Retour sur le passage de paramètres par variable

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

int init1();
void init2(int *);

int main(){
 int a,b;
 a=init1();
 init2(&b);
 printf("%d, %d",a,b);
 return EXIT_SUCCESS;
}

int init1(){
 return 5;
}

void init2(int *n){
 *n=7;
}
```


Retour sur le passage de paramètres par variable

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

int init1();
void init2(int *);

int main(){
 int a,b;
 a=init1();
 init2(&b);
 printf("%d, %d",a,b);
 return EXIT_SUCCESS;
}

int init1(){
 return 5;
}

void init2(int *n){
 *n=7;
}
```

Type simple

Tableau

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

int *init1(int);
void init2(int **,int);
void rempli(int *,int);

int main(){
 int *ta,*tb,i;
 ta=init1(7);
 init2(&tb,7);
 rempli(tb,7);
 for(i=0;i<7;i++) printf("%d, ",tb[i]); printf("\n");
 for(i=0;i<7;i++) printf("%d, ",tb[i]); printf("\n");
 return EXIT_SUCCESS;
}

int *init1(int n){ return (int *)malloc(n*sizeof(int));}

void rempli(int *tt,int n){
 int i;
 for(i=0;i<n;i++) tt[i]=2*(n-i);}

void init2(int **t,int n){
*t= (int *) malloc (n*sizeof(int));}
```


Retour sur le passage de paramètres par variable

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

int init1();
void init2(int *);

int main(){
 int a,b;
 a=init1();
 init2(&b);
 printf("%d, %d",a,b);
 return EXIT_SUCCESS;
}

int init1(){
 return 5;
}

void init2(int *n){
 *n=7;
}
```

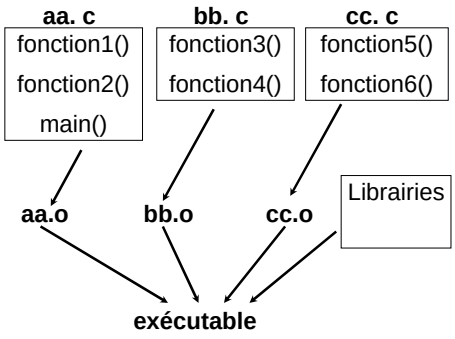
Type simple

Tableau

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

int *init1(int);
void init2(int **,int);
void rempli(int *,int);

int main(){
 int *ta,*tb,i;
 ta=init1(7);
 init2(&tb,7);
 rempli(tb,7);
 for(i=0;i<7;i++) printf("%d, ",tb[i]); printf("\n");
 for(i=0;i<7;i++) printf("%d, ",tb[i]); printf("\n");
 return EXIT_SUCCESS;
}

int *init1(int n){ return (int *)malloc(n*sizeof(int));}

void rempli(int *tt,int n){
 int i;
 for(i=0;i<n;i++) tt[i]=2*(n-i);}

void init2(int **t,int n){
*t= (int *) malloc (n*sizeof(int));}
```


Compilation séparée

Fonctions réparties dans plusieurs fichiers.



Principales options du compilateur C

-g pour deboguer

-O/- O1/- O2 ... pour optimiser

-Wall pour afficher plus d'infos (avec gcc)

(TOUJOURS METTRE CETTE OPTION!)

-Idirectory : chercher les headers dans ce répertoire (dans l'ordre s'il y a plusieurs -I)

Principales options de l'éditeur de liens

-Ldirectory : chercher les librairies dans ce répertoire (dans l'ordre s'il y a plusieurs -L)

-llibrary : chercher les fonctions dans cette librairie dans un des répertoires précédemment spécifiés par –L

attention l'ordre importe !

Répertoires standard sous Unix

/usr/ include et /usr/ local/ includ

/usr/ lib et /usr/ local/ lib

(suivant système et compilateur)

Makefile : Fichier de règles indiquant:

•les fichiers à compiler

•les compilateurs et leurs options

•les librairies, etc.

La commande **make**

•lit le **Makefile**

•appelle automatiquement les outils adéquats

•vérifie les dates et ne recompile **QUE** ce qui doit l'être

On a typiquement 1 Makefile par répertoire

Règle d'or :

•toujours utiliser un Makefile (ou, au choix, un outil plus évolué)

•ne pas s'amuser à retaper les commandes de compilations à la main (toujours fausses dès que l'application devient importante !)

Exemple de Makefile simplifié

```
# le compilateur C et ses options
CC= gcc
CFLAGS= -g -Wall -I/usr/local/qt/include
# les librairies utilisées
LDLIBS = -L/usr/local/qt/lib -lqt
# les fichiers objet de l'application et le nom de exécutable
OBJS= tri.o donnees.o
EXEC= tri
# règle de production de l'application
# ATTENTION: la 2e ligne commence par une tabulation
$(EXEC) : $(OBJS)
        $(CC) $(CFLAGS) -o $@ $(OBJS) $(LDLIBS)
# nettoyage
clean:
        -@$(RM) $(EXEC) $(OBJS)
make : lance la 1ere règle (qui compile tout)
make clean : lance la règle "clean"
```


Compléments sur les Makefiles

Le Makefile précédent est incomplet :

• pas de règles pour créer les .o => fait implicitement

• mais sans tenir compte des .h !!!

Il faut soit:

• mettre les règles à la main

• utiliser configure ou un outil plus évolué

Autres outils utiles

• xemacs, etc.

• gdb, xxgdb, ddd

• grep, nm

• ar

• tar, gtar, gzip, gunzip

• workshop

Compléments sur les Librairies

Librairies statiques :

•extension .a

•simples "archives" de fichiers .o

•le code est inséré dans l'exécutable à la compilation

Librairies dynamiques :

•extension .so, dylib, etc.

•le code n'est PAS inséré dans l'exécutable

•il est chargé DYNAMIQUEMENT à l'exécution

Avantages/ inconvénients

•les programmes sont beaucoup moins gros

•moins de swapping car le binaire est partagé

•mais la librairie doit exister à l'exécution !

•=> éventuels problèmes de licences

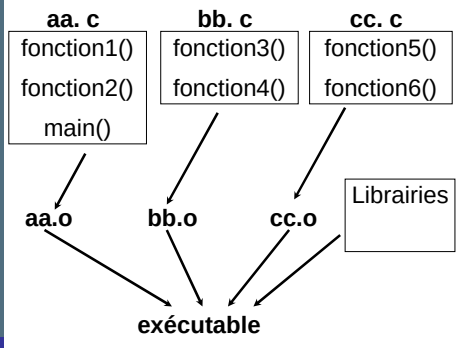
•et il faut trouver la bonne !

•Les libs dyn sont recherchées suivant la variable:

•LD_ LIBRARY_ PATH (ou équivalent)

Retour sur la compilation séparée

Fonctions réparties dans plusieurs fichiers.



Appel d'une fonction définie dans un autre fichier

Le langage C :

•permet d'appeler une fonction non déclarée

•mais ne vérifie pas ses arguments !

•car le compilateur n'examine qu'un seul fichier à la fois !

=> source d'erreurs considérable !!!

Pour imposer une vérification

•la fonction doit être déclarée

•dans le fichier où elle est appelée (avant l'appel)

Déclaration de fonction

extern int ChercheVal(float tab[ ], int taille, float val) ;

Différence avec la définition:

•pas de corps de fonction mais un ;

•le mot clé (optionnel) extern

Première version (pas satisfaisante) ***fichier cherche. c :***

```
int ChercheVal ( float tab[ ], int taille, float val ) {
 int k;
 for (k = 0; k < taille; k++)
   if (tab[ k] == val) return k;
 return -1;
}
```

fichier main. c :

```
#include <stdio. h>
int ChercheVal ( float tab[ ], int taille, float val ) ;
int main() {
 float donnees[ ] = {1., 2., 3., 4., 5., 6., 7., 8., 9., 10.};
 int indice = ChercheVal (donnees, 10, 4.);
 if (indice >= 0)
   printf( "indice = %d , valeur = %f \n", indice, donnees[ indice] );
}
```


Problème

•cohérence de la déclaration et de la définition

Solution

•inclure la même déclaration dans les fichiers

•où la fonction est définie

•et ceux où elle est appelée

•via des fichiers partagés appelés " headers "

Principe

•cherche. c --> définition de *ChercheVal*

•cherche. h --> déclaration de *ChercheVal*

inclure cherche. h

•dans main. c

•et dans cherche. h !

header cherche. h :

```
extern int ChercheVal( float tab[ ], int taille, float val );
```

---

header utile. h :

```
#define CARD( tab) ( sizeof(tab) / sizeof( tab[0]) )
```

---

fichier cherche. c :

```
#include "cherche. h"
int ChercheVal( float tab[ ], int taille, float val ) {
.......
}
```


fichier main. c :

```
#include <stdio. h>
#include "utile. h"
#include "cherche. h"
int main() {
.......
indice = ChercheVal (donnees, CARD( donnees), 4.);
.......
}
```


Headers standards

Déclaration des fonctions des bibliothèques

(printf( ), cos( ), strcpy( ) ...)

• stdio. h : fonctions d'entrées- sorties

• math. h : fonctions mathématiques

• string. h : chaînes de caractères

• etc.

Il y a un header pour chaque librairie

Recherche des headers

• header utilisateur #include " truc. h "

• header standard #include < truc. h >

< > recherche dans

• répertoires standard (/usr/include ...)

• répertoires indiqués en option -I de cc

cc -I.. -I../.. truc. c -o truc

! Attention

• fonction externe non déclarée

• qui retourne autre chose qu'un int

=> PROGRAMME FAUX !!!

Ne PAS oublier d'inclure les headers y compris pour les fonctions des librairies

Exemple: faux sans le #include

```
#include <math. h>
int main(){
double x = cos( 0.5);
printf(" cos( 0.5) = %f \n", x);
}
```

Remarque importante

• l'édition de liens est indépendante de la compilation

• => un programme peut "compiler" (en trouvant les bonnes librairies) même s'il n'utilise pas les bons headers !

Variables

Règles

- **une seule définition**
- **déclaration cohérente avec définition**
- **sinon le programme est faux !**

Comparaison avec les fonctions

- **même principe que pour les fonctions**

sauf que:

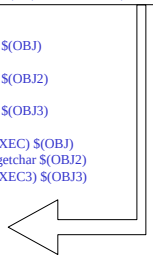
- **une variable doit forcément être définie ou déclarée**
- **le mot clé extern est obligatoire pour déclarer une variable externe (il est optionnel pour les fonctions)**

Conseils

- **éviter les variables globales**
- **toujours les déclarer dans un header**

Exemple de makefile

```
CC = gcc
CFLAGS = -Wall -Werror -ansi -pedantic
CFLAGS += -g

OBJ = b_facto.o facto.o
EXEC= factorielle
OBJ2= putgetchar.o
EXEC2=putgetchar
OBJ3= sort.o
EXEC3=sort

.PHONY: clean realclean
clean:
        $(RM) $(OBJ)
        $(RM) $(OBJ2)
        $(RM) $(OBJ3)
realclean : clean
        $(RM) factorielle.exe  $(EXEC2).exe  sort.exe

factorielle: $(OBJ)
        $(CC) $(CFLAGS) -o $@ $(OBJ)
$(EXEC2): putgetchar.o
        $(CC) $(CFLAGS) -o $@ $(OBJ2)
$(EXEC3) : $(OBJ3)
        $(CC) $(CFLAGS) -o $@ $(OBJ3)
all : $(OBJ) $(OBJ2) $(OBJ3)
        $(CC) $(CFLAGS) -o $(EXEC) $(OBJ)
        $(CC) $(CFLAGS) -o putgetchar $(OBJ2)
        $(CC) $(CFLAGS) -o $(EXEC3) $(OBJ3)
```



Utilisation

- **make**
- **make –f makefile**
- **make factorielle**
- **make -f makefile factorielle**

Produisent le même résultat : **b_facto.o facto.o factorielle**(.exe)

make putgetchar produit **putgetchar.o putgetchar**(.exe)

make sort produit **sort.o sort**(.exe)

make all produit **b_facto.o facto.o factorielle**(.exe) **putgetchar.o putgetchar**(.exe) **sort.o sort**(.exe)

make clean efface **b_facto.o facto.o putgetchar.o sort.o**

Make realclean efface **b_facto.o facto.o putgetchar.o sort.o factorielle**(.exe) **putgetchar**(.exe) **sort**(.exe)