4 Procédures

© J. Morinet Lambert , M. Cadot UHP, L. Pierron
version :03/11/05

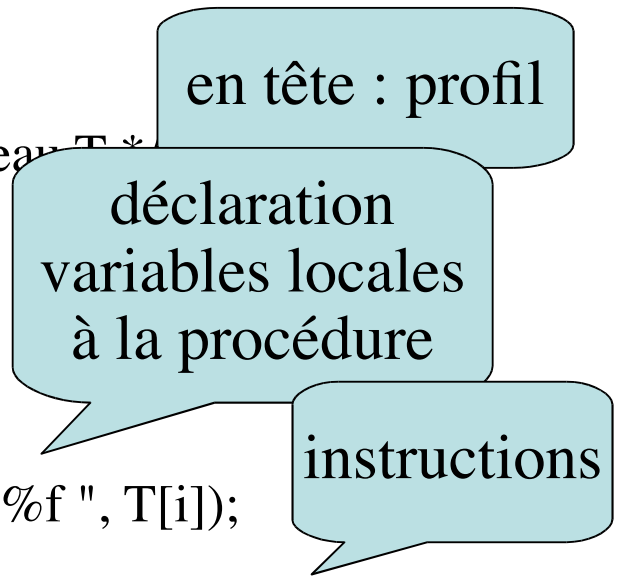
Programmation procédurale

- Comment éviter de réécrire les morceaux de programmes qui permettent
  - d'initialiser un tableau
  - de l'afficher
- Programmation procédurale : définition
  - Une procédure est une liste d'instructions ayant un rôle fonctionnel précis
  - exemples : init_tableau, affiche_tableau
- Objectif : réutiliser
  - Il n'est pas nécessaire de savoir comment on réalise la procédure pour l'utiliser
- Il faut savoir
  - quelles informations donner, quels résultats attendre
- Deux étapes : créer la procédure, utiliser la procédure

Exemple de procédure

- Exemple

```
/* rôle afficher les valeurs du tableau T */
/* données : …*/
/* résultat : … */
void affiche_tableau( tableau T)
{ int i;
    for (i=deb; i<=fin; i++) printf("%f ", T[i]);
}
```



- Rôle du profil****
  - interface d'utilisation, seules informations à connaître pour utiliser la procédure : les **paramètres**
  - équivalent en μprog à un composant avec pattes pour E/S

Structure d'une procédure

- En tête
  commentaires sur le rôle
  Sert à créer la documentation

- Déclarations
  variables locales nécessaires

- Corps
  comment faire pour obtenir
  les résultats ?

```
/* profil */
/* role :              */
/* données :           */
/* résultats :         */
type nom( type parametre_formel)
{




}
```

Types de procédures

- **fonction** : rend un résultat de type simple (int, float)
  - exemple :

```
int min( int a,int b)
{ if  (a<b)  return a; else return b;
}
```

- **procédure** : pas de résultat explicite on passe l'adresse du résultat
  - exemples connus : printf("%d \n", i); scanf("%d",&i);
  - exemple :

```
void mini(int a, int b, int *res)
{if (a<b) *res=a; else *res=b;
}
```

Étapes de définitions d'une procédure

- Ce qu'il y a à faire
  - Permet de définir l'en tête et l'interface d'utilisation
  - Un contrat entre **l'utilisateur** et le **programmeur**
  - Partie visible par l'**utilisateur**
- Comment le faire
  - Permet de définir le corps de la procédure : les instructions
  - Réservé au **programmeur**
- Tester si c'est correct
  - Utiliser dans un programme principal avec de bons jeux d'essais
  - C'est **l'appel**
- (RE)Utiliser ailleurs…
  - On est sûr que cela fonctionne… même si on ne sait pas comment : cf. **boîte noire**

Place de la procédure dans le programme

- Placer la définition (déclaration) de la procédure
  - dans la partie déclaration d'un programme
  - avec les paramètres **formels** (représentent les informations à traiter)
  - requis pour définir la procédure
- Placer les appels à la procédure
  - dans le programme principal ou dans tout autre programme souhaitant l'utiliser
  - avec les paramètres **effectifs** : requis pour exécuter la procédure
  - exemple :

```
tableau T1, T2;                                  /* déclaration de variables */
affiche_tableau(T1); affiche_tableau(T2); /* appels */
T1 et T2 paramètres effectifs
```

Les paramètres des procédures

- **Les** paramètres **formels** (pour définir)
- variables connues pour **définir** la procédure
    - figurent dans le profil (la règle d'utilisation)
    - sont utilisés dans le corps de la procédure
    - exemple **T** pour la procédure **affiche**
- **Les** paramètres **effectifs** (pour utiliser)
  - ceux du programme appelant (principal) nécessaires pour **l'exécution**
  - exemple : **P1 et P2**
- La correspondance **entre** paramètres formels et effectifs **se fait par position**
  - même **nombre** de paramètres, mêmes **types**
  - même **ordre** dans la liste des paramètres
    - du profil (déclaration) et de l'appel (utilisation)

Modes de passage des paramètres

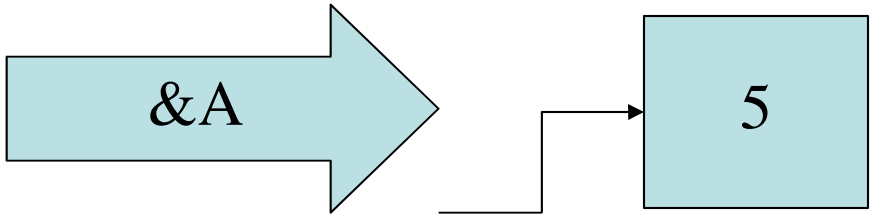
- par valeur
  - on **recopie** la **valeur** du paramètre effectif dans le paramètre formel
  - similitude avec une constante : paramètres données
  - printf("%d", i);
- par adresse
  - on utilise une **indirection**
  - le paramètre formel est de type "***pointeur sur le type de donnée***"
  - on **recopie l'adresse** du paramètre effectif dans le paramètre formel
  - permet de **changer** la valeur du paramètre : résultat
  - exemple : scanf("%d", &i);
- Comment passer les paramètres données et résultats?
  - heuristique : donnée : par valeur, résultat : par adresse
- Certains objets complexes passés par adresse : tableaux, fichiers

Type pointeur sur…* (rappel)

- Type **pointeur** : une adresse
- Adresse de quoi ? :
  - pointeur sur entier
  - pointeur sur réel etc.
- Symbolisation : *
- Opérateur : &

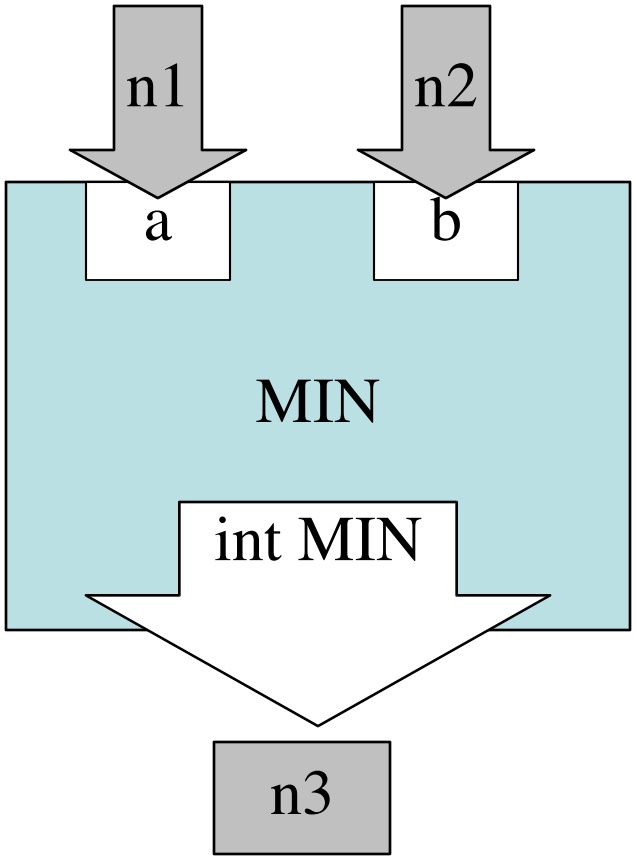
variable A
valeur de A : 5
adresse de A : &A

variable p



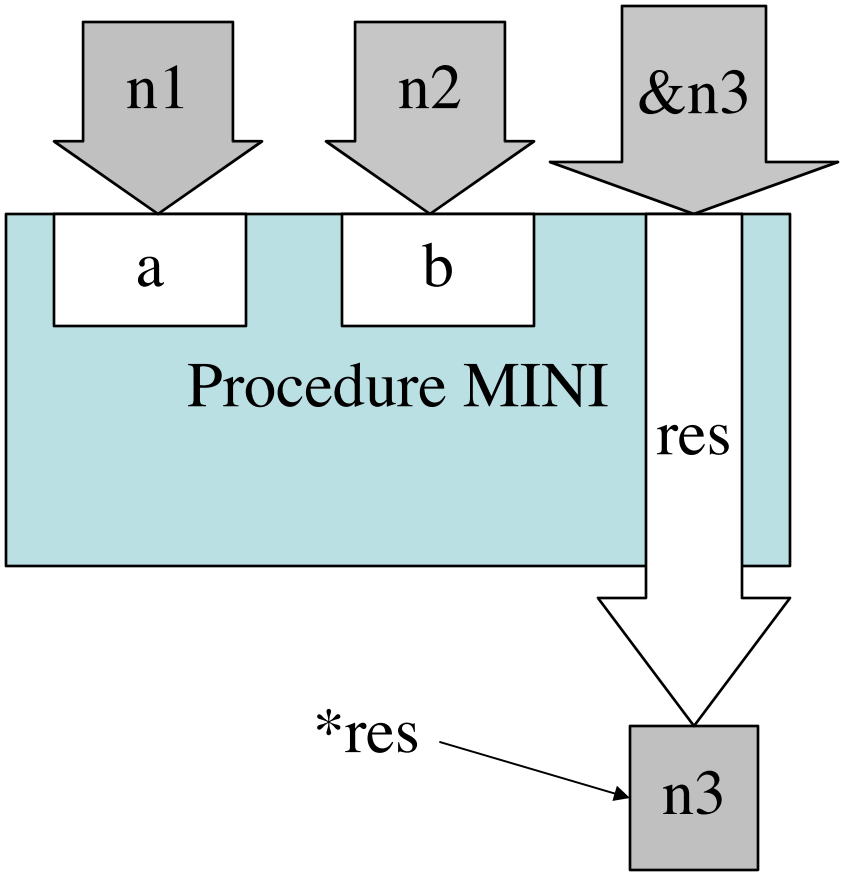

```
int A =5;
int *p;
p= &A;  /* p  pointeur : on lui affecte l'adresse de a */
*p= 7;   /*  *p variable pointée cad A prend la valeur 7 */
```

Passage par valeur : on **recopie la valeur** des paramètres effectifs dans les paramètres formels




```
int MIN ( int a, int b)
{ if (a<b) return a; else return b;
}
int main(void)
{
        int n1=45, n2=32, n3 ;
        n3= MIN(n1,n2);
        printf("min %i\n",n3);
}
```

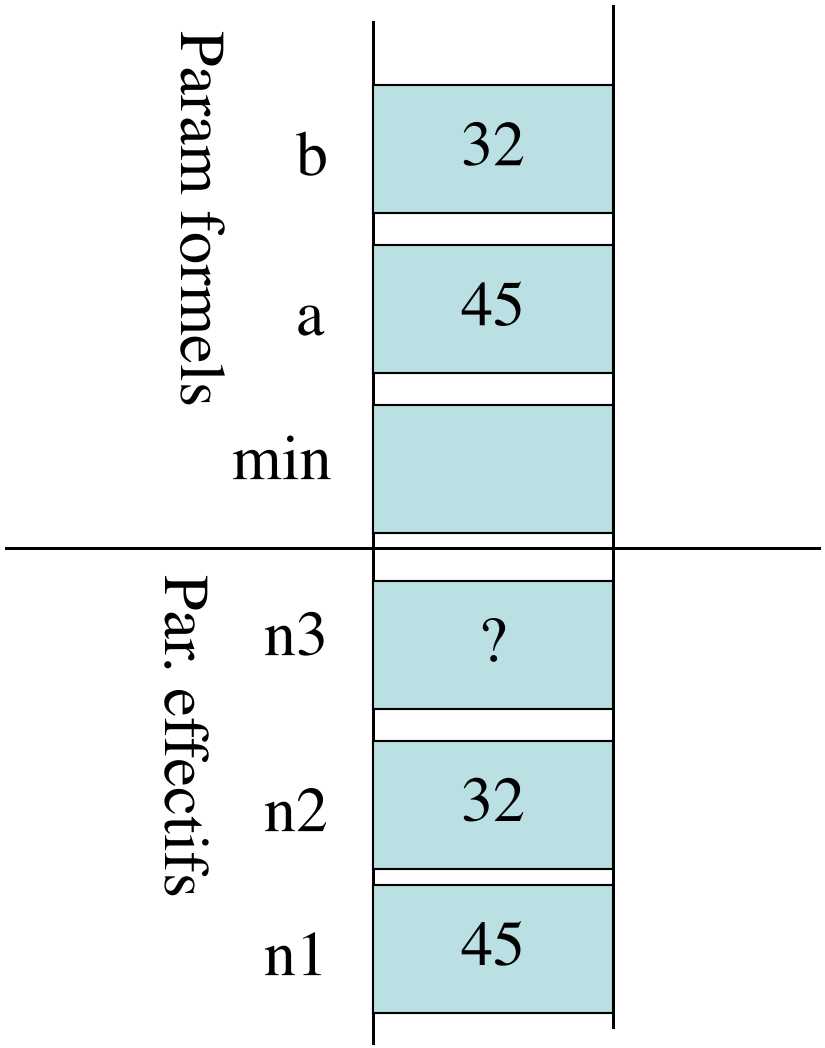
Passage par adresse : on recopie la valeur de l'**adresse** à une variable de type "pointeur sur"




```
void MINI ( int a, int b, int *res)
{
if (a<b) *res=a; else *res=b;
}
int main(void)
{
        int n1=45, n2=32, n3 ;
        MINI(n1, n2, &n3);
        printf("mini %i\n",n3);
}
```

Pile à l'exécution : passage par valeur

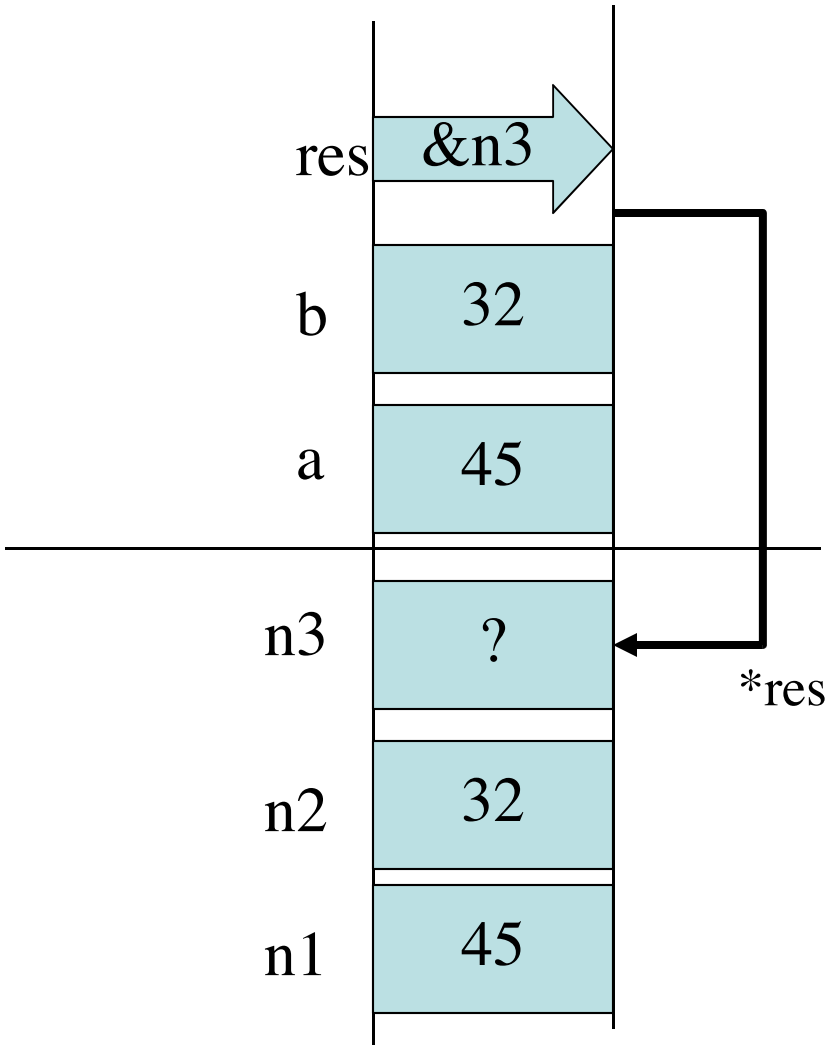
```
int MIN ( int a, int b)
{ if (a<b) return a; else return b;
}
int main(void)
{
    int n1=45, n2=32, n3 ;
    n3= MIN(n1,n2);
    printf("min %i\n",n3);
}
```



Pile à l'exécution : passage par adresse

```
void MINI ( int a, int b, int *res)
{
if (a<b) *res=a; else *res=b;
}
int main(void)
{
    int n1=45, n2=32, n3 ;
    MINI(n1, n2, &n3);
    printf("mini %i\n",n3);
}
```

res de type : pointeur sur int
on passe l'adresse de n3



Portée des variables dans un fichier programme

- Variables et déclarations **globales**
  - dans la partie déclaration du programme
  - réserver pour les types communs, les constantes
  - et quelques très rares cas particuliers (grands tableaux)
- Variables **locales** à une procédure
  - dans la procédure
  - ou le programme principal
- On peut utiliser des variables ou des déclarations issues d'un autre programme
  - statut external , déclarée **extern** ***type nom_var*** **;**
  - cf. cours sur la compilation séparée

Un point de vocabulaire

- Ce sont les **paramètres** qui permettent de **communiquer** les informations entre la procédure et le programme utilisateur
- Une procédure qui **rend** un ou plusieurs résultats : il calcule, transforme et place le résultat dans des variables, *il n'affiche pas*
- Une procédure **reçoit** des données depuis un programme utilisateur : *il ne fait pas de saisie*.
- ***SAUF les procédures spéciales*** de **SAISIE et AFFICHAGE**
- Les **saisies** (depuis clavier/souris/capteur) et les **affichages** (écran /imprimante/commande) se font à l'aide de procédures spéciales où on traite
  - les difficultés de transcodage et la validité des données
  - ou la gestion de l'espace d'affichage.

Effets de bord

- Utilisation de variables globales à l'intérieur de procédures

- faux avantage : on ne déclare pas plusieurs fois un index
- exemple : int i ; /* index dans un tableau */
- inconvénients : la procédure n'est plus réutilisable telle quelle
  - il faut penser à déclarer l'index
  - il faut connaître son nom
  - donc il faut connaître comment est faite la procédure :-((
- source d'erreurs à l'exécution : les plus difficiles à retrouver et à corriger

Inclusion de fichier : quel est le problème ?

```
/*prog1.c */
#define debut 0
#define fin 20
#define taille 21
typedef int TAB[taille];

void proc(...)
{TAB T1,T2;
...
}
```

```
/*prog2.c */
#define debut 0
#define fin 20
#define taille 21
typedef int TAB[taille];

void autreproc(...)
{TAB T,TT;
...
}
```

prog1.c et prog2.c utilisent les mêmes types de données
comment maintenir si on doit changer la taille par exemple ?

Inclusion : déclarations en C

un fichier commun
fichier header : myinclude**.h**
pour les déclarations

```
/*myinclude.h */
#define debut 0
#define fin 20
#define taille 21
typedef int TAB[taille];
```

```
/*prog1.c */
#include "myinclude.h"
void proc(...)
{TAB T1,T2;
...
}
```

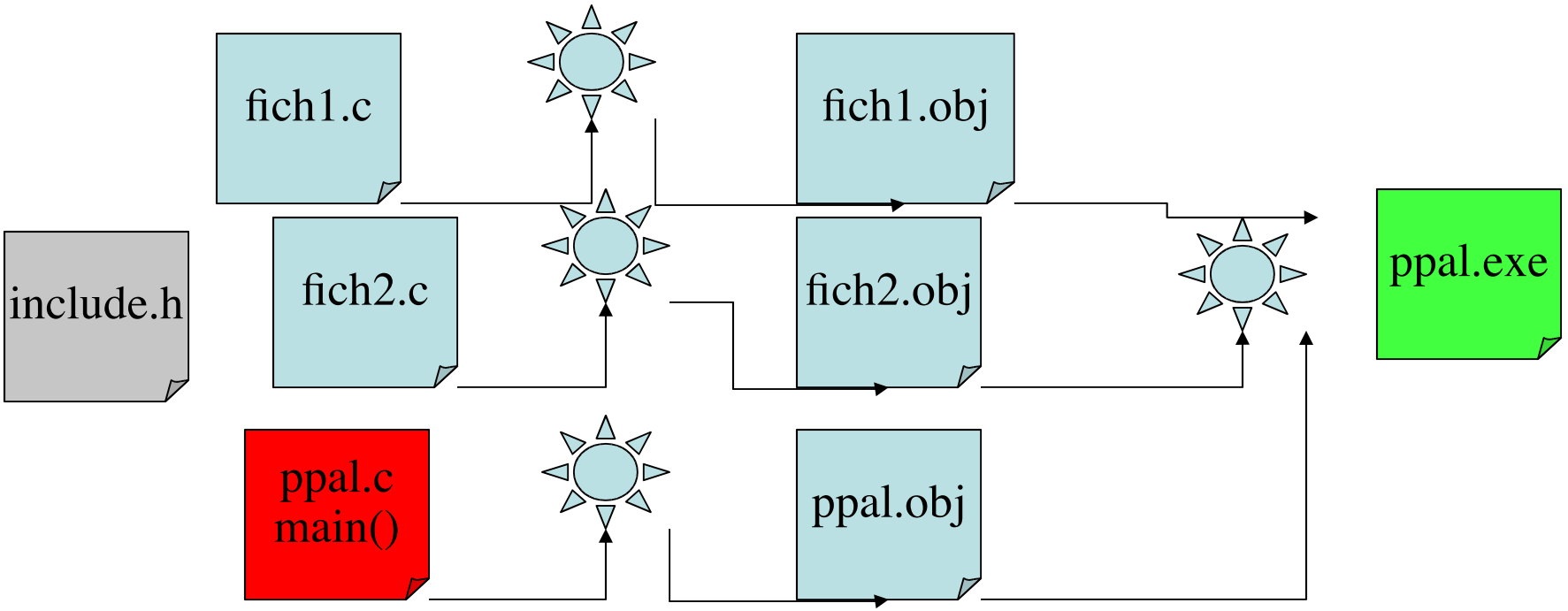
```
/*prog2.c */
#include "myinclude.h"

void autreproc(...)
{TAB T,TT;
...
}
```

attention à pouvoir "atteindre" le fichier inclus depuis les programmes

Compilation séparée : objectif

- Pour de grosses applications
- éviter de recompiler les parties de programmes "correctes"
- Méthode : répartir les procédures dans plusieurs fichiers



Compilation séparée : un "projet"

```c
/* prog1.c */
int MIN ( int a, int b)
{ if (a<b) return a;
          else return b;
}
int truc=5;
```

```c
/* prog2.c */
extern int MIN ( int a, int b);
extern int truc;
int main(void)
{
int n1=45, n2=32, n3 ;
   n3= MIN(n1,n2);
   printf("%i %i\n",n3,truc);
}
```

on déclare les procédures dans un programme

on compile le programme

gcc prog1.c **-c**

- **prog1.o** n'est pas exécutable
- (vérifier les droits -rw-r--r--)

on utilise les procédures externes

on compile le programme

gcc prog2.c **-o prog2 prog1.o**

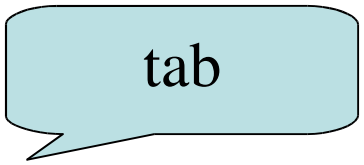
- **prog2** est exécutable (main)

Un utilitaire : makefile

- Un fichier particulier unique
  - Contient toutes les directives de compilation
  - Les fichiers à compiler ou inclure (avec les chemins d'accès)
  - Les références aux programmes précompilés pour l'édition de liens
- Fonctionnement
  - teste les dates de programmes sources et des programmes objets
  - si *date du source* postérieure à *date de l'objet* c'est qu'il a été modifié donc recompiler et refaire l'édition de lien
- Avantage
  - gain de temps de compilation pour tout ce qui n'est pas modifié
- Appel : make

Exemple de makefile

```
# fichier pour compiler en programmation séparée
# fichier prog2 exécutable dépend de prog1.o et prog2.o
prog2 : prog2.o prog1.o
     gcc prog1.o prog2.o  -o prog2
# fichiers non exécutables  et prog1.c avec inclusion
prog1.o :  prog1.c myinclude.h
        gcc  -c  prog1.c
prog2.o : prog2.c myinclude.h
     gcc -c prog2.c
# on peut ajouter les fichiers requis pour l'édition de lien
# dans la commande de compilation -lm -link nomdelibrairierequise
```



à vérifier sous linux et gcc dont la syntaxe diffère de cc

Gestion des variables en C

- complément de déclaration définissant la manière dont on va gérer les informations en mémoire
- auto (automatic)
  - variables internes à un bloc, détruites quand on le quitte (cf. pile exécution)
- static
  - variables internes à un bloc, conservent la valeur jusqu'au retour dans le bloc, réservation permanente de place en mémoire
- extern (external)
  - existent et conservent leur valeur pendant toute l'exécution d'un programme même si elle sont dans des blocs compilés séparément
- register
  - semblables à automatic mais mémorisée dans les registres,

Portée des variables

```
static int a;
extern int b;

int proc(int param)
{int c;
/*instructions */
/* utilise param a et b */
return c;
}
```

```
int b;
int compose(int para)
{int e ;
/* utilise para, e et b */
}
int main (void)
{int d;
/* instructions */
/* utilise compose, b et d */
}
```

globales: a,b
locales à proc : param, c
inutilisables par autrui : a

globales : b
locales à compose : para, e
locale à main : d