UE Socle commun disciplinaire
- Langage C -

Katia Jaffrès-Runser, Xavier Crégut, Yamine Ait-Ameur

ENSEEIHT, Dept. SN,
kjr@n7.fr

1SN

Année 2017-2018

Plan du cours I

CM 1 **Programmation impérative en C**
- Introduction
- Quelques mots sur la compilation
- Types, Constantes, Variables, Portée des variables
- Structures de contrôle
- Enumération, Enregistrement, Tableaux
- Chaines de caractères
- Pointeurs
- Sous-programmes
- Gestion des entrées / sorties

TD 1 **Spécificités du C**

TP 1 **Mise en oeuvre - TP noté**

Plan du cours I

CM 2 **Modules, Make et Allocation dynamique**
Allocation dynamique
Les modules en C
Make
Misc : arguments ligne de commande, fichiers, pointeurs de fonctions

TD 2 **Make, allocation dynamiques et gestion de la mémoire**

TP 2 **Mise en oeuvre - TP noté**

Introduction au langage C

Qu'est-ce que le langage C ?
Bibliographie
Un premier exemple

Un peu d'histoire

- Défini en 1972 par Dennis Ritchie et Ken Thompson, aux Bell labs, NJ, USA.
- Pour développer le système d'exploitation multi-tâches UNIX.
- Fait suite au langage B
- Régulièrement dépoussiéré
  - Version originale du C K&R (1970),
  - Mises à jour C90 (1989-90), C99 (1999), C11 (2011)
- Forte source d'inspiration pour de nombreux langages de programmation ultérieurs. Un des plus récents étant **Go**[1] de Google.

1. `https://golang.org/`

Introduction

Qu'est-ce que le langage C ?

- un langage impératif procédural
- un langage compilé (vérification statique).
- un langage qui fait confiance au programmeur (*"trust the programmer"*) :
  $\rightarrow$ aucune vérification n'est faite à l'exécution du programme (gestion des indices dans un tableau, etc.)
- un langage soucieux de performances : gestion fine de la mémoire

Pourquoi le C ?

Un langage universel :

- Connu par tous (?) les programmeurs
- Présent sur toutes les plateformes
- Enorme base installée
- Coeur de nombreux systèmes Unix, Linux, embarqués ..

Avertissement sur le contenu de ce cours

- **Ce cours n'est pas un manuel de référence du langage C** :
  Il fait suite aux cours de programmation impérative !
  Il se base donc sur les notions d'algorithmique vues dans ces cours.
  $\rightarrow$ Il se focalise donc sur **les spécificités du C**.

- par manque de place (et pour ne pas avoir à écrire trop petit), certains codes ne respectent pas les règles de bonne programmation (identifiants aux noms significatifs, commentaires, etc. . .)

- le C permet d'écrire des choses illisibles
  - Il existe un concours du code le plus obscur
    `http://www.ioccc.org/` [2]

2. Exemples IOCCC 2014 en suivant

```
#include<stdio.h>/*IOCCC2014           2014    2014    IOCCC2014IOCCC2014IOCCC201*/
char*s="\"nsu{AntynCnuq}Bnu{          sEot    ln>b    )+c^g+@`+]_osk{;j@bkg&c<'^o\
r'Q]                    bh'l  vQ^k                    g&c:                  %n|\
N]_o                    ptj9  lwg+                    )d:b                  kg$\
c8#^    #g+)d8`a%g+)    d8`_            g&;bh'oq      Q^)g    +&kcNlyMc+)d  8`a\
`g+@    u)|d8ak=bl}(    Q^og            (O{MK6lM      L(rR    pOpM866OsRlm  N(q\
Q]##    OsR#M_(lQoOa    N9$m  vOwwRor~        }(cN    mkM:    q(Q]%_(uU]}_  {8b\
%mRh    #S^`#mOaaD%/    RI4$  4SNH$%N4        RlMG    /2MJ    24O3NF(tQ?1l  N*N\
+Q]l    mq9l8b$^#h$#    .d,d  xv#mSOPm    R8`/        lM;b    h&^/lM:k8b%&  Q^-\
h%c9    .#,/&N$McPc%    -d8,  $c8``7:b    %^h%        :79b    $^%$7ON8r%Qr  h$Q\
On%q                    N%O~      M$qN$OMp    RmPQ    O%rQ                  $rQ\
MkQN                    77O#      dj#Nkd$7    O%d8    ``(r                  RmM\
:(luN](%mRMmO%lNRmO_loa8%%O<  b(lQ    h'lQ    O^ln    ;b'%N&O^6sN#M9sllltwzh&TmS$'\
%R#ON$oNS&pRM7O$'%S#ON$oNS&p  SMd9    ``(t    pkO9    $:nk8b#^%}kx#O%lMtpk#OQOP:#\
                              mR8`}(qQ        O<b\
                              '7M{N^kh        'Q]\
    7sN'    M8q;k'N_7vN'nSM8  'nR;    $%M'    TamS&7O#d#7O    &d8`z9b$    ^h$\
    rQ]6    yN$_$$pQaM8m{px$  TmSt    6O#N    sM$nRmUd#7Os    d77sNOsd    8`j\
=nk;        |8k9    b1g(    Q^)$c:%]  k*Q]g$9%_h*]+wnS        _+sTaa:&%{R\
M<%{        S{O_    &<#w    kP8#a;'(  Od:`184>b)^15)OM        :23h)QN<'/M\
    =b(^'h(Q]Oh#    c_kah%Od9`b'/O(Q^(/SlM9O%M(/            RM;&    $c&kckQ,    -$c\
    .&kccMcOP$&d    9g(=``6:69j=b5g(Q^3(2lNUO<b4            'Q^s    ;b&^%%pQ    M8k\
        $7O#cQ]O(2(Q]'2O    3'Q]'_3a  N_1'    5OMaMch&T$mR#nRMPmSk    U$7\
        O#d8_a%%mS]l_%mR    mS]$]h$9  j_la    _$6N]g$9j_laaaN:`g'<    ``7\
oM:6%N9b%g$Q        ^6%N8b%g  #Q^l            peoqemtemw#jQ7#QO$jQ    O7$Q    Ok#\
$7OMQ]k_#$7O        ckQaOrON  epne            luelue`lpeoqempepneu    e`kf    _a`\
                              lf))    **,**,*)-)/),0).0(6/2+667,(&    $##\
                              #;=@    D*0;#include<stdio.h>@intYH[    9`%\
.],*s,*c,d,t;`;'main(`-9n,ch          ar*v    []){        for(s=c=    H+3`XI;\
(***s=Q[d++]););for(;n>1&&(*          ++s=    v[1]        [tgANs=H    ;d=*c++\
%93,                    d-9;  ){in    tYv=*s,g        []={    n+v,        v-n,n*_\
,#^_                    ,,<v  ,n?v    /n:0`,#%        `,,v    >>n,        v==n}ay\
>t=0    ;d<4&&d>=2*!    !n&&  (c-=d/3*2_KK3+*c++,t||v!=98+d);)              t+=v++/6-16\
?0:v    /2%3-1cq*d-1    4;t>  0_t$<3_z(105<*c_X'=t*21aq@-106;n              =d>76?s--,g\
[d-7    7]:d>55?H_1)    21]=  n,*_@_8(        9?***s_4&12>d_C`f3]+=21-d*2\
:d<3    4?t:_Zadat57    <d?p  utchar(n        ),v:6<a](g+99]=a{d2#:`xa6,n\
;}re    turnY_.",*p,    b,d[      9338    ],*q,x,*r=d;            int main(){;for(
p=q=    5000+0+d;*s;    s++)      if(*    s>32)*p++=*s            -89?*s:32;for(p=
1152                    +q;(  b=*p++);){for(d[17]=              10;x    =*p++,b<
92&&                    34<b  --;*r++=x)if(x==9*9)              for(    ;*q;x=34
)*r++=*q++;for(p-=b<92;b-->4      *23;r++)    *r=r[36-    x];}puts    (d);
return(0);}/*IOCCC2014IOCCC2      014IOCCC    2014IOCC    C2014IOC    CC*/
```

```
#include /*!!TAB=4!*/<stdio.h>
#include /*IOCCC2*/"SDL/SDL.h"
#define b/*{{IOCCC257}}*/ if (
#define a(b, c) for (b = 0 ; /*IOCCC/2014*/b<d; b++,c)
e,h,f,g,i,j,k,l,m,n,o,p=1,*q,r,s=5, t,*u,x,y,z,A,B, C[
333*7],d=333; D,E,F,G [2 ],H, I, J, K, L, M,N= 1,O,P,Q
,*R;char * S, **T;          /*33*/     ;
SDL_Surface*U,* V;          int*W(     X
){ u=C+X *7; b*u){          u++; x     =u[0]; y=u[1
]; H=u     [2]; z          =(H%8)           *
i; A=(     H/8)*i          ; B=u[           3
]/**/;     return          u; ; }            return
0; } Y     (){ O=50; r=          0; t=1                ;
} Z (X      ,m, n,o)/**/          { /**/               ;
return     W(X)&&B&&(m<          x+i&&n               <
y+i&&x<m+i&&                     y<n+o); } int*ba(int*u){ int X
,*bb=C; a(X,                     bb+=7){ b!*bb){ *bb=1; R
=bb +1; H=6;                     while(H--)*R++=*u++; /*W
        N                                 N
        E                                E
        S                                 S
    */return bb; } } return 0; } bc(e){ q[2]=e; } bd
    (be,bf){ int X,bg; m+=be; n+=bf; I=e
    -i; m=m<0?0:(m>I?I:m); a(X,0){ b Z(X
    ,m,n,o)){ bg=B&1; b D&&bg) continue;
m-=be; n-=bf; b B&8){ u[-1]=0; j=1; bc(8); b B&32){ n-= i;
o=i*2; } } b B&16&&!O){ bc(32+(o>i?8:0)); Y(); u[- 1]=0; }
b(B&128&&bf&&s<0)||(B&64))u[-1]=0; b bg&&!O){ b bf&&s >0){
u[2]--; u[3]=bf=0; s=-6; } else { b j){ bc(0); b o>i) n+=o=i; D=30; j=0;
} else { bc(24); Y(); L=-1; } } } b B&4){ b bf&&s<0){ int I[]={ x,y-i,u[
5],u[4],rand()%2?1:-1,2} ; u[2]++; u[3]=2; ba(I); } } b bf)s=1; break; }
}         } bh (m,e,k,     bi){ H=k/ 2;       G[bi]=m>e-H?         k
-         e:(m>H?H-m:0      ); } bj(X,be       ,bf){ int bk         ;
u          [0]+=be; u[1      ]+=bf; W(X);       E=x,F=y,I=0;         a
(               bk,0){ b I=(X!=bk&&Z(bk,E,F,i)&&(B&6                )
)               )break; } W(X); b I){ b bf)u[1]-=bf;                b
be             ){ u[0]-=be; u[4]*=-1*be; } } W( X);                }
bl          (){ int bm=n,X; SDL_FillRect(U,0,M); X=4; while(         X
--         )bd(r,0); X=3; while(X--)bd(0,s); b n>h&&!O)Y ()          ;
t          =bm==n; q[0]=m; q[1]=n; bh(m,e,k,0); bh(n,h,l,1)         ;
a(X,0){ b W(X)){ b B&9){ bj(X,u[4],0); bj(X,0,2); } b B&1){ *u +=u[4]; b
        ++u[5]>20){ u[4]*=          -1; u[5]=0; }   z+=K
        %2?i:0; } J=i; b q          ==u){ J=o; b!O){ b
        r)z+=i*(K%2); b!t)          =48; z+=p<0?i*4:0
        ; }} b q!=u||!(D&&          0==D%3)){ SDL_Rect
```

Bibliographie I

Beaucoup de livres contiennent des erreurs.
Voici quelques sources assez fiables :


Le langage C
Max Buvry – Polycopié N7 – **disponible sur Moodle**


Le langage C
B.W. Kernighan et D.M. Ritchie, 2ème édition
très populaire mais insuffisant


C : a reference manual
P. Harbison et Guy L. Steele
devrait être sur la table de tout programmeur C (à moins de trouver mieux . . .)


The C standard
ISO/IEC 9899 :1999, la norme du langage C, disponible chez Wiley

Premier exemple : pgcd.c

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

/* Afficher le pgcd de deux entiers strictement positifs. */
int main() {
   // Saisir deux entiers
   int a, b;   // deux entiers saisis au clavier
   printf("Entrer deux entiers strictement positifs : ");
   scanf("%d%d", &a, &b);
   assert (a > 0);
   assert (b > 0);

   // Déterminer le pgcd de a et b
   int na = a, nb = b;  // gain de place ! À éviter !
   while (na != nb) {   // na et nb différents
      // Soustraire au plus grand le plus petit
      if (na > nb) {
         na = na - nb;
      } else  {
         nb = nb - na;
      }
   }
   int pgcd = na;     // le pgcd de a et b

   // Afficher le pgcd
   printf("pgcd = %d\n", pgcd);
   return EXIT_SUCCESS;
}
```

Premier exemple : pgcd.c

Elements visibles dans ce premier exemple

- L'inclusion de bibliothèque avec une commande pré-processeur : `#include <stdio.h>` inclut la bibliothèque qui permet de gérer les entrées / sorties standard
- L'écriture de commentaires avec les balises
  - `//` pour commenter le reste de la ligne
  - `/* .... */` pour commenter une portion de code
- La déclaration d'une fonction principale `int main()`. Sa définition se trouve entre les accolades. La donnée entière est retournée avec l'instruction `return`.
  La fonction principale doit toujours s'appeler `int main()`.
- La déclaration de variables entières `int a, b;`
- La gestion des entrées/sorties formatées avec `printf` et `scanf` de la bibliothèque `stdio.h`
- L'utilisation d'une assertion avec la bibliothèque `assert.h`.
- L'utilisation d'une répétition `while` et d'une conditionnel `if`.

Quelques mots sur la compilation en C

Chaine de compilation
Le pré-processeur

Compilation d'un programme en C

Compilation :

- à l'N7, le compilateur s'appelle `c99`.
  Commande pour la compilation :

```
c99 -Wextra -pedantic prog.c -o prog
```

- le compilateur donne deux types de messages :
  - avertissement (*Warning*) : code semble étrange au compilateur
  - erreur : refuse de compiler
- options de compilation recommandées : `-Wextra -pedantic`
- autre option utile :
  - `-lm` : placé en fin de ligne de commande si `#include <math.h>`, la bibliothèque mathématique est utilisée.

Aide sur les fonctions standard (manuel)

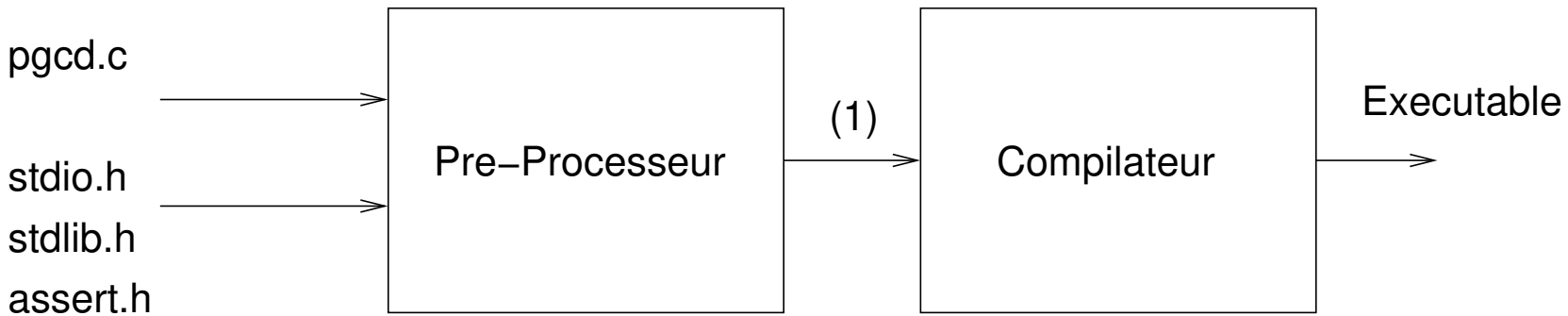
On utilise la commande `man`. Exemple :

- Manuel de la commande `printf` : `man printf`
- Aide sur la commande `man` : `man man`

Compilation d'un programme en C

Chaine de compilation C

Elle lance successivement le pré-processeur, puis le compilateur :



(1) Genere un unique fichier

Compilation d'un programme en C

Le pré-processeur

Le pré-processeur est *indépendant* du compilateur

- Le pré-processeur se moque de travailler sur du C, du Pascal ou du HTML : il prend en entrée un fichier texte avec des *directives* et produit un *unique fichier texte* (mais il a été conçu pour le C)
- On peut invoquer directement le pré-processeur avec les commandes

  `cpp` ou `cpp -P` ou `c99 -E`
- Toutes les directives exécutées par le pré-processeur sont listées dans `pgcd.c` ou les fichiers inclus avec **des commandes pré-processeur** qui commencent par le caractère # :
  `#include`, `#define`, `#if`, `#ifndef`, etc.

Compilation d'un programme en C

Le pré-processeur

Quelques actions du pré-processeur :

- Il supprime les commentaires (texte entre `/*` et `*/` ou après `//`) du fichier `pgcd.c`.
- Il remplace les constantes pré-processeur (#define) par leur valeur de substitution.
- Il inclut le texte des fichiers qui suivent `#include` (lesquels sont aussi traités par le pré-processeur)[3].

3. Une description plus précise de `#include` sera donnée lors de la présentation des modules.

Les constantes avec #define

Syntaxe

```
#define <ident> <valeur de substitution>
```

Sémantique.

A chaque fois que le pré-processeur rencontre `<ident>`, il le remplace par `<valeur de substitution>`
Remarques :

- `<ident>` doit être un identificateur C (par convention, on n'utilise que des majuscules)
- le pré-processeur prend toute la ligne pour `<valeur de substitution>`

Usage courant

Pour définir la capacité d'un tableau

```
#define CAPACITE 255
```

Les constantes avec #define

Attention avec #define :

pas de ; à la fin d'une instruction pré-processeur

Exemple :

```
1  #define SIZE_MAX 65535 /* sans ; */
2  #define SIZE_MIN 0 ; /* avec ; */
3  int main(){
4    int i= SIZE_MAX;
5    int j= SIZE_MIN
6     * SIZE_MAX;
7    return j;
8  }
```

A la compilation :

```
kjaffres@ader:~/CoursC$ c99 -Wextra -pedantic main.c -o main
main.c: In function ‘main’:
main.c:5:18: error: invalid type argument of unary ‘*’ (have ‘int’)
  int j= SIZE_MIN * SIZE_MAX;
```

Types, Constantes et Variables en C

Types Fondamentaux
Constantes et Variables
Portée des variables et blocs

Les commentaires

Commentaires standards

Notation - - en algorithmique

- Equivalent en C99 à //....
  Tout ce qui est après // jusqu'à la fin de la ligne est en commentaire
- Commentaire par bloc /* ... */
  Utile pour commenter plusieurs lignes de code d'un coup.
- Attention : commentaire par bloc ne peuvent pas être imbriqués !
  /* commentaire /* commentaire */ pas commentaire */

Commentaires de vérification de propriété

Notation { } en algorithmique

- Utilisation de la commande `assert` :

```
assert (a>0);
```

  Nécessite l'utilisation de la bibliothèque `<assert.h>`

Les identificateurs

Définition

C'est le nom donné à une entité d'un programme : le programme lui-même, les variables, les constantes, les types, les sous-programmes...

Les identificateurs permettent :

- à l'ordinateur de distinguer les entités et
- aux hommes de comprendre leur rôle et intérêt.

Règle

Un identificateur commence par une lettre, suivie de chiffres, lettres et soulignés (_) :

```
rayon, prix_ttc, n1, n2, etc.
```

Les variables

Déclaration d'une variable

en algorithmique	en C
`le_nom: Le_Type` `valeur : Entier`	`Le_Type le_nom;` `int valeur ;`

- Attention : le langage C distingue majuscules et minuscules !
- On peut déclarer une variable à tout moment dans le programme C

Affectation d'une variable

L'affectation permet d'enregistrer une donnée dans une variable avec l'opérateur $=$ en C.

en algorithmique	en C
`valeur <- 8`	`valeur = 8;`

Les types de base

Plusieurs types de base (ou fondamentaux) existent en C :

Entier (`int`), Réel (`float, double`), Booléen (`bool`), Caractère (`char`) et Pointeur.

Opérateurs de comparaison

Tous les types fondamentaux sont munis des opérateurs de comparaison (à valeur booléenne) :

En algorithmique : `>, <, <=, =>, =  et /=`
En C : `>, <, <=, =>, ==  et !=`

Le type des entiers relatifs

en algorithmique	en C
`l_entier` : **Entier**	`int l_entier;`

Il existe des modificateurs de taille (`short`, `long`) qui modifient la taille mémoire (et l'intervalle des valeurs) de `int` :

- `signed`, `unsigned` : pour des `char` ou des `int` signés (implicite pour `int`) ou non
- La taille (en octets) et l'intervalle des types entiers sont stockés dans la bibliothèque `<limits.h>`

Le type des entiers

Un exemple de valeurs possibles (dépend du compilateur) :

Type	Taille (octets)	Valeur min	Valeur max
`short int`	2	$-2^{15}-1$	$2^{15}-1$
`int`	4	$-2^{31}-1$	$2^{31}-1$
`unsigned int`	4	0	$2^{32}-1$
`long int`	8	$-2^{63}-1$	$2^{63}-1$
`unsigned long int`	8	0	$2^{64}-1$

Le type des booléens

En général en C, on utilise **un entier pour représenter un booléen** avec la convention suivante pour `int b` :

`(b == 0)` $\rightarrow$ `b est FAUX`
`(b != 0)` $\rightarrow$ `b est VRAI`

Il existe aussi la bibliothèque `stdbool` (il faut donc faire un `#include <stdbool.h>`) qui définit le type `bool` avec deux valeurs : `true` et `false`

en algorithmique	en C
`le_booleen:` **Booleen**	`bool le_booleen;`
`le_booleen <-` **VRAI**	`le_booleen = true;`

Attetion : `b == true` ne donnera pas toujours le résultat attendu.

Le type des booléens

Les opérateurs logiques en C :

en algorithmique	en C
a Et b	n'existe pas
a Ou b	n'existe pas
Non a	!a
a EtAlors b	a && b
a OuSinon b	a || b

Evaluation en court-circuit (ou partielle) en C

L'évaluation d'une expression booléenne s'arrête dès que le résultat est connu :

false && expr -- toujours FAUX sans avoir a évaluer expr

true || expr -- toujours VRAI sans avoir a évaluer expr

Intérêt Dans l'expression `(n<>0)` **Et** `((s` **Div** `n) >= 10)`, que se passe-t-il si n vaut 0 ?

Attention Tous les langages n'ont pas d'évaluation partielle. Le langage C implante l'évaluation partielle des expression booléennes.

Les types de Réels

Il existe plusieurs types de réels en C

3 types de types Réels (flottants) en C

- `float` : Un nombre décimal à simple précision
- `double` : Un nombre décimal à double précision
- `long double` (C99) : Un nombre décimal à précision étendue

Ils sont toujours signés.

Précision	Type	Taille	Exp.	Mantisse	Valeur
Simple	`float`	32 bits	8 bits	23 bits	$(-1)^s \times m \times 2^{(e-127)}$
Double	`double`	64 bits	11 bits	52 bits	$(-1)^s \times m \times 2^{(e-1023)}$

Les types de base

Le type caractère

en algorithmique	en C
`le_car : Caractère`	`char le_char;`

En C

- Il est codé sur 1 octet
- Chaque caractère est représenté par un entier : code ASCII

Attention en C

→ `char` est un type entier non signé (`unsigned int`) !

Les types `char` et `int` sont donc compatibles (car identiques) :

```
char c = 'A';
int val = c; // val vaut 65 (code ASCII de c)
```

Convertir un chiffre en char, et réciproquement.

Il n'existe pas d'opérateur de conversion dédié pour ces opérations en C.

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

int main(){
  // Conversion du char '1' en l'entier 1
  char c_char = '1';
  int c_int = c_char - '0'; //on retire le code ascii de '0'
  printf("'%c' en ascii = %d et c_int = %d \n", c_char, c_char, c_int);

  // Conversion de 1 en un char qui vaut '1'
  int new_int = 1;
  char c_char2 = new_int + '0'; //on ajoute le code ascii de '0'
  printf("new_int = %d et c_char2 en ascii = %d \n", new_int, c_char2);
}
```

On obtient à l'exécution :

```
'1' en ascii = 49 et c_int = 1
new_int = 1 et c_char2 en ascii = 49
```

Alias de type : typedef

Contexte et définition

Il est possible de définir des nouveaux types à partir de types existants à l'aide de la commande :

```
typedef <type_existant> <nouveau_type>;
```

Le type <nouveau_type> est un alias (ou synonyme) du <type_existant>

Exemple

```
typedef int Entier;
```

Les constantes en C

- Les constantes littérales (déconseillées ailleurs qu'en initialisation)
  1, −12, 'a', '5', "Hello"
  Notons trois constantes caractères particulières (parmi d'autres) : `'\t'` (tabulation), `'\n'` (retour à la ligne) et `'\\'` (anti-slash).
- Les constantes typées (à partir de C89) : ajout de `const` devant une déclaration initialisée

```
const int nb_max_fiches = 65535;
```

- Les constantes pré-processeur (historiques) : substituées par leur définition

```
#define NB_MAX_FICHES  65535
```

Attention - Rappel

Pas de `;` après un `#define`

Différence entre contantes typées et pré-processeur

- Le code :

```
#define TAILLE_EN_TETE 6
#define TAILLE_CHARGE  16
#define TAILLE_PAQUET  TAILLE_EN_TETE+TAILLE_CHARGE
const int taille_paquet=  TAILLE_EN_TETE+TAILLE_CHARGE;
int trame1, trame2;
trame1= TAILLE_PAQUET*10; /* = 6+16*10 = 166 */
trame2= taille_paquet*10;  /* = 22*10   = 220 */
```

- Après exécution du pré-processeur (commande c99 -E) :

```
const int taille_paquet= 6 +16;
int trame1, trame2;
trame1= 6 +16*10;
trame2= taille_paquet*10;
```

Conclusions :

1. préférez les constantes typées (quand c'est possible)
2. parenthésez les expressions avec #define

Compatibilité entre types en C

En C, on peut manipuler des expressions qui comportent des expressions de types différents[4] si elles sont compatibles.

Définition

Si un type A est compatible avec un type B, on peut mettre une expression de type A partout où une expression de type B est attendue.

Règles de compatibilité

- Un type est compatible avec lui même !
- Un type A est compatible avec un type B ssi le passage d'un type A à un type B se fait sans perte d'information.
  Par exemple, `Entier` est compatible avec `Réel`, mais l'inverse est faux.
- Il y a coercition : transformation des données.

```
double x = 1; // l'entier 1 est converti en un réel 1.0
```

4. Ce qui n'est pas toléré en ADA.

Opération de conversion explicite (*cast*)

Conversion explicite

On peut convertir explicitement une expression en un autre type :

```
(type) expression
```

Exemple

```
int somme = 125;
int nb_echantillons = 10;
double fausse_moyenne = somme / nb_echantillons;
                    /* ici fausse_moyenne vaut 12.0 */
double moyenne = somme / (double) nb_echantillons;
                    /* moyenne vaut 12.5 */
```

Evaluation d'une expression

Une expression est évaluée :

- En commançant par les opérateurs de priorité la plus élevée.
- A priorité égale, les opérateurs les plus à gauche.
  `a + b * c + d` ≡ `(( a + (b * c)) + d)`

Table de priorité des opérateurs

En langage pseudo-algorithmique.

Priorité	Opérateurs	
1	`+, -,` **Non** (Unaires)	Priorité la plus forte
2	`*, /,` **Div, Mod, Et**	
3	`+, -,` **Ou**	
4	`<, >, <=, =>, =, <>`	Priorité la plus faible

Les parenthèses permettent de modifier les priorités.

Opérateurs arithmétiques et relationnels en C

+	-	*	/ (entier)	/ (flott.)	%	<	<=	>=	>	==	!=

Le langage fait de nombreuses conversions implicites ([H&S5th,§6.3.4]), qui se passent souvent assez bien.

Entre entiers	$2 + 4 == 6$	$3 / 2 == 1$
Entre flottants	$2.0+4.0 == 6.0$	$3.0/2.0 == 1.5$
Conversions	$2 + 4.0 == 6.0$	$3.0/2 == 1.5$

Affectation avec opération

Forme

Les instructions de la forme `x = x # y` étant très fréquentes, le langage C offre une forme raccourcie.

`x #= y` équivaut à `x = (<type de x>)(x # (y))`

Exemples :

```
4     i+= 4;     /* ajoute 4 a i */
5     j+= -2*5; /* ajoute -10 a j */
6     j*= 2+5;  /* multiplie j par 7 */
```

Ne pas affecter deux fois une même variable dans une expression !

Pré- et post- incrementation et décrémentation

Forme

- Pré-incrementation : `++x` :
  ajoute 1 à `x` avant d'évaluer l'expression et vaut la nouvelle valeur de `x`.
- Post-incrementation : `x++` :
  vaut `x` et ajoutera 1 à `x` avant de passer à la prochaine expression ou instruction

Il en est de même avec `--x` et `x--`.

Exemples :

```
3     int i= 5;
4     int j= ++i;  /* j vaut 6, i aussi */
5     j= (i--)+1;  /* j vaut 7, i vaut 5 */
```

Priorités des operateurs en C

Dans l'ordre des priorités décroissantes :

```
15   -> . [] ()
14   sizeof ++ -- ~ ! * & (cast) + - /* unaires */
13   * / %
12   + - /* binaires */
11   <<   >>
10   <   <=   >   >=
 9   ==   !=
 8   &
 7   ^
 6   |
 5   &&
 4   ||
 3   ?: (conditionnelle)
 2   = *= /= %=  +=  -=  <<=  >>=  &= |= ^=
 1   ,
```

Bloc et portée des variables en C

Définition d'un bloc

Les délimiteurs de *bloc* $\{\ldots\}$ transforment une *suite* d'instructions séparées par ; en *une unique* instruction.
Comme un bloc est une instruction, un bloc peut contenir des blocs.

Attention !

pas de ; après le $\}$

Portée des variables

Une variable est visible (et donc accessible) de sa déclaration jusqu'à la fin du bloc qui contient sa déclaration.

Remarque : En C historique les déclarations de variables ne pouvaient être qu'en début de bloc. Depuis C99, elles sont possibles partout où on peut mettre une instruction.

Bloc, portée et masquage des variables en C

Portée des variables : Exemple

```c
int main(){  // Début de bloc
   int mon_entier = 10;
   {   // Début de bloc
      float mon_reel = 1.5;
      int mon_entier = 15;  // masquage
      printf("mon_entier dans le bloc : %d \n", mon_entier);
   }
   // mon_reel n'est plus accessible
   printf("mon_entier après le bloc : %d \n", mon_entier);
   return EXIT_SUCCESS;
}
```

Résultat

```
mon_entier dans le bloc : 15
mon_entier après le bloc : 10
```

Conditionelles et répétition

Les structures de contrôle

Les structures suivantes sont détaillées :

- Conditionnelle `Si-Alors-Sinon`
- Conditionnelle `Selon`
- Répétition `Tant_Que`
- Répétition `Répéter` et `Tant_Que`
- Répétition `Pour`

Le Si-Alors-Sinon

Pseudo-langage	en langage C

`Si condition Alors`
`  sequence1`
`Sinon`
`  sequence2`
`Fin_Si`
	`if (condition) {`
	`  sequence1;`
	`} else {`
	`  sequence2;`
	`}`

En langage C :

- La partie `else` est optionnelle
- il *faut* des parenthèses autour de la condition
- pas de mot-clé `then` (d'où les parenthèses obligatoires)
- on ne met pas de `;` en fermeture de bloc
- conseil : toujours mettre les `{ }`

L'opérateur de choix en C

```
cond ? val_vrai : val_faux
```

Évalue `cond`. Si `cond` est vraie (différente de 0), évalue `val_vrai` et vaut cette valeur. Sinon (`cond` vaut 0), évalue `val_faux` et vaut cette valeur.

Exemple

```
nb_jours_fevrier = est_bissextile ? 29 : 28
```

La structure Selon

Évalue l'expression (entière !) dans le `switch`, saute au `case` correspondant (`default` sinon, s'il existe) et exécute toutes les instructions de tous les `case` suivants, jusqu'à rencontrer un `break`.

Pseudo-Langage	Langage C

`Selon <expr> Dans`
`  valeurs_seq_1 : sequence1`
`  valeurs_seq_2 : sequence2`
`  ...`
`  valeurs_seq_N : sequenceN`
`  Autres : parDefaut`
`Fin_Selon`
	`switch (expr) {`
	`  case choix1:`
	`  instruction1;`
	`  instruction2;`
	`  break;`
	`  case choix2:`
	`  instruction1;`
	`  instruction2;`
	`  break;`
	`  default:`
	`  parDefaut;`
	`  break;`
	`}`

Attention

Ne pas oublier l'instruction `break;` !

La structure Selon : Exemple

On ne peut pas lister plusieurs valeur de expr dans un case :

algorithmique	langage C

Selon valeur Dans
  1, 2 : valeur <-- valeur + 1
  3..5 : valeur <-- 100
  Autres : valeur <-- 0
Fin_Selon
	switch(valeur){
	case 1 :
	case 2 :
	  valeur += 1;
	  break;
	case 3 :
	case 4 :
	case 5 :
	  valeur = 100;
	  break;
	default :
	  valeur = 0;
	}

La boucle Tant_Que

Forme générale

Pseudo-langage	Langage C

`Tant_Que <cond> Faire`
`  sequence`
`Fin_Tant_Que`
	`while (cond) {`
	`  sequence;`
	`}`

Exemple

a % b par soustractions successives

```
1  int n = 0;
2  while (a > b) {
3    a = a - b;
4    n++;
5  }
```

La boucle Répeter Jusqu_à

Forme générale

Pseudo-langage	Langage C

Répéter
  sequence
Jusqu_à <cond>
	do {
	  sequence;
	} while (cond);

Exemple : Saisie contrôlée d'une valeur

```
1  do {
2    printf("Donnez un nombre dans [0..100] : ");
3    scanf("%d", &i);
4    if (i < 0 || i > 100) {
5      do_something();
6    }
7  } while (i > 100 || i < 0);
```

La boucle Pour

En pseudo-langage

```
Pour <ident> De <init> A <final> Faire
    instruction
Fin_Pour
```

En C

```
for (expr_ini; condition_continuation; increment ){
    instruction;
}
```

On peut omettre `expr-ini`, `condition_continuation` et `increment`.

La boucle Pour

Exemple

Somme des N premiers entiers

```
int sum = 0;
for (int i = 1; i <= N; i++ ){
    sum = sum + i;
}
```

Conseil : Il faut raisonner comme pour une boucle `while` :

```
int sum = 0;
int i = 1;
while ( i<= N){
    sum = sum + i;
    i++;
}
```

La boucle Pour

Attention en C

Ce n'est pas un vrai *pour* comme vu en algorithmique :

- l'arrêt peut-être autre chose qu'une borne
- c'est le programmeur qui contrôle l'incrémentation
- on peut modifier la variable de boucle dans une instruction du corps de la boucle : A proscrire ! !

En C99, on peut déclarer des variables dans `expr-ini`, visibles uniquement dans `condition-sortie, increment` et `instruction`.

La boucle Pour

Usages courants :

```
for (int i=0; i<10; i++){
    ...
}
```

```
for (int i=9; i>=0; i--){
    ...
}
```

```
for (char c='a'; c<='z';c++){
    // Si codage ASCII
    // ...
}
```

Les instructions de saut

Existent mais **très fortement déconseillées**

- `break` [Buv, §1.8.8], [K&R2d, §3.7] (sauf dans la structure switch)
- `continue` [Buv, §1.8.9], [K&R2d, §3.7]
- `goto` [K&R2d, §3.8]

Indentation

- Il existe plusieurs types d'indentation acceptables en C, nous utiliserons le style K&R en 1SN [5].

```c
// Indentation K&R
int main(){
  int i = lire_choix();
  if (i < 0 ){
    // i negatif
    i= -i;
  } else {
    i= i+1;
  }
  while( i > 0 ){
    i= i-3;
  }
  return EXIT_SUCCESS;
}
```

5. cf. brace policy section at https://users.ece.cmu.edu/~eno/coding/CCodingStandard.html

Les types utilisateurs

Énumérations - Enregistrements – Tableaux

Rappel : Enumération en Algorithmique

Déclaration d'un type énuméré Jour

```
TYPE Jour EST ENUMERATION ( Lundi, Mardi, Mercredi,
                            Jeudi, Vendredi, Samedi, Dimanche )
```

- Chaque valeur possible est nommée par un *identifiant*

Déclaration et affectation d'une variable

```
Variable
    mon_jour : Jour
Début
    mon_jour <- Jeudi
```

Enumération en C

Déclaration d'un type énuméré en C

```
enum Jour { LUNDI, MARDI, MERCREDI,
                    JEUDI, VENDREDI, SAMEDI, DIMANCHE };
```

- Chaque identifiant est une constante symbolique initialisée par le compilateur
  `LUNDI` vaut 0, `MARDI` vaut 1, `MERCREDI` vaut 2, etc ...
- Convention en C : identifiants des constantes en majuscules
- ATTENTION : Deux types énumérés de ne peuvent définir le même identifiant ! [6]

6. Ce qui n'était pas le cas en Ada ou en algorithmique.

Enumération en C

Déclaration et affectation d'une variable

Création d'une variable de type énuméré `enum Jour`, initialisée au choix par exemple à `MARDI` :

```
enum Jour   mon_jour = MARDI;
```

Remarques

- `enum X {...};` définit un type `enum X` et non un type `X`
- il est d'usage de créer un alias de type pour raccourcir le nom d'un type énuméré en C :

  ```
  typedef enum Jour Jour;
  ```

  On peut alors utiliser le type `Jour` pour manipuler un `enum Jour`

Opérateurs sur le type énumération

Opérateurs de comparaison

$$=,\ <>,\ <,\ <=,\ >,\ >=$$

Si e1 et e2 apparaissent *dans cet ordre* dans la définition du type énuméré, alors (e1 $<$ e2) est vrai.
Par exemple :

```
LUNDI < MARDI < MERCREDI < DIMANCHE
mon_jour == MERCREDI // vrai si mon_jour vaut MARDI
```

Note :

- Il n'existe pas d'opérateurs spécifiques en C qui permettent de trouver le successeur ou le prédécesseur d'une variable de type `enum X` comme en algorithmique ou en Ada.
  Il suffit de faire $+1$ (modulo le nombre de valeurs possibles du type) pour passer au successeur.

Rappel : Enregistrement en algorithmique

Déclaration d'un enregistrement Date

```
TYPE Date EST ENREGISTREMENT
    jour : Jour
    mois : Mois
    année : Année
FIN ENREGISTREMENT
```

Déclaration et affectation d'une variable

```
ma_date, copy_date : Date -- variables de type Date
ma_date.jour <- 3
ma_date.mois <- AOUT
ma_date.annee <- 1979
copy_date <- ma_date
```

Enregistrement en C

Déclaration du type enregistrement `struct Date`

```
struct Date {
    Jour jour;
    Mois mois;
    int annee;
};
```

Déclaration et affectation d'une variable

```
struct Date debut, fin, ma_date;
debut.jour = 1;
debut.mois = JANVIER;
debut.annee = 2012;
fin = {31, DECEMBRE, 2012} ;
ma_date = debut;
```

Enregistrement en C

Remarques :

- `struct X {...};` définit un type `struct X` et non un type `X`
- ne pas oublier le ; après le }
- Un membre du type `struct X` ne peut être du type `struct X`.
- Par contre, si `struct X1` est défini avant `struct X2`, un membre de `struct X2` peut être de type `struct X1`.
- Il est d'usage de créer un alias de type pour raccourcir le nom d'un struct en C :

```
typedef struct Date Date;
```

Enregistrement en C

Exemple

- Création d'un `struct Point` avec pour membres les coordonnées du point.

```
1  #include <math.h>   /* pour sqrt */
2  struct Point {
3    double x;
4    double y;
5  };

7  struct Point a= { 1.0, 2.0 };
8  struct Point b= { 0.0, 2.0 };
9  double distance= sqrt( (a.x-b.x)*(a.x-b.x)
10                        +(a.y-b.y)*(a.y-b.y) );
```

Différence entre synonyme et constructeur de type

```
20  struct X {int i;};
21  struct Y {int i;}; //struct Y de même structure que struct X!
22  typedef struct X X; //X est un synonyme de struct X.
23
24  struct X x;
25  struct Y y;
26  X sx;
27
28  x= sx;  /* OK car synonymes*/
29  sx= x;  /* OK car synonymes*/
30  x= y;   /* KO car constructeurs différents*/
31  y= sx;  /* KO car constructeurs différents*/
```

Usage ‘concentré’ du typedef

- On peut faire un synonyme dans la même instruction que la construction du type :

```
typedef enum GroupeMatieres {Maths, Info,
                             Telecoms} eGroupeMatiere;
```

  au lieu de :

```
enum GroupeMatieres {Maths, Info, Telecoms};
typedef enum GroupeMatieres eGroupeMatiere;
```

- On peut faire un synonyme d’un type enregistrement ou enum *anonyme*

```
12 typedef struct {
13     double hauteur;
14     double largeur;
15 } Rectangle;
```

Attention

L’utilisation du type enregistrement anonyme + typedef est fortement déconseillé (mauvaise lisibilité du code)

Rappel : le type tableau en algorithmique

Définition d'un type tableau à une dimension

Un tableau est un type de données qui permet de regrouper *un nombre fini d'éléments*, ayant *tous le même type*.

```
CAPACITE : Constante Entier <-- 10
TYPE NomType EST TABLEAU (1 .. CAPACITE) DE TypeElement
```

- NomType : Tdentificateur du type
- CAPACITE : Nombre maximal d'éléments du tableau
- TypeElement : Type de chaque élément du tableau

Déclaration de variables de type tableau

```
CAPACITE : Constante Entier <-- 10
TYPE T_Vecteur EST TABLEAU (1 .. CAPACITE) DE Entier
monVecteur : T_Vecteur // declaration
```

Les tableaux en C

Déclaration d'un **type** tableau

typedef `type-elem T_tab[Capacite_1]...[Capacite_dN];`

Déclaration d'une **variable** tableau

En C, on peut déclarer directement un variable tableau, sans passer par le définition explicite d'un type :

- Déclaration d'une variable tableau à une dimension de `type-elem` :
  `type-elem nom[Capacite];`
- Déclaration d'une variable tableau à N dimensions :
  `type-elem nom[Capacite-d1]...[Capacite-dN];`

Exemples :

```
2    #define CAPA 6
3    int le_tableau[CAPA]; // variable
4    double une_matrice[CAPA][CAPA]; // variable
5    typedef int hypercube[2][2][2]; // type
6    hypercube c1, c2;   // variables
```

Tableau à une dimension en C

Accès à un élément

```
int tab[2];                 // Déclaration tableau taille 2
tab[0] = 12;                // Affectation du 1er élément
tab[1] = 23;                // Affectation du 2e et dernier élément
int dernier = tab[1];       // Accès au 2e élément
```

Attention

- La **numérotation commence à 0 !**
- On utilise des `[ ]` pour les indices
- Aucun contrôle de borne ! ! !

Tableau à une dimension en C

Remarques

- Contrairement à l'algorithmique, on ne peut pas affecter un tableau

```
int vect1[5], vect2[5];
int test[5] = {1, 2, 3, 2, 1};
for (int i=0; i<5; i++){
    vect1[i] = 0;          -- initialisation
}
```

- On peut ne pas donner de capacité si on initialise le tableau en le déclarant

```
float x[] = {1.25, -5.1, -0.6};
```

Tableau à plusieurs dimensions en C

Description

- Un tableau à plusieurs dimensions est un tableau de tableaux

```
int c[2][3]= { {1,2,3}, {-1,-2,-3}};
```

- utilisation de `[ ][ ]...[ ]` pour l'accès
- on peut ne pas donner la taille *de la première dimension* si on initialise la variable en la déclarant
- attention à mettre le bon nombre de `[ ]...[ ]`

Exemple d'utilisation

```
23      for(int i = 0; i < 2; ++i) {
24        for (int j = 0; j < 3; ++j) {
25          c[i][j]= c[i][j]*b[j];
26        }
27      }
```

Tableaux et constantes en C

- On peut déclarer la taille d'un tableau avec une constante typée (attention, cela ne marche que pour les compilateurs récents) :

```
const int MAX = 50;
int test[MAX];
```

- Il vaut mieux utiliser une constante littérale ou pré-processeur (ou un identifiant d'un `enum`...).

```
#define CAPA 6
int le_tableau[CAPA]; // variable
```

Tableaux et chaîne de caractères

Il n'existe pas de *type* chaîne de caractères en C !

C'est un tableau de caractères avec un marqueur de fin

Par convention, si on enregistre une chaine de `N` caractères, la case d'indice `N` contient le caractère `'\0'`. Il faut donc un tableau de `N+1` cases au moins pour enregistrer une chaine de `N` caractères.

- Les constantes chaine de caractère littérales sont définies avec les guillemets doubles.
  Exemple : `"BONJOUR"`
- Les chaînes étant des tableaux, on ne peut pas les affecter avec = ou les comparer avec `<`, `<=`, etc.. Mais on peut *initialiser* un tableau avec une constante littérale.
  Exemple : `char str[] = "Bonjour!";`
- Pour manipuler des chaînes, on utilise `strcpy` et `strncpy` (copie), `strcmp` (comparaison lexicographique), `strlen` (longueur), `strncat` (ajout) déclarés dans `string.h`.

Chaînes : exemple

```
1  #include <string.h>
2  #include <assert.h>
3  void main(){
4  char nom[10]= "Ermont";
5       // <=> {'E','r','m','o','n','t','\0'}
6  char prenom[]= "Jerome";
7       /* on a 6+1 == 7 caracteres */
8  char homepage[1]="http://irt.enseeiht.fr/ermont";
9       /* Signale mais accepte */
10 char nom_complet[50];
11 strncpy(nom_complet, prenom, 50);
12 strncat(nom_complet, " ", 50 - 1 - strlen(nom_complet));
13 strncat(nom_complet, nom, 50 - 1 - strlen(nom_complet));
14 assert( strcmp(nom_complet, "Jerome Ermont") == 0);
```

Ligne 14 : marche car `strcpy` copie le marque de fin `'\0'` et `nom_complet` a une capacité suffisante !

Adresse mémoire – pointeurs

Adresse mémoire

Adresse et variable

En C, il est possible de connaitre l'adresse à laquelle est stockée la variable `var1` avec l'opérateur unaire & :

```
&var1
// retourne l'adresse mémoire de var1
```

Pointeur et adresse mémoire

Déclaration d'un pointeur en C

On déclare une variable de type **pointeur sur** <type_pointé> comme on déclare une variable en ajoutant l'opérateur * placé devant le nom de cette variable.

```
type_pointé *identifiant_pointeur;
```

Remarque : `int *ptr` est équivalent à `int* ptr`

Initialisation d'un pointeur

- On peut l'initialiser avec `NULL` : `identifiant_pointeur = NULL;`
- On peut l'initialiser avec l'adresse d'une variable déjà déclarée

```
identifiant_pointeur = &var_pointée;
```

Pointeur et adresse mémoire

Accès au contenu d'un pointeur

- On veut accéder *au contenu de la variable pointée* par le pointeur pour le lire ou le mettre à jour
- On utilise aussi l'opérateur *, comme pour la déclaration.

```
*identifiant_pointeur = valeur;
```

Exemple de mise à jour du contenu du pointeur `ptr` avec la valeur 10 :

```
int var1 = 1;
int *ptr = &var1; // déclaration et initialisation de ptr
int a = *ptr;     // accès à la donnée pointée
*ptr = 10;        // mise à jour de la donnée pointée
```

La dernière instruction est équivalente à l'instruction `var1 = 10;`.

Affectation de pointeur

Affectation de pointeurs

```
p2 = p1;
```

- L'affectation du pointeur `p1` au pointeur `p2` recopie l'adresse dans `p1` dans `p2`.
- `p2` pointe sur la même zone mémoire que `p1`.
- Attention p1 et p2 doivent être déclarés comme des pointeurs sur le même type !

Pointeur et adresse mémoire

En résumé en C

`type *ptr`	Déclaration d'un pointeur sur type
`ptr`	Adresse (référence) de la donnée
`*ptr`	Accès à la donnée pointée par le pointeur
`ptr = NULL`	Initialisation à `NULL`
`ptr = &var`	Initialisation avec l'adresse d'une variable
`ptr2 = ptr1`	Initialisation avec un autre pointeur

Pointeur et adresse mémoire

Exemple : pointeur sur type différent

```c
int main(){
  float *p_float; //pointeur sur un flottant
  int *p_int;   //pointeur sur un entier
  p_float = NULL;
  int val = 2;
  p_int = &val;  //init avec un entier
  p_float = p_int;
  printf("*p_float = %f et *p_int = %d \n", *p_float, *p_int);
  return 0;
}
```

Résultat de la compilation et de l'exécution

```
kjaffres@ader:~/CoursC$ c99 -Wall -pedantic main.c -o main
test.c: In function ‘main’:
test.c:10:16: warning: assignment from incompatible pointer type
        p_float = p_int;
                  ^
kjaffres@ader:~/CoursC$ ./main
*p_float = 0.000000 et *p_int = 2
```

Pointeurs : petits exemples

Qu'ès aquò ?

```
1     int a = 1;
2     int b = 2;
3     int *pa = &a;
4     int *pb = &b;
5     b = *pa;  /* a vaut 1, b vaut 1 */
6     *pa = 3;  /* a vaut 3, b vaut 1 */
7     a = 5;    /* a vaut 5, b vaut 1 */
8     b = *pa;  /* a vaut 5, b vaut 5 */
```

Pointeurs et types

Précision par rapport à la déclaration

Un pointeur est une adresse mémoire qui peut pointer vers tout type de variable :

- des types de base

```
char *p_caractere;   float *p_reel;   ...
```

- des types utilisateurs (on les verra plus tard !)

```
enum Mois *p_mois; // p vers un type énuméré Mois
struct Date *p_date; // p vers un type struct Date
```

- des pointeurs eux mêmes !

```
char **pp_caractere;   float **pp_reel;   ...
```

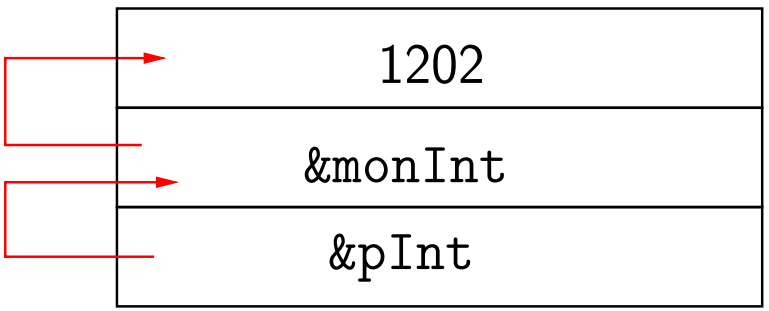
`char **pp_caractere` est un pointeur de pointeur de caractère.

Pointeur de pointeur

Pointeur de pointeur

L'adresse d'une adresse d'une variable.

```
type-element **ppElement;
```




```
int monInt;
int *pInt = &monInt;
int **ppInt = &pInt;
```

Dans cet exemple, on a donc :

```
**ppInt == 1202;
```

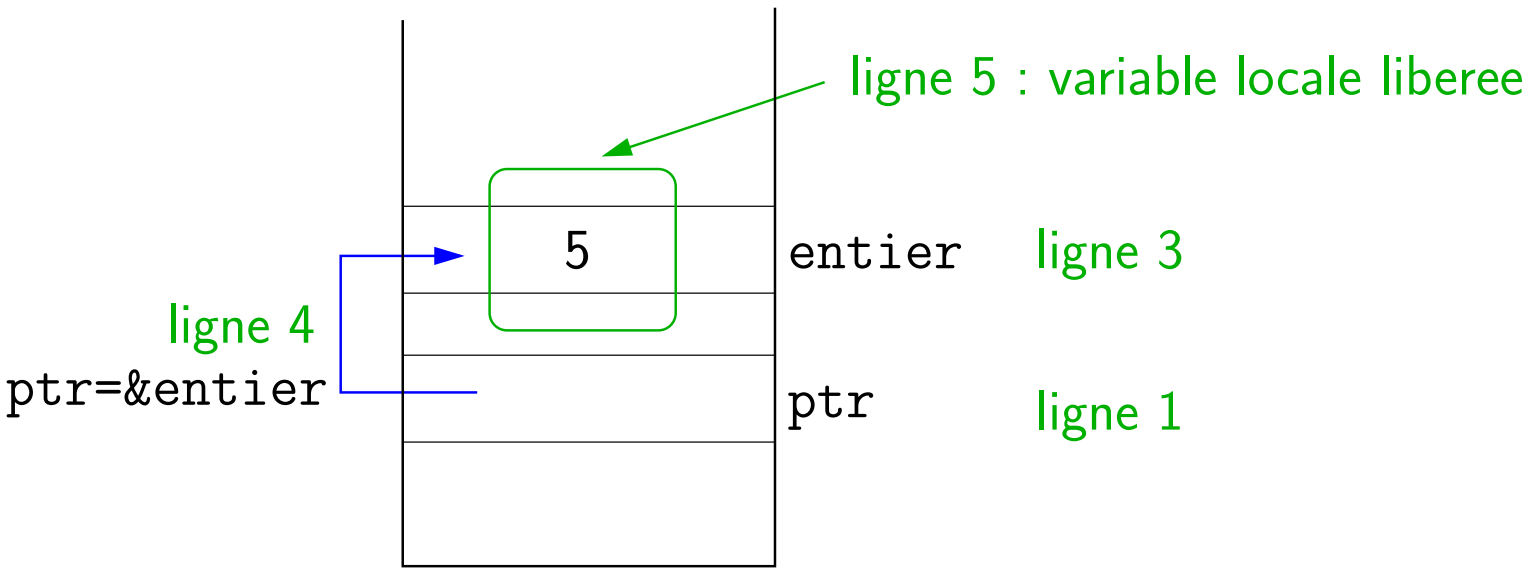
Pointeur et portée des variables

Un exemple

```
1       int *ptr;
2       {                       //debut de bloc
3           int entier = 5; //variable locale au bloc
4           ptr = &entier;
5       }                       //variable locale liberee
6       printf("*p = %d\n",*ptr);
7
8               /* Que se passe-t-il ? */
9   }
```

Pointeur et portée des variables

Un exemple



Note : Pointer sur une variable en Ada

Adresse d'une variable en Ada

- Il est possible d'obtenir l'adresse d'une variable en Ada. Comme cette opération a des effets de bord dangereux, l'opération est rendue plus complexe en Ada.
- Pour se faire, il faut déclarer explicitement qu'on souhaite accéder à l'adresse de la variable lors de sa déclaration :

```
N : aliased Integer ;
```

- Et définir un pointeur qui permette de pointer sur une variable avec `is access` **all** :

```
Type T_Pointeur is access all integer ;
```

- On peut alors initialiser une variable de type pointeur avec l'adresse d'une variable en utilisant 'access.

```
ptr_N : T_Pointeur;
N := 10;
ptr_N := N'access; -- Ptr_N pointe sur N
```

Pointeur et type enregistrement

Déclaration et initialisation

- La déclaration et l'initialisation se font comme pour un type de base :

```
struct tStruct *pStruct;   // déclaration du pointeur
struct tStruct la_variable;// déclaration de l'enregistrement
pStruct = &la_variable;    // initialisation du pointeur
```

Accès à la donnée de type enregistrement

- L'accès aux *éléments* de l'enregistrement pointé utilise aussi l'`*` :

```
(*pStruct).element1 // 1er élément du struct via le pointeur
(*pStruct).element2 // 2e élément
```

- Notation équivalente fortement conseillée car plus lisible :

```
pStruct->element1
```

Pointeur et tableau en C

Rappel sur la définition d'un tableau

Un tableau se déclare sous la forme

```
type-element nomTab[taille]
```

Pointeur et tableau

En C, un tableau est géré en interne à l'aide des pointeurs :

- `nomTab` est *un pointeur constant sur le premier élément* du tableau :

```
assert(nomTab == &nomTab[0]);
```

- On a donc `*nomTab == nomTab[0]`
- Les `taille-1` autres valeurs sont stockées dans les cases mémoires consécutives à `nomTab[0]`.

Arithmétique des pointeurs

Il est possible d'utiliser la notation pointeur pour effectuer des opérations sur un tableau. Prenons l'exemple du tableau `int monTab[4]`.

- Par construction, on a `*monTab == monTab[0]`
- L'expression `monTab + 1` a pour valeur l'adresse de la 2e case du tableau. On a donc

  ```
  *(monTab + 1) == monTab[1]
  ```

  Ici l'incrément $+1$ représente le décalage d'adresse d'un entier en mémoire puisque `monTab` est un pointeur constant sur un entier.
- On peut généraliser : `*(monTab + i) == monTab[i]`
- Et donc récupérer un pointeur sur n'importe quelle case d'un tableau :

  ```
  int* ptr = nomTab + 3; // ptr pointe sur Tab[3]
  ```

  et modifier l'adresse avec l'arithmétique des pointeurs :

  ```
  ptr = ptr - 1; // ptr pointe sur Tab[2]
  ```
- À utiliser avec parsimonie !

Fonction et Procédure en C
Passage de paramètres

Rappel : sous-programmes en algorithmique

Sous-programmes

Il existe 2 types de sous-programmes :

- Les procédures
  Bloc d'instructions définies par le programmeur
- Les fonctions
  Opérateurs définis par le programmeur. Cette opérateur retourne le résultat de l'opération définie.

Signatures et modes en algorithmique

En algorithmique, les sous-programmes sont spécifiés en précisant le mode de passage des paramètres formels du SP :

mode **in**	Le paramètre réel est uniquement lu par le SP
mode **out**	Le paramètre est initialisé par le SP (il est créé)
mode **in out**	Le paramètre réel est lu et modifié par le SP appelé pour le SP

Le mode de passage des paramètres d'un sous-programme est définis dans sa signature. Voici des exemples de `Fonction` et `Procédure` :

En **in** :

```
Fonction min(a, b : in Entier) Retourne Entier Est
```

En **out** :

```
Procedure min(a, b : in Entier, résultat : out Entier)
```

En **in out** :

```
Procedure incrementer(val : in out Entier, inc : in Entier)
```

Sous-programmes en C

Caractéristiques d'un SP en C

- Un SP peut retourner une valeur ou non
- Le seul mode de passage de paramètre possible est le passage par valeur.
  Les paramètres réels du SP ne sont pas modifiables par les instructions du SP !
- `int main()` est un sous-programme : la fonction principale.
- Il est possible de créer des *variables locales* au SP qui sont libérées à la fin du SP.
- Pas de surcharge : deux SP différents doivent avoir des noms différents

Une application C est donc définie comme

- Un ensemble de variables globales (Rappel : à éviter) et constantes
- Un ensemble de SP dont un seul se nomme `main()`

Syntaxe d'un sous-programme en C

La spécification d'un SP

```
typeRes nomSP (type1 param1, ... , typeN paramN)
```

Avec :

- `typeRes`, le type de la valeur de retour.
  Si `void`, il n'y a pas de valeur de retour.

  ```
  void nomProcedure(type1 param1, ...)
  ```

- `type1 param1, type2 param2, ...` sont des paramètres formels.

Attention

Procédures et fonctions suivent la même syntaxe en C.
Néanmoins dans ce cours, fonctions et procédures seront *différenciées aussi en C*. On représentera une procédure avec un SP à type de retour `void`.

Syntaxe d'une fonction C

L'implantation

- Toutes les instructions sont contenues dans un bloc { }
- Les variables locales à la fonction sont définies au début *ou plus loin* dans l'implantation
- Si la valeur de retour est définie (différente de `void`), on retourne une valeur avec l'instruction `return`. Attention, cette instruction termine l'exécution du SP.

```
return expression;
```

Appel de la fonction

Réalisé par l'appel de son identificateur suivi des paramètres réels (ceux qui remplacent les paramètres formels lors de son appel).

```
typeRes vRetour = nomFonction(p1, p2, .., pN);
```

avec `p1, p2, ..., pN` les variables passées en paramètre et `vRetour` la variable qui contient la valeur de retour de la fonction.

Fonction et procédures en C

Exemple de procédure

```
1  void progresser (int n) {
2      // Faire avancer le robot
3      //
4      // Necessite :
5      //  n>0      -- avec n > 0
6
7      for (int i = 1; i <= n; i++) {
8          AVANCER();  // Dans la librairie du robot
9      }
10 }
```

Exemple de fonction

```
1  int max(int a, int b) {
2      // le plus grand des deux entiers a et n
3      //
4      // assure
5      //           Resultat >= a
6      //           Resultat >= b
7      //           (Resultat == a) || (Resultat == b)
8
9      if (a >= b) {
10         return a;
11     } else {
12         return b;
13     }
14 }
```

Exemple d'appel des fonctions en C

Affichage d'un feu tricolore

```c
enum CouleurFeu {ROUGE, VERT, ORANGE};
typedef enum CouleurFeu CouleurFeu;

void afficher_feu(CouleurFeu couleur) {
   // Afficher la couleur du feu
   switch (couleur) {
      case ROUGE:
         printf("Feu rouge\n");
      break;
      case VERT:
         printf("Feu vert\n");
         break;
      case ORANGE:
         printf("Feu orange\n");
         break;
   }
}
```

```c
void exemple_afficher_feu() {
   // affichages de feux

   CouleurFeu c;
   afficher_feu(ROUGE);
   afficher_feu(ROUGE+1);
   for (c=ROUGE; c<=ORANGE; c++) {
      afficher_feu(c);
   }
}

int main() {
   exemple_afficher_feu();
}
```

Passage de paramètres en C

Types de passage de paramètres en C

Il existe deux types de passage de paramètres en C :

- passage *par valeur*
- passage *par adresse*

Passage par valeur

Passage par valeur

```
void f1 (int p1){
  p1 = 10;
}
void f2 (){
  int var1 = 0;
  f1(var1); // vaudra toujours 0 à la fin de f1
}
```

- La variable `var1` du SP `f2` appelant est passée en paramètre au SP appelé `f1`.
- Le contenu de `var1` n'est jamais modifié par `f1`.
- A la fin de `f1`, la valeur de `var1` dans `f2` n'a pas changé.

C'est LE mode de passage de paramètre par défaut en C.

Le paramètre formel peut être interprété comme une variable locale au SP initialisée avec la valeur du paramètre effectif correspondant.

Passage par valeur : exemple

```
9   #include <stdio.h>
10  #include <assert.h>
11  void f1(int p1){
12      p1 = 10 * p1;
13      printf("p1 == %i\n",p1);
14  }
15
16  void f2(){
17      int var1 = 70;
18      f1(var1);
19      printf("var1 == %i\n",var1);
20      assert(var1 == 70)
21  }
```

A l'exécution de `f2`, on affiche `p1 == 700` et `var1 == 70` ;

Passage par valeur en C

Pour un sous-programme C

Un SP ne peut pas modifier la donnée d'une variable passée en paramètre. On peut interpréter l'appel du SP comme suit :

1. Recopie de la valeur des variables passées en paramètre dans des variables *locales au SP*.
   Les identifiants de ces variables locales sont les identifiants des paramètres du SP.
2. Déroulement du code de l'implantation
3. Recopie de la valeur de retour si elle existe
4. Libération des variables locales et des paramètres du SP

Passage par adresse

Comment écrire `Permuter` en C ?

alors que le seul mode disponible en C est le passage par valeur.

Passage par adresse en C

Solution : Passage par adresse

On utilise les pointeurs !

Au lieu de passer la valeur de la variable en paramètre, on donne l'adresse mémoire que le SP va manipuler :

On fait un passage par valeur de l'adresse mémoire

Passage par adresse en C

Spécification et appel

```
void permuter(float *r1, float *r2)
// Spécification avec les pointeurs en paramètre

permuter(&valeur1,&valeur2)
// Appel de la fonction avec des paramètres réels
```

Définition du sous-programme dans ce cas

Il faut manipuler *r1 et *r2 dans le corps du sous-programme :

```
1  void permuter(float *r1, float *r2){
2     assert(r1 != NULL);
3     assert(r2 != NULL);
4
5     float tmp = *r1;
6     *r1 = *r2;
7     *r2 = tmp;
8  }
```

Passage de paramètres formels en C

Passage par adresse en C : exemple

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

// Permuter les flottants r1 et r2
void permuter(float *r1, float *r2){
   assert(r1 != NULL);
   assert(r2 != NULL);

   float tmp = *r1;
   *r1 = *r2;
   *r2 = tmp;
```

```c
// Tester permuter
void tester_permuter(){
   float a,b;
   // Initialiser a et b
   a=10;
   b=20;
   // Permuter a et b
   permuter(&a,&b);
   assert(a==20);   //verifier a
   assert(b==10);   //verifier b
}

int main(){
   tester_permuter();
```

Passage de paramètres formels en C

Passage par adresse

Ce qu'il se passe avec un pointeur en paramètre formel :

1. Recopie des pointeurs (i.e. des adresses) dans les paramètres du SP
2. Déroulement du code de l'implantation avec accès déréférencé au pointeurs (`*ptr`) : on met bien à jour la variable pointée.
3. A la sortie, recopie de la valeur de retour si elle existe
4. Destruction des *copies locales* des pointeurs (i.e. des adresses), et non de leur contenu.

Paramètre tableau

Pour une fonction, un paramètre tableau d'éléments de type T est vu comme un paramètre *pointeur sur T*

- On a donc toujours un passage par ADRESSE pour les éléments du tableau

On peut donc définir un paramètre formel de type tableau de plusieurs façons :

`int f1(int tab[5])` $\simeq$ `int f1(int tab[])` $\simeq$ `int f1(int *tab)`

Note : on ne connait plus la capacité de `tab` dans le SP.

Tableaux et sous-programmes en C

Usages : risques de confusions !

- On peut passer un pointeur à une fonction faite pour des tableaux (à vos risques et périls)
- On peut passer un tableau à une fonction faite pour des pointeurs (équivaut à passer le premier élément)
- On peut passer un tableau sans commencer par le début

Que faire ?

Pensez que votre code peut être relu par quelqu'un qui ne sait pas que c'est pareil.

- Si votre fonction attend un tableau de taille inconnue utilisez []
- Si votre fonction attend un pointeur sur un unique élément utilisez *

Passage par valeur d'un tableau

Constantes typées pour un pointeur

Utilisation de `const` pour empêcher la modification d'un pointeur :

```
2    int i=1, j=2;
3    const int * c= &i;  /* interdit de modifier i à travers c */
4    int const * d= &i;  /* idem que const int* */
```

Passage par valeur d'un tableau

On utilise un paramètre `const`-*qualifié* :

```
int f1(const int tab[30]) ≃ int f1(const int tab[])
                          ≃ int f1(const int *tab)
```

Ici, les éléments du tableau ne seront pas modifiés.

Modes de passage en C et Algorithmique

Correspondance

Les modes de passages vus en algorithmique sont matérialisés en C par :

mode **in**	un passage par valeur
mode **out**	un passage par adresse
mode **in out**	un passage par adresse

Ordre des SP en C

Ordre nominal

`f2` peut faire appel au sous-programme `f1` si celui-ci est déclaré avant `f2`.

(cf. exemple précédent où `afficher_feu` *est spécifié et implanté avant* `exemple_afficher_feu`*).*

Déclaration en avant

Si l'implantation du SP `f1` se trouve après celle de `f2`, C permet d'utiliser `f1` en insérant *sa spécification* avant l'implantation de `f2`.
On déclare `f1` *en avant*.

Exemple de déclaration en avant

On insère la déclaration, *suivie d'un point-virgule* :

```
void afficher_feu (CouleurFeu couleur);
 // Afficher la couleur du feu

void exemple_afficher_feu() {
   // affichages de feux

   CouleurFeu c;
   afficher_feu(ROUGE);
   afficher_feu(ROUGE+1);
   for (c=ROUGE; c<=ORANGE; c++) {
      afficher_feu(c);
   }
}
```

```
void afficher_feu(CouleurFeu couleur)
   // Afficher la couleur du feu
   switch (couleur) {
      case ROUGE:
         printf("Feu rouge\n");
      break;
      case VERT:
         printf("Feu vert\n");
         break;
      case ORANGE:
         printf("Feu orange\n");
         break;
   }
}
```

Les exceptions

N'existent pas

Le langage C n'offre pas de mécanisme de gestion des exceptions comme vu en algorithmique.
On utilise des codes d'erreurs.

Gestion des entrées et sorties d'un programme

i.e. Afficher et Lire des données

Les entrées / sorties

Le plus gros point faible du C ?
Un problème difficile quelque soit le langage.

Notions générales : clavier, écran et flux ?

Notion de flux de données

Les flux permettent d'interagir avec le clavier, l'écran ou un fichier pour échanger des données.
Il existe des *flux* de deux types uniquement :

texte suite de caractères, séparées par des retour chariot

binaires suite de `char`

Un flux est une suite où il est facile de se déplacer de proche en proche :

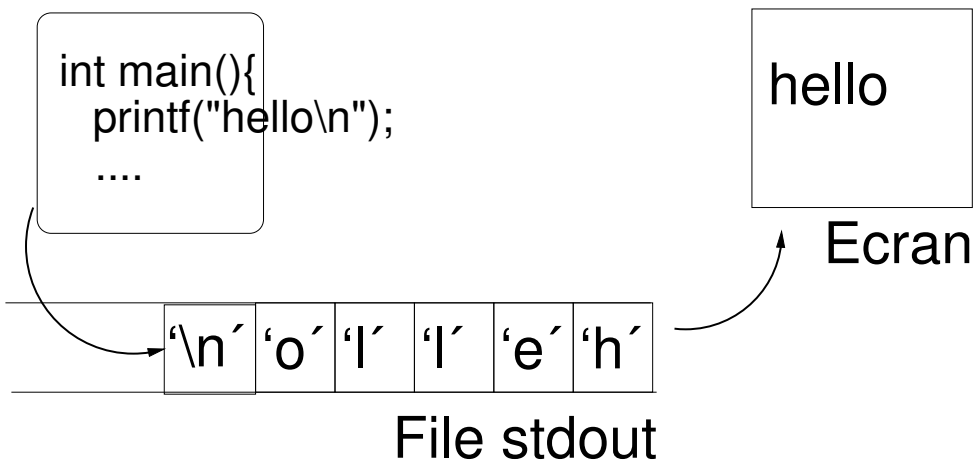
- on peut *lire* ou *écrire* dans un flux,
- *avancer le curseur* dans le flux.

Les flux standard

Il existe 3 flux standards. Un flux est de type `FILE*`

- `FILE* stdout` : **flux de sortie**
- `FILE* stderr` : flux de sortie d'erreur
- `FILE* stdin` : flux d'entrée.

Ils sont de type `FILE*`

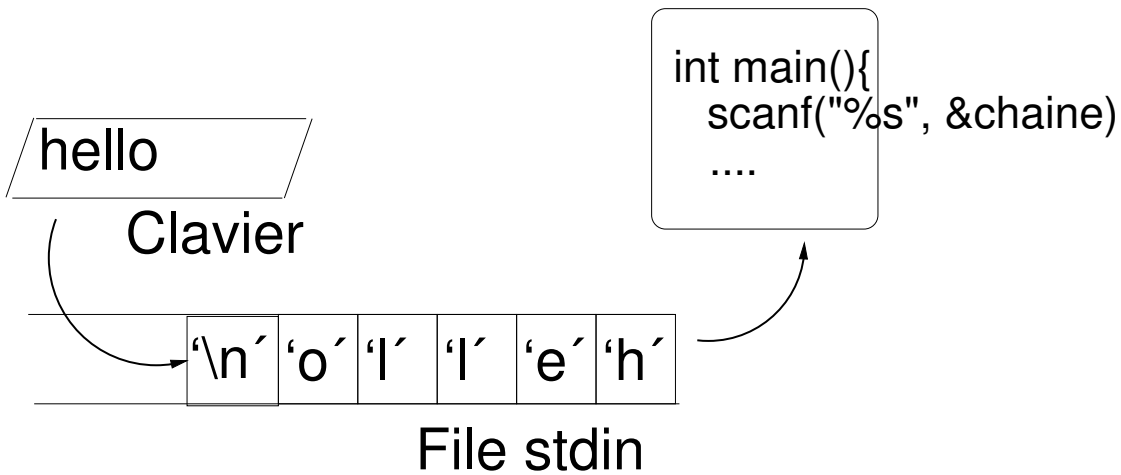


Les flux standard

Il existe 3 flux standards. Un flux est de type `FILE*`

- `FILE* stdout` : **flux de sortie**
- `FILE* stderr` : flux de sortie d'erreur
- `FILE* stdin` : flux d'entrée.

Ils sont de type `FILE*`



Affichage caractère par caractère

Avec ces sous-programmes, on envoie un caractère (ou une chaine de caractères) après l'autre dans la file (mot-clé **put**).

- Pour afficher un caractère dans un flux :

```
int fputc(int c, FILE *flux);
int putc(int c, FILE *flux);  //equivalent a fputc
int putchar(int c); // <=> putc(c,stdout)
```

- Pour afficher une chaîne de caractères :

```
int fputs(const char s[], FILE *stream);
int puts(const char s[]); // <=> fputs(s,stdout)
```

Exemple d'affichage orienté caractère

```c
#include <stdio.h>

int main(void)
{
    char i;
    for(i = 'A'; i <= 'Z'; i++)
        fputc(i,stdout);
    putchar('\n');  // Ajout du retour chariot
    return 0;
}
```

On obtient la sortie suivante à l'écran :

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Les affichages formatés : `printf`

- Principe :

  ```
  printf("format", param1, param2,...,paramN);
  ```

  La chaîne `"format"` est une chaîne de caractères, parsemée de *spécificateurs de format*, commençant par %, chacun s'appliquant à un paramètre écrit dans l'ordre des spécificateurs.

Attention : le compilateur ne vérifie pas la cohérence (mais signale quelques erreurs avec l'option `-Wall`)

Les affichages formatés : printf

Principaux spécificateurs de format

c	char
d	int décimal
u	unsigned int
f	double numérique
x	int en hexa

s	chaîne caractères
hd	short int
ld	long int
e	double scientifique
%	%

```
1  #include <stdio.h>
2  int main(){
3    int i=12;
4    printf("%d",i);  /* => 12 */
5    double d=1.25;
6    printf("%lf",d); /* => 1.250000 */
7    printf("i= %d -- d= %f\n",i,d);
8    /* i= 12 -- d= 1.250000 */
9    printf("d= %1.2f\n",i,d); /* %1.2f : 2 decimales seulement */
10   /* d= 1.25 */
11 }
```

Les sorties sont asynchrones

- Un ordre de la forme printf ("x= %d", x); ne signifie pas « écrit la valeur de l'entier x maintenant », mais « il faudra écrire la valeur de l'entier x ».
- On peut forcer le système à vider la file en ajoutant un caractère retour chariot

```
printf ("x= %d\n", x);
```

ou utiliser un ordre plus ferme

```
fflush (stdout );
```

Entrées caractères par caractère.

Pour lire le flux caractère par caractère, on peut utiliser :

```
int fgetc(FILE *stream);
int getc(FILE *stream);
int  getchar(); // <=> getc(stdin);
```

- lit le prochaine caractère dans le flux
- fait avancer le curseur dans le flux d'un caractère
- retourne le caractère ou `EOF`

Comme on est obligé de consommer pour voir ce qu'il y a après, on avance souvent 'un coup de trop'. Le C offre la possibilité de remettre *un* caractère dans le flux.

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

Exemple d'une lecture caractères par caractère

- Lire tous les blancs jusqu'au prochain caractère :

```
char c;
do {
c= getchar();
} while ( isspace(c) ); // Tant que c'est un blanc
ungetc(c,f);  // Remet le premier non blanc
```

Les entrées formatées : `scanf`

Principe similaire à `printf`.

```
int nb = scanf("format", &param1, &param2, ..., &paramN);
```

sauf que `format` ne doit contenir que des spécificateurs de format (pas de caractères)

Attention !

- pensez aux & car passages par adresse des paramètres
- retourne le nombre de donnée bien lues : contrôle des entrées

Exemple d'utilisation de scanf

```
1  #include <stdio.h>
2  int main(){
3    int i;
4    printf("Entrez un entier ");
5    scanf("%d", &i);
6    double d;
7    printf("Entrez un flottant ");
8    scanf("%lf", &d);
9
10   int nbLus;
11   printf("Entrez un couple entier flottant");
12   nbLus= scanf("%d%lf", &i, &d);
13   switch (nbLus){
14   case 0: printf("Impossible de lire l'entier\n");
15   case 1: printf("Impossible de lire le flottant\n");
16   }
17   return 0;
18 }
```

Gestion des espaces et caractères vides

Un espace dans le format consomme tous les caractères blancs :

```
1  char c1, c2;
2  int nbValides;   //nb de char correctement lus par scanf
3  do {
4     printf("Entrer deux caractères.\n");
5     // On consomme les blancs avant c1 ET entre c1 et c2
6     nbValides = scanf(" %c %c", &c1, &c2);
7     if (nbValides<2) {
8        printf("Entrée non valide.\n");
9     }
10 } while (nbValides < 2);
```

Les entrées formatées

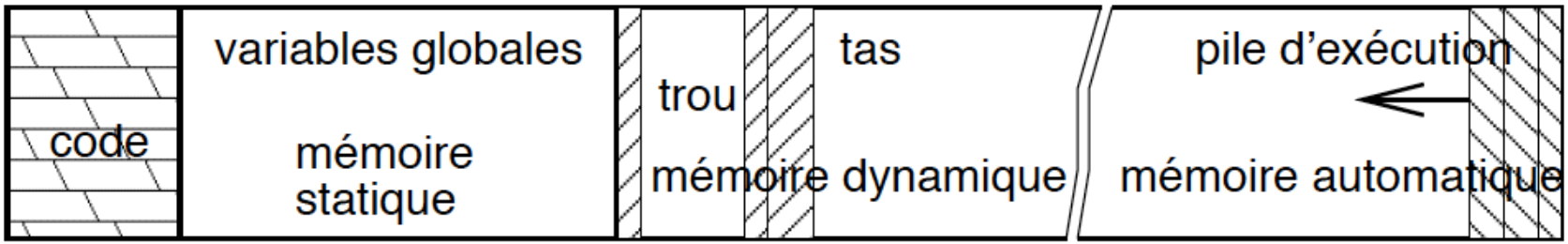
Attention

Quand il reste encore des caractères d'une entrée précédente dans la file qui n'ont pas été consommés, on peut vider tous les caractères jusqu'au prochain retour chariot :

```
5     int c;
6     do {
7       c= getchar();
8     } while ( c != EOF && c != '\n' );
```

L'allocation dynamique de mémoire

Types de mémoires - mémoire dynamique



Mémoire statique

Zone de la mémoire où sont stockées les données qui ont la même durée de vie que le programme (variables globales).

Mémoire automatique

Zone de la mémoire appelée *pile d'exécution* où sont stockés les blocs d'activation, paramètres et variables locales des SP.
Cette mémoire est gérée automatiquement par le compilateur (réservation et libération). La mémoire est contiguë (sans trous).

Mémoire dynamique

Mémoire dynamique : Zone de la mémoire appelée *tas* dans laquelle le programmeur peut explicitement réserver (allouer) de la place.
Il devra la libérer explicitement. Cette zone est fragmentée (trous).

Mémoire dynamique

Accès à la mémoire dynamique

La mémoire réservée en mémoire dynamique est équivalente à une "variable anonyme". Elle n'est accessible qu'à travers un pointeur.

Allocation dynamique de mémoire en algorithmique

```
Type t_ptr Est Pointeur Sur Entier
ptr : t_ptr
ptr <-- New Entier
```

- `ptr` est une variable de type pointeur sur un entier (`t_ptr`)
- `New` réserve de la mémoire dans le tas
- `ptr` contient alors l'adresse de la mémoire allouée (`NULL` si plus de mémoire disponible)
- la quantité de mémoire allouée permet de stocker une valeur du type pointé par `ptr` (Attention : ce n'est pas le cas en C).
- `Allouer` réserve de la mémoire mais *ne l'initialise pas en C*.

Libération de la mémoire dynamique

`Libérer ptr`

`Libérer` libère la mémoire allouée dynamiquement (donc dans le tas).

Il est interdit :

- de libérer un pointeur de valeur `NULL`
- de libérer une zone mémoire déjà libérée
- de libérer de la mémoire allouée automatiquement par le compilateur
- d'accéder à une zone mémoire libérée

Si vous essayez, votre programme est faux (erreur de programmation)...
...même si le programme ne plante pas forcément à l'exécution ! ! !

Libération de la mémoire dynamique : exemple

`Libérer` ne modifie pas la valeur du pointeur

```
1   Allouer ptr
2   *ptr <- 10
3   Libérer ptr
4   Écrire(*ptr) -- Affiche 10 mais c'est une erreur
5                -- de programmation
```

Pour lever toute ambiguïté, il vaut mieux mettre `ptr <- NULL` après avoir libéré la mémoire.

L'opérateur de taille : `sizeof`

Donne la taille en nombre de `char` (donc en octets) de son argument [7]

`sizeof(variable)` ou `sizeof(type)`

Attention :

→ `sizeof` *retourne un entier non signé !*

7. Surtout utile pour l'allocation dynamique.

Allocation de mémoire

nécessite `#include <malloc.h>`

Déclaration	`T_Chose* ptr;`
Réservation	`ptr= malloc( sizeof(*ptr) );` `ptr = malloc(sizeof(T_Chose);`
Destruction	`free(ptr);`

- `malloc` prend en paramètre la taille de la donnée à créer
- s'il ne reste pas assez de mémoire, `malloc` retourne `NULL`
- `free` n'a pas besoin de la taille
- à un `malloc` doit correspondre un unique `free`
- `free(NULL)` est permis

Exemple

```c
#include <malloc.h>
#include <stdio.h>
int main(){
  int *pi = NULL;
  pi = malloc( sizeof(*pi) ); // ou sizeof(int)
  if (pi != NULL) {
    *pi = 5;
    printf("*pi = %d\n", *pi);
    printf("Donnez une nouvelle valeur a *pi: ");
    scanf("%d", pi); // pi == &*pi
    printf("*pi = %d\n", *pi);
    free(pi);
    pi = NULL; // bonne habitude, inutile ici
  } else {
    fprintf(stderr, "Pas assez de memoire");
    return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
}
```

Conseil de programmation

un pointeur devrait toujours
valoir `NULL` *ou* pointer sur une donnée valide !

⇒ toujours initialiser un pointeur lors de sa déclaration

- avec une adresse de variable valide
- avec la constante `NULL`
- avec un allocateur (`malloc` ou équivalent)
- avec la valeur d'un autre pointeur

⇒ toujours tester la valeur de retour d'un allocateur

⇒ toujours mettre un pointeur à `NULL` après un `free`

Allocation d'un tableau

- L'allocation dynamique de tableau de taille dynamique se fait en passant par un pointeur.
- On utilise la possibilité d'utiliser les `[]` sur un pointeur.
  - création d'une variable dynamique tableau de `N` cases de type `T` :

    ```
    malloc( N * sizeof(T) )
    ```

  - on utilise un pointeur sur `T` pour désigner la variable

    ```
    T* tab= malloc( N * sizeof(*tab) );
    ```

  - on s'appuie sur les relations pointeurs-tableaux

    ```
    for(int i=0 ; i< N ; ++i){
      tab[i]= getSomething(i);
    }
    ```

  - On verra plus loin qu'on peut aussi utiliser `calloc`

Attention !

Il faut conserver la taille `N` quelque part !

```
sizeof( *tab ) == sizeof( T ) != N * sizeof( T )
```

Exemple

```
3     int *tab;                    /* Pointeur vu comme tableau */
4     tab = malloc( CAPACITE * sizeof(int) ); // ou sizeof(*tab)
5     if (tab) {
6       for(int i=0; i<CAPACITE; ++i){
7         tab[i] = i;
8       }
9       int * copieTab = tab;              /* Copie du pointeur */
10      free(tab);
11      tab = NULL;
12      copieTab[0] = copieTab[1];           /* BOUM ! */
13    }
```

Autres allocateurs : `calloc`, `realloc`

Il existe d'autres allocateurs de mémoire

- `void* malloc(size_t s);` – déjà vu

- `void* calloc(size_t nbElem, size_t size);`
  alloue la mémoire pour un tableau de `nbElem` cases d'objets de taille `size`.

```
int *p= calloc(50, sizeof(int));
if (p){ // p[] est un tableau de 50 int
```

  - retourne `NULL` en cas d'échec
  - `calloc(nbElem, s)` $\simeq$ `malloc(nbElem * s)`
  - la zone a tous ses bits à 0

Note : `size_t` est un synonyme de `unsigned int`

Autres allocateurs : calloc, realloc

- `void * realloc (void * ptr, size_t size);`
  tente de changer la taille mémoire du bloc désigné par `ptr` à la valeur `size`
  - suppose que `ptr` désigne de la mémoire obtenue par un des allocateurs (ou vaut `NULL`)
  - en cas d'échec : ne fait rien et retourne `NULL`
  - si `s == 0`, équivalent à `free`
  - en cas de réussite : retourne l'adresse d'un (nouveau) bloc, en ayant recopié le contenu de l'ancien
- Exemple d'usage :

```
11    // Agrandir le tableau de double 'tab'
12    // de 'size' case a 'size + INC'
13    double nouveau = realloc(tab, (size+INC) * sizeof(*tab)):
14    if (nouveau != NULL){
15      tab = nouveau;
16    } else {
17      gestion_erreur_alloc();
18    }
```

Modules, pré-processeur et compilation séparée

Rappel : Les modules en algorithmique

Définition d'un module

Partie d'un programme définissant une **unité structurelle et fonctionnelle**. Un module regroupe :

- Un ensemble de déclarations, de constantes, de types, d'attributs et de sous-programmes
- L'ensemble des implantations (corps) de ces sous-programmes satisfaisant au principe de séparation.

Structure d'un module

- **D'une interface** (ou spécification) qui permet de déclarer les constantes, types, attributs au module et de spécifier les sous-programmes.
- **D'un corps** (ou définition) où on regroupe l'implantation des différents sous-programmes spécifiés dans l'interface.
  Et éventuellement d'autres constantes, types, attributs et sous-programmes internes au module.

Les modules en C

Les modules en C n'existent pas !

Le langage C n'offre pas de support syntaxique à la définition des modules. Le principe est le suivant. Un fichier est :

- Soit un programme principal avec un (et un seul) `int main()`,
- Soit un "module".

Déclarations en avant

Pour utiliser ce qui est définit dans un module, on peut utiliser des déclarations en avant dans le fichier programme principal.

- Pour simplifier la vie des utilisateurs, expliciter ce que fournit le module, on écrit un fichier d'entête (.h) qui regroupe les déclarations en avant.
- Ce fichier d'en-tête correspond à la spécification du module. Il est inclut avec la commande `#include` du pré-processeur dans un programme principal qui a besoin de ses services.

Les modules en C

Les modules en C, c'est :

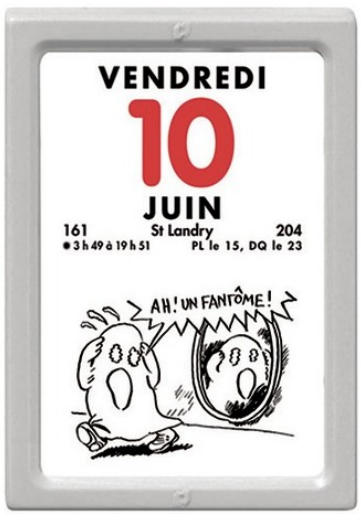
- Une convention de nommage de fichiers pour séparer l'interface du corps du module.
  Un module `complexe` se décompose alors en deux fichiers :
  - `complexe.h` : l'interface du module,
  - `complexe.c` : le corps du module.
- Un outillage pour compiler cette structure de fichiers :
  - Comme le compilateur C ne sait travailler que sur un unique fichier qui regroupe interface et corps, il faut inclure l'interface (`complexe.h`) au début du corps (`complexe.c`) à l'aide de la commande pré-processeur `#include "complexe.h"`.
  - Pour utiliser un module `complexe` dans un programme principal (`calculer.c` par exemple), on inclut son interface `complexe.h` avec `#include "complexe.h"`

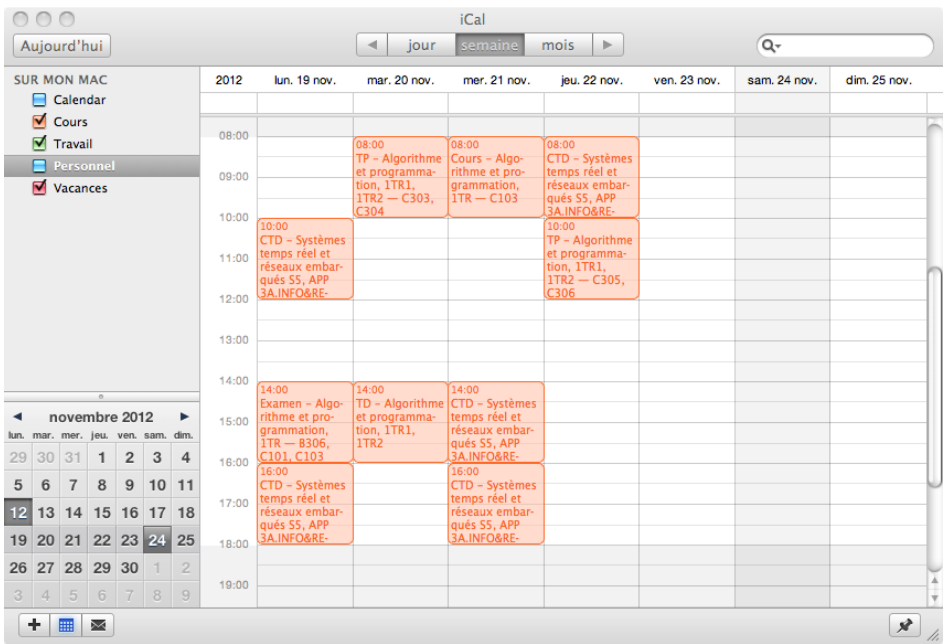
Exemple illustratif

Applications cibles

On souhaite développer les deux applications suivantes :

- Une première application de type Éphéméride qui permet d'afficher le jour et la date courante.
- Une seconde application de type Agenda qui permet de stocker des événements à différentes dates.





Module Date

Les deux applications manipulent des dates. Nous allons donc définir un module `date` en C, puis nous verrons comment ce module est utilisé dans l'application Ephéméride.

On va donc définir : :

- l'interface dans le fichier `date.h`
- le corps dans le fichier `date.c`

Module date en C

Interface du module date : date.h

```
// Inclusion des bibliothèques nécessaires à l'interface
#include <time.h>

// Declaration des types
enum NomJour { DIMANCHE, LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI};
enum Mois {JAN, FEV, MAR, AVR, MAI, JUIN, JUIL, AOUT, SEPT, OCT, NOV, DEC };
typedef enum NomJour NomJour;
typedef enum Mois Mois;

struct Date {
   // Invariant :      jour>=1 && jour<=31; annee>0
   int jour;
   NomJour nomJour;
   Mois mois;
   int annee;
};
typedef struct Date Date;
```

Interface du module date : date.h - suite

```
// Declaration (en avant !) des fonctions et procedures

// Initialise une date. Elle vaut alors Jeudi 01/01/1970.
void initialiser(Date* date);
// Retourne la date d'aujourd'hui
Date date_aujourd_hui();
// Affiche dans stdout la date d'aujourd'hui au format d.jour/(d.mois+1)/d.anne
void afficher_date(Date d);
// Convertit la date au format time_t de time.h en une date de type Date
void convertir_vers_date(time_t t, Date* date);
// Retourne le nombre de jours qui separent d1 de d2
int nbJoursDeDifference(Date d1, Date d2);
// Retourne de numero de la semaine de la date en parametre
int numeroSemaine(Date d);
// Retourne la date du lendemain connaissant la date d en parametre
void dateDemain(Date auj, Date* demain);
// Calcule la date du jour precedent la date auj en parametre
void dateHier(Date auj, Date* hier);
```

Module date en C

Corps du module date : date.c (début uniquement)

```c
// Inclure l'interface Date.h
#include "Date.h"

// Inclure les bibliothèques nécessaire à Date.c
#include <stdio.h>
#include <math.h>

void initialiser(Date *date){
   date->jour = 1;
   date->nomJour = JEUDI;
   date->mois = JAN;
   date->annee = 1970;
}
void convertir_vers_date(time_t t, Date* date){
   struct tm now;
   localtime_r(&t, &now);   //convertion fuseau horaire
   date->jour = now.tm_mday;          //jour
   date->nomJour = now.tm_wday;       //jour de la semaine
   date->mois = now.tm_mon;          //mois
   date->annee = now.tm_year+1900;    //annee (a partir de 1900)
```

Compiler un module

On peut compiler un module une fois l'interface et le corps défini avec l'option `-c` :

```
c99 -Wextra -pedantic -c date.c
```

Dans cette étape, les commandes pré-processeur (`#define`, `#include`, etc) sont réalisées, puis le compilateur vérifie la correction syntaxique du fichier et **génère un binaire (non-exécutable) appelé** `date.o`.

Utiliser un module en C

Comment utiliser un module ?

On l'inclut dans l'application voulue à l'aide de la commande *pré-processeur* :

```
#include "nom_module.h"
```

Il existe des modules standards (que vous avez déjà manipulés) : `string.h`, `stdio.h`, `stdlib.h` ....
Ces modules se trouvent dans un répertoire système (e.g. `/usr/include/`). Attention, la syntaxe pour les inclure est différente :

```
#include <module_standard.h>
```

Utiliser un module en C

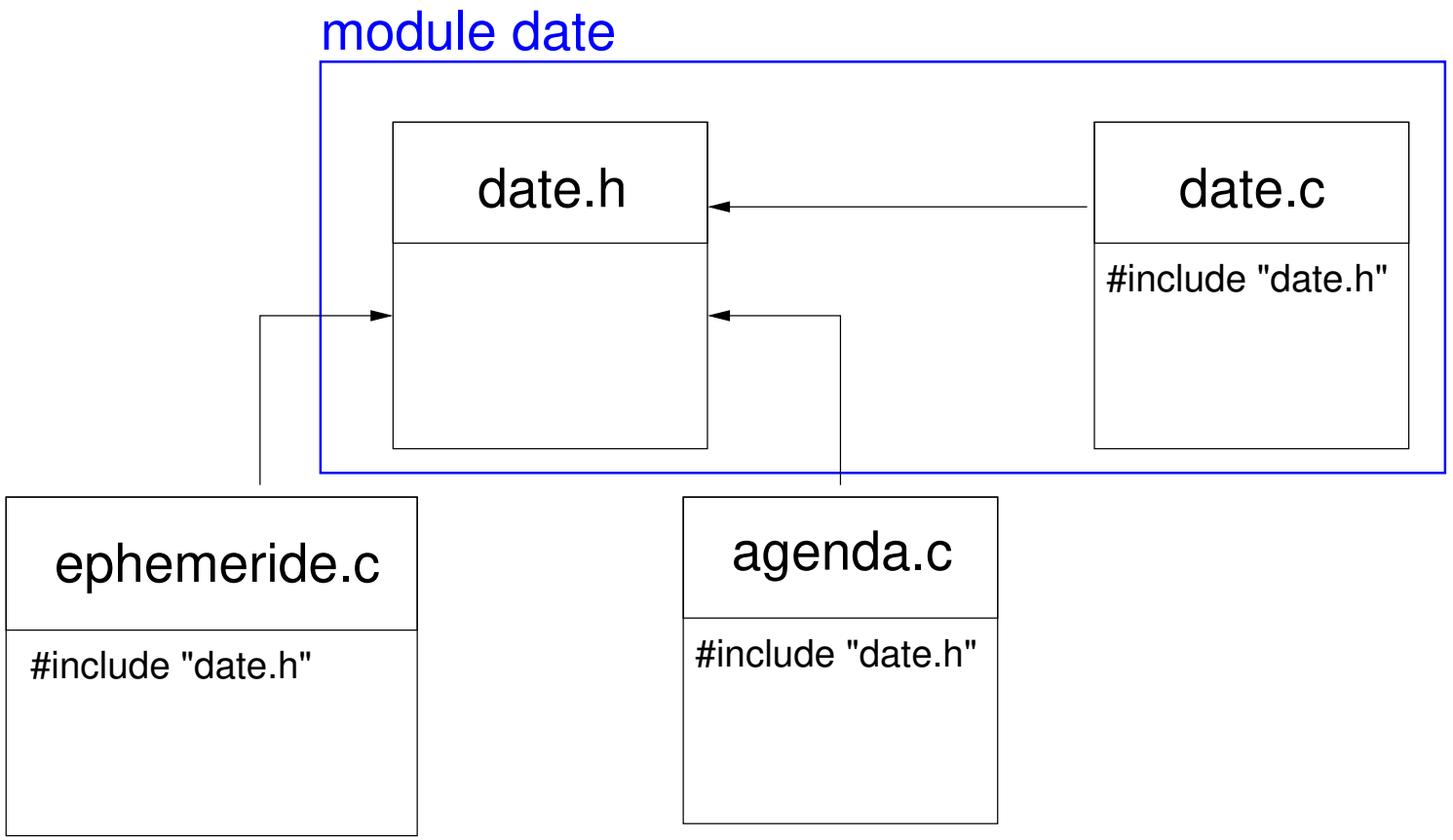
Exemple de l'éphéméride

```c
#include "date.h"   //Inclure le module Date

int main(){
    Date auj = date_aujourd_hui();
    afficher_date(auj);
}
```

Avec la commande `#include "date.h"`, la fonction `main()` peut utiliser tous les services définis dans `date.h`.

L'inclusion de fichier : `#include`



Traitement du pré-processeur : #include

```
# 1 "Ephemeride.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 328 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "Ephemeride.c" 2
# 1 "./date.h" 1
# 13 "./date.h"
# 1 "/usr/include/time.h" 1 3 4

[ ... ]
[ Inclusion des bibliothèques systemes incluses par time.h ]
[ Inclusion de time.h]
# 72 "/usr/include/time.h" 2 3 4
struct tm {
 int tm_sec;
 int tm_min;
 int tm_hour;

[ ...]
[ Inclusion de date.h]
# 14 "./date.h" 2
enum NomJour { DIMANCHE, LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI};
enum Mois {JAN, FEV, MAR, AVR, MAI, JUIN, JUIL, AOUT, SEPT, OCT, NOV, DEC };
```

Compilation et édition de liens

Rappel : on peut compiler un module sans générer d'exécutable

avec l'option `-c` : `c99 -Wextra -pedantic -c fichier.c`
Cette commande produit un fichier binaire : `"Fichier.o"`

Pour générer un fichier exécutable

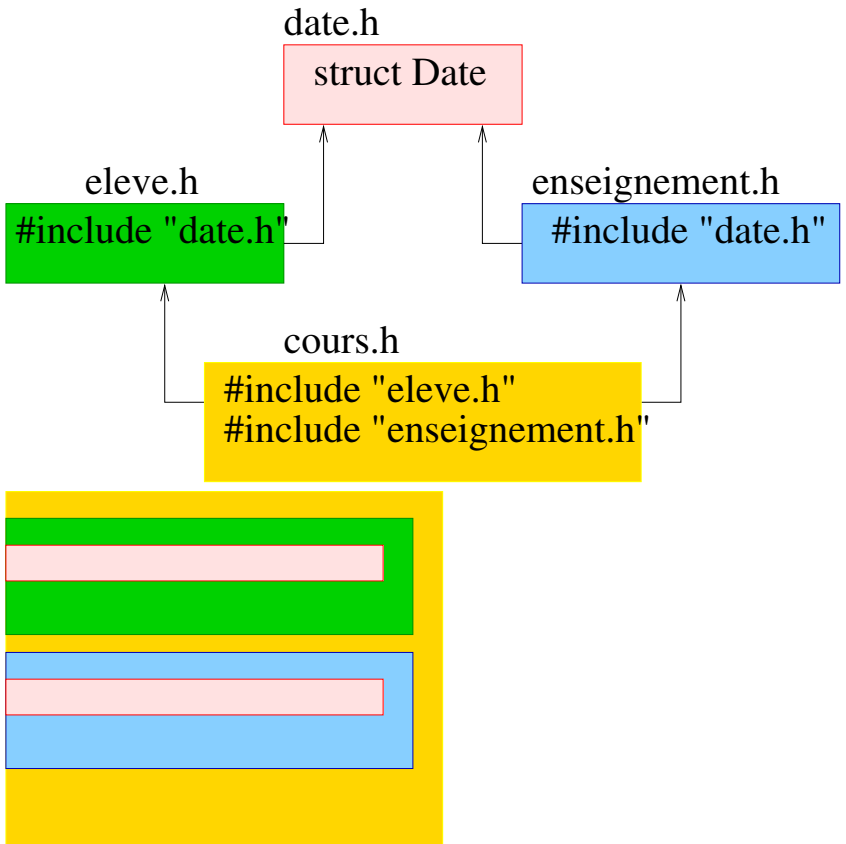
Il faut lier les fichiers `.o` entre eux pour créer l'exécutable final. C'est la phase **d'édition de liens**. Pour cela il faut :

- Ne pas mettre l'option `-c`
- Lister l'ensemble des `.o` nécessaires
- Qu'il n'y ait qu'une unique fonction `main()` dans tous les `.o`
- Donner le nom de l'exécutable après `-o` (sinon, `a.out` est créé)

```
c99 -Wextra -pedantic -c date.c
c99 -Wextra -pedantic -c ephemeride.c
c99 date.o ephemeride.o -o main
```

Remarque : la dernière ligne est l'appel à l'éditeur de lien `ld` (avec les options adaptées au C).

Problème de l'inclusion multiple



`date.h` définit un type `struct Date`, et comme `Cours.h` inclus deux fois `date.h`, il possède deux définitions de `struct Date`, et refuse de compiler...

comment faire ? ? ?

Compilation conditionnelle avec le pré-processeur

Commandes pré-processeur `#ifdef` ou `#ifndef`

Elles permettent de faire de la compilation conditionnelle.

Syntaxe

```
#ifdef <ident>
code-if
#else
code-else
#endif
```

Sémantique

Si la macro `<ident>` a été définie, la partie `code-if` est compilée. Sinon la partie `code-else` est compilée.

La partie `#else` est optionnelle.

`#ifndef` teste si la macro n'a *pas* été définie.

la solution classique : la garde conditionnelle

Chaque fichier `header.h` *commence par les deux lignes*

```
#ifndef HEADER_H
#define HEADER_H
```

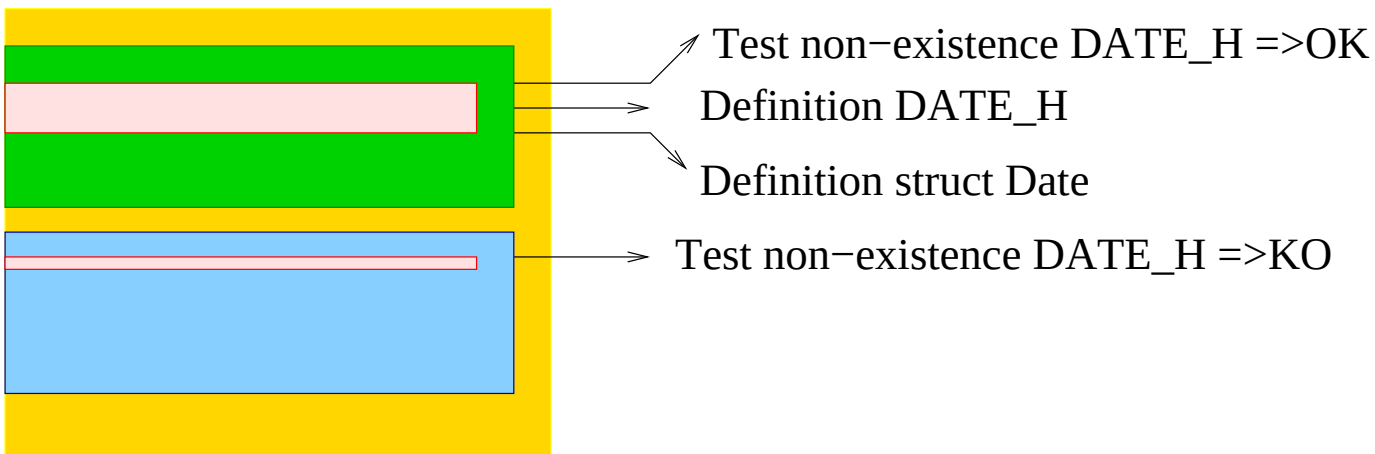
et se termine par

```
#endif
```

Exemple :

```
#ifndef DATE_H
#define DATE_H
struct Date{
   int jour;
   int mois;
   int annee;
};
#endif
```

Avec garde



Rappels d'algorithmique : Modules types et objets, Encapsulation

Familles de modules

- **Les modules objet :** c'est l'abstraction **d'une entité** ou d'un objet.
  Il définit des attributs (i.e variables) et des sous-programmes en mesure de contrôler ces attributs.
- **Les modules type :** c'est l'abstraction d'un **ensemble d'entités** caractérisées par le même type.
  Il définit des types ainsi que des sous-programmes qui sont en mesure de contrôler des variables des types choisis.

Le principe d'encapsulation consiste à

- Cacher les attributs et types d'un module en les déclarant dans le corps.
- A ne manipuler les attributs (module objet) ou les entités du type (module type) uniquement au travers des sous-programmes (ou services) définis dans l'interface.

Problèmes du C

Visibilité des variables et fonctions en C

Par défaut,[8] *toutes* les fonctions et variables globales définies dans le corps `module.c` sont visibles.

→ Il faut explicitement rendre une variable ou fonction locale au module.

8. et c'en est un

Limites des modules en C

La propriété **static**

Pour déclarer ou définir une variable ou une fonction *locale* à un module, on la fait précéder du mot clef **static**.

Attention : on ne peut pas rendre un type `static`

Fichier "Static.h"

```
#ifndef _Static_h
#define _Static_h

// Unique fonction
// visible par
// les autres modules
int f();

#endif
```

Fichier "Static.c"

```
#include "Static.h"
// fonction locale au module Static
static int max(int a, int b){
  if (a>b){
    return a;
  } else {
    return b;
  }
}
// fonction f() presente dans le .h,
// visible par les autres modules
int f(){
  int val1 =2, val2=8;
  int maxi = max(val1, val2);
  return maxi;
}
```

Identifiants des sous-programmes

L'éditeur de liens n'accepte pas que plusieurs entités d'une même application portent le même identifiant.

Illustration :

```
kjaffres@ader:~/CoursC$ c99 ephemeride.o ephemeride.o date.o -o ephemeride
duplicate symbol _main in:
    ephemeride.o
ld: 1 duplicate symbol for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

Création de modules type et objet

Création d'un module type en C

Le module `date` est un module type. Dans sa forme, le principe d'encapsulation n'a pas été respecté car le type Date est visible dans l'interface.
Pour encapsuler le type, on doit **définir le type Date comme un pointeur sur un Struct Date**, et déplacer la définition du struct Date dans le corps du module.

Module type encapsulé en C

Interface du module date_encap

```
// Inclusion des bibliothèques nécessaires à l'interface
#include <time.h>

// Encapsulation du type : on definit un pointeur dans l'interface
typedef struct Date* Date;

// Initialise une date. Elle vaut alors Jeudi 01/01/1970.
void initialiser(Date* date);
// Retourne la date d'aujourd'hui
Date date_aujourd_hui();
// Affiche dans stdout la date d'aujourd'hui au format d.jour/(d.mois+1)/d.anne
void afficher_date(Date d);
// Convertit la date au format time_t de time.h en une date de type Date
void convertir_vers_date(time_t t, Date* date);
// Retourne le nombre de jours qui separent d1 de d2
int nbJoursDeDifference(Date d1, Date d2);
// Retourne de numero de la semaine de la date en parametre
int numeroSemaine(Date d);
// Retourne la date du lendemain connaissant la date d en parametre
```

Module type encapsulé en C

Corps du module date_encap

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

....
// Definition des types internes - Encapsulation
enum NomJour { DIMANCHE, LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI};
enum Mois {JAN, FEV, MAR, AVR, MAI, JUIN, JUIL, AOUT, SEPT, OCT, NOV, DEC };
typedef enum NomJour NomJour;
typedef enum Mois Mois;
struct Date{
   // Invariant :      jour>=1 && jour<=31; annee>0
   int jour;
   NomJour nomJour;
   Mois mois;
   int annee;
};
```

Note : Les types `enum NomJour`, `NomJour`, `enum Mois`, `Mois` et `struct Date` ne sont pas visibles des modules appelant.

Module type encapsulé en C

Corps du module date_encap (Suite)

```
// Attention, ici Date est un pointeur sur un struct Date !
void initialiser(Date *date){
   //Encapsulation : allocation dynamique ici
   *date = malloc(sizeof(struct Date));
   assert(*date);
   (*date)->jour = 1;
   (*date)->nomJour = JEUDI;
   (*date)->mois = JAN;
   (*date)->annee = 1970;
}
void convertir_vers_date(time_t t, Date* date){
   struct tm * now = localtime(&t);
   (*date)->jour = now->tm_mday;
   (*date)->nomJour = now->tm_wday;
   (*date)->mois = now->tm_mon;
   (*date)->annee = now->tm_year+1900;
}
Date date_aujourd_hui(){
```

Nécessite l'utilisation de l'allocation dynamique de mémoire !

Création de modules type et objet

Création d'un module Objet en C

La mise en oeuvre d'un module Objet passe la définition d'une variable globale, unique, visible et modifiable par tous les modules ou applications qui incluent le module objet.
Comme vu en algorithmique, il faut encapsuler la variable globale dans le corps. On utilisera la propriété `static`.

Module objet en C

Voici un exemple de module objet qui définit un compteur.

Interface du module compteur

```
1   #ifndef _COMPTEUR_H
2   #define _COMPTEUR_H
3
4   // Specification de la procédure re-initialiser
5   void re_initialiser();
6   // Specification de la procedure incrementer
7   void incrementer();
8   // Specification de la function valeur
9   int valeur();
10
11  #endif
```

Module objet en C

Corps du module compteur

```c
#include <stdio.h>

// Encapsulation : elle est static dans compteur.c
static int compteur=0;

void re_initialiser(){
   compteur=0;
}
void incrementer(){
   compteur ++;
}
int valeur(){
   return compteur;
}
```

Utilisation du module objet en C

Programme de test de compteur

```c
#include "compteur.h"
#include <stdio.h>

int main(){
  // initialiser
  re_initialiser();
  printf("Init Compteur c=%d \n", valeur());
  // incrementer
  incrementer();
  printf("Incrementer c=%d \n", valeur());
  // access au compteur sans appel a valeur()
  incrementer();
  incrementer();
  printf("Acces direct a compteur c=%d \n", compteur);

  return 0;
}
```

Utilisation du module objet en C

Exécution du programme de test

```
Undefined symbols for architecture x86_64:
  "_compteur", referenced from:
      _main in test_compteur-58df5c.o
ld: symbol(s) not found for architecture x86_64
```

Manipuler une variable externe

Propriété `extern`

Il est possible d'utiliser une variable déjà définie dans un module.
Pour cela, il faut la re-déclarer dans son programme en la pré-fixant par le mot-clef `extern`.
Le compilateur sait alors que cette variable existe dans un autre module.

Manipuler une variable externe

Exemple

Ici, on accède à la variable `compteur` du module compteur. Voici la forme du programme principal qui demande l'accès à `compteur` :

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

// acces au compteur de compteur.h
extern int compteur;

int main(){
  // initialiser
  re_initialiser();
  // access direct au compteur du module
  printf("Acces direct a compteur c=%d \n", compteur);
  return 0;
}
```

Attention : Cette opération n'est possible que si la variable `compteur` n'a pas été déclarée `static` dans `compteur.c`

Rappel : Généricité en Algorithmique

Définition en algorithmique

C'est la spécification et la réalisation de sous-programmes et modules qui travaillent avec des types abstraits (symboliques).

- Il faut après créer une instance de ces modules ou sous-programmes génériques avec des types concrets (entier, enregistrement, etc).
- Et les utiliser dans une application

Généricité en C

Pas si simple que cela...

La généricité n'a pas été prévue dans C. Tout comme la modularité. Donc pour y arriver, il faut utiliser des moyens détournés et souvent, écrire des scripts spécialisés pour instancier un module générique.

Nous n'irons pas très loin dans ce cours par manque de temps. Seules les grandes étapes sont présentées.

Généricité en C : quelques solutions

1. Masquer le type avec `(void *)`
2. Utiliser des macros pré-processeur.
3. Instancier statiquement (manuellement) le module pour le type réel voulu.
   - Définir le type réel dans `typedef t_reel t_abstrait` : remplacer `t_reel` par `int` par exemple
   - En pré-fixant tous les sous-programmes par le nom du type réel si on veut utiliser plus d'une instance du même module en même temps.
   - En spécialisant les fonctions d'affichage / lecture du type réel.
   - En incluant la bibliothèque qui définit le type réel si besoin.

Généricité en C

Petit exemple

Fonction générique

```
typedef t_reel t_gen; // definition du type generique
// permuter deux elements
void permuter(t_gen* X, t_gen* Y){
  t_gen tampon;
  tampon = *X;
  *X = *Y;
  *Y = tampom;
}
```

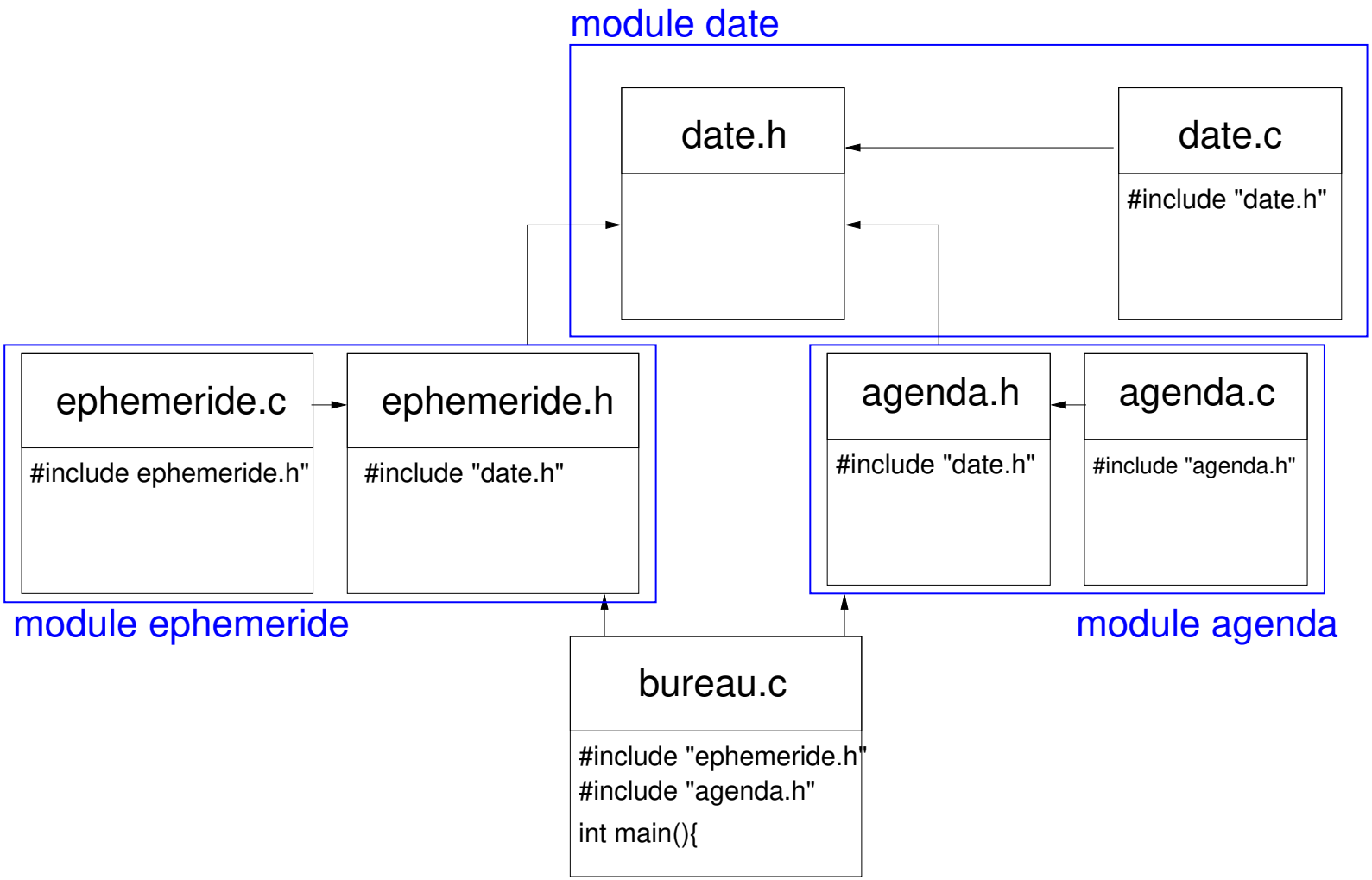
Instanciation pour un type réel double

```
typedef double t_gen; // definition avec le type réel
// instancier : pre-fixer avec le type reel
void double_permuter(t_gen* X, t_gen* Y){
  t_gen tampon;
  tampon = *X;
  *X = *Y;
  *Y = tampom;
}
```

L'outil Make

Ou comment automatiser la compilation d'une application

Exemple d'application 'bureau'



Exemple d'application 'bureau'

Pour compiler cette application

- On compile tout en une seule ligne de commande :

```
c99 bureau.c agenda.c date.c ephemeride.c -o bureau
```

- On compile séparément :
  1. On compile tous les fichiers .c pour obtenir des fichiers objets .o :

```
c99 -c bureau.c
c99 -c date.c
c99 -c agenda.c
c99 -c ephemeride.c
```

ou [9]

```
c99 -c *.c
```

  2. On lance la phase d'édition des liens :
     (i.e. on génère l'exécutable bureau à partir des fichiers objets).

```
c99 agenda.o date.o ephemeride.o bureau.o -o bureau
```

ou

```
c99 *.o -o bureau
```

9. Compile tous les fichiers .c présents dans le répertoire courant

Automatisation de la compilation

`make`

Il est possible d'automatiser la compilation des fichiers d'une application à l'aide de l'outil `make`.

- La commande `make` :
  1. Recherche un fichier dénommé `makefile ou Makefile` dans le répertoire courant
  2. Construit la *première règle* du fichier `makefile ou Makefile`
- Il est possible de lui spécifier un nom de fichier différent, ou d'exécuter plusieurs Makefile avec l'option `-f` :

```
make -f mon_makefile
```

Structure du fichier Makefile

Un fichier Makefile contient une liste de *règles*.

Définition d'une règle

Une règle se compose du nom du fichier exécutable qui sera généré par `make`, que l'on nomme `cible`, des dépendances et d'une commande :

```
nom_cible1: dependances_regle1
[TAB] commande_regle1

nom_cible2: dependances_regle2
[TAB] commande_regle2
```

Une dépendance liste tous les fichiers nécessaires à la création de la cible.

- Attention : Pas d'espace entre `nom_cible1` et les deux points.
- Pour générer la cible `nom_cible1` (et l'exécutable du même nom), on tape la commande :

```
make nom_cible1
```

Exemples de fichiers Makefile

Pour la compilation de l'application `bureau`.

```
bureau: date.c ephemeride.c agenda.c bureau.c date.h ephemeride.h agenda.h
    c99 date.c ephemeride.c agenda.c bureau.c -o bureau

clean :
    rm bureau
```

- Il y a deux cibles dans le fichier `Makefile` : `bureau` et `clean`.
- La création de la cible `bureau` nécessite les fichiers .c et .h de tous le modules inclus dans `bureau.c`.
- La création de la cible `clean` n'a pas de dépendances.

Pour lancer la cible `clean`

```
make clean
```

Question : Que fait-elle ?

Exemples de fichiers Makefile

Pour la compilation séparée de l'application `bureau`.

```
bureau: date.o ephemeride.o agenda.o bureau.o
    c99 -Wextra -pedantic date.o ephemeride.o agenda.o bureau.o -o bureau
date.o: date.c date.h
    c99 -Wextra -pedantic -c date.c
ephemeride.o: ephemeride.c ephemeride.h date.h
    c99 -Wextra -pedantic -c ephemeride.c
agenda.o: agenda.c agenda.h date.h
    c99 -Wextra -pedantic -c agenda.c
bureau.o: bureau.c agenda.h date.h ephemeride.h
    c99 -Wextra -pedantic -c bureau.c
clean:
    rm *.o bureau
```

Il y a trois types de cibles dans ce fichier `Makefile` :

- Celle qui génère l'exécutable `bureau`
- Celles qui compilent les fichiers .c pour créer des fichiers objects.o
- Celle qui supprime les fichiers .o et `bureau`

Comportement de `Make`

Quand on tape la commande `make` pour la première fois (ou après un `make clean`), voici la sortie écran obtenue :

```
c99 -c date.c
c99 -c ephemeride.c
c99 -c agenda.c
c99 -c bureau.c
c99 date.o ephemeride.o agenda.o bureau.o -o bureau
```

Que se passe-t-il ?

Comportement de Make

Quand on tape la commande `make` pour la première fois (ou après un `make clean`) :

```
c99 -c date.c
c99 -c ephemeride.c
c99 -c agenda.c
c99 -c bureau.c
c99 date.o ephemeride.o agenda.o bureau.o -o bureau
```

Que se passe-t-il ?

1. `make` recherche la première cible.
2. Pour créer la cible `bureau`, il a besoin du fichier Date.o. Il recherche alors la cible qui lui permet de la construire et exécute la commande correspondante.
3. Il fait de même pour toutes les dépendances de la cible `bureau` (càd `Ephemeride.o, Agenda.o, bureau.o`), puis il construit la cible `bureau`.

Comportement de Make

Quand on tape la commande `make` encore une fois, on obtient :

```
make: `bureau' is up to date.
```

Rien ne se passe car toutes les cibles sont à jour.

Comment sait-il que la cible est à jour ?

Comportement de Make

Quand on tape la commande `make` encore une fois, on obtient :

```
make: `bureau' is up to date.
```

Rien ne se passe car toutes les cibles sont à jour.

Comment sait-il que la cible est à jour ?
Il vérifie qu'aucune dépendance (fichier .c ou .h listé dans les dépendances) n'est plus récent que la cible.

Comportement de Make

On modifie le fichier `date.c`. On tape `make` et on obtient :

```
c99 -c date.c
c99 date.o ephemeride.o agenda.o bureau.o -o bureau
```

Deux cibles ont été re-générées : `Date.o` et `bureau`.

Conséquence

Si on change un des fichiers listés dans une dépendance, on re-créé uniquement les cibles qui en dépendent.

Q : La règle `clean` n'a pas de dépendances. Quand est-elle appliquée ?

Variables et commentaires

Dans le `Makefile`

- Il est possible d'insérer des commentaires après le caractère dièse ♯
- On peut positionner des variables pour rendre un `Makefile` plus générique

```
#Je suis un commentaire qui vous explique que la variable CC stocke le
#nom du compilateur
CC=c99
#Et les variables CFLAGS et LDFLAGS comportent les options de compilation
#et d'edition des liens
CFLAGS=-Wextra -pedantic -c
LDFLAGS=-Wextra -pedantic
#Voici une utilisation des variables CC et CFLAGS
bureau: date.o ephemeride.o agenda.o bureau.o
    $(CC) $(LDFLAGS) date.o ephemeride.o agenda.o bureau.o -o bureau
date.o: date.c date.h
    $(CC) $(CFLAGS) date.c
ephemeride.o: ephemeride.c ephemeride.h date.h
```

Variables et commentaires

Principales variables

- `CC` : le nom du compilateur
- `CFLAGS, LDFLAGS` : les options de l'étape de compilation et d'édition des liens
- `SOURCES` : liste les fichiers sources .c
- `OBJECTS` : liste les fichiers objets .o
- `EXECUTABLE` : le nom de la cible a générer

Variables automatiques

- `$@` : le nom de la cible de la règle courante
- `$<` : le nom du premier fichier dans les dépendances
- `$^` : le nom de tous les fichiers (séparés par un espace) listés dans les dépendances de la règle courante
- `$?` : le nom de tous les fichiers listés dans les dépendances et qui sont plus récents que la cible courante

Règle implicite %.o %.c

En C

Transformer un fichier .c en fichier objet .o est très courant.
Il est possible d'écrire toutes les règles suivant ce modèle avec une seule règle récurrente :

```
clean :
	rm *.o $(EXECUTABLE)
```

De plus, la plupart du temps, `make` est configuré pour connaitre la règle `%.o: %.c` de toute façon en C, qu'elle soit écrite ou non !
→ Même pas besoin de l'écrire !

Utilisation des ces variables et règles

```
CC=c99
CFLAGS=-Wextra -pedantic -c
LDFLAGS=-Wextra -pedantic
SOURCES=date.c ephemeride.c agenda.c bureau.c
OBJECTS=date.o ephemeride.o agenda.o bureau.o
EXECUTABLE=bureau

#Regles
All : $(EXECUTABLE) $(SOURCES)

$(EXECUTABLE): $(OBJECTS)
	$(CC) $(LDFLAGS) $(OBJECTS) -o $@
clean :
	rm *.o $(EXECUTABLE)
depend:
	makedepend $(SOURCES)
```

L'outil `makedepend`

Pour des applications de grande taille, où plusieurs modules sont inclus à différents niveaux, il est parfois fastidieux de lister tous les .h dans le fichier makefile.

☺☺ Pour se simplifier la vie, il existe la commande `makedepend` ☺☺

```
makedepend - create dependencies in makefiles
```

Syntaxe de la commande

```
makedepend sourcefile
```

- Traite par défaut le fichier `makefile` ou `Makefile`.
- Rajoute à la fin du fichier `Makefile` toutes les dépendances relatives au fichier source `sourcefile`.

L'outil `makedepend` : Exemple

La commande

```
makedepend *.c -Y.
```

Rajoute les lignes suivantes à la fin du fichier `Makefile` :

```
# DO NOT DELETE

agenda.o: agenda.h date.h
date.o: date.h
ephemeride.o: ephemeride.h date.h
bureau.o: agenda.h date.h ephemeride.h
```

L'outil `makedepend` : Utilisation courante

Règle dédiée dans le Makefile

Il est d'usage de rajouter une règle dédiée à l'utilisation de `makedepend` dans le fichier `Makefile` :

```
depend:
	makedepend date.c ephemeride.c agenda.c bureau.c
```

Ainsi, si l'on souhaite mettre à jour les dépendances, il faut simplement lancer la commande

```
make depend
```

avant de lancer la commande `make`

Conclusion sur `make`

Utilisation

Il faut savoir :

- Comprendre un `Makefile` et l'utiliser
- Créer un `Makefile` avec des règles explicites
- Modifier un `Makefile` avec des règles implicites et des variables pour l'utiliser

Nota Bene

`make` peut s'utiliser avec bien d'autres langages de programmation, pour archiver des données, installer des application, etc ...

Attention !

- Bien respecter la position des tabulations et espaces
- Ne pas oublier `makedepend`

Derniers éléments de C

Miscellaneous

Points abordés

- Arguments de la ligne de commande
- Les pointeurs de fonctions
- Les fichiers
- Entiers non signés
  - Le type `size_t`
  - Décalage bit à bit
- Les Macros du pré-processeur
- Tableaux et types énumérés

Le programme et son environnement

L'environnement d'exécution

Le programme C s'exécute dans un environnement et peut interagir avec celui-ci.
Pour cela, il existe principalement trois modes pour interagir :

- Les entrées/sorties `printf, scanf ...`
- Les arguments de la ligne de commande
- Ecriture et lecture de fichiers

Les arguments de la ligne de commande

Il est possible de prendre en compte des paramètres donnés par l'utilisateur au moment de l'appel du programme.
Ceci permet de paramétrer le comportement de l'exécutable au moment de son lancement.
On appelle ces paramètres les *arguments de la ligne de commande*

Syntaxe complète du programme principal en C

```
int main(int argc, char *argv[])
```

Il a des arguments en entrée :

- `argc` le nombre d'arguments
- `argv` tableau de chaîne de caractères qui contient les arguments `argv[0]` contient toujours le nom de l'exécutable

et une valeur de retour. Il existe deux valeurs standard : `EXIT_SUCCESS` et `EXIT_FAILURE` (définies dans `stdlib.h`)

Les arguments de la ligne de commande

Exemple d'utilisation

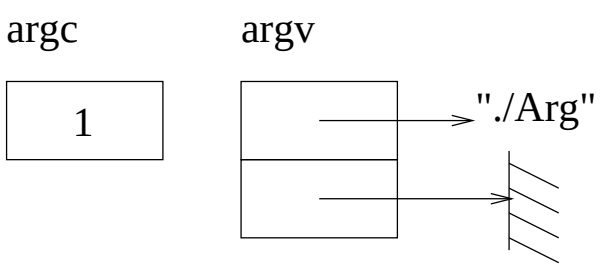
Affichage des arguments de la ligne de commande à l'écran (fichier "Arg.c") :

```
1  #include <stdio.h>
2  int main(int argc, char *argv[]){
3    for(int i= 0 ; i < argc ; ++i)
4      printf("%s ",argv[i]);
5    putchar('\n');
6    return 0;
7  }
```

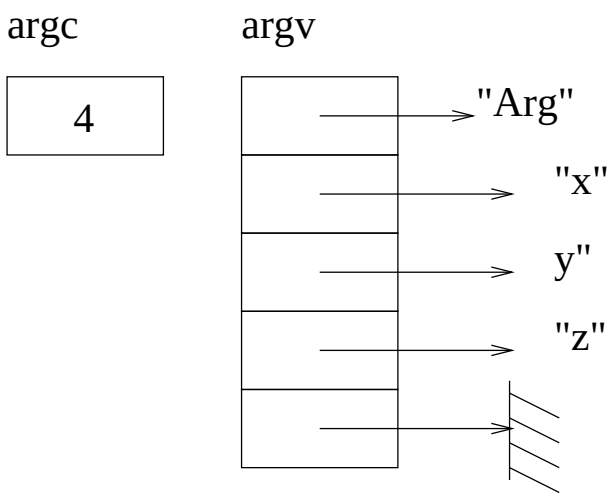
Exemple d'utilisation

Affichage des arguments de la ligne de commande à l'écran : exemple d'exécution

```
CoursC> ./Arg
Arg
```




```
CoursC> ./Arg x y z
Arg x y z
```



Pointeurs de fonctions : Syntaxe

Soit la signature suivante d'un SP :

```
Tret laFonctionPointee(Tpar1 p1, Tpar2 p2, ..., TparN pN);
```

avec

- `Tret` : type de retour
- `Tpar1, ..., TparN` : type des paramètres `p1, ..., pN`

Déclaration d'une variable pointeur sur fonction

```
Tret (*nomPointeur)(Tpar1, Tpar2, ..., TparN);
```

Déclaration d'un type pointeur sur fonction

```
typedef Tret (*nomType) (Tpar1, Tpar2, ..., TparN);
```

Pointeurs de fonctions : Syntaxe

Initialisation du pointeur sur fonction

```
nomPointeur = laFontionPointée;
```

Utilisation du pointeur sur fonction

```
nomPointeur(le_p1, le_p2, ..., le_pN);
```

ou encore

```
*nomPointeur(le_p1, le_p2, ..., le_pN);
```

Pointeur de fonctions : Premier exemple

```
// Une fonction (int*,int)->int
int incrCarre(int* val, int inc){
  *val+= inc*inc;
  return *val;
}

// Declaration variable ptrFct sur fct (int*,int)->int
int (*ptrFct)(int*, int);
// Declaration type modifInt sur fct (int*,int)->int
typedef int (*modifInt)(int*, int);
// Declaration d'une autre variable de type modifInt
modifInt ptrFct2;
```

Pointeur de fonctions : Premier exemple

```
int main(){
  int a= 2, b= 2, c;
  // Initialisation du pointeur ptrFct
  ptrFct= incrCarre;

  // Utilisation de incrCarre() a travers ptrFct
  c= ptrFct(&a, b);
  assert( c == 6 && a == c);
      // -> 2+4

  ptrFct2= incrCarre;
  c= ptrFct2(&b, a);
  assert( c == 38 && b == c);
      // -> 2+36
  return 0;
}
```

Pointeur de fonctions : 2e exemple

Pour réaliser un passage de paramètre d'une fonction, on utilise un pointeur de fonction

```
typedef void (*modifyInt)(int*);
void applyToAll(int tab[], size_t size, modifyInt fct){
/* ou bien
void applyToAll(int tab[], size_t size, void (*fct)(int*) ){ */
  for(int i=0; i<size ; ++i){
    fct(&tab[i]);
  }
}
//Deux fonctions avec une signature (int*)->void
void incrementer(int* x){  (*x)++; }
void cuber      (int* x){   *x = *x * *x * *x; }
```

Pointeur de fonctions : 2e exemple

Pour réaliser un passage de paramètre d'une fonction, on utilise un pointeur de fonction

```
#define TAILLE 5U  // Constante 5 non signee
int main(){
  int values[TAILLE];
  for(int i=0;i<TAILLE;++i){ values[i]=i; } // values= 0,1,2,3,4
  //Utilisation de la fonction incrementer
  applyToAll(values, TAILLE, incrementer);  // values= 1,2,3,4,5
  //Utilisation de la fonction cuber
  applyToAll(values, TAILLE, cuber);        // values= 1,8,27,64,75
  return 0;
}
```

Ecriture et lecture de fichiers

Il est possible de directement lire et d'écrire des données dans des fichiers.

Avertissement

- La gestion fiable (Gestion des erreurs, gestion des caractères accentuées...) des fichiers est un problème difficile.
- Ces quelques transparents sont une introduction.
- Vous trouverez plus de renseignements dans [Buv, §8.2.3.1].
- Si vous devez programmer professionnellement une manipulation de fichier, lisez attentivement la documentation.

Ecriture et lecture de fichiers

- Utiliser `#include <stdio.h>`.
- Pour déclarer une variable qui permet de manipuler un fichier, on utiliser le type `FILE *`

Ouvrir un fichier

```
FILE* leFichier = fopen(char nom[], char mode[]);
```

- Retourne `NULL` en cas d'erreur.
- Le nom contient le chemin relatif ou absolu du fichier dans l'arborescence de l'OS.
- Le mode détermine si le fichier est en lecture seule (`"r"`), en écrire (`"w"`), ou les deux (`"rw"`)
  (cf. [Buv, §8.2.3.1] pour les modes possibles)

Ecritures / lectures formatées

Il est possible d'utiliser des fonction d'écriture et de lecture. Elles peuvent être :

- Formatées : `fprintf`, `fscanf`, `fgets`, `fgetc`, `fputs`, `fputc`

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

  Elles s'utilisent comme leurs équivalents `printf`, `scanf`, ...
- 'Brutes' : on lit bit à bit le fichier : `fread`, `fwrite`.
- Il est possible de remettre un caractère

  `int ungetc (int c, FILE * stream);`
- On peut aussi se déplacer dans le fichier : `fseek`, `ftell`, `rewind`, `fgetpos`, `fsetpos`

Ecriture et lecture de fichiers

Fermer un fichier

```
int fclose(FILE* file);
```

Retourne 0 en cas de succès

Détecter la fin du fichier

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

- Retourne une valeur différente de 0 si la fin du fichier est détectée.
- On ne peut détecter la fin d'un fichier en C qu'après une lecture ayant échoué.
- On peut aussi détecter la fin du fichier avec la valeur de retour des fonctions de lecture (`fgetc()` retourne `EOF`)

Gérer des erreurs

`ferror, clearerr, perror, errno`

Exemple de copie d'un fichier

```c
#include <stdio.h>
#include <assert.h>
int main(int argc, char*argv[]){
  FILE* input;
  FILE* output;
  assert(argc >= 3);
  input= fopen(argv[1], "r");
  assert(input != NULL );
  output= fopen(argv[2], "w");
  assert( output != NULL);
  char c= getc(input);
  /* Copie */
  while ( !feof(input) ){
    putc(c,  output);
    c= getc(input);
  }
  /* Fermeture */
  fclose(input);
  fclose(output);
  return 0;
}
```

Entiers non-signés

Le type `size_t`

On retrouve le type `size_t` dans la déclaration des fonctions d'allocation dynamique (`malloc, realloc, calloc`) entre autres.

- C'est un `unsigned int`
- Rappel : quand un signé et un non signé sont dans une même expression, le signé est souvent convertit en non-signé
- La conversion de fait modulo 2^N, avec N la taille en bit du type

Conséquence : la constante -1 est le plus grand entier non signé représentable par un type. On écrit souvent :

```
unsigned int max= -1;
```

en lieu et place de

```
# include <limits.h>;
unsigned int max= UINT_MAX ;
```

Entiers non-signés

On utilise des entiers non-signés pour réaliser des manipulations binaires.

Décalage bit à bit (rappel)

L'expression `entierNS << N` décale de N bits vers la gauche la représentation binaire de l'entier non-signé `entierNS`.

```
res1 = (1<4)|5;       // vaut 5
res2 = res1 << 1;     // decallage gauche de 1 bit
res3 = res1 << 2;     // decallage gauche de 2 bits
```

- Les éléments qui sortent à gauche sont perdus,
- On rajoute des 0 à droite.

L'expression `entierNS >> N` décale de N bits vers la droite la représentation binaire de l'entier non-signé `entierNS`.

- Les éléments qui sortent à droite sont perdus,
- On rajoute des 0 à gauche

Les macros : #define

C'est une substitution avec paramètres réalisée par le pré-processeur.

Syntaxe

```
#define <ident>(<param> (, <param> )*) <subst.>
```

Sémantique : désormais, chaque fois que le pré-processeur rencontre `<ident>(P1,...,PN)`, il le remplace par la valeur de substitution en remplaçant les `<param>` par le `Pi` correspondant

Exemple

```
#define free(ptr) free_purify(ptr)
```

Remplace toutes le occurrences de `free` par `free_purify`.

Tableaux et type énumérés en C

- Les indices des tableaux sont toujours entiers, commençant par 0.
- Mais les types énumérés sont implicitement convertis en `int` et commencent à 0.
- Ce qui permet en fait d'utiliser les `enum` comme indices de tableaux.

Tableaux et types énumérés en C : Exemple

```
3
4   int main(){
5     typedef enum{Maths, Info, Reseaux, Telecoms} groupMatieres;
6     int coefMatiere[4] = {3, 2, 4, 8};
7     char cMat;
8     printf("Entrer la matiere (M)aths, (I)nfo, (R)eseaux, (T)elecoms\n");
9     scanf("%c",&cMat);
10    switch(cMat){
11    case 'M':
12       printf("La matiere a un coef de %d\n", coefMatiere[Maths]);
13       break;
14    case 'I':
15          printf("La matiere a un coef de %d\n", coefMatiere[Info]);
16       break;
17    case 'R':
18          printf("La matiere a un coef de %d\n", coefMatiere[Reseaux]);
19       break;
20    case 'T':
21          printf("La matiere a un coef de %d\n", coefMatiere[Telecoms]);
22       break;
23    default:
24         printf("Erreur\n");
25    }
26    return EXIT_SUCCESS;
27  }
```