

Nombre variable de paramètres et fichiers

Marie Pelleau

`marie.pelleau@univ-cotedazur.fr`

Basé sur les transparents de Jean-Charles Régin

Conversions implicites

- Elles sont provoquées par des opérateurs arithmétiques, logiques et d'affectation, lorsque les types des opérandes sont différents mais comparables
- Pour les expressions arithmétiques et logiques :
 - les conversions sont effectuées du type le plus faible vers le plus fort (`char` vers `int`, `short` vers `int` ...)
 - ...
- Pour les affectations :
- le résultat de la partie droite est converti dans celui de la partie gauche (`int x = 3 / 4;`)

Conversions de type

- Une conversion est en fait un changement de représentation
 - Un entier en un double
 - Un double en un entier
 - ...
- Attention : le résultat d'une conversion de type peut être indéterminé

Conversions explicites

- Elles sont faites par le transtypage
- La valeur de l'expression ainsi construite est le résultat de la conversion de l'expression dans le type

```
int i;
double d;
enum E {rouge = 1, bleu = 2, vert = 3} couleur;
...
d = 3;
couleur = (enum E)((int) d);
i = (int) couleur;
float f = (float) 3;
int j = (int) 1.2345e2;
```

Ligne de commande

- En fait, la fonction `main` a plusieurs paramètres permettant de faire le lien avec UNIX
 - `int main (int argc, char *argv[])` s
- On utilise par convention `argc` et `argv` (ce ne sont pas des identificateurs réservés)
 - Le paramètre `argc` (entier) indique le nombre de paramètres de la commande (incluant le nom de celle-ci)
 - Le paramètre `argv` (tableau de chaînes de caractères) contient la ligne de commande elle-même

Ligne de commande

- `commande -option toto 1234 tata`
- Dans le programme C
 - `argc = 5`
 - `argv` a 6 éléments significatifs et on peut les représenter comme suit:
 - `argv[0] = "commande"`
 - `argv[1] = "-option"`
 - `argv[2] = "toto"`
 - `argv[3] = "1234"`
 - `argv[4] = "tata"`
 - `argv[5] = NULL`

Ligne de commande

```
#include <stdio.h>

int main (int argc, char *argv[]) {
    int i;
    for (i = 1; i < argc; i++) {
        printf("* %s *\n", argv[i]);
    }

    while (*++argv) {
        printf("= %s =\n", *argv);
    }

    return 0;
}
```

Nombre variable de paramètres

- La liste variable de paramètres est dénotée par ... derrière le dernier paramètre fixe
- Il y a au moins un paramètre fixe
 - `int printf(const char *format, ...);`
- 4 macros sont définies dans le fichier `stdarg.h`
 - Le "type" `va_list` pour déclarer le pointeur se promenant sur la pile d'exécution `va_list ap;`
 - La macro `va_start` initialise le pointeur de façon à ce qu'il pointe après le dernier paramètre nommé `void va_start (va_list ap, last);`
 - La macro `va_arg` retourne la valeur du paramètre en cours, et positionne le pointeur sur le prochain paramètre `type va_arg (va_list ap, type);` Elle a besoin du nom du type pour déterminer le type de la valeur de retour et la taille du pas pour passer au paramètre suivant
 - La macro `va_end` permet de terminer proprement

Nombre variable de paramètres

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

void imp (int nb, ...) {
    int i;
    va_list p;

    va_start (p, nb);
    for (i = 0; i < nb; i++) {
        printf("%d ", va_arg(p, int));
    }
    fputc ('\n', stdout);
    va_end(p);
}

int main (int argc, char *argv[]) {
    imp(10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
    imp(5, 'a', 'b', 'c', 'd', 'e');
    imp(2, 12.3, 4.5);
    return 0;
}
```

Nombre variable de paramètres

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

int max (int premier, ...) {
    /* liste d'entiers positifs terminée par -1 */
    va_list p;
    int M = 0, param = premier;

    va_start(p, premier);
    while (param >= 0) {
        if (param > M) {
            M = param;
        }
        param = va_arg(p, int);
    }
    va_end(p);
    return M;
}

int main (int argc, char *argv[]) {
    printf("%d\n", max(12, 19, 17, 21, 0, 35, 4, -1));
    printf("%d\n", max(12, 19, -1, 17, 21, 0, 35, 4, -1));
    return 0;
}
```

Entrées/Sorties

- Fichier de déclarations `stdio.h`
- Sorties simples sur la sortie standard
 - Écriture d'un caractère


```
char c = '1';
putchar(c);
putchar('\n');
```
 - Écriture d'une chaîne de caractères (avec retour à la ligne)


```
char *s = "Coucou";
puts(s);
puts("Salut");
```
- Entrées simples sur la l'entrée standard
 - Lecture d'un unique caractère


```
char c;
c = getchar();
```
 - Lecture d'une chaîne de caractères (jusqu'à EOF ou `\n`) et `'\0'` est mis à la fin


```
char s[256];
printf("Nom ?");
gets(s);
```

Entrées/Sorties formatées

- Écriture sur la sortie standard


```
printf("décimal = %d, hexa = %x\n", 100, 100);
printf("nb réel = %f, %g\n", 300.25, 300.25);
```
- Lecture sur l'entrée standard


```
int i;
float f;
scanf("%d %f", &i, &f);
```

Entrées/Sorties formatées

- Écriture dans une chaîne de caractères

```
char s[256];
sprintf(s, "%s", "Allô");
printf("%s\n", s);
sprintf(s, "%d", 1234567);
printf("%s\n", s);
```

- Lecture dans une chaîne de caractères `int i`;

```
sscanf("1234", "%d", &i);
printf("%d\n", i);
```

Entrées/Sorties fichiers de caractères

- `FILE *` est un descripteur de fichier
- Trois fichiers standard déclarés dans `stdio.h`
`FILE *stdin, *stdout, *stderr;`
- Déclaration d'un descripteur de fichier
`FILE *fd;`

Entrées/Sorties fichiers de caractères

- Ouverture d'un fichier (liaison entre le nom logique et le nom physique)
`FILE* fopen (const char *filename, const char *mode);`
 - `"r"` pour lecture,
 - `"w"` pour écriture et
 - `"a"` pour ajouter en fin de fichier`"b"` à la fin du mode pour les fichiers binaires
- Fermeture d'un fichier
`int fclose (FILE *stream);`

Entrées/Sorties fichiers de caractères

```
FILE *fl, *fe;
fl = fopen("../ficLec", "r");
/* si "../ficLec" n'existe pas, fl == NULL
   sinon fl contient le descripteur de fichier
   correspondant au fichier physique de nom "../ficLec" */

fe = fopen("ficEcr", "w");
/* si problème fe == NULL sinon fe contient le descripteur
   de fichier correspondant au fichier physique de nom "
   ficEcr" si "ficEcr" existe, effacement du contenu, sinon
   création du fichier vide */
```

Entrées/Sorties fichiers de caractères

Écriture dans un fichier

- un caractère

```
int fputc (int c, FILE *stream);
```

```
int putc (int c, FILE *stream);
```
- une chaîne de caractères

```
int fputs (const char *s, FILE *stream);
```
- un peu plus compliqué

```
int fprintf (FILE *stream, const char *format, ...);
```

Lecture dans un fichier

- un caractère

```
int fgetc (FILE *stream);
```

```
int getc (FILE *stream);
```
- une chaîne de caractères

```
char * fgets (char *s, int n, FILE *stream);
```
- un peu plus compliqué

```
int fscanf (FILE *stream, const char *format, ...);
```

Commande cat

```
#include <stdio.h>

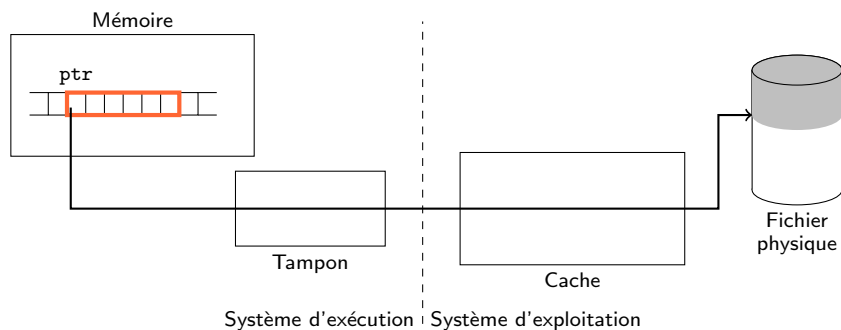
void print (FILE *f) {
    int c;
    while ((c = fgetc(f)) != EOF) {
        fputc(c, stdout);
    }
}

int main (int argc, char *argv[]) {
    FILE *f;
    if (argc == 1) {
        print (stdin);
    } else {
        while (--argc) {
            if ((f = fopen(++argv, "r")) == NULL) {
                fprintf(stderr, "Ouverture impossible de %s\n", *argv);
                return 1;
            } else {
                print(f);
                fclose(f);
            }
        }
    }
    return 0;
}
```

Entrées/Sorties fichiers quelconques

- Ouverture fopen et fermeture fclose
- Écriture

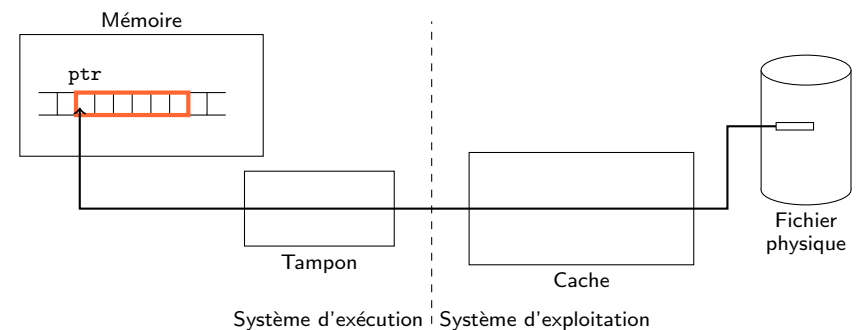
```
size_t fwrite (const void *ptr, size_t size, size_t nitems, FILE *stream);
```



Entrées/Sorties fichiers quelconques

- Ouverture fopen et fermeture fclose
- Lecture

```
size_t fread (void *ptr, size_t size, size_t nitems, FILE *stream);
```



Entrées/Sorties fichiers quelconques

```
FILE *f1 = fopen("entree.bin", "rb");
char c;
fread(&c, sizeof(char), 1, f1);

FILE *fe = fopen("sortie.bin", "wb");
int i[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
fwrite(&i, sizeof(int), 5, fe);
fwrite(&i, sizeof(int), 1, fe);
fwrite(&c, sizeof(int), 1, fe);
fwrite(&i, sizeof(char), 10, fe);
```

Entrées/Sorties fichiers quelconques

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

FILE * ecrire (char *nom, int n) {
    int i;
    FILE *f = fopen(nom, "w");
    for (i = 0; i < n; i++) {
        fwrite(&i, sizeof(int), 1, f);
    }
    fclose(f);
    return f;
}

FILE * lireEtAfficher (char *nom) {
    int i;
    FILE *f = fopen(nom, "r");
    while (fread(&i, sizeof(int), 1, f)) {
        printf("%d ", i);
    }
    fputc('\n', stdout);
    fclose(f);
    return f;
}

int main (int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: %s filename\n", argv[0]);
        exit(2);
    }
    ecrire(argv[1], 20);
    lireEtAfficher(argv[1]);
    return 0;
}
```

Entrées/Sorties positionnement

- Déplacement en octets
`int fseek (FILE *stream, long offset, int whence);`
 SEEK_SET pour le début du fichier, SEEK_CUR pour la position courante et SEEK_END pour la fin du fichier
- Indication de position
`long ftell (FILE *stream);`
- Fin de fichier
`int feof (FILE *stream);`

Entrées/Sorties positionnement

```
FILE *f = fopen(nom, "r");
fseek(f, 0, SEEK_SET); /* f au début du fichier */
fseek(f, 10, SEEK_SET); /* f 10 octets après le début du
    fichier */
fseek(f, -4, SEEK_CUR); /* f 4 octets avant la place
    courante */
fseek(f, -7, SEEK_END); /* f 7 octets avant la fin du
    fichier */

while (!feof(fichier)) {
    char c;
    fscanf(fichier, "%c", &c);
    printf("%c", c);
}
```

Structure des programmes

- Deux types de durée de vie
 - **statique** ou permanente (celle du programme)
 - **automatique** ou dynamique (celle du bloc qui la déclare)
- Trois types de portée
 - bloc (ou fonction)
 - fichier (au sens .c = fichier source)
 - programme

Variables et durée de vie

Définition d'une variable

- Variable réellement créée, mémoire allouée


```
{ /* bloc */
  int v; /* variable automatique */
  /* déclare v et alloue la mémoire nécessaire au
    rangement d'un entier */
}
```
- La définition est **UNIQUE**

Déclaration d'une variable

- Pas de mémoire allouée, juste la nature de la variable est donnée


```
extern int v; /* définit v comme étant une variable
    de type entier */
```
- La déclaration de référence peut être **MULTIPLE**

Classes de variables

La classe de mémorisation est spécifiée par

- **auto** : pas ou plus utilisé
- **extern** : variable globale définie dans une autre fichier
- **register** : variable mise dans un registre
- **static** : variable dans un bloc conservant sa valeur d'un appel à l'autre

Variables internes

- Paramètres
- Variables automatiques (ou locales) : elles sont locales à un bloc
 - naissent à l'appel de la fonction (ou à l'entrée d'un bloc)
 - meurent lorsque la fonction se termine (ou quand on sort du bloc)
- Classe **auto**

Variables externes

- Elles servent à la communication entre fonctions (comme les paramètres) :
 - visibles du point de déclaration jusqu'à la fin du fichier physique
 - définies hors de toute fonction (niveau 0)
 - ont des valeurs permanentes, durant l'exécution du programme
 - initialisées lors de la définition
- Chaque fonction doit la déclarer si elle veut l'utiliser :
 - de façon explicite, grâce à `extern`
 - de façon implicite, par contexte (si la déclaration apparaît avant son utilisation)

Variables registres

- Cela permet d'indiquer au compilateur que la variable va être beaucoup utilisée et que le compilateur va pouvoir la ranger dans un registre (car l'accès à un registre est plus rapide qu'un accès à la mémoire)
- Seules les variables automatiques et les paramètres formels d'une fonction peuvent avoir cette caractéristique
- L'ensemble des types autorisés varie
- Il est impossible de connaître l'adresse d'une variable `register`

Variables statiques

- Les variables internes statiques sont locales à une fonction particulière mais "restent en vie" d'un appel sur l'autre
- Les variables externes statiques sont locales au fichier source (fichier physique), cela permet de ne pas les exporter

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

```
void f (void) {
    static int S = 0;
    int L = 0;
    L++;
    S++;
    printf("L = %d, S = %d\n", L, S);
}
```

```
int main (void) {
    f();
    f();
}
```

Variables volatiles

- Indiquer à l'optimiseur qu'une variable peut changer de valeur même si cela n'apparaît pas explicitement dans le source du programme

```
int main (int argc, char *argv[]) {
    volatile int var = -1;
    int i = 1;
    while (i) {
        i = var /* sans volatile, cette instruction est
                 supprimée par l'optimiseur */
    }
    return 0;
}
```

- Variables susceptibles d'être modifiées indépendamment du déroulement normal du programme : variable modifiée sur réception d'un signal ou d'une interruption, variable implantée à une adresse directement utilisée par la machine

Initialisation des variables

- Si pas d'initialisation explicite, les variables sont initialisées à n'importe quoi
- Si initialisation explicite, les variables `static` et `extern` doivent l'être avec une expression constante car initialisations élaborées à la compilation
- Les initialisations des variables `auto` et `register` sont élaborées à chaque entrée dans le bloc (exécution), donc n'importe quelle expression est acceptée

Domaine d'application

Le domaine d'application d'une déclaration = région de texte du programme C dans laquelle cette déclaration est active

- **Variables globale** : entre son emplacement de déclaration et la fin du fichier physique
- **Paramètre formel** : entre son emplacement de déclaration et la fin du corps de la fonction
- **Variable automatique** : entre son emplacement de déclaration et la fin du bloc
- **Etiquette d'instruction** : l'ensemble du corps de la fonction dans laquelle elle apparaît
- **Macro** : entre `#define` et la fin du fichier physique ou jusqu'à un `#undef` correspondant

Fonctions

- Par défaut, toute fonction est `extern` et elle est supposée rendre un `int` ou `char` s'il n'y a pas eu de déclaration explicite avant
- Une fonction peut être `static` explicitement (limite la portée de la fonction au fichier physique)

Conseils

- Avoir un seul emplacement de définition (fichier source) pour chaque variable externe (omettre `extern` et avoir un initialiseur)
`int cpt = 0;`
- Dans chaque fichier source référençant une variable externe définie dans un autre module, utiliser `extern` et ne pas fournir d'initialiseur
`extern int cpt;`

Modularité

- Forme très simple de “modularité” reposant sur la notion de fichiers (et inclusion de fichiers)
- Rappels sur les déclarations de variables :
 - au début d'un bloc : locale, temporaire
 - au niveau 0 : globale au programme, permanente
 - déclaration **static** : locale (fichier ou fonction), permanente
 - déclaration **extern** : référence à une définition
- Rappel sur le mot-clé **static** : il permet de rendre une variable externe ou une fonction “privée” à un fichier; il permet de déclarer des variables internes permanentes

Règles de modularité

- Les fonctions et variables internes du module sont déclarées locales à ce module au moyen du mot-clé **static**
- Un fichier de déclarations ne contient aucune définition de variable, mais seulement :
 - des définitions de types
 - des déclarations de variables
 - des déclarations de fonctions
- Le fichier de déclarations d'un module est inclus dans le source de ce module, afin de permettre un contrôle des déclarations par le compilateur

Pourquoi la modularité ?

- Un module est une unité de programme, c'est-à-dire un ensemble de fonctions réalisant un même traitement et un ensemble de variables utilisées par ces fonctions
- Le découpage d'un programme en modules est indispensable à
 - la lisibilité
 - la maintenance
 - la ré-utilisation
- Dans un programme en langage C, on définira un module **nom** au moyen du couple de fichiers :
 - **nom.c** : le fichier d'implémentation contenant les définitions de toutes les fonctions et variables du module
 - **nom.h** : le fichier de définitions contenant les déclarations de types, de constantes, de variables et de fonctions (prototypes)

Exemple

generator.h

```
#ifndef _GENERATOR_H
#define _GENERATOR_H

extern void generator_reset (int);
extern int generator_current (void);
extern int generator_next (void);

#endif
```

generator.c

```
#include "generator.h"

static int value = 0;

void generator_reset (int v) {
    value = v;
}

int generator_current (void) {
    return value;
}

int generator_next (void) {
    return value++;
}
```

- Compilation :
gcc -c -Wall -pedantic -ansi generator.c
- Ne crée que le fichier objet et pas d'exécutable

Modularité compilation

main.c

```
#include <stdio.h>
#include "generator.h"

int main (int argc, char *argv[]) {
    printf("%d\n", generator_current());
    generator_reset(4);
    printf("%d\n", generator_next());
    printf("%d\n", generator_current());
    return 0;
}
```

- Compilation :
gcc -Wall -pedantic -ansi generator.c main.c
- Crée l'exécutable a.out

Options de compilations

Répertoires

- -I dir : ajoute dir à la liste des répertoires où chercher les fichiers à inclure
- -L dir : ajoute dir à la liste des répertoires où chercher les bibliothèques -l

Déboguer

- -g : met les informations nécessaires dans l'exécutable pour le débogueur
- -g format : pour un format précis (gdb, coff, xcoff, dwarf)

Options de compilations

Générales (principes généraux mais syntaxe spécifique gcc)

- -c : compile et assemble les fichiers sources et stoppe avant l'édition de liens (fichier .o ou .obj)
- -S : stoppe après la compilation propre, n'assemble donc pas (fichier .s)
- -E : stoppe après le passage du préprocesseur (sur la sortie standard)
- -o file : (output) redirige la sortie sur le fichier file
- -v : (verbose) option intéressante, car liste toutes les commandes appelées par gcc

Options de compilations

Optimisations

- -O1, -O2, -O3 : afin d'optimiser
- -O0 : pour ne pas optimiser

Cible

- -b machine : pour la cross-compilation
- -V version : quelle version utiliser (option utilisée bien sûr si plusieurs versions sont installées)

Avertissements

- -w : pour supprimer les avertissements du compilateur
- -Wall : à utiliser si on veut avoir des programmes "vraiment propres"
- -pedantic -ansi : pour être sûr de "coller" la norme ansi

Outil make

- Vocation de `make` = gérer la construction de logiciels modulaires
- Réaliser des compilations (ou toute autre action) dans un certain ordre (compatible avec des règles de dépendance) : fichier `makefile`
- Dans le cas où `make` réalise des actions autres que la compilation, cet outil est équivalent à un script SHELL, si ce n'est qu'il y a la gestion des règles de dépendance en plus
- `make` ne produit les cibles que si les cibles dont elles dépendent sont plus récentes qu'elles

Exemple

```
math.h
/* Retourne a^b (ou 1.0 si b < 0) */
double puissance (int , int);
```

```
math.c
#include "math.h"
double puissance (int a, int b) {
    double z = 1.0;
    while (b > 0) {
        z *= a;
        b--;
    }
    return z;
}
```

Utilitaire make et makefile

- Existe partout (`make` sur Linux, `nmake` sur windows)
- Exécute une suite d'instructions contenues dans un fichier dit "makefile"
- Souvent le fichier "makefile" s'appelle `Makefile`
- Structure du fichier
 - entree : dépendances
 - action à réaliser
- Attention tabulations **importantes** avant actions

Exemple

```
essai.c
#include <stdio.h>
#include <stdlib.h>
#include "math.h"

int main (int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "usage: %s x y >= 0 (x^y)\n", argv[0]);
        return 1;
    }
    int x = atoi(argv[1]);
    int y = atoi(argv[2]);
    if (y < 0) {
        fprintf(stderr, "usage: %s x y >= 0 (x^y)\n", argv[0]);
        return 2;
    }
    printf("x = %d, y = %d, x^y = %.2f\n", x, y, puissance(x, y));
    return 0;
}
```

Exemple

Makefile

```
CC = gcc
CFLAGS = -Wall -pedantic -ansi
OBJECTS = math.o essai.o

vasy : $(OBJECTS)
    $(CC) -o vasy $(OBJECTS)
    @echo "La compilation est finie"

essai.o : essai.c math.h
    $(CC) -c $(CFLAGS) essai.c

math.o : math.c math.h
    $(CC) -c $(CFLAGS) math.c

clean :
    -rm -f $(OBJECTS) vasy *~

print :
    a2ps math.h math.c essai.c | lpr
```

Structure générale du makefile

- Ce fichier de texte contient une suite d'entrées qui spécifient les dépendances entre les fichiers
- Il est constitué de lignes logiques (une ligne logique pouvant être une ligne physique ou plusieurs lignes physiques séparées par le signe \)
- Les commentaires débutent par le signe # et se terminent à la fin de ligne
- Contenu (ordre recommandé) :
 - définitions de macros
 - règles implicites
 - règles explicites ou entrées

Exemple

Remarque : Si math.o n'est pas "lu" lors de l'édition de liens, il y aura une référence non résolue de la fonction puissance

```
% gcc essai.c
...: In function "main":
...: undefined reference to "puissance":
...: ld returned 1 exit status
```

Règles explicites

```
cible1 ... ciblem : dépendance1 ... dépendancen
    action1
    action2
    ...
    actionp
```

- Traduction : mettre à jour les cibles : cible₁ ... cible_m quand les dépendances dépendance₁ ... dépendance_n sont modifiées en effectuant les opérations action₁, action₂ ... action_p
- La première entrée est la cible principale

makefile : exemple

```

onyva : entiers.o matrices.o pal.o
    @gcc -o onyva entiers.o matrices.o pal.o
    @echo "exécutable $@ créé"

entiers.o : entiers.h entiers.c
    @echo "compilation de $*.c..."
    @gcc -c -Wall -pedantic -ansi entiers.c

matrices.o : matrices.h entiers.h elements.h matrices.c
    @echo "compilation de $*.c..."
    @gcc -c -Wall -pedantic -ansi matrices.c

pal.o : matrices.h pal.c
    @echo "compilation de $*.c..."
    @gcc -c -Wall -pedantic -ansi pal.c

clean :
    -rm -f *.o; rm -f onyva; rm -f *~

print :
    a2ps entiers.h entiers.c | lpr
    a2ps matrices.h matrices.c | lpr
    a2ps elements.h pal.c | lpr

```

Commandes usuelles

Il est bien pratique d'avoir des entrées d'impression, de nettoyage ou d'installation

impression :

```
-lpr *.c *.h
```

menage :

```
@-rm *.o *.out core *~
```

install :

```
mv a.out /usr/bin/copy
chmod a+x /usr/bin/copy
```

makefile

- Si une action s'exécute sans erreur (code de retour nul), make passe à l'action suivante de l'entrée en cours, ou à une autre entrée si l'entrée en cours est à jour
- Si erreur (et - absent), make arrête toute exécution
- Les actions peuvent être précédées des signes suivants :
 - - : si l'action s'exécute avec un code de retour anormal (donc erreur), make continue
 - @ : l'impression de la commande elle-même est supprimée
 - @-, -@ : pour combiner les précédents

Appel de make

`make [-f nom_du_makefile] [options] [nom_des_cibles]`

Options :

- -f : si option manquante, make prendra comme fichier de commandes un des fichiers makefile, Makefile, s.makefile ou s.Makefile (s'il le trouve dans le répertoire courant)
- -d : permet le mode "Debug", c'est-à-dire écrit les informations détaillées sur les fichiers examinés ainsi que leur date
- -n : imprime les commandes qui auraient dû être exécutées pour mettre à jour la cible principale (mais ne les exécute pas)
- -t : permet de mettre à jour les fichiers cible

Appel de make

`make [-f nom_du_makefile] [options] [nom_des_cibles]`

Options :

- `-p` : affiche l'ensemble complet des macros connues par make, ainsi que la liste des suffixes et de leurs règles correspondantes
- `-s` : n'imprime pas les commandes qui s'exécutent; make fait son travail en silence
- `-S` : abandonne le travail sur l'entrée courante en cas d'échec d'une des commandes relatives à cette entrée (L'option opposée est `-k`)
- `nom_des_cibles` : si aucun nom n'est donné, la cible principale sera la première entrée explicite du makefile

Macros

Définition de macros

- Syntaxe
 - `chaîne1 = chaîne2`
 - `chaîne2` est une suite de caractères se terminant au caractère `#` de début de commentaire ou au caractère de fin de ligne (s'il n'est pas précédé du caractère d'échappement `\`)
- Dans la suite du makefile, chaque apparition de `$(chaîne1)` sera remplacée par `chaîne2`
- Exemples
 - `OBJETS = f1.o f2.o f3.o`
 - `SOURCES = f1.h f1.c f2.h f2.c f3.h f3.c`
 - `REPINST = /usr/bin`

Appel de make

Exemple

```
make
make pal.o
make clean
make onyva
make (@print@)
```

Macros

Remplacement d'une sous-chaîne par une autre dans une chaîne

- Syntaxe
 - `$(chaîne:subst1=subst2)`
 - `subst1` est remplacé par `subst2` dans `chaîne`
- Exemples
 - `$(OBJETS:f2.o=)`
 - `$(OBJETS:f2.o=fn.o)`
 - `$(REPINST:bin=local/bin)`

Macros internes

- `$*` : le nom de la cible courante sans suffixe
- `$@` : le nom complet de la cible courante
- `$<` : la première dépendance
- `$^` : la liste complète des dépendances
- `$?` : la liste des dépendances plus récentes que la cible

Exemple

```
CC = gcc
CFLAGS = -Wall -pedantic -ansi
OBJECTS = entiers.o matrices.o pal.o
SOURCES = *.h *.c Makefile ALIRE
CARDIR = /usr/profs/Licence

onyva : $(OBJECTS)
    @$(CC) -o onyva $(OBJECTS)
    @echo "$ (USER), l'exécutable $@ est cr
    éé"

entiers.o : entiers.h entiers.c
    @echo "compilation de $*.c..."
    @$(CC) -c $(CFLAGS) $*.c

matrices.o : matrices.h entiers.h \
    elements.h matrices.c
    @echo "compilation de $*.c..."
    @$(CC) -c $(CFLAGS) $*.c
```

```
pal.o : matrices.h pal.c
    @echo "compilation de $*.c..."
    @$(CC) -c $(CFLAGS) $*.c

clean :
    -rm -f *.o; rm -f onyva; rm -f *~

print :
    a2ps entiers.h entiers.c | lpr
    a2ps matrices.h matrices.c | lpr
    a2ps elements.h pal.c | lpr

copy :
    tar czf $(CARDIR)/binomes.tgz $(
    SOURCES)
    chmod o+r $(CARDIR)/binomes.tgz
```

- Les variables d'environnement sont supposées être des définitions de macros
 - Les variables d'environnement l'emportent sur les macros internes définies par défaut
 - Les macros définies dans le makefile l'emportent sur les variables d'environnement
 - Les macros définies dans une ligne de commande l'emportent sur les macros définies dans le makefile
- L'option `-e` change tout ça de telle façon que les variables d'environnement l'emportent sur les macros définies dans le makefile

Règles implicites

- Elles servent à donner les actions communes aux fichiers se terminant par le même suffixe
 - `.SUFFIXES` : liste de suffixes
 - `.source.cible` : actions
- Dans `.SUFFIXES` : on définit les suffixes standard utilisés par les outils pour identifier des types de fichiers particuliers
- Traduction : À partir de `XX.source`, on produit `XX.cible` grâce à actions
- Pour supprimer les règles implicites par défaut, appeler make avec l'option `-r`, ou écrire `.SUFFIXES :` seulement

Règles implicites

Exemple

Pour tous les fichiers sources C (ayant comme suffixe .c), on appelle le compilateur C avec l'option -c

```
.SUFFIXES : .out .o .h .c
```

```
.c.o :
```

```
gcc -c -Wall -pedantic -ansi $*.c
```

Exemple : règles implicites par défaut

```
CC = gcc
CFLAGS = -Wall -pedantic -ansi
OBJECTS = entiers.o matrices.o pal.o
SOURCES = *.h *.c Makefile ALIRE
CARDIR = /usr/profs/Licence
```

```
onyva : $(OBJECTS)
@$(CC) -o onyva $(OBJECTS)
@echo "$(USER), l'exécutable $@ est cr
ée"
```

```
entiers.o : entiers.h entiers.c
```

```
matrices.o : matrices.h entiers.h \
elements.h matrices.c
```

```
pal.o : matrices.h pal.c
```

```
clean :
-rm -f *.o; rm -f onyva; rm -f *
```

```
print :
a2ps entiers.h entiers.c | lpr
a2ps matrices.h matrices.c | lpr
a2ps elements.h pal.c | lpr
```

```
copy :
tar czf $(CARDIR)/binomes.tgz $(
SOURCES)
chmod o+r $(CARDIR)/binomes.tgz
```

Exemple : changement des règles implicites par défaut

```
CC = gcc
CFLAGS = -Wall -pedantic -ansi
OBJECTS = entiers.o matrices.o pal.o
SOURCES = *.h *.c Makefile ALIRE
CARDIR = /usr/profs/Licence
```

```
.c.o :
@echo "compilation de $*.c..."
@$(CC) -c $(CFLAGS) $*.c
```

```
onyva : $(OBJECTS)
@$(CC) -o onyva $(OBJECTS)
@echo "$(USER), l'exécutable $@ est cr
ée"
```

```
entiers.o : entiers.h entiers.c
```

```
matrices.o : matrices.h entiers.h \
elements.h matrices.c
```

```
pal.o : matrices.h pal.c
```

```
clean :
-rm -f *.o; rm -f onyva; rm -f *
```

```
print :
a2ps entiers.h entiers.c | lpr
a2ps matrices.h matrices.c | lpr
a2ps elements.h pal.c | lpr
```

```
copy :
tar czf $(CARDIR)/binomes.tgz $(
SOURCES)
chmod o+r $(CARDIR)/binomes.tgz
```

Exemple : avec demande à l'utilisateur

```
CC = gcc
CFLAGS = -Wall -pedantic -ansi
OBJECTS = entiers.o matrices.o pal.o
SOURCES = *.h *.c Makefile ALIRE
CARDIR = /usr/profs/Licence
```

```
.c.o :
@echo "avec Debug ?"
@-read -q REP; \
case ${REP} in \
y) @$(CC) -c $(CFLAGS) -g -o $@ $
*.c;; \
n|N) @$(CC) -c $(CFLAGS) -o $@ $*.
c;; \
*) @echo "$*.c non compilé"
esac
```

```
onyva : $(OBJECTS)
@$(CC) -o onyva $(OBJECTS)
@echo "$(USER), l'exécutable $@ est cr
ée"
```

```
entiers.o : entiers.h entiers.c
```

```
matrices.o : matrices.h entiers.h \
elements.h matrices.c
```

```
pal.o : matrices.h pal.c
```

```
clean :
-rm -f *.o; rm -f onyva; rm -f *
```

```
print :
a2ps entiers.h entiers.c | lpr
a2ps matrices.h matrices.c | lpr
a2ps elements.h pal.c | lpr
```

```
copy :
tar czf $(CARDIR)/binomes.tgz $(
SOURCES)
chmod o+r $(CARDIR)/binomes.tgz
```

Le préprocesseur

Fonctions du préprocesseur

Il est appelé avant chaque compilation par le compilateur. Toutes les directives commencent par un `#` en début de ligne

- Inclusion textuelle de fichiers (`#include`)
- Remplacements textuels (`#define`)
 - Définition de constantes
 - Définition de macros
- Compilation conditionnelle (`#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, `#endif`)
- Remarque : Récursivité des définitions

Définition des constantes

- `#define nom expression`
- `#undef nom`
- Dans le fichier concerné, `nom` sera remplacé textuellement par `expression` (sauf dans les chaînes de caractères et les commentaires)

Remarque

- Certains préprocesseurs produisent un message d'avertissement s'il y a re-définition d'une macro, mais remplacent la valeur par la nouvelle
- D'autres ont une pile de définitions
- La norme ansi ne permet pas l'empilement

Définition des constantes

- `#define nom expression`
- `#undef nom`
- Dans le fichier concerné, `nom` sera remplacé textuellement par `expression` (sauf dans les chaînes de caractères et les commentaires)

Exemple

```
#define FALSE 0
#define TRUE 1
#define NULL ((char*) 0)
#define T_BUF 512
#define T_BUFDBLE (2 * T_BUF)
```

Inclusion de fichiers sources

- `#include "nom_du_fichier"`
- `#include <nom_du_fichier>`
- Avec les `<>`, le préprocesseur ne va chercher que dans le ou les répertoires standards
 - `/usr/include`
 - `/include`
- Avec les guillemets, le préprocesseur va chercher à l'endroit spécifié, puis dans le ou les répertoires standards
- On peut passer une option au compilateur pour lui expliquer où chercher (`-I`)

Définition de macros

- `#define nom(par1, ..., parn) expression`
- Dans expression, il est recommandé de parenthéser les par1 afin d'éviter des problèmes de priorité lors du remplacement textuel des paramètres (rien à voir avec le passage des paramètres lors d'un appel de sous-programme)

Exemple

```
#define getchar() getc(stdin)
#define putchar(c) putc(c, stdout)
#define max(a, b) ((a) > (b)) ? (a) : (b)
#define affEnt(a) fprintf(stdout, "%d", a)
#define p2(a) ((a) * (a))
```

Définition de macros

- Macro `concat(a, b)` je veux concaténer a et b : `##` va permettre de concaténer
 - `#define concat(a, b) a##b;`
- ATTENTION a la priorité des opérateurs : on parenthèse
 - `#define max(a, b) (a < b) ? b : a;`
 - `#define max(a, b) ((a) < (b)) ? (b) : (a);`
- ATTENTION les macros font du remplacement de texte
 - `#define max(a, b) ((a) < (b)) ? (b) : (a);`
 - `max(i++, j++)` : mauvais résultat !

Définition de macros

- Une macro est définie à partir du `#define` jusqu'à la fin de ligne
- Pour passer à la ligne, sans utiliser une fin de ligne, on doit utiliser le caractère `\`

Exemple

```
#define PRINT_TAB(tab,n) \
    int i; \
    for (i = 0; i < n; i++) { printf("%d ", tab[i]); } \
    printf("\n");
```

Macros prédéfinies

- `__LINE__` ligne courante dans le fichier source
- `__FILE__` nom du fichier source
- `__DATE__` date de compilation du programme
- `__TIME__` heure de compilation du programme
- `__STDC__` à 1 si implémentation conforme à ansi

Exemple

```
#include <stdio.h>
int main (int argc, char *argv[]) {
    printf("fichier %s", __FILE__);
    printf(" compilé le %s à %s\n", __DATE__, __TIME__);
    printf("Ligne %d\n", __LINE__);
    printf("Ligne %d\n", __LINE__);
}
```

Compilation conditionnelle

```
#if expression_constante
#ifdef expression_constante
#ifndef expression_constante
    liste_instructions_ou_déclarations
#elif expression_constante
#else
    liste_instructions_ou_déclarations
#endif
```

- Il est possible d'emboîter des commandes de compilation conditionnelle
- La compilation conditionnelle permet :
 - la paramétrisation à la compilation des structures de données
 - de gagner de la place en ôtant le code inutile à l'exécution
 - de prendre des décisions à la compilation plutôt qu'à l'exécution

Autres directives

- `#line` fournit un numéro de ligne
 - `#line 42`
 - `#line 99 "toto.c"`
- `defined(nom)` détermine si `nom` est défini comme une macro de préprocesseur
 - `#if defined(TOTO)`
 - `#ifdef TOTO`
- `#error "m"` arrête la compilation avec le message d'erreur `"m"`
- `#warning "m"` produit l'avertissement `"m"` à la compilation

Exemple

```
#include <stdio.h>
#if 0
    /* partie de programme en commentaires */
    ...
#endif

int main (int argc, char *argv[]) {
    printf ("%d\n", __STDC__);

    #if __STDC__
        printf ("ansi\n");
    #else
        printf ("non ansi\n");
    #endif

    return 0;
}
```

Options du compilateur (très utilisées)

- `-Dmacro=defn` définit la macro `macro` avec la chaîne `defn` comme valeur
- `-Umacro` pour ôter la définition de la macro `macro`

Exemple

generator.h

```
#ifndef _GENERATOR_H
#define _GENERATOR_H

extern void generator_reset (int);
extern int generator_current (void);
extern int generator_next (void);

#endif
```

Compilation

```
gcc -c -DSELFEXEC=1 generator.c
```

generator.c

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

static int value = 0;
void generator_reset (int beg) {
    value = beg;
}
int generator_current (void) {
    return value;
}
int generator_next (void) {
    return value++;
}

#ifdef SELFEXEC
int main (int argc, char *argv[]) {
    generator_reset(argv[1] != NULL ? atoi(argv[1]) :
0);
    do {
        printf("%d\n", generator_current());
        while (generator_next() < 10);
        return 0;
    }
#endif
```

Édition de liens

- L'édition de lien permet de constituer un module binaire exécutable à partir de bibliothèques et de fichiers objets compilés séparément, en résolvant les références (externes) qui n'ont pas été résolues lors des passes précédentes du processus de compilation
- Elle extrait des bibliothèques les modules utiles aux fonctions effectivement utilisées
- Chaque objet externe doit avoir une et une seule définition dans l'ensemble des modules à assembler

Assert et NDEBUG

```
#include <assert.h>
void assert(int exp);
```

- La macro assert est utilisé pour tester des erreurs
- Si exp est évalué à 0, alors assert écrit des informations sur la sortie erreur standard (stderr)
- Si la macro NDEBUG (Not Debug) est définie alors assert est ignorée

```
x = t[i] + 2;
```

- Dangereux si i n'est pas dans les bornes (i >= 0 et i < n)
- Problème tester coûte cher
- Solution assert

```
assert(i >= 0 && i < n);
x = t[i] + 2;
```
- Avec NDEBUG définie ne fait rien du tout = coût nul
- Avec NDEBUG non définie, alors on teste les valeurs et si erreur alors message du type "Assertion failed in line XXX"

Bibliothèque (library)

- C'est un fichier "un peu spécial" contenant la version objet d'une fonction ou de plusieurs traitant d'un sujet particulier (ou la version objet d'un "module")
- Sous UNIX, les répertoires standard des bibliothèques sont /lib ou /usr/lib
- La bibliothèque d'archives standard C est en fait le fichier /usr/lib/libc.a, et elle contient entre autres fprintf.o, atoi.o, strcat.o, ...
- La bibliothèque d'archives mathématique est en fait le fichier /usr/lib/libm.a, contenant entre autres e_pow.o, s_sin.o, ...

Utilisation explicite d'une bibliothèque

```
power.c
#include <stdio.h>
#include <math.h>

#define dem(n, v) printf(n " = ? "); \
    fscanf(stdin, "%d", &v)

int main (int argc, char *argv[]) {
    int x, y;

    dem("x", x);
    dem("y", y);
    printf("%d^%d = %.2f\n", x, y, pow(x, y));
    return 0;
}
```

Compilation et édition de liens

- `gcc -Wall -pedantic -ansi -c power.c`
 - `gcc power.o : ERROR`
 - `...: In function "main":`
 - `...: undefined reference to "pow"`
 - `...: ld returned 1`
- `gcc power.o -lm`
- `gcc power.o /usr/lib/libm.a`
- `gcc power.o /usr/lib/libm.so`

Édition de liens statique

- Elle extrait le code de la fonction et le recopie dans le fichier binaire
- Tout est donc contenu dans le fichier binaire, ce qui permet une exécution directe du programme
- Inconvénients :
 - Problème de mémoire : le code de la fonction est chargé en mémoire autant de fois qu'il y a de processus l'utilisant
 - Problème de version : si la bibliothèque change, les applications déjà construites continueront d'utiliser l'ancienne version...

- `gcc power.o -static /usr/lib/libm.a`
taille de a.out = 1656301 octets
- `gcc power.o -static -lm`
taille de a.out = 1656301 octets

Édition de liens dynamique

- Dans ce cas, l'éditeur de liens ne résout plus totalement les références, mais construit une table de symboles non résolus contenant des informations permettant de localiser ultérieurement les définitions manquantes
- Les résolutions sont alors seulement faites lors de l'exécution
 - liaison dynamique immédiate (lors du chargement du programme)
 - liaison dynamique différée (à la première référence d'un objet)
- Inconvénient : ralentissement du chargement du programme

Édition de liens dynamique

- Les bibliothèques de fonctions “reliables”, dynamiquement sont appelées
 - objets partagés (fichiers .so)
 - bibliothèques partagées (fichiers .sl)
 - Dynamic link library (dll) sous Windows
- `gcc power.o /usr/lib/libm.so`
taille de a.out = 14298 octets
- `gcc power.o -lm`
taille de a.out = 14298 octets

Options de compilation

Pour Windows

- `"nom_de_biblio.lib"` : l'éditeur de liens va chercher, dans la liste des répertoires standard des bibliothèques, la bibliothèque `nom_de_biblio`, qui est en fait un fichier nommé `lib`
- `/LIBPATH:"nom_de_chemin"` : ajoute `nom_de_chemin` à la liste des répertoires standard des bibliothèques
- `/MT` : édition de lien statique
- `/MD` : édition de liens dynamique

Options de compilation

- Tous les fichiers dont les noms n'ont pas de suffixe connus sont considérés par gcc comme des fichiers objets (et sont donc reliés par l'éditeur de liens)
- `-lnom_de_biblio` : l'éditeur de liens va chercher, dans la liste des répertoires standard des bibliothèques, la bibliothèque `nom_de_biblio`, qui est en fait un fichier nommé `lib` (`nom_de_biblio.a` ou `.so` et `.sl` si dynamique)
- `-Lnom_de_chemin` : ajoute `nom_de_chemin` à la liste des répertoires standard des bibliothèques
- `-static` : sur les systèmes acceptant l'édition de liens dynamique, elle permet d'éviter l'édition de liens avec des bibliothèques partagées

Construction de bibliothèques

- Il est nécessaire de fournir un (ou des) `.h`, qui servira d'interface avec la bibliothèque :
 - définitions de constantes symboliques et de macros
 - définitions de types
 - déclarations de variables globales externes
 - déclarations de fonctions (en fait, leur prototype)
- Et bien sûr, il y a le (ou les) `.c`, qui définit les variables et les fonctions déclarées dans le (ou les) `.h`, et autres objets privés

Création de bibliothèques

Après la mise au point de la fonction ou du module, il suffit de créer la bibliothèque

- Bibliothèque d'archives : réunir un ensemble d'“objets” en une seule bibliothèque d'archives, en vue de leur reliure statique
 - `ar [options] archive fichiers`
- Options :
 - `-t` : affiche le contenu de l'archive
 - `-r` : remplace ou ajoute le(s) fichier(s) dans l'archive
 - `-q` : ajoute le(s) fichier(s) à la fin de l'archive (sans contrôler leur existence)
 - `-d` : supprime le(s) fichier(s) spécifié(s) de l'archive
 - `-x` : extrait le(s) fichier(s) de l'archive sans que le contenu de l'archive soit modifié

Bibliothèque partagée

Bibliothèque partagée (produite par gcc ou ld)

Exemple

- `gcc -c -shared -o pow.so pow.c`
- `gcc power.o pow.so`

Exemples

- `ar -r lib/libdiv.a generator.o`
- `ar -t lib/libdiv.a generator.o`
- `ar -r lib/libdiv.a pow.o`
- `ar -t lib/libdiv.a generator.o pow.o`