 0x50ff4
(gdb)
```

# Purify (4)

- Possibilité d'inspecter le code à chaque erreur

```
(gdb) run
ABW: Array bounds write:
* This is occurring while in:
    main          [ccNJJa0Ew.o]
    _start        [crt1.o]
* Writing 4 bytes to 0x83990 in the heap.
* Address 0x83990 is 1 byte past end of a malloc'd block at
0x83968 of 40 bytes.
* This block was allocated from:
    malloc        [rtlib.o]
    main          [ccNJJa0Ew.o]
    _start        [crt1.o]
Breakpoint 1, 0x50ff4 in purify_stop_here ()
(gdb) up
#1  0x54b78 in main () at bro1.c:6
6      x[10] = 0;
```

# Purify (5)

- Analyse des fuites mémoire à la fin

```
(gdb) c
Purify: Searching for all memory leaks...
Memory leaked: 40 bytes; potentially leaked: 0 bytes
MLK: 40 bytes leaked at 0x83968
  * This memory was allocated from:
      malloc          [rtlib.o]
      main            [brol.c:5]
      _start          [crt1.o]
Purify Heap Analysis (combining suppressed and unsuppressed
blocks)

```

	Blocks	Bytes
Leaked	1	40
Potentially Leaked	0	0
In-Use	0	0
-----		
Total Allocated	1	40

# Compilation multi-fichiers (1)

---

- Trois étapes distinctes s'enchaînent au cours de la compilation
  - Pré-traitement du code source (cpp)
    - Inclusion de sous-fichiers
    - Remplacement de macros
  - Compilation du code source pré-traité (cc)
    - On a un fichier objet pour chaque fichier source compilé
  - Édition de liens (ld)
    - Fusion de tous les fichiers objets pour créer le programme binaire exécutable

# Compilation multi-fichiers (2)

- Schéma réel

Fichiers de code source en langage C

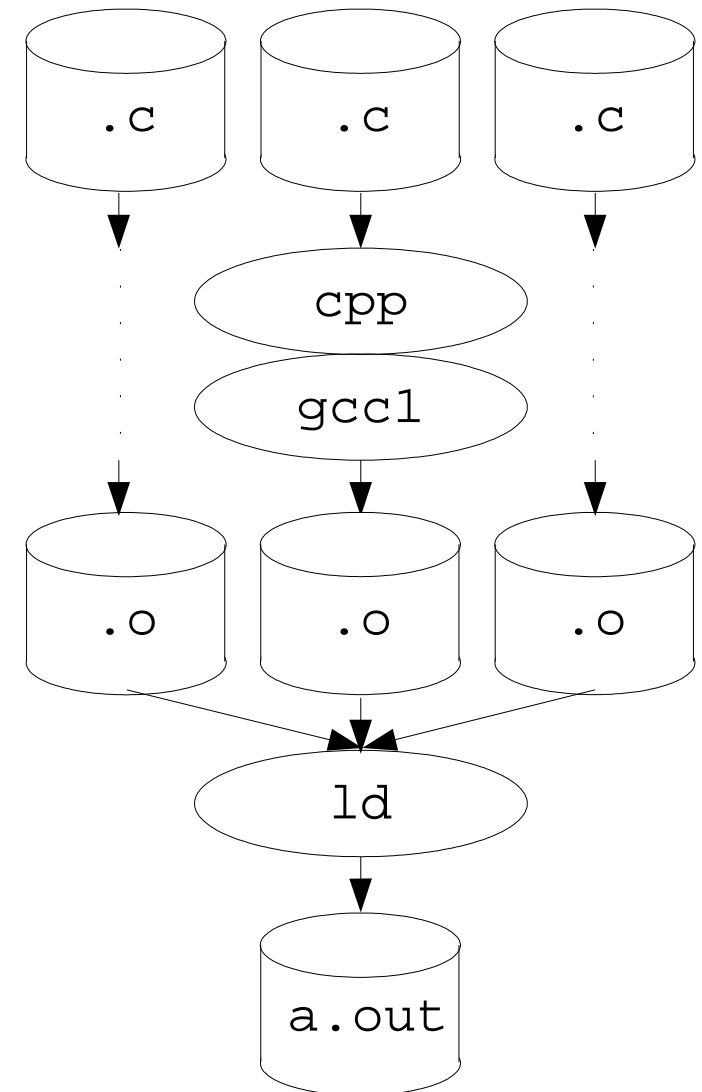
Préprocesseur

Compilateur proprement dit

Fichiers objet

Éditeur de liens (*linker*)

Fichier exécutable



# Compilation multi-fichiers (3)

- On limite la compilation à la seule production d'un fichier objet « .o » au moyen de l'option « -c » du compilateur
- Pour effectuer l'édition de liens, on peut aussi passer par le compilateur, en lui fournissant en entrée la liste des fichiers « .o » à lier

```
% gcc -c brol.c -o brol.o
% gcc -c trol.c -o trol.o
% gcc -c chmol.c -o chmol.o
% gcc brol.o trol.o chmol.o -o brol
```

- On peut automatiser la compilation en utilisant un script ou le programme `make`

# Compilation multi-fichiers (4)

---

- Pour les gros projets organisés en sous-répertoires, il faut pouvoir inclure les fichiers d'en-tête et les bibliothèques provenant d'autres répertoires
- L'option « `-I` » du compilateur permet d'augmenter la liste des répertoires dans lesquels chercher les fichiers d'en-tête
- L'option « `-L` » du compilateur permet d'augmenter la liste des répertoires dans lesquels chercher les fichiers de bibliothèques

# Gestionnaire de compilation

---

- Dès qu'un projet comprend plusieurs modules se pose le problème de la recompilation après modification d'un ou plusieurs fichiers
- La compilation « à la main » des fichiers modifiés est pénible et hasardeuse
- Créer un fichier script shell pour tout recompiler à chaque fois est exagérément coûteux
- Nécessité d'automatiser la recompilation en fonction des dépendances entre fichiers



# Make (1)

---

- Make est un programme permettant de n'effectuer que les traitements nécessaires à l'obtention d'un nouvel exécutable
  - Seule recompilation des fichiers sources modifiés
  - Mise à jour des bibliothèques les contenant
  - Édition des liens pour générer les exécutables

# Make (2)

---

- Afin de remplir sa fonction, Make doit connaître :
  - Les dépendances entre fichiers
  - Les traitements à appliquer
- Ces informations ne sont pas spécifiques aux programmes C, et make peut donc servir à tout type de traitement de mise à jour partielle
  - Documents LaTeX multi-fichiers
  - Remise à jour de bases de données réseau à partir de fichiers texte (anciennes « *Yellow Pages* » / NIS)

# Make (3)

---

- Les informations de dépendances sont contenues dans un fichier appelé `makefile` ou `Makefile`
  - Make utilise le fichier « `makefile` » s'il existe
  - Sinon, il utilise le fichier « `Makefile` »
- L'option « `-f` » de make permet de spécifier un autre nom de fichier si besoin

# Fichier `makefile` (1)

---

- Un fichier `makefile` est un ensemble de clauses « cible : dépendances  $\leftarrow$  actions »
- Pour savoir si une cible doit être reconstruite, `make compare`, éventuellement récursivement, la date de la cible et la date chacune de ses dépendances
- Une fois les dépendances de la cible vérifiées et éventuellement mises à jour, si le fichier cible est moins récent que l'un au moins des fichiers de dépendance, les actions sont effectuées

# Fichier makefile (2)

- Les actions sont situées sur des lignes commençant par une tabulation
  - Une ligne ne commençant pas par une tabulation indique une nouvelle cible (source d'erreurs !)

```
bro1      TAB      : bro1.o trol.o chmol.o
TAB      TAB      gcc bro1.o trol.o chmol.o -o bro1 -lm

bro1.o    TAB      : bro1.c bro1.h trol.h chmol.h
TAB      TAB      gcc -c bro1.c

trol.o    TAB      : trol.c trol.h
TAB      TAB      gcc -c trol.c

chmol.o   TAB      : chmol.c chmol.h trol.h
TAB      TAB      gcc -c chmol.c
```

makefile

# Fichier makefile (3)

---

- Par défaut, Make évalue uniquement la première clause rencontrée, et son arbre de dépendances
- On peut donner en paramètre à Make le nom d'une autre cible
  - Possibilité de définir des actions différentes
  - Possibilité de définir des noms de cibles mnémotechniques sans interférences avec des cibles existantes au moyen de la cible `.PHONY`

# Fichier makefile (4)

- Exemples de cibles mnémotechniques :  
default (en première position), all,  
clean, install, archive ...

```
.PHONY    TAB    : default all clean
default  TAB    : bro1
all      TAB    : bro1 bro1_test
clean    TAB    :
TAB      TAB    rm -f *.o bro1 bro1_test
bro1.o   TAB    : ...
```

makefile

# Variables d'environnement de Make (1)

- Sous Make, il est possible d'utiliser des variables, analogues aux variables d'un script
  - Permettent de factoriser les actions des règles

```
CC=gcc
LD=gcc
...
bro1      TAB      : bro1.o trol.o chmol.o
TAB      TAB      $(LD) bro1.o trol.o chmol.o -o bro1 -lm

bro1.o    TAB      : bro1.c bro1.h trol.h chmol.h
TAB      TAB      $(CC) -c bro1.c -o bro1.o
...
clean     TAB      :
TAB      TAB      $(RM) *.o bro1 bro1_test
```

makefile



# Variables d'environnement de Make (2)

---

- Toutes les variables d'environnement visibles par Make sont définies comme variables
- Toutes les variables de Make sont transmises sous forme de variables d'environnement aux programmes appelés pour réaliser les actions

# Variables d'environnement de Make (3)

---

- Principales variables d'environnement gérées par Make (il y en a d'autres !)
  - CC : Nom du compilateur C
  - CFLAGS : Options de la première phase de la compilation (comprend « -c »)
  - LD : Nom de l'éditeur de liens
  - LDFLAGS : Options de l'édition de liens
  - CPP : Nom du pré-processeur
  - CPPFLAGS : Options du pré-processeur
  - RM : Nom de la commande d'effaçage

# Variables automatiques de Make (1)

---

- Lors de l'évaluation d'une règle, Make configure un certain nombre de variables utilisables au sein des actions
  - $\$@$  : le nom du fichier cible de la règle
  - $\$*$  : le nom du fichier cible, sans son suffixe
  - $\$<$  : le nom de la première dépendance
  - $\$^$  : la liste de toutes les dépendances
  - $\$?$  : la liste de toutes les dépendances qui sont plus récentes que la cible (utile par exemple pour la mise à jour des fichiers bibliothèques)

# Variables automatiques de Make (2)

makefile

```
CC=gcc
LD=gcc
...
bro1      TAB      : bro1.o trol.o chmol.o
TAB      TAB      $(LD) $(^) -o $(@) -lm

bro1.o    TAB      : bro1.c bro1.h trol.h chmol.h
TAB      TAB      $(CC) -c $(<) -o $(@)

trol.o    TAB      : trol.c trol.h chmol.h
TAB      TAB      $(CC) -c $(<) -o $(@)

chmol.o   TAB      : chmol.c chmol.h
TAB      TAB      $(CC) -c $(<) -o $(@)

libbro1.a TAB      : bro1.o trol.o chmol.o
TAB      TAB      $(AR) r $(@) $(?)
```

# Règles implicites de Make (1)

- Pour ne pas avoir à récrire les mêmes actions pour chaque cible de même type, Make offre un mécanisme de règles implicites
  - Format « classique » commun à tous les Make
  - Format étendu GNU Make
- Une fois les règles implicites déclarées, il suffit juste d'indiquer dans les règles le nom de la cible et de ses dépendances

```
bro1      TAB      : bro1.o trol.o chmol.o
```

```
bro1.o    TAB      : bro1.c bro1.h trol.h chmol.h
```

# Règles implicites de Make (2)

- Les règles implicites classiques sont basées sur les suffixes

```
.c.o:  
TAB      $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

- Si des suffixes « exotiques » sont utilisés, ils doivent être déclarés au moyen de la cible factice `.SUFFIXES`

```
.SUFFIXES: .brol  
...  
.brol.o:  
TAB      $(BROLCC) $@ $<
```

# Règles implicites de Make (3)

- Les règles implicites de GNU Make sont basées sur la reconnaissance de motifs
  - Caractère « % » servant de souche
  - Permet de prendre en compte les chemins

```
OBJDIR = ./obj
```

```
...
```

```
$(OBJDIR)/%.o: %.c
```

```
TAB      $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

- Ce format n'est pas standard mais, GNU Make étant un logiciel libre, on peut demander à l'utilisateur de l'installer s'il n'en dispose pas, et l'inclure dans la distribution logicielle

# Règles implicites de Make (4)

---

- Au lancement, Make dispose de nombreuses règles prédéfinies pour compiler des fichiers « .c », « .f », « .p », « .y », ... en fichiers « .o » ou en exécutable
  - Utilisent les variables  $\$(CC)$ ,  $\$(CFLAGS)$ , ...



# Facteurs externes de qualité

---

- **Validité** : aptitude à réaliser les tâches définies par les spécifications
- **Robustesse** : aptitude à fonctionner dans des conditions non spécifiées
- **Extensibilité** : facilité d'adaptation aux changements de spécifications
- **Réutilisabilité** : aptitude à être réutilisé en tout ou partie pour de nouvelles applications

# Validité

- La définition du cahier des charges est essentielle aux processus de programmation et de test de conformité (recette par le client)
  - Domaine de validité des types manipulés
  - Sémantique des opérateurs s'y appliquant
- La validité d'un logiciel est évaluée par des moyens empiriques ou formels
  - Tests de couverture
  - Preuve de programmes

# Robustesse (1)

- La robustesse est principalement liée à la qualité de la programmation

```
main ()
{
  int   solde[2];      /* Soldes du compte courant et épargne */
  char  nom[64];

  solde[0] = 12;
  solde[1] = 34;
  ...
  scanf ("%s", nom);  /* Risque d'écrasement (si pile descendante) */
  ...
  printf ("Solde compte courant :%d\nSolde compte épargne : %d\n",
          solde[0], solde[1]);
}
```

# Robustesse (2)

- Si elle est prise en compte à un plus haut niveau, c'est qu'elle fait alors partie des spécifications

```
char  nom[64];  
...  
scanf ("%63s", nom); /* Pas de risque d'écrasement */
```

- Plus les spécifications sont précises et définissent le comportement attendu du programme en toute circonstance, plus on rentre dans le cadre de la validité plutôt que de la robustesse

# Robustesse (3)

---

- Pour augmenter la robustesse d'un programme, il faut programmer de façon à empêcher la propagation des bogues d'une section du programme aux suivantes
  - Fermeture de tous les tests par des clauses « else » et « default » (tiers exclus)
  - Vérification de la validité des valeurs d'entrée et de retour des procédures
    - Assertion ou retour de valeur d'erreur
  - Définition de comportements par défaut les moins dommageables possibles

# Robustesse (4)

```
/* Programme de déverrouillage d'un sas : les deux portes A et B
** ne doivent jamais être déverrouillées simultanément.
** Etat : 0 : A et B verrouillées
**        1 : B reste verrouillée
**        2 : A reste verrouillée
*/

void
deverrouille (
int          etat)
{
    if (etat & 1)
        deverrouille_A ();
    if (etat & 2)
        deverrouille_B ();
}          /* Danger si variable d'état trafiquée et positionnée à 3 */
```

# Robustesse (5)

```
void
deverrouille (
int          etat)
{
    switch (etat) {
        case 1 :
            deverrouille_B ();
        case 0 :
            break;
        default :
            /* Par défaut, une porte reste ouverte : */
            deverrouille_A (); /* Évite les enfermements en cas de panne */
            break;           /* Pas nécessaire mais aide-mémoire */
    }
}
```

# Extensibilité (1)

---

- Il est souvent très difficile de prédire de quelle manière un logiciel pourra être étendu
  - Ajout de nouvelles fonctionnalités opérant sur les types de données existants
  - Extension du domaine des types de données
- L'extensibilité d'un programme peut être améliorée au moyen de conventions de codage
  - Définition de types spécifiques à chaque usage
    - Concrétisation d'un type abstrait
  - Valeurs passées par référence



# Extensibilité (2)

```
/* Routines appelées par l'interface d'une calculette
** manipulant des nombres en virgule flottante.
*/

float    valeurLit      (FILE * stream);
void     valeurAffiche  (float val, FILE * stream);
float    valeurAjoute   (float val1, float val2);
float    valeurMultiplie (float val1, float val2);
...

/* Étendre le programme de calculette pour manipuler des
** fractions est très coûteux, car il faut remplacer toutes
** les variables « float » par des structures, ce qui oblige
** à remanier en profondeur le code de l'interface.
*/
```

# Extensibilité (3)

```
/* Version utilisant un type propre.
*/

typedef float Valeur;          /* On peut aussi mettre un « double » */

void valeurLit (Valeur * val, FILE * stream);
void valeurAffiche (Valeur * val, FILE * stream);
void valeurAjoute (Valeur * res, Valeur * val1, Valeur * val2);
void valeurMultiplie (Valeur * res, Valeur * val1, Valeur * val2);
...

void
valeurAffiche (
Valeur * val,
FILE * stream)
{
    fprintf (stream, "%lf", (double) *val); /* Accepte un « double » */
}
```

# Extensibilité (4)

```
/* On peut facilement étendre le fonctionnement de la calculette
** pour manipuler des fractions, en redéfinissant le type Valeur
** et le comportement des fonctions associées. Le code de
** l'interface, lui, ne changera pas.
*/

typedef struct Valeur_ {
    union {
        float  nombre;          /* « union » pour économiser de la place */
        int    numerateur;     /* Valeur à virgule flottante */
    }         fmt;             /* Numérateur de la fraction */
    int       denominateur;    /* Format du numérateur ou du nombre */
} Valeur;

void  valeurLit      (Valeur * val, FILE * stream); /* Pareil ! */
void  valeurAffiche (Valeur * val, FILE * stream);
void  valeurAjoute  (Valeur * res, Valeur * val1, Valeur * val2);
void  valeurMultiplie (Valeur * res, Valeur * val1, Valeur * val2);
...
```

# Extensibilité (5)

```
void
valeurAffiche (
Valeur *      val,
FILE *        stream)
{
    if (val->denominateur == 0)
        fprintf (stream, "%lf", (double) val->fmt.nombre);
    else
        fprintf (stream, "%ld/%ld",          /* Accepte des « long » */
                 (long) val->fmt.numerateur,
                 (long) val->denominateur);
}
```

# Réutilisabilité

- Afin de pouvoir réutiliser facilement un fragment de code, il faut que celui-ci possède plusieurs qualités :
  - Plus grande indépendance possible entre le fragment de code et le reste de son programme
    - Isolation des structures de données manipulées
    - Absence de variables globales
    - Clarté de l'interface entre le fragment et l'extérieur
    - Programmation modulaire
    - Mise à disposition sous forme de bibliothèque
  - Clarté de la documentation

# Modularité

---

- Technique de décomposition visant à réduire la complexité d'un système en le considérant comme un assemblage de sous-systèmes plus simples
- Un module est un sous-système dont le couplage avec les autres est faible par rapport au couplage de ses propres parties
- La capacité à modulariser dépend du degré de couplage entre les sous-systèmes

# Abstraction

---

- Une abstraction fait ressortir les caractéristiques essentielles d'une entité
- Définition des frontières conceptuelles par rapport au point de vue de l'observateur
  - Notion d'interface
- Notion de type abstrait

# Type abstrait (1)

---

- Un type abstrait est composé :
  - D'un comportement, c'est-à-dire d'un ensemble d'opérations applicables sur ce type :
    - Constructeurs
    - Accesseurs
    - Modificateurs d'état
  - D'un ensemble de valeurs possibles
    - Découle du comportement que l'on souhaite
    - Conditionne le choix du type support (et non l'inverse !)



# Type abstrait (2)

- Exemple : le type Point

- Construction

point: Réel X, Réel Y → Point

- Accesseurs

abscisse: Point → Float

ordonnée: Point → Float

- Modificateurs d'état

translater: Point P, Réel DX, Réel DY → Point

pivoter: Point P, Point C, Réel A → Point

- Sémantique de composition

abscisse (translater (p, dx, dy))  $\Leftrightarrow$  abscisse (p) + dx

# Encapsulation (1)

---

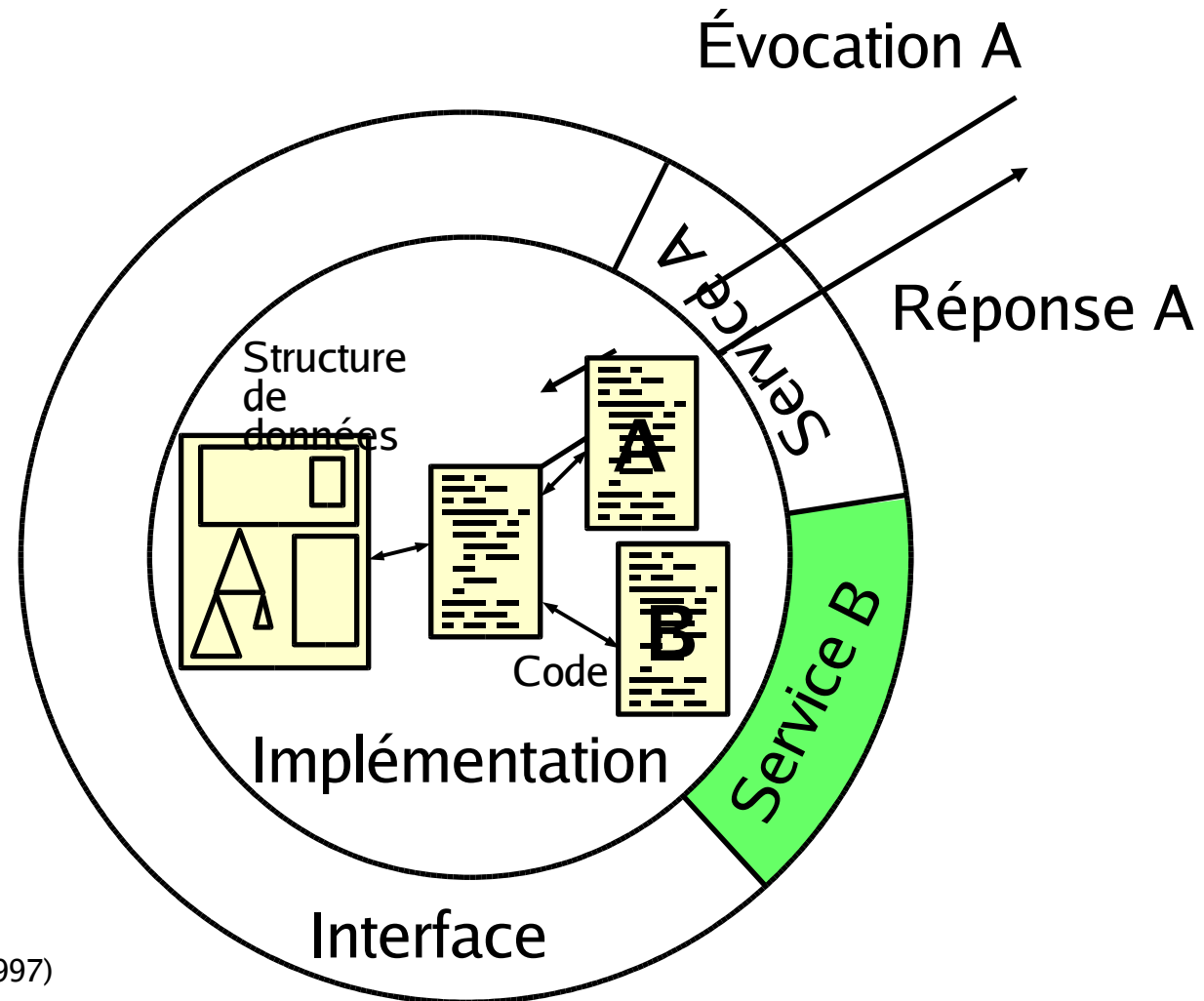
- Technique favorisant la modularité des sous-systèmes, par séparation de l'interface d'un module de son implémentation
- Interface (partie publique)
  - Liste des services offerts
  - Présentation du type abstrait
- Implémentation (partie privée)
  - Réalisation des services offerts
    - Structures de données
    - Algorithmes et implémentation

# Encapsulation (2)

---

- L'encapsulation doit être assurée :
  - Au niveau du langage : mots-clés qualificateurs **public, private, ...**
  - Au niveau de l'exécution : interdiction de manipulations hasardeuses de pointeurs, ...
- Les modules communiquent par évocation du comportement et non par accès mutuels à leurs données
  - Interface de programmation : API
  - Permet la programmation par contrat

# Module



D'après G. Falquet (1997)

# Intérêt de la modularité (1)

---

- Séparation entre les services et leur réalisation
  - Évolution de l'implémentation du fournisseur sans remise en cause des clients
- Compatibilité ascendante de l'interface
  - Les services initialement fournis doivent perdurer

# Intérêt de la modularité (2)

---

- Programmation contractuelle
  - Utilisation de pré- et post-conditions
- Pré-condition : vérification par le module que l'appel est conforme au contrat
  - Évite la propagation des erreurs
- Post-condition : phase de mise au point
  - Vérifie que l'objet est dans un état conforme
    - Vérification de certains axiomes du type abstrait

# Conventions de codage (1)

- Un module consiste en la définition d'un ou plusieurs fichiers « .c » possédant tous le même nom de base, représentatif du service rendu par le module
  - Par exemple pour un module de gestion de matrices, on pourra avoir : `matrice.c`, `matrice_es.c`, `matrice_addition.c`, `matrice_verifie.c`, etc.
- À chaque module doit correspondre un fichier d'en-tête externe donnant l'API du module
  - Par exemple : `matrice.h`

# Conventions de codage (2)

- Lorsque, dans un module, on définit un type, les fonctions manipulant ce type doivent être préfixées par le nom du type suivi du type de la méthode appliquée
  - Exemple : pour le type `Matrice` dérivant de la structure `struct Matrice_`, on aura les fonctions `matriceInit`, `matriceLit`, `matriceEcrit`, `matriceVerifie`, etc.
- Il est préférable que ces noms de méthodes (`Init`, `Lit`, `Ecrit`, `Verifie`, etc.) soient normalisés pour le plus de types possibles



# Tests

- Les tests sont une activité essentielle du développement logiciel
  - Ce n'est pas au client d'effectuer ce travail !
  - Ou alors, il doit en retirer un avantage...
- Il existe différents types de tests, correspondant à des objectifs différents
  - Tests de couverture : vérification du comportement de la plus grande fraction du code possible
  - Tests de non-régression : persistance de la validité du module vis à vis des spécifications de la version précédente

# Tests unitaires (1)

---

- Ont pour but d'attester la validité de chacun des modules qui constituent le projet
- Ils doivent mettre en œuvre l'ensemble des fonctionnalités décrites dans les spécifications, et explorer le fonctionnement du module dans des conditions non spécifiées
  - Tests de couverture et de non-régression de l'ensemble du code, y compris des blocs dédiés à la gestion des erreurs
  - Définition de jeux de tests représentatifs
  - Comparaison à des résultats attendus

# Tests unitaires (2)

---

- Le code dédié aux tests unitaires doit faire partie du code du module
  - Permet la réutilisabilité des tests en même temps que du code du module proprement dit
  - Facilite l'extensibilité des tests et donc la mise en œuvre des tests de non régression

# Tests unitaires (3)

---

- Tout module `module.c` doit disposer d'au moins un fichier `module_main_test.c` contenant la procédure de test unitaire
  - Contient une fonction `main()`
  - Construit par la commande « `make test` »
  - Renvoie un code de succès « `exit(0)` » ou d'échec
    - Permet la conduite automatique des tests au moyen de scripts shell
- Procédure documentée dans le manuel de maintenance

# Tests d'implémentation (1)

---

- Tout module `module.c` gérant un type `Module` doit contenir une méthode `moduleVerifie` destinée à vérifier la cohérence de l'instance de `Module` qui lui est passée en paramètre
  - Utile seulement s'il existe des conditions vérifiables
  - Sert à vérifier la cohérence des objets de type `Module` calculés par les méthodes du module
    - Assertion ou test en mode « debug »
    - Utilisation d'un drapeau « `MODULE_DEBUG` »
  - Mise à la disposition des tiers désireux d'étendre les fonctionnalités du module

# Tests d'implémentation (2)

```
int
matriceFaitQqch (
Matrice *    source,
Matrice *    destination,
int          paramètre)
{
...
#ifdef MATRICE_DEBUG                /* Test de pré-condition */
    if (matriceVerifie (source) != 0) { /* Test avec retour d'erreur */
        ...
        return (1);
    }
#endif /* MATRICE_DEBUG */
...
#ifdef MATRICE_DEBUG                /* Test de post-condition */
    assert (matriceVerifie (destination) == 0); /* Assertion (exit) */
#endif /* MATRICE_DEBUG */

    return (0);                        /* On y est arrivé */
}
```

# Tests d'intégration (1)

---

- Ont pour but d'attester la validité du projet dans son ensemble
  - Mettent en œuvre des jeux de tests de taille réelle
  - Utilisés comme éléments contractuels pour la phase de recette du logiciel

# Tests d'intégration (2)

---

- Tout projet doit disposer d'un ou plusieurs fichiers contenant la procédure de test d'intégration
  - Programmes ou scripts shell
- Procédure documentée dans le manuel de maintenance
  - Nécessaires au bon déroulement des tests de non régression



# Analyse de performance (1)

---

- La performance d'un logiciel est évaluée en fonction des ressources nécessaires à l'obtention du résultat demandé
  - Temps mis
  - Ressources consommées (mémoire centrale, espace disque; bande passante réseau, ...)

# Analyse de performance (2)

---

- Lorsque la performance est insuffisante, il faut déterminer l'origine de cette insuffisance afin d'essayer d'y porter remède
- Les remèdes pourront être :
  - Matériels : ajout ou remplacement de ressources
    - Plus le temps passe, plus les matériels sont puissants !
  - Logiciels : simple recodage des routines critiques ou bien modifications profondes de la structure du logiciel

# Analyse de performance (3)

---

- Lorsque le manque de performance concerne le temps, les problèmes d'accès à la mémoire en sont les causes principales
  - En moyenne, près de la moitié des cycles consommés par les processeurs sont des cycles d'attente de la mémoire
- Il est donc essentiel de concevoir les algorithmes afin de minimiser les attentes mémoire

# Principes de localité

- Dans tout programme, il est possible de mettre en évidence un phénomène de localité des accès mémoire
  - Localité temporelle : plus une zone mémoire a été accédée récemment, plus sa probabilité de ré-accès est élevée
    - Lecture des instructions par le processeur (boucles), ...
  - Localité spatiale : plus une zone mémoire est proche de la dernière zone mémoire accédée, et plus la probabilité qu'elle soit à son tour accédée est importante
    - Parcours de tableaux, ...

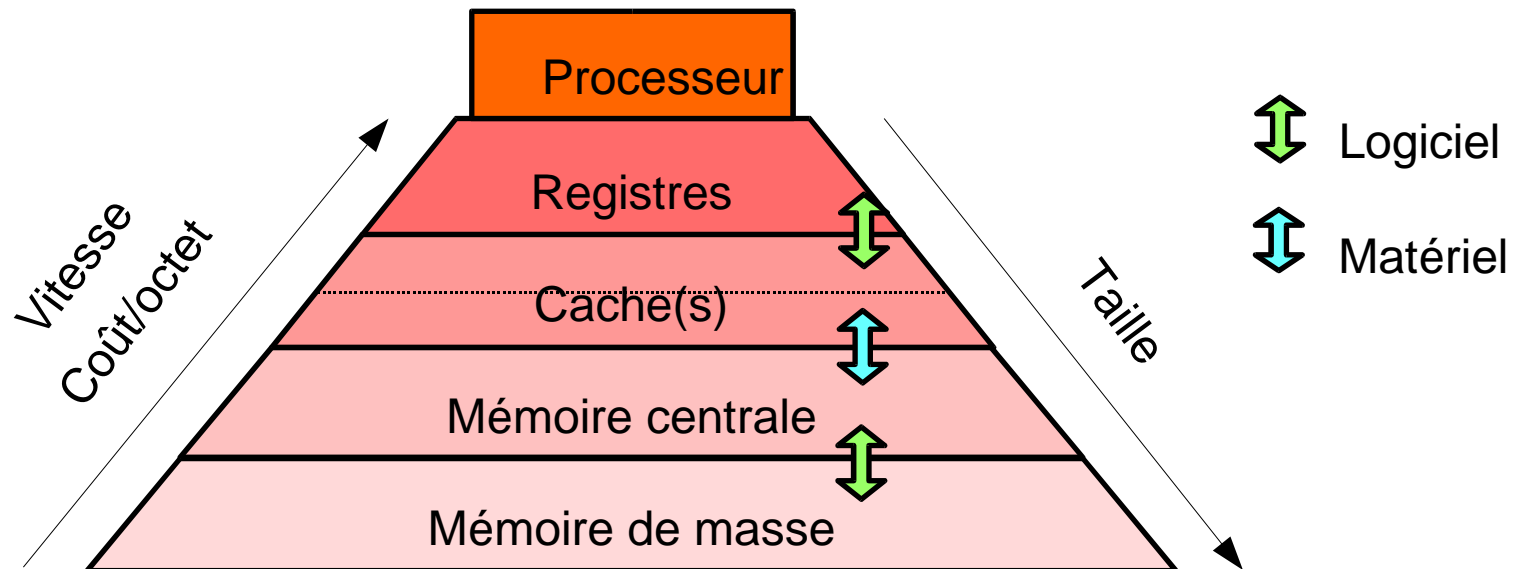
# Hiérarchie mémoire (1)

---

- Pour minimiser l'attente de la mémoire, il faut que les informations les plus fréquemment utilisées soient disponibles le plus rapidement possible
- On s'appuie sur les principes de localité pour mettre en place une hiérarchie de la mémoire
  - Mémoires rapides de faible capacité, proches du processeur
  - Mémoires de grande capacité aux temps d'accès plus longs, situées plus à distance

# Hiérarchie mémoire (2)

- Met en œuvre les principes de localité
- Rend disponibles plus rapidement les données les plus fréquemment utilisées



- Les algorithmes doivent s'appuyer dessus !

# Mémoires cache (1)

---

- Mémoire(s) rapide(s) située(s) entre le processeur et la mémoire centrale
- On trouve habituellement plusieurs niveaux de cache sur les architectures modernes
  - Cache de premier niveau
    - Sur le chip du processeur lui-même
    - Quelques dizaines de Ko seulement
    - Caches dissociés pour les instructions et les données
  - Caches de second/troisième niveau
    - Dans le boîtier du processeur ou à proximité
    - De quelques centaines de Ko à quelques Mo

# Mémoires cache (2)

---

- Les caches ne manipulent pas des octets ou des mots, mais des lignes (« *cache lines* »)
  - Lorsque le processeur demande un mot mémoire, qui n'est pas déjà présent dans le cache, le cache charge toute la ligne contenant le mot voulu à partir de la mémoire (de 32 à 128 octets)
  - Mise en œuvre du principe de localité spatiale
- Lorsque le cache est plein, la ligne la plus ancienne est effacée pour faire de la place à la nouvelle
  - Politique de remplacement de type LRU



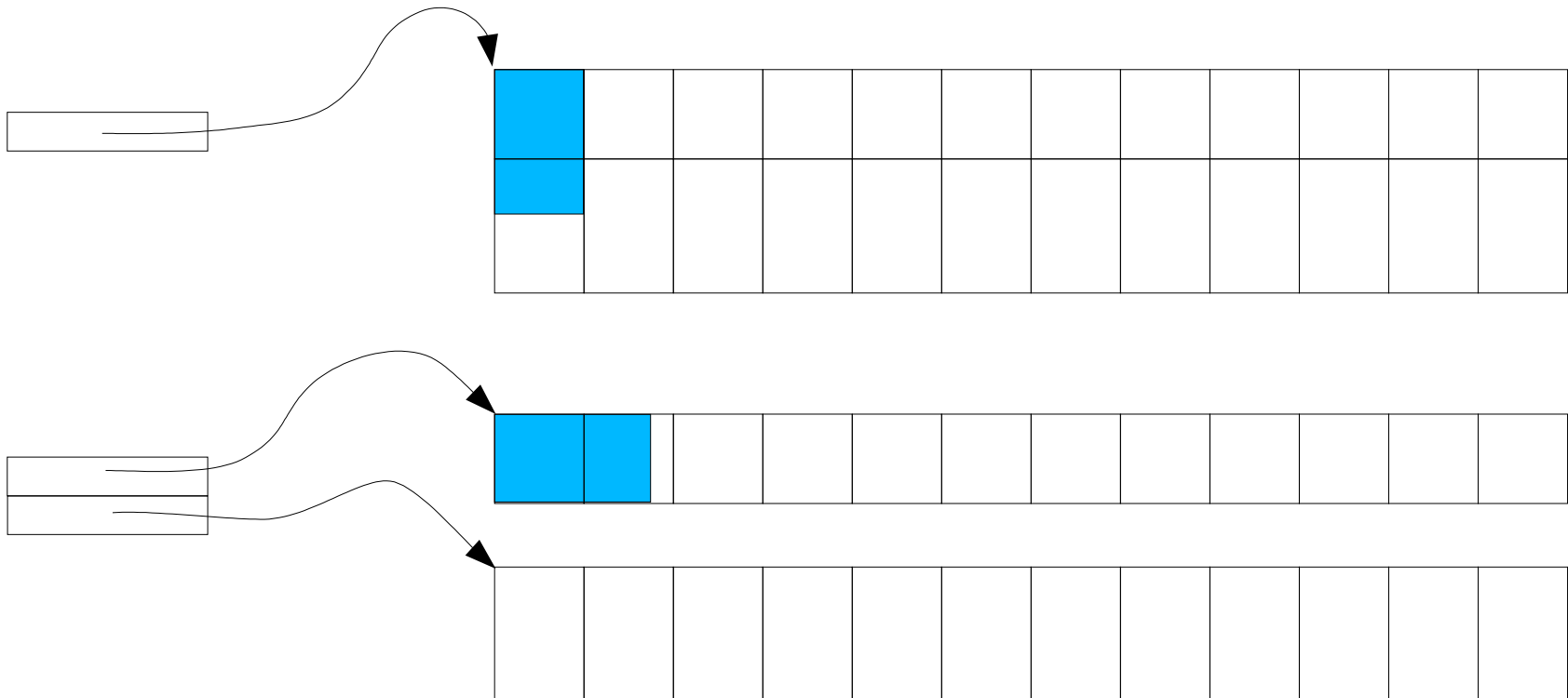
# Programmation « *cache friendly* » (1)

---

- Pour qu'un algorithme soit efficace sur une architecture disposant d'une hiérarchie mémoire, il faut :
  - Effectuer la majorité des parcours de façon continue
    - Croissante ou décroissante
    - Maximise l'utilisation des données des lignes nouvellement chargées
  - Effectuer les parcours en écriture de façon continue
    - Les écritures sont plus chères car on doit répercuter les modifications en mémoire centrale

# Programmation « *cache friendly* » (2)

- Manipuler plusieurs tableaux de sous-structures plutôt qu'un unique tableau de structures de grande taille
  - Les lignes de cache ne stockent que des données utiles



# Mesure de la performance (1)

---

- La mesure de la performance d'un logiciel est complexe, car la prise de mesures doit perturber le moins possible le fonctionnement du système exécutant le logiciel
- Les mesures de performance peuvent être effectuées au niveau :
  - Du processeur
  - Du système d'exploitation
  - Du programme

# Profilage de code

---

- Le profilage de code est l'action d'instrumenter le code source afin d'en obtenir des mesures d'usage lors de l'exécution de jeux de tests
  - Ajout de routines de comptage de passage en différents points du code source
  - Activation de registres matériels du processeur dédiés à cette activité
    - Comptage du nombre de cycles consommés, des accès mémoire, des défauts de cache...

# Time

- Commande Unix classique donnant un résumé des ressources consommées par la commande passée en paramètre
  - Temps CPU consommé en mode utilisateur
  - Temps CPU consommé en mode système
  - Temps réel écoulé depuis le lancement
  - Tailles mémoire utilisées par le code et les données
  - Nombre de défauts de page, etc...

```
% time gcc -O3 bro1.c -o bro1
0.05user 0.02system 0:00.18elapsed 43%CPU (0+0)k 0in+0out
(9major+2815minor)pagefaults 0swaps
```

# Routines de mesure

---

- La routine `clock()`, appartenant à la bibliothèque standard, renvoie le temps CPU écoulé depuis une date d'origine
  - On calcule le temps consommé dans une routine par soustraction entre la valeur à la sortie et la valeur à l'entrée
- La routine `getrusage()` permet d'obtenir, au niveau du processus, les informations affichées par la commande `Time`
  - Temps CPU utilisateur et système, mémoire, ...

# Compteurs matériels (1)

---

- Les processeurs modernes disposent tous de circuits destinés à la mesure de performance
- Compteurs paramétrables d'événements internes au processeur, tels que nombres de cycles consommés, de lectures ou écritures, d'opérations à virgule flottante, de défauts de cache de premier ou deuxième niveau, ...
  - Deux registres compteurs de 40 bits disponibles sur le Pentium II, quatre de 48 bits sur l'Athlon

# Compteurs matériels (2)

---

- Différentes bibliothèques permettent de sélectionner le type d'événement à compter et de lire la valeur des compteurs
  - Dépendantes du processeur et du système d'exploitation
- Tentatives d'offrir une interface unifiée pour plusieurs processeurs et systèmes
  - Papi, de l'Université du Tennessee



# Gprof (1)

---

- Gprof est un outil de profilage, permettant de savoir dans quelles routines un programme passe le plus de temps, et quel est l'arbre d'appel du programme
- Gprof analyse a posteriori les traces générées par l'exécution d'un programme, et produit un relevé statistique du temps passé dans chaque routine

# Gprof (2)

- Pour que l'exécution du programme génère des traces exploitables, il faut compiler avec l'option « `-pg` » de `gcc`
  - Réalise l'édition de liens avec les bibliothèques adaptées
- À la fin de l'exécution, les traces sont collectées dans le fichier « `gmon.out` »

```
% gcc -pg brol.c -o brol
% ./brol
% ls
brol
brol.c
gmon.out
```

# Gprof (3)

- Pour que l'exécution du programme génère des traces exploitables, il faut compiler avec l'option « `-pg` » de `gcc`
  - Réalise l'édition de liens avec les bibliothèques adaptées
- À la fin de l'exécution, les traces sont collectées dans le fichier « `gmon.out` »

```
% gcc -pg bro1.c -o bro1
% ./bro1
% ls
bro1
bro1.c
gmon.out
```

# Gprof (4)

- Le rapport d'exécution est créé par la commande `gprof` proprement dite
  - Rapport d'exécution en format texte
- Classe les fonctions par ordre décroissant de temps consommé dans la fonction et dans les sous-fonctions qu'elle a appelé

```
% gprof bro1 gmon.out > rapport.txt  
% more rapport.txt
```

# Gprof (5)

index	% time	self	children	called	name
...					
[6]	87.7	0.04	35.95	1099+5870	<cycle 2 as a whole> [6]
		0.03	19.27	4773+1647	vgraphSeparateSt [7]
		0.00	0.00	1098	vgraphSeparateMl [42]
-----					
				1647	vgraphSeparateSt [7]
				3674	vgraphSeparateMl2 [9]
		0.04	35.95	1099/1099	hgraphOrderNd [5]
[7]	47.0	0.03	19.27	4773+1647	vgraphSeparateSt [7]
		15.06	1.72	2576/2576	vgraphSeparateFm [8]
		2.10	0.39	1098/1098	vgraphSeparateGg [14]
		0.00	0.00	1099/1099	stratTestEval [41]
		0.00	0.00	1098/1098	vgraphStoreInit [44]
		0.00	0.00	1098/1098	vgraphStoreSave [45]
		0.00	0.00	1098/1098	vgraphStoreExit [43]
		0.00	0.00	714/714	vgraphStoreUpdt [47]
				1098	vgraphSeparateMl [42]
				1647	vgraphSeparateSt [7]

# Bibliothèques (1)

---

- Lorsqu'on fournit un ensemble de fonctions rendant un ensemble cohérent de services, il serait préférable de grouper les fichiers objets associés sous la forme d'un unique fichier
  - Facilite la manipulation et la mise à jour
- C'est le rôle des fichiers de bibliothèque
  - Fichiers servant à archiver des fichiers objet
  - Utilisables par l'éditeur de liens comme banques de fichiers objets qui seront inclus dans l'exécutable s'ils contiennent la définition de symboles non encore définis

# Bibliothèques (2)

- Deux types de bibliothèques
  - Bibliothèques statiques
    - Format en « `lib*.a` » (Unix) ou « `*.lib` » (DOS)
    - Liées à l'exécutable lors de la compilation
      - Augmentent (parfois grandement) la taille des exécutables
      - On n'a plus besoin que de l'exécutable proprement dit
  - Bibliothèques dynamiques
    - Format en « `lib*.so` » (Unix, « *shared object* ») ou « `*.dll` » (Windows, « *dynamic loadable library* »)
    - Liées à l'exécutable lors de l'exécution
      - Permettent la mise à jour indépendante des bibliothèques
      - Problème si pas présentes (variable « `LD_LIBRARY_PATH` »)

# Ar (1)

---

- La commande Ar (« archive ») est un outil de gestion de fichiers d'archives
  - Un fichier archive est un fichier contenant d'autres fichiers
  - Peut archiver tout type de fichier
- Bien qu'il existe d'autres outils similaires et plus efficaces tels que Tar, Ar perdure car son format est reconnu par les éditeurs de liens
  - Définition de fichiers de bibliothèques servant à stocker des fichiers objets



# Ar (2)

- La commande Ar permet :
  - D'ajouter des fichiers à une archive
  - De remplacer des fichiers contenus dans une archive
    - Commande « ru » pour remplacer par les plus récents
  - De supprimer des fichiers contenus dans une archive
  - De modifier l'ordre des fichiers de l'archive
    - L'ordre des fichiers dans l'archive est important !

```
% gcc -c brol.c -o brol.o
% gcc -c brol_io.c -o brol_io.o
% gcc -c brol_check.c -o brol_check.o
% gcc -c brol_compute.c -o brol_compute.o
% ar ruv libbrol.a brol.o brol_io.o brol_check.o brol_compute.o
```

## Ar (3)

- Par défaut, lors de l'édition de liens, les fichiers de bibliothèques sont parcourus linéairement
  - Si deux fichiers contiennent le même symbole, c'est le premier fichier rencontré qui sera inclus
  - Si, lors de l'édition de liens, le fichier objet membre de bibliothèque dont on a besoin a lui-même besoin d'un symbole présent dans un fichier objet déjà vu mais pas inclus, l'édition de liens échouera
- Même problème avec l'ordre dans lequel on demande à l'éditeur de liens de consulter les fichiers bibliothèques

# Ranlib

- La commande Ranlib ajoute dans le fichier d'archive un fichier d'index listant les symboles définis dans les fichiers objets de l'archive
  - Permet d'éviter le problème de visibilité induit par le parcours linéaire des fichiers de l'archive

```
% ranlib libbrol.a
```

```
RANLIB=ranlib  
...  
libbrol.a TAB      : brol.o trol.o chmol.o  
TAB          TAB      $(AR) r $(@) $(?)  
TAB          TAB      $(RANLIB) $(@)
```

makefile

# Nm (1)

- La commande Nm sert à lister les symboles présents dans les fichiers objets ou les bibliothèques
  - Permet de vérifier si une donnée ou un symbole existe bien dans le fichier objet ou de bibliothèque
    - Aide au diagnostic si la compilation plante à l'édition de liens en indiquant un symbole non résolu
    - Permet de déterminer quelle bibliothèque utiliser lors de l'édition de liens
  - Permet de vérifier si l'interface d'une bibliothèque correspond bien à sa documentation
    - Liste des fonctions définies

## Nm (2)

- Les symboles contenus dans les fichiers objets sont identifiés par une lettre code :
  - T : Fonction définie (symbole du segment « text »)
    - Le code de la fonction est disponible dans le fichier
  - t : Fonction statique définie
    - Le symbole n'est visible, et donc la fonction n'est directement appellable, qu'à partir du fichier lui-même
  - U : Fonction ou variable non définie
    - L'éditeur de liens devra trouver une définition de cette fonction dans un autre fichier objet ou il y aura échec

# Nm (3)

- B : Donnée définie dans la zone des données non initialisées (symbole du segment « BSS »)
  - La place est réservée, mais le contenu de la zone n'est pas stocké dans l'exécutable
- D : Donnée définie dans la zone des données initialisées (symbole du segment « data »)
  - Le contenu initial de la zone sera stocké dans l'exécutable
- R : Donnée définie dans la zone des données initialisées et en lecture seule
- b, d, r : Versions statiques des précédents
  - Le symbole n'est pas visible en dehors du fichier objet

# Nm (4)

```
% nm common_integer.o
          U __ctype_b_loc
          U fprintf
0000014a T intAscn
00000000 T intLoad
00000190 T intPerm
0000023f T intRandInit
00000121 T intSave
00000267 T intSort1asc1
00000287 t intSort1asc1_2
000002b0 T intSort2asc1
000002d0 T intSort2asc2
000002f0 t intSort2asc2_2
          U _IO_getc
          U qsort
00000000 b randflag.4042
          U random
          U srandom
          U ungetc
```

# Nm (5)

```
% nm libcommon.a
common.o:
00000000 T clockGet
          U fprintf
          U fwrite
          U getrusage
00000065 T usagePrint
common_error.o:
00000050 T errorPrint
000000b6 T errorPrintW
00000000 T errorProg
00000000 d errorProgName
          U fflush
          U fprintf
          U fputc
          U stderr
          U strncpy
          U vfprintf
common_integer.o:
...
```



# Nm (6)

- Sur certaines implémentations, l'option « -s » permet de voir le contenu du fichier d'index
  - Uniquement les symboles non statiques

```
% nm -s libcommon.a
Archive index:
clockGet in common.o
usagePrint in common.o
intRandInit in common_integer.o
intSort1asc1 in common_integer.o
intSort2asc1 in common_integer.o
intSort2asc2 in common_integer.o
memAllocGroup in common_memory.o
memReallocGroup in common_memory.o
common.o:
00000000 T clockGet
          U fprintf
...
```

# Utilisation des bibliothèques (1)

- L'option « `-L` » du compilateur/éditeur de liens permet d'augmenter la liste des répertoires dans lesquels chercher les fichiers de bibliothèques
- L'option « `-lxxx` » du compilateur permet d'ajouter le fichier bibliothèque « `libxxx.a` » à la liste des fichiers consultés pour y trouver les symboles manquants, dans l'ordre dans lequel ces fichiers sont listés

```
% cd ../main/  
% gcc main.c -o main -L../libbrol/ -lbrol -lm
```

# Utilisation des bibliothèques (2)

---

- Lorsqu'un symbole non encore résolu est défini dans un fichier objet d'une bibliothèque, le fichier objet est ajouté à l'exécutable en construction pour résoudre le symbole
- Les fichiers objets, issus de la compilation de fichiers sources, contiennent le code objet de l'ensemble des fonctions présentes dans le code source
- Il suffit donc que l'une des fonctions seulement soit nécessaire pour que toutes soient incluses

# Utilisation des bibliothèques (3)

---

- Pour optimiser la taille des exécutables, il faut répartir les fonctions d'un module en autant de fichiers sources que de groupes de fonctions utilisables séparément
  - Laisser dans le même fichier les fonctions utilisées conjointement
    - `broInit()` et `broExit()` peuvent rester ensemble
  - Isoler dans des fichiers individuels les fonctions servant peu et/ou de façon individuelle
    - Il est préférable de placer `broLit()`, `broEcrit()` et `broVérifie()` dans des fichiers sources différents