

# Compiler un programme C

- Compilation simple

```
$gcc structureProgrammeC.c  
$./structureProgrammeC
```

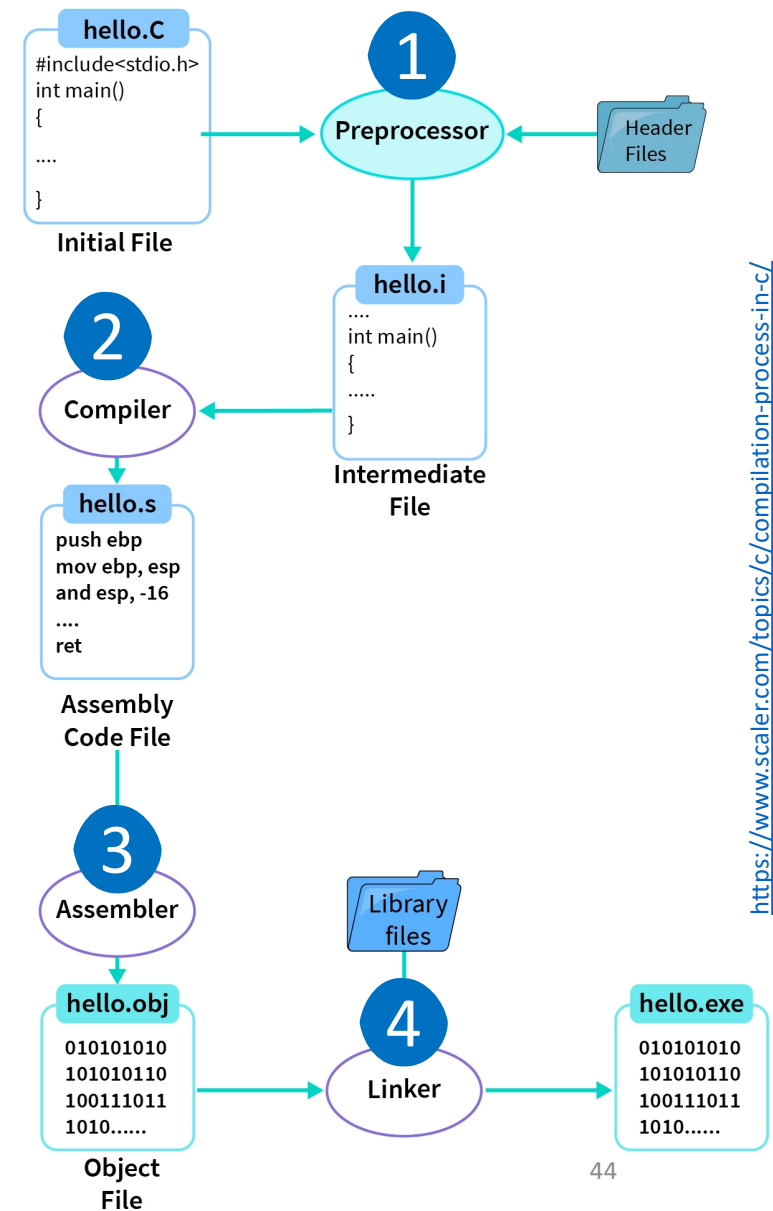
On obtient un exécutable a.out (Windows) ou structureProgrammeC (Linux/MacOS)

- En spécifiant le fichier de sortie

```
$gcc structureProgrammeC.c -o monexecutabledestructureProgramme  
$./monexecutabledestructureProgramme
```

# Les 4 étapes de la compilation C

- Compilation : opération de traduction du code source – langage de haut niveau – en code machine prêt à être exécuté
- Les étapes :
  1. Appel du pré-processeur :
    - prise en compte de toutes les directives de pré-processing (#)
      - include
      - define
    - et suppression des commentaires
  2. Compilation : traduction par le compilateur du langage source en langage d'assemblage correspondant au jeu d'instructions du processeur
  3. Assemblage : obtention à partir du code en assembleur, du programme binaire équivalent, c'est le module objet.
  4. Edition de liens : obtenir le programme binaire (code machine) final à partir de tous les modules objets et libraires compilés et assemblés séparément.



# Pré-processeur

- Modification du texte du fichier source
  - Commentaires ôtés
- Les directives commençant par `#` lui sont adressées
- Inclusion d'autres fichiers sources : `#include`
  - Recopie le contenu d'un fichier dans le fichier courant
  - Si `#include <...>` recherche dans l'include paths standard
  - Si `#include "..."` recherche dans le répertoire courant
- Compilation conditionnelle
  - `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif` `#endif`

# Pré-processeur

- Définition de macros (fonction) ou de symboles (constantes) :  
`#define`

```
#define cube (r) r*r*r  
#define somme (a,b) a+b  
#define volume (r) (4*PI*cube(r))/3
```

```
#define PI 3.14159
```

- Remplacement du nom de la macros (ex : `cube ( r )`) ou du symbole (`PI`) par le code (`r*r*r`) ou le littéral (`3.14`) correspondant dans le corps du code avant compilation

Attention : la valeur assignée via un `define` peut être écrasée et modifiée par un `define` ultérieur.



# Exemple après pre-processeur

```
//Commentaires pour le lecteur du code
int main(int argc, char const *argv[])
{
    int variable = 0;
    return 0;
}
```

simplemainforcompilerexplained.c

```
//Commentaires pour le lecteur du code
#define VALEUR 50
int main(int argc, char const *argv[])
{
    int variable = VALEUR;
    return 0;
}
```

simplemainwithmacrosforcompilerexplained.c

Arrêt après le preprocesseur

Spécification du fichier de sortie

```
$ gcc -E simplemainforcompilerexplained.c -o simpleaprespreprocess.e
$ cat simpleaprespreprocess.e
# 1 "simplemainforcompilerexplained.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 400 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "simplemainforcompilerexplained.c" 2
```

```
int main(int argc, char const *argv[])
{
    int variable = 0;
    return 0;
}
```

```
$ gcc -E simplemainwithmacrosforcompilerexplained.c -o simpleaprespreprocess.e
(base) udc-1-1:simpleC nivet_m$ cat simpleaprespreprocess.e
# 1 "simplemainwithmacrosforcompilerexplained.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 400 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "simplemainwithmacrosforcompilerexplained.c" 2
```

```
int main(int argc, char const *argv[])
{
    int variable = 50;
    return 0;
}
```

Arrêt de la compilation  
après le preprocesseur  
gcc -E fichier.c

Faites le sur votre propre  
machine avec votre  
compilateur, en ligne de  
commande



# Exemple après pre-processeur

```
//Commentaires pour le lecteur du code
#include <stdio.h>

int main(int argc, char const *argv[])
{
    int variable = 0;
    printf("La valeur de la variable est %d\n",variable);
    return 0;
}
```

simplemainwithincludeforcompilerexplained.c

Regardez ce que fait le pré-processeur dans le cas des include...

Arrêt après le preprocesseur

Spécification du fichier de sortie

\$ gcc -E simplemainwithincludeforcompilerexplained.c -o simpleaprespreprocess.e

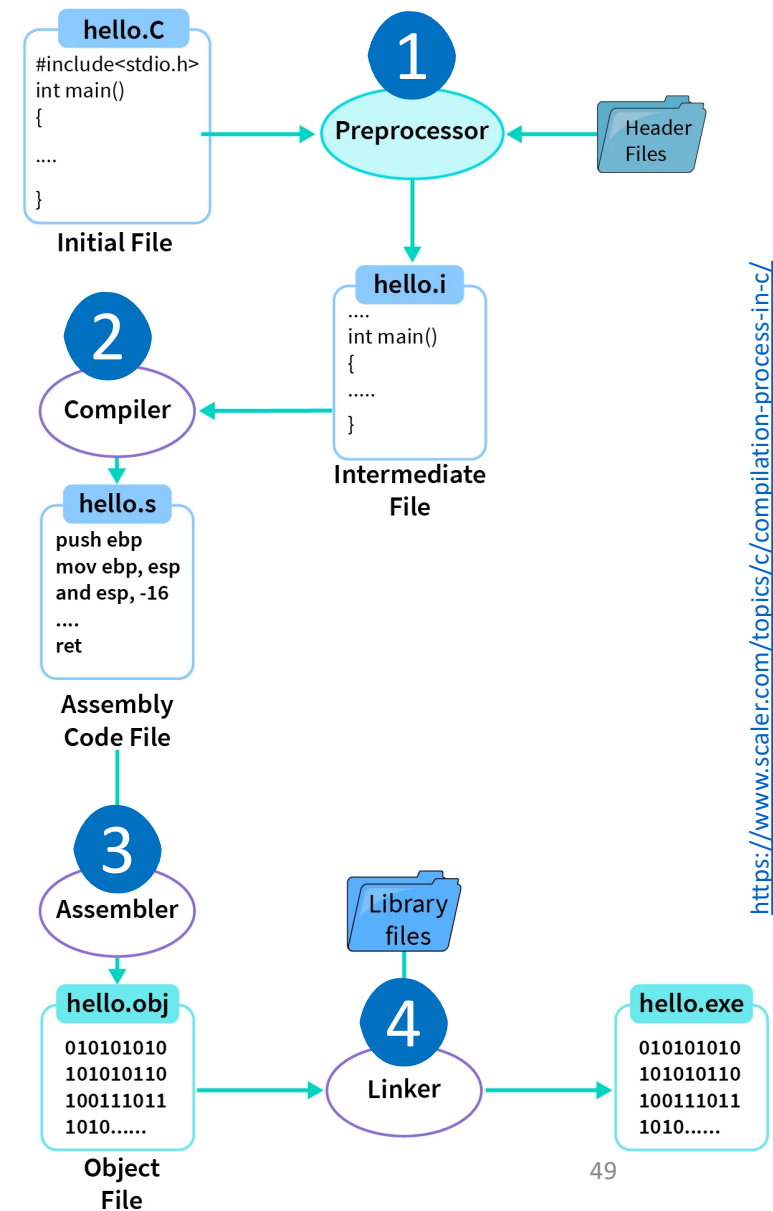
Faites le sur votre propre machine avec votre compilateur, en ligne de commande



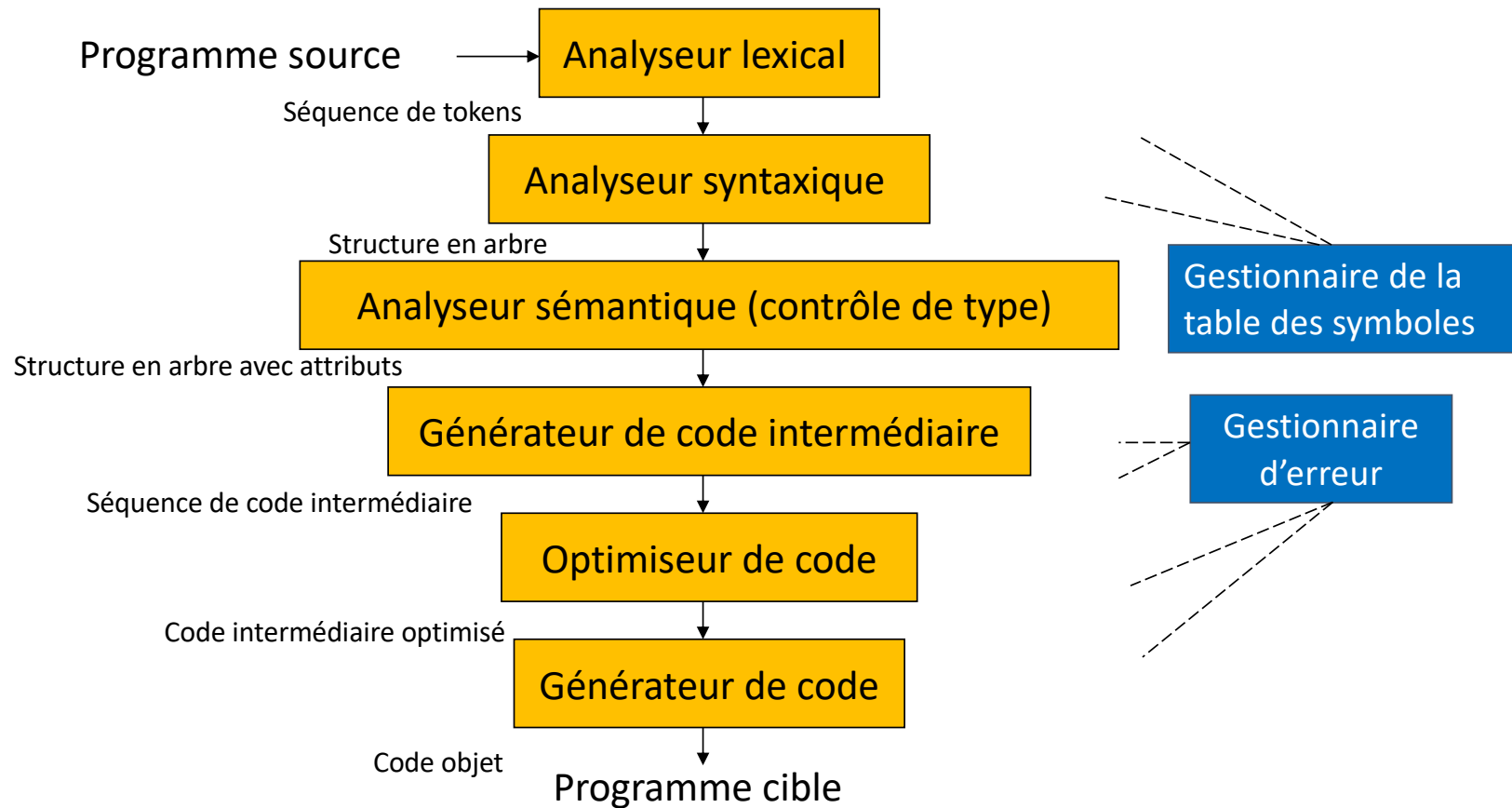
UNIVERSITÀ DI CORSICA  
PASQUALE PAOLI

# Les 4 étapes de la compilation C

- Compilation : opération de traduction du code source – langage de haut niveau – en code machine prêt à être exécuté
- Les étapes :
  1. Appel du pré-processeur :
    - prise en compte de toutes les directives de pré-processing (#)
      - include
      - define
    - et suppression des commentaires
  2. Compilation : traduction par le compilateur du langage source en langage d'assemblage correspondant au jeu d'instructions du processeur
  3. Assemblage : obtention à partir du code en assembleur, du programme binaire équivalent, c'est le module objet.
  4. Edition de liens : obtenir le programme binaire (code machine) final à partir de tous les modules objets et libraires compilés et assemblés séparément.



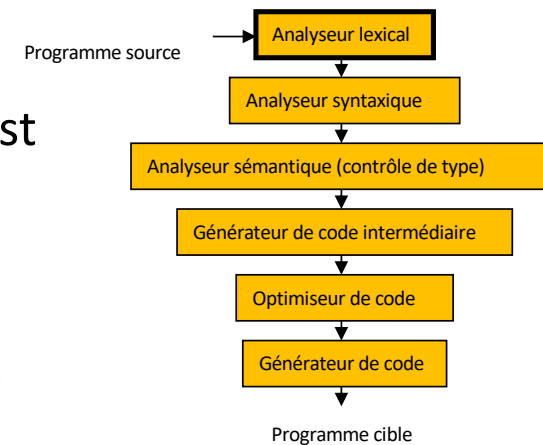
# Comprendre et voir les différentes étapes de la compilation





# Analyseur lexical (scanner)

- Transforme une suite de caractères en une suite d'entités de plus haut niveau
  - Lus de gauche à droite
  - Groupés en unités lexicales lexèmes (*token* : suite de caractères ayant une signification collective)
  - Caractères superflus supprimés
- But de l'analyse
  - Générer des unités lexicales
  - Déterminer si chaque unité lexicale est un mot du vocabulaire
- Résultat
  - Erreur générée si mot non autorisé



# Exemple : analyse lexicale

```
result := somme + test/20
```

Identificateur `result`

Symbole d'affectation

Identificateur `somme`

Symbole d'addition

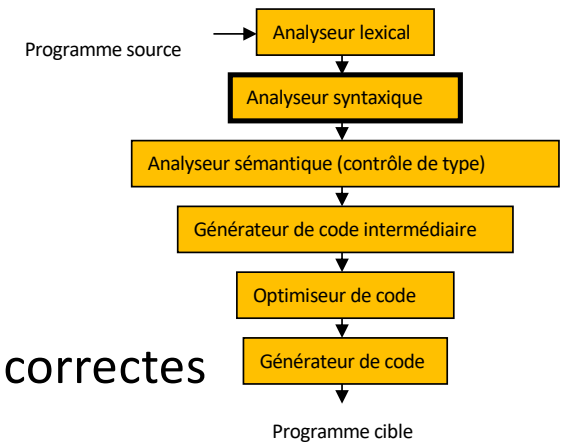
Identificateur `test`

Symbole de division

Nombre 20

# Analyseur syntaxique (parser)

- But de l'analyse
  - Regrouper les unités lexicales en structures grammaticales correctes
  - Déterminer si la syntaxe (grammaire) est correcte
- La grammaire d'un langage est définie par un ensemble de règles (récursives ou non)
- Résultat
  - Construction à partir des tokens d'un arbre syntaxique
  - Erreur générée si structure non valide c'est à dire ne respecte pas la grammaire

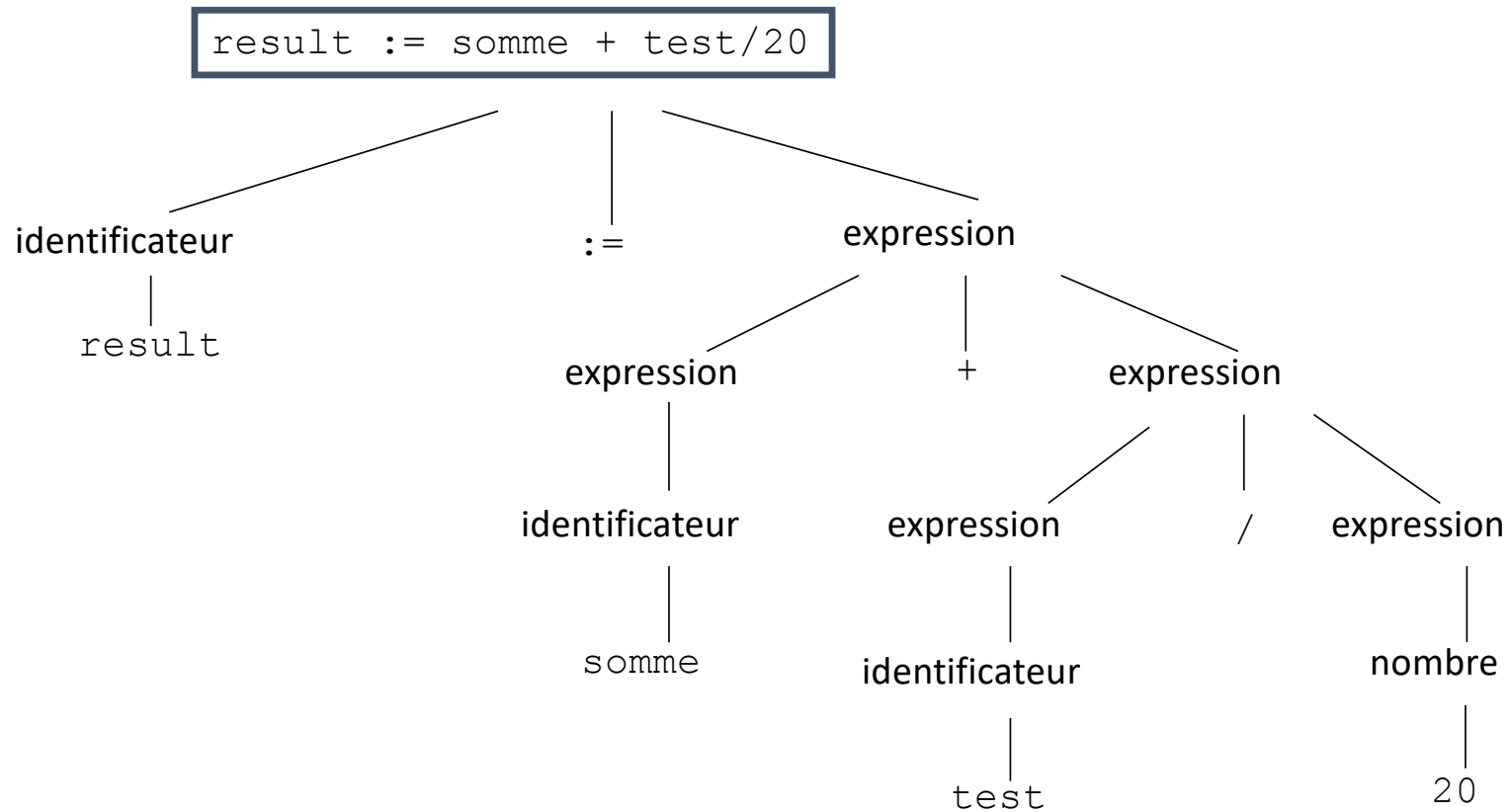


# Analyseur syntaxique (parser)

## Comment ?

- Construction d'arbres d'analyse
- Deux types d'analyses
  - Descendante (racine vers feuilles) ou descendante récursive ou prédictive
  - Ascendante (feuilles vers racine)

# Exemple : analyse syntaxique, Arbre d'analyse ou de dérivation

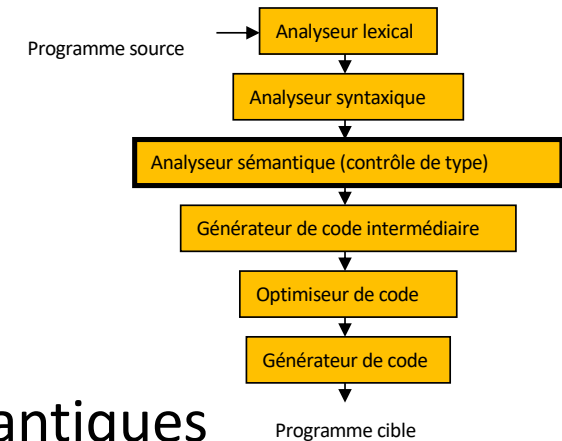


# Table des symboles

- Structure de données
- Stockage d'informations concernant les identificateurs du programme source
  - Type, emplacement mémoire, portée, visibilité, nombre et types des paramètres d'une fonction
- Le remplissage de cette table a lieu lors des phases d'analyse

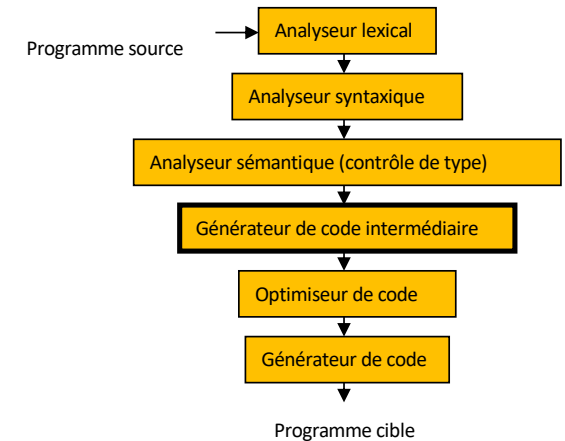
# Analyseur sémantique

- Déterminer si le programme contient des erreurs sémantiques statiques
  - Contrôle de type des identificateurs
  - Portée des identificateurs
  - Unicité des identificateurs
  - Flot d'exécution
    - exemple : utilisation d'un `break` alors qu'il n'y a pas de boucle englobante



# Générateur de code intermédiaire

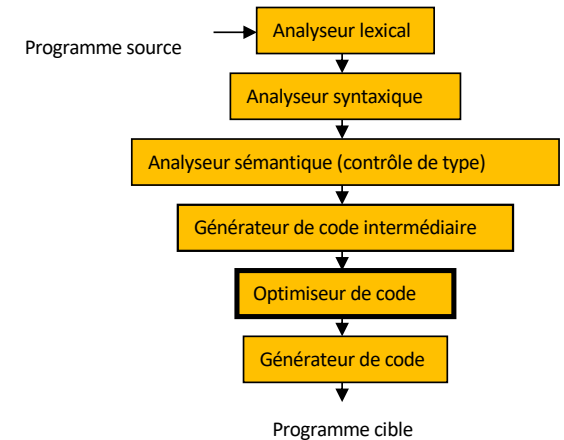
- Le code intermédiaire doit être
  - Indépendante de la machine cible
  - Décrire en détail les opérations à effectuer
- C'est un code exécutable d'une machine abstraite
- Exemple de code intermédiaire
  - AST : Abstract Syntax Tree
  - TAC : Three-Address Code (Code à trois adresses)
    - Au plus un opérateur, trois opérandes, une affectation





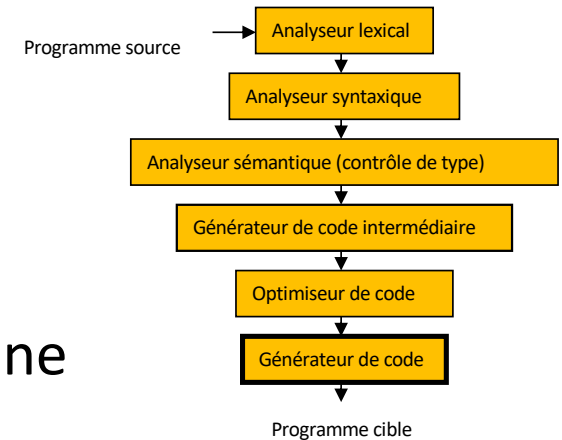
# Optimisation de code

- Amélioration du code intermédiaire suivant
  - Le temps : réduire la durée d'exécution
  - La mémoire : occupation à minimiser
- Les deux étant incompatible il faut faire des compromis...



# Production de code

- Code cible en langage assembleur ou machine
- Le code engendré peut
  - Être exécutable
  - Nécessiter une édition de liens avec des librairies
- Phase dépendant de la machine sur laquelle tournera le programme



# Exemple, assembleur généré

Arrêt de la compilation après génération de l'assembleur  
Obtenu via `gcc -S simplemain.c`

```
int main(int argc, char const *argv[])
{
    int variable;
    variable=3;
    return 0;
}
```

simplemain.c

Faites le sur votre propre  
machine avec votre  
compilateur, en ligne de  
commande

Instruction du processeur

Étiquette (:) nom symbolique @ mémoire

Directive donnée à l'assembleur (.)

```
$ gcc -S simplemain.c
$ cat simplemain.s
        .section      __TEXT,__text,regular,pure_instructions
        .build_version macos, 12, 0      sdk_version 12, 3
        .globl _main                      ; -- Begin
function main
        .p2align      2
        _main:                                     ; @main
        .cfi_startproc
; %bb.0:
        sub          sp, sp, #32
        .cfi_def_cfa_offset 32
        mov          x8, x0
        mov          w0, #0
        str          wzr, [sp, #28]
        str          w8, [sp, #24]
        str          x1, [sp, #16]
        mov          w8, #3
        str          w8, [sp, #12]
        add          sp, sp, #32
        ret
        .cfi_endproc
; -- End function
        .subsections_via_symbols
```

# Exemple

test\_Asembleur.c

```
int t[5];

int somme () {
    int s = 0;
    int i;
    for (i=0; i<10; i++)
        s = s + t[i];
    return s;
}
```

Faites le sur votre propre machine avec votre compilateur, en ligne de commande

Arrêt de la compilation après génération de l'assembleur  
Obtenu via `gcc -S test_Asembleur.c`

test\_Asembleur.s

```
.file "test_Asembleur.c"
.text
.globl _somme
.def _somme; .scl 2;
.type 32; .endef
_somme:
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    movl $0, -4(%ebp)
    movl $0, -8(%ebp)
L2:
    cmpl $9, -8(%ebp)
    jg L3
    movl -8(%ebp), %eax
    movl _t(,%eax,4), %edx
    leal -4(%ebp), %eax
    addl %edx, (%eax)
    leal -8(%ebp), %eax
    incl (%eax)
    jmp L2
L3:
    movl -4(%ebp), %eax
    leave
    ret
.comm _t, 32, # 20
```

si `i<10 ($9)` on va en L3

Jump to L2, fin de boucle

# Édition de liens

test\_Asembleur.c

```
#include <stdio.h>

int main(int argc, char *argv){
    printf("Bonjour");
    return 0;
}
```

Il faut utiliser l'éditeur de liens pour faire le liens entre le symbole `_printf` et l'adresse mémoire à laquelle on peut trouver son code

Arrêt de la compilation après génération de l'assembleur  
Obtenu via `gcc -S test_Asembleur.c`

test\_Asembleur.s

```
.file
"test_Asembleur_edition_lien.c"
.def    __main; .scl    2;
.type   32;     .endef
.section .rdata,"dr"

LC0:
.ascii "Bonjour \12\0"
.text
.globl __main
.def    __main; .scl    2;
.type   32;     .endef
__main:
    pushl    %ebp
    movl     %esp, %ebp
    subl     $8, %esp
    andl     $-16, %esp
    ...
    ...

    sall     $4, %eax
    movl     %eax, -4(%ebp)
    movl     -4(%ebp), %eax
    call     __alloca
    call     __main
    movl     $LC0, (%esp)
    call     printf
    leave
    ret
.def    _printf; .scl    3;
.type   32;     .endef
```

# L'éditeur de lien

- L'assembleur génère le code objet (.o)
  - `call _printf`
  - `_printf` est un symbole utilisé mais non résolu
- L'éditeur de liens prend tous les fichiers objets (.o) et fabrique l'exécutable (binaire)
  - Il résout les références symboliques entre les fichiers

Vous pouvez observer ce qui se passe en utilisant l'option verbose  
`gcc -v test_Asembleur.c`

# Editeur de liens avec des librairies externes

- Si vous faites appel à des librairies externes

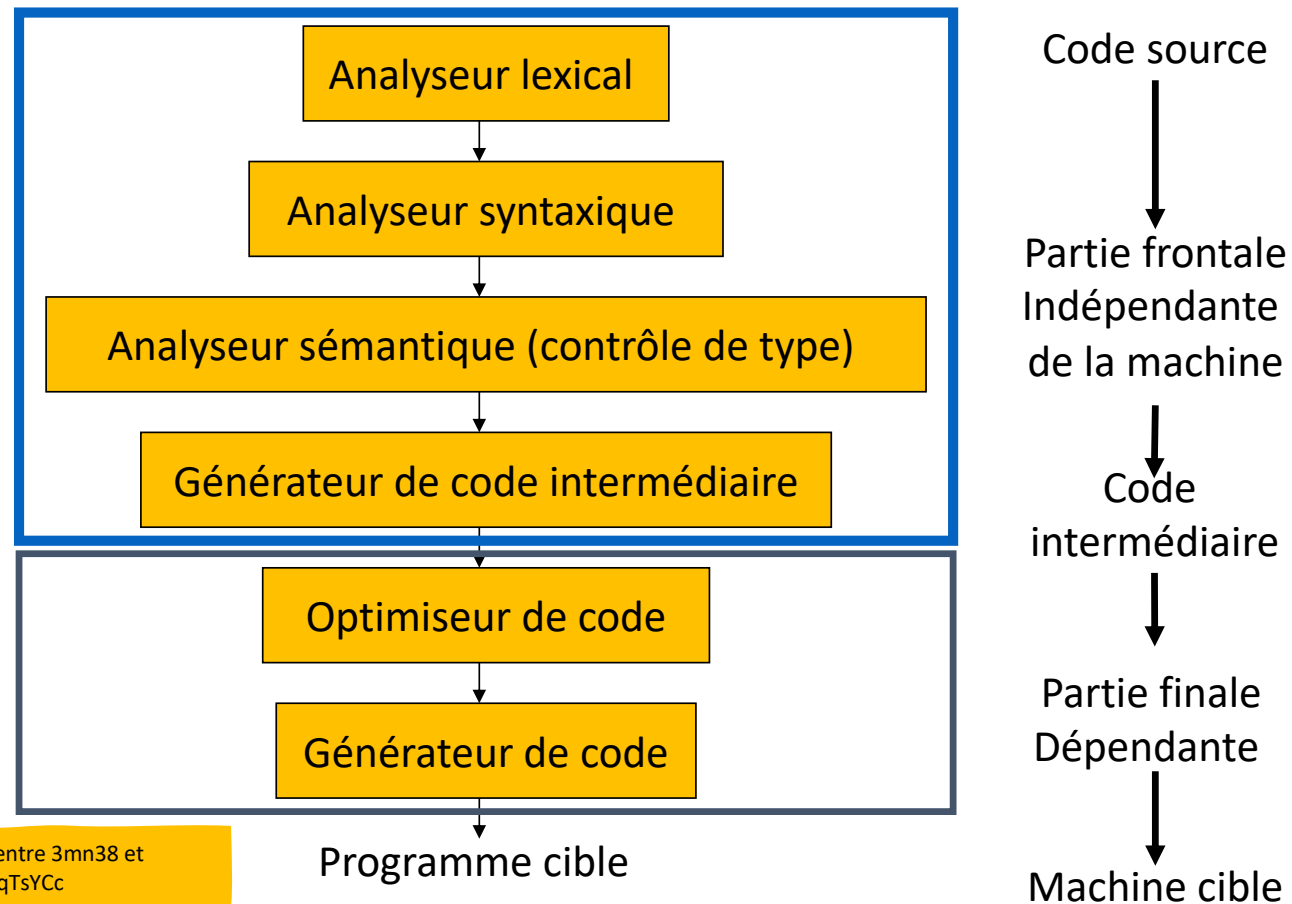
- Exemple mode console :

- ncurses (Linux/Macos/Windows Cygwin)

```
gcc -lncurses jeuModeConsole_0.c -o jeu
```

- windows et conio (Windows)

# Parties frontale (*front-end*) et finale (*back-end*)



How do computer read code ? (Frame of essence) entre 3mn38 et 9mn10 <https://www.youtube.com/watch?v=QXjU9qTsYCc>



# La compilation C

- Le programme compilé dépend du SE et du matériel (architecture processeurs)
- La compilation peut optimiser le programme et favoriser :
  - La vitesse d'exécution
  - L'empreinte mémoire du binaire, etc...

## Exercice :

- Les différentes phases de la compilation
- Les directives de compilation gcc



# Pour réviser un peu...

Connaissez-vous les différentes étapes de ce qu'on appelle la compilation C, leur succession et leur rôle respectif ?

Comment s'adresse-t-on au pré-processeur ?

Qu'est-ce qu'un fichier objet (.o) ?

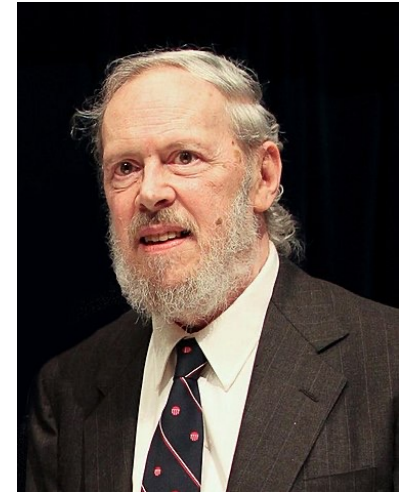
Qu'est-ce que l'éditeur de lien ?



# Historique et filiation de C

Des origines jusqu'à aujourd'hui

# Dennis Ritchie (1941-2011)



Informaticien américain, inventeur du langage C et développeur avec Ken Thompson du système d'exploitation Unix, pour lequel les deux scientifiques reçoivent en 1983 le prix Turing et d'autres récompenses prestigieuses. Pionnier de l'informatique moderne, il passe sa carrière comme employé aux Bell Labs jusqu'à sa retraite en 2007, terminant à la tête du département de recherche sur les logiciels système.

Il est l'auteur avec Brian Kernighan d'un livre référence et célèbre sur le **langage C, qu'il a inventé en 1972**, The C programming language (1978).

Au départ C avait été uniquement pensé pour concevoir un OS : Unix



# Les langages ayant précédé C

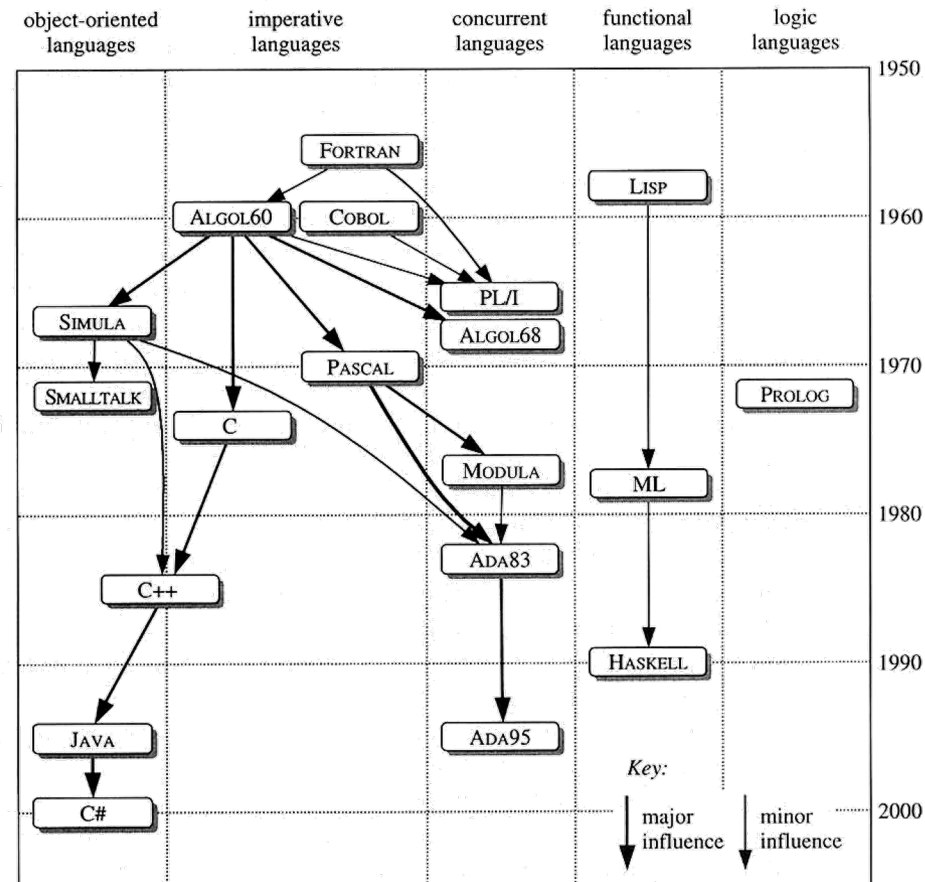


Figure 1.1 Dates and ancestry of major programming languages.

Pour un historique plus complet  
<http://www.levenez.com/lang>

Source :  
« Programming Language  
Design Concepts »,  
David A. Watt, Edition Wiley

# Bonnes pratiques

Discipline de programmation en C mais pas que

- Spécifications
- Code design guide lines, etc.
- Du code, plus propre, plus lisible, plus...



UNIVERSITÀ DI CORSICA  
PASQUALE PAOLI

# Pourquoi suivre/s'imposer de bonnes pratiques ?

En premier lieu pour vous faciliter la vie !!!!!!!

Pas pour : le prof/votre boss/votre collègue/je ne sais qui...



[http://imgs.xkcd.com/comics/future\\_self.png](http://imgs.xkcd.com/comics/future_self.png)



UNIVERSITÀ DI CORSICA  
PASQUALE PAOLI

# Conseils/obligations ? (1)

- Règles de nommage : lisibilité, constance (définition d'une norme, respect d'un standard), pertinence des noms
- Structuration : aération du code, indentation
- Organisation : modularité du code (fichiers/fonctions/procédures/blocs)
  - Soyez cohérent : regroupez des choses ayant un lien dans le même fichier
  - Limitez les dépendances : si vous changez une ligne de code dans le fichier X cela ne doit pas impacter les autres fichiers
- Privilégier les solutions simples et intuitives
- Chercher les conventions et s'il y en a les respecter



Ce n'est pas parce qu'on peut  
le faire qu'on doit le faire !

Curiosité :

Il existe un concours annuel « *The International Obfuscated C Code Contest* » récompensant le code C le plus obscur.

Cf : <https://www.ioccc.org/>

```
#include <math.h>
#include <sys/time.h>
#include <X11/Xlib.h>
#include <X11/keysym.h>

double L, o, P,
, _dt, T, Z, D=1, d,
s[999], e= 0, i,
J, K, w[999], M, m, 0
,n[999], j=33e-3, l=
1E3, r, t, u, v, W, S=
74.5, l=221, X=7.26,
a, B, A=32.2, c, F, H;
int N, q, C, y, p, U;
Window z; char f[52];
; GC k; main(){ Display=
XOpenDisplay( 0); z=RootWindow(e,0); for (XSetForeground(e,k=XCreateGC (e,z,0,0),BlackPixel(e,0))
; scanf("%l%l%l%l",y, "n,w,y, y+s)=1; y ++); XSelectInput(e,z= XCreateSimpleWindow(e,z,0,0,400,400,
0,0,WhitePixel(e,0) ),KeyPressMask); for(XMapWindow(e,z); ; T=sin(0)){ struct timeval G={ 0,dt=1e6};
K= cos(j); N=1e4; M= He_ ; Z=D*K; F=_=P; r=E*K; W=cos( 0); m=K*W; H=K*T; O=D=_=F/ K/d/K/e=_ B=
sin(j); a=B*T=D-E*W; XClearWindow(e,z); t=T-E=D*B*W; j+=d=_=F=E; P=W+E*B-T*D; for (o=(1=D+W+E
+T*B,E=D/K-B+W/K+F*D)+_ , p=y; ){ T=p[s]-1; E=c-p[w]; D=m[p]-L; K=D+m-B*T-H*E; if(p [n]=l p]-p[s
j]= 0)K -fabs(W-T-r-1+E -D*p) /fabs(D+t-D*Z -T-a +E); K=N+1e4; else{ q=W/K +dE2+2e2; C= 2E2+4e2/ K
+D; N-1E466 XDrawLine(e ,z,k,N ,U,q,C); N=q; U=C; } ++p; } L=_= (X+t +P*M+m+l); T=X*X+ l+l*M +M;
XDrawString(e,z,k ,20,380,f,17); D=m/l+15; i=(B +L-M-r -X+Z)+_ ; for( ; XPending(e); u =C5=N){
XEvent z; XNextEvent(e ,z);
+++(N-XLookupKeysym
(z.xkey,0))-IT?
N-L? UP-N? E:6
1:6 ut gh); --(
DN -N? N-DT ?N=
RT?6u: 6 W:6h:6J
); } m=15+F/1;
c=+I+M, l, 1+H
+I+M+X)+_ H
+Arriv+X-F+L+
E=, 1+X+4.9/1, t
-T+m/32-1+T/24
)/5; K=F+M+
hw 1e4/L-(T+
E+S+T+E)/3e2
)/5-X+d-B+A;
a=2.63 /L+d;
X=( deL-T/5
+(-1.0+E +a
+.64+3/1e3
)-M+ v +Ae
2)+_ 1 L +=
K =_ W:d;
sprintf(f,
"%5d %3d"
"%7d",p, l
/1.7, (C+9E3+
0+57.3)%8550, (int)1); d=Tw(.45-14/L+
X-a+130-3e .14)+_ /125e2+F+_v; P=(T+147
+I-m 52+E+04 +D+T+38+ue-21+E) /1e2+W+
179+v)/2312; select(p+0,0,0,66); v=(
W+F-T+(-.63+m-I+.086+m+E+10-D+25-.11+u
)/107e2)+_ D=cos(a); E=sin(a); } }
```

Simulateur de vol gagnant en 1998



UNIVERSITÀ DI CORSICA  
PASQUALE PAOLI

# Conseils/obligations ? (2)

- Commentaires indispensables
  - Faciliter la lecture du code
  - Indiquer des choix de conceptions/résolutions de pb
  - Indiquer les spécifications
- Programmation défensive : augmenter la robustesse du code
  - Codes d'erreur en C

# Conseils/obligations ? (3)

Cours S2 :

- Concepts des Langages
- Qualité logicielle

- DRY : Don't Repeat Yourself !
  - Encapsulez votre code dans une fonction au lieu de le copier/coller partout
- Une fonction = une opération, une fonctionnalité, évitez les usines à gaz
- Validation et tests de code
  - Différences entre résultats attendus (spécifications) et effectifs...
  - Pensez les tests en amont...
  - Test dynamiques / Tests statiques
- Maintenance
  - La vie du code **commence** à l'écriture
  - Suppression du superflu, optimisation, ré-écriture, choix d'un meilleur algorithme, révision des commentaires, etc.

Dynamiques : exécution du code  
Statiques : lecture du code



# Pour le prochain cours...

Proposez par binôme une liste de bonnes pratique que vous vous engagez à respecter dans le cadre de ce module...

Rechercher/proposer une Cheat Sheet C

Déposez votre « manifeste » sous la forme d'un fichier PDF (nom\_prenom1\_nom\_prenom2\_manifesteC.pdf) dans la zone travaux sur l'ENT

Vous veillerez à bien citer vos sources en étayant votre texte par la citation de références précises en Français et en Anglais.

# Spécifications algorithmiques

- Décrire le plus formellement possible le problème qui est censé être résolu par l'algorithme
  1. Les entrées qui caractérisent une instance – un exemple – du problème à résoudre
  2. Une précondition – prédicat – sur les entrées, si elles ne sont pas vérifiées le problème n'a pas lieu d'être
  3. Les sorties qui décrivent une solution au problème
  4. Une postcondition – prédicat – qui doit être vérifié après l'appel de l'algorithme. Si elle n'est pas vérifiée la solution n'est pas correcte

Ces quatre éléments définissent les **spécifications de l'algorithme**.

Ils doivent obligatoirement apparaître dans le code sous la forme de commentaires.

# Exemple de spécifications

Problème : « Factoriser dans  $\mathbb{R}$  le trinôme du second degré  $ax^2 + bx + c = 0$  et  $b^2 - 4ac \geq 0$  »

1. Les entrées sont  $a, b$  et  $c$  réels
2. La précondition est  $\{a \neq 0, b^2 - 4ac \geq 0\}$
3. La sortie est  $x_1, x_2 \in \mathbb{R}$ , tel que  $x_1 =$
4. La postcondition est  $ax_1^2 + bx_1 + c = ax_2^2 + bx_2 + c = 0$

Ces spécifications devront être conservées dans le code et apparaître sous la forme de commentaires



# Code résultant

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

typedef struct
{
    double x1,x2;
}double_pair;

/*
@inputs: double a,b,c polynom parameters
@outputs: double_pair roots;
@precondition: a != 0 and b*b - 4 ac >= 0
@postcondition: ax1*x1 + bx1 + c = ax2*x2 + bx2 + c = 0
@description: compute if there exists the roots of the second order quadratic equation
@author:Nivet ML
@date: 2022-11-01
@version: v1
*/
double_pair compute_roots(double a, double b, double c){
    double delta = b*b - 4*a*c;
    double_pair result;
    if (a!=0){
        if (delta>=0){
            double square_delta = sqrt(delta);
            result.x1 = (-b - square_delta)/(2*a);
            result.x2 = (-b + square_delta)/(2*a);
        }else{printf("Error, precondition isn't respected, 'a' must not be equal to 0");
        }
    }
    return result;
}
```

# Abstractions et paramétrisation



# Variables

- Qu'est-ce qu'une variable ?
  - Espace mémoire réservé et accessible permettant de stocker et utiliser des valeurs
  - Abstraction sur la mémoire

# Comment parler des variables ?

Portée :

Zone de code dans un programme dans laquelle il est possible de faire référence à la variable

Durée de vie :

Temps pendant lequel une variable est présente en mémoire vive

En C la portée est statique, elle dépend uniquement du code source à la compilation



# Nature d'une variable en C ?

- Globale
  - Portée étendue à l'ensemble du programme, durée de vie identique à celle du programme
  - Déclarée en dehors de toute fonction
- Locale
  - Portée limitée à la fonction dans laquelle elle est déclarée, durée de vie identique à celle de la fonction (créée à chaque appel, détruite à la fin de l'exécution de la fonction)
- Locale statique :
  - Sa durée de vie dépasse sa portée

```
void f (int x)
{
    static int compteur = 0;
    ++compteur;
    printf("%d-e appel de la fonction f\n", compteur);
    // Traitements de la fonction
    ...
}
```

Compte le nombre de fois où la fonction f est invoquée. Variable locale à f – inconnue en dehors de f – initialisée uniquement lors du premier appel à f – garde sa valeur d'un appel à l'autre – et détruite à la fin de l'exécution du programme.

# Transparence référentielle ?

- Toute variable, tout appel à une fonction peut être remplacé par sa valeur sans changer le comportement du programme
- En C – en général dans le paradigme impératif – pas de transparence référentielle

```
int i = 0;  
  
int g(int n){  
    i += n;  
    return i;  
}
```

Évaluez :

```
int appel1 = g(1) + g(1);  
int appel2 = 2*g(1);
```

- La présence de variables globales – à proscrire le plus possible ! – rend la transparence référentielle impossible.
- Les calculs dépendent de l'état du système.



# Exercice portée

- Soit le code ci-contre donnez pour chacune des variables la portée de celle-ci (ligne X à ligne Y)

```
1  #include"stdio.h"
2  float v = 50;
3
4  float f(float x)
5  {
6      float i = 2.*x+v;
7      return i;
8  }
9  int main(void)
10 {
11     float val = 4.;
12     float res = f(val);
13     printf("%f",res);
14     return 1;
15 }
```

Variable	Lignes de portée
v	
x	
i	
val	
res	
f	



# Abstraction

## Différence entre expression et commande ?

- Nommage d'expression / Définition de constantes

```
#define TAILLE (elt*10)
...
TAILLE ...
```

- Nommage de commande

```
void f (void) {
    g1 = e1;
    g2 = e2;
}
f();
```

- Nommage de type

```
typedef
```



# Constantes

- Dans le code avec le modificateur `const`
  - Vérification du compilateur qui assure la non modification de la variable

- Pré-processeur `#define`

- Dans le corps du programme, de préférence au début
- Dans un fichier d'entête

Toutes les références à la constante seront remplacées par la valeur dans le code au moment de la compilation

- Constantes énumération, avec `enum`

Le plus lisible et le plus sécurisé demeure l'utilisation du `const`

```
1  #include <stdio.h>
2
3  #define TEST 0;
4  #define AUTRE_CONSTANTE 5;
5  #define LANGAGE_DEV "C";
6
7  int main()
8  {
9      const float UNE_AUTRE = 5.5;
10
11      enum DIRECTION {NORD=100, SUD, EST, OUEST}; // Si on ne donne pas de valeur de départ NORD sera à 0
12      printf("Nord : %d\n", NORD); // affiche 100
13      printf("Sud : %d\n", SUD); // affiche 101
14      printf("Est : %d\n", EST); // affiche 102
15      return 0;
16 }
```



# Types et abstractions de types en C

Types abstraits de données

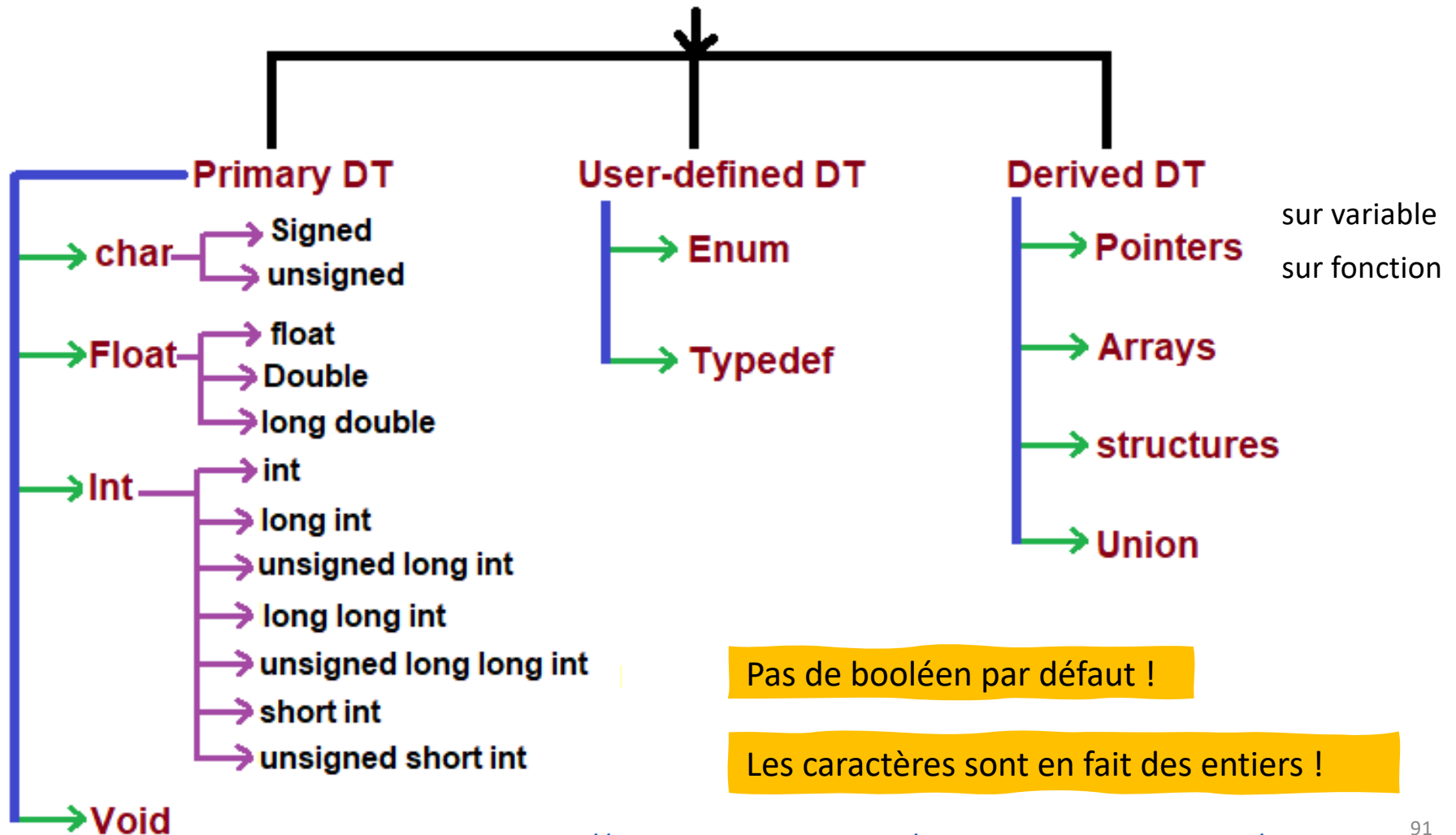
Built-in types ? Types primitifs ? Création de nouveaux types ?

Types composés ?



DT - Data type

# Data Types in C



<https://www.hellocodies.com/data-types-in-c-language/>

# Les types simples/types primitifs (1) : domaine de valeur. Les caractères.

- Pas de véritable type caractère en C.
- Même chose que le type entier.
- char est juste un type entier codé sur 1 Octet.
- Comme tous les autres types entiers, il peut-être signé ou non signé

char	1 byte	-128 to 127	%c
signed char	1 byte	-128 to 127	%c
unsigned char	1 byte	0 to 255	%c

```
char a,b;  
a = 'A';  
b = 'B';  
printf("%d\n", a); //65  
printf("%c\n", a); //A  
printf("%d\n", a+b); //131
```

## Les types simples/types primitifs (2) : domaine de valeur. Les booléens.

Introduit dans C99

- Des booléens, prenant leur valeur dans {true, false}

```
//Entête à inclure  
#include <stdbool.h>  
//Permet d'utiliser bool au lieu de _Bool
```

```
#define bool unsigned int  
#define true 1  
#define false 0  
#define __bool_true_false_are_defined 1
```

stdbool.h

- Attention : toutes les affectations à \_Bool/bool qui ne sont pas 0 (faux) sont stockées comme 1 (vrai)



# Types primitifs (3)

## Entiers et Char

- Les tailles en octets allouées ne sont pas les mêmes sur tous les systèmes en particulier sur les systèmes embarqués...

Tableau 3.2 : contraintes imposées à la représentation des entiers

Nature de l'entier	Contrainte imposée par la norme	Remarque
Non signé	Codage binaire pur	Un entier non signé de taille donnée est donc totalement portable.
Signé	Un entier positif doit avoir la même représentation que l'entier non signé de même valeur.	La plupart des implémentations utilisent, pour les nombres négatifs, la représentation dite du « complément à deux ».

« Complément à deux » :

- on exprime la valeur en base 2 ;
- tous les bits à gauche du premier bit de droite à 1 sont « inversés » : 1 devient 0 et 0 devient 1 ;

4 = 0000 0100

-4 = 1111 1100

Type	Explication courte	Formateurs
<code>char</code>	Plus petite unité adressable d'une machine, elle peut contenir les caractères de base. C'est une valeur entière qui peut être signée ou non.	<code>%c</code>
<code>signed char</code>	Type char signé, capable de représenter au moins les nombres [-127 ; +127].	<code>%c</code>
<code>unsigned char</code>	Type char non-signé, capable de représenter au moins les nombres [0 ; 255].	<code>%c</code>
<code>short</code> <code>short int</code> <code>signed short</code> <code>signed short int</code>	Type entier minimum, court, entier et signé, capable de représenter au moins les nombres [-32 767 ; +32 767].	<code>%i</code>
<code>unsigned short</code> <code>unsigned short int</code>	Type entier minimum, court, idem que le type entier standard non signé.	<code>%hu</code>
<code>int</code> <code>signed</code> <code>signed int</code>	Type entier standard, signé, capable de représenter au moins les nombres [-32 767 ; +32 767].	<code>%i</code> ou <code>%d</code>
<code>unsigned</code> <code>unsigned int</code>	Idem que le type entier standard, mais non signé, capable de représenter au moins les nombres [0 ; 65 535].	<code>%u</code>
<code>long</code> <code>long int</code> <code>signed long</code> <code>signed long int</code>	Type entier long, entier et signé, capable de représenter au moins les nombres [-2 147 483 647 ; +2 147 483 647].	<code>%li</code> ou <code>%ld</code>
<code>unsigned long</code> <code>unsigned long int</code>	Idem type entier long mais non signé, capable de représenter au moins les nombres [0 ; 4 294 967 295].	<code>%lu</code>
<code>long long</code> <code>long long int</code> <code>signed long long</code> <code>signed long long int</code>	Type entier long long, entier et signé, capable de représenter au moins les nombres [-9 223 372 036 854 775 807 ; +9 223 372 036 854 775 807].	<code>%lli</code>
<code>unsigned long long</code> <code>unsigned long long int</code>	Idem type entier long long mais non signé, capable de représenter au moins les nombres [0 ; +18 446 744 073 709 551 615].	<code>%llu</code>

[https://fr.wikipedia.org/wiki/Types\\_de\\_donnée\\_du\\_langage\\_C](https://fr.wikipedia.org/wiki/Types_de_donnée_du_langage_C)

```
int i1,i2;
i1 = 4; i2 = -5;
printf("%d\n",i1+i2);// -1
// Division entre entiers
printf("%f\n",i1/i2);// 0.00
```

# Types primitifs (4)

## Entiers

- Les tailles en octets allouées ne sont pas les mêmes sur tous les systèmes en particulier sur les systèmes embarqués...
- Si vous voulez des tailles fixes utilisez

```
#include <stdint.h>
```

Specific integral type limits

Specifier	Signing	Bits	Bytes	Minimum Value	Maximum Value
int8_t	Signed	8	1	$-2^7$ which equals $-128$	$2^7 - 1$ which is equal to 127
uint8_t	Unsigned	8	1	0	$2^8 - 1$ which equals 255
int16_t	Signed	16	2	$-2^{15}$ which equals $-32,768$	$2^{15} - 1$ which equals 32,767
uint16_t	Unsigned	16	2	0	$2^{16} - 1$ which equals 65,535
int32_t	Signed	32	4	$-2^{31}$ which equals $-2,147,483,648$	$2^{31} - 1$ which equals 2,147,483,647
uint32_t	Unsigned	32	4	0	$2^{32} - 1$ which equals 4,294,967,295
int64_t	Signed	64	8	$-2^{63}$ which equals $-9,223,372,036,854,775,808$	$2^{63} - 1$ which equals 9,223,372,036,854,775,807
uint64_t	Unsigned	64	8	0	$2^{64} - 1$ which equals 18,446,744,073,709,551,615

The limits of these types are defined with macros with the following formats:

- `INTN_MAX` is the maximum value ( $2^{N-1} - 1$ ) of the signed version of `intN_t`.
- `INTN_MIN` is the minimum value ( $-2^{N-1}$ ) of the signed version of `intN_t`.
- `UINTN_MAX` is the maximum value ( $2^N - 1$ ) of the unsigned version of `uintN_t`.

Wikibooks C Programming/stdint.h

[https://en.wikibooks.org/wiki/C\\_Programming/stdint.h](https://en.wikibooks.org/wiki/C_Programming/stdint.h)

## Types primitifs (5)

### Réels

- Les tailles en octets allouées ne sont pas les mêmes sur tous les systèmes
- Problématique de la précision

float	Type flottant. Simple précision (4 octets, ou 32 bits) sur quasiment tous les systèmes.	%f %F %g %G %e %E %a %A
double	Type flottant. Double précision (8 octets, ou 64 bits) sur quasiment tous les systèmes.	%lf %lF %lg %lG %le %lE %la %lA
long double	Type flottant. En pratique, selon le système, de la double précision à la quadruple précision.	%Lf %LF %Lg %LG %Le %LE %La %LA

[https://fr.wikipedia.org/wiki/Types\\_de\\_donnée\\_du\\_langage\\_C](https://fr.wikipedia.org/wiki/Types_de_donnée_du_langage_C)

[illegible]

L'opérateur `sizeof()` retourne la taille en octet de la variable qui lui est passée

```
float f1,j;
i = 4.; j = -5.;
printf("%f\n",i+j);// -1.000000
printf("%f\n",i/j);// 0.800000
```

# Codage des nombres

## 2.2 Codage des nombres

Un `int` est codé sur 32 bits, en base 2 (signe codé par complémentation).  
Donc `x=19` est codé par le mot sur 32 bits :  
000000000000000000000000000010011

On peut jongler avec la représentation binaire, décalage à gauche : `19 << 1` (...100110=38), à droite `19 >> 1` (...1001=9), masquage (`&`, `|`) etc.

Pour les types `float` et `double`, la différence avec les nombres idéaux (les réels dans ce cas), est encore pire, d'une certaine façon : Il s'agit d'un codage en précision finie : la *mantisse* est codée en base 2, l'*exposant* également, et le tout sur un nombre fini de bits (cf. norme IEEE 754).



Attention, à cause de tout cela, et des erreurs d'arrondi dues au nombre fini de bits utilisés pour le codage des nombres, il n'y a pas associativité de l'addition, multiplication etc.

Considérons le programme suivant :

```
float x, y;  
x = 1.0f;  
y = x+0.00000001f;
```

Alors, `x` et `y` ont la même valeur !

Un grand classique (Kahan-Muller) de programme qui donne un résultat surprenant est le suivant :

```
float x0, x1, x2;  
x0=11/2;  
x1=61/11;  
for (i=1; i<=N; i++) {  
    x2=111 - (1130-3000/x0)/x1;  
    x0=x1;  
    x1=x2;  
}
```

Une exécution produit la suite : 5.5902; 5.6333; 5.6721; 5.6675; 4.9412; -10.5622; 160.5037; 102.1900; 100.1251; 100.0073; 100.0004; 100.0000; 100.0000; 100.0000; ... (stable) alors que la vraie limite dans les réels est 6 !

Ou un autre exemple, beaucoup plus simple.

```
float x0, x1;  
x0=7/10;
```

14

CHAPITRE 2. PROGRAMMATION IMPÉRATIVE

```
for (i=1; i<=N; i++) {  
    x1 = 11*x0 - 7;  
    x0 = x1;  
}
```

Ce programme produit la suite 0.7000; 0.7000; 0.7000; 0.6997; 0.6972; 0.6693; 0.3621; -3.0169; -40.1857; -449.0428; -4946.4712; -54418.1836; -598607.0000; -6584684.0000; diverge vers  $-\infty$ , alors que dans les réels c'est la suite constante égale à 0.7 !

# Les pointeurs

Qu'est-ce qu'un pointeur ? Quelles sont ses utilisations ? Quels sont les opérateurs associés ? Qu'est-ce que la valeur NULL ? Qu'est-ce qu'un void \* ? Qu'est-ce qu'une allocation en programmation ? Qu'est-ce qu'une allocation dynamique ? Qu'est-ce qu'une variable statique ? Qu'est-ce qu'une variable dynamique ? Que signifie libérer la mémoire ? Comment et pourquoi la libérer ? Qu'est-ce qu'une adresse mémoire non valide ? Quelles sont les erreurs classiques provoquées par des adresses mémoire non valide ?



# Démo... Test... Expérimentations

- Ouvrez votre IDE/Editeur
- Créez un nouveau fichier pointeurtests.c
- Déclarez un nouveau main
- Amusez-vous !

## Les opérateurs/instructions à connaître

```
type *pt_sur_un_type = NULL; //Déclaration d'un pointeur sur...  
&variable //pour récupérer l'adresse mémoire de variable  
*pt_sur_un_type // pour atteindre l'objet pointé en lecture ou en écriture
```



# Expérimentation, les pointeurs

A sa création en C il est recommandé de mettre le pointeur à la valeur NULL.

```
#include <stdio.h>

int main(int argc, char const *argv[]){
    int *pt_int; // i pointeur sur entier
    int un_int = 5;
    printf("Valeur du pointeur pt_int à la déclaration %p\n", pt_int);
    //A la déclaration en C un pointeur il est recommandé de l'assigner à NULL
    pt_int = NULL;
    printf("Valeur du pointeur pt_int après assignation a NULL %p\n", pt_int);
    pt_int = &un_int; //Récupération de l'adresse mémoire de un_int
    printf("Vérification du fait que la valeur de pt_int %p et l'adresse de un_int %p sont les mêmes\n", pt_int, &un_int);
    printf("Vérification du fait que la valeur un_int %i et la valeur de l'objet pointé par pt_int %i sont les mêmes\n", un_int, *pt_int);
    //Modification de la valeur de un_int en passant par le pointeur
    *pt_int = 2;
    printf("Vérification du fait que la valeur de un_int %i et donc de l'objet pointé par pt_int %i ont bien été modifiés\n", un_int,
    *pt_int);
    return 0;
}
```



# Définition : pointeur (i)

- Un pointeur est une variable qui contient l'adresse mémoire codée en binaire (Notée en Hexa) d'un autre objet
  - Le nombre d'octets k à réserver pour un pointeur dépend de la machine et du "modèle" de mémoire choisi : taille du "mot machine"
  - En général 32 bits sur proc 32 bits, 64 bits sur proc 64...
- Un pointeur a toujours la même taille sur un même système, quel que soit le type de la variable pointée

```
sizeof(void*) = sizeof(int*) =  
sizeof(double*) = sizeof(char*)
```

## Définition : pointeur (ii)

- Permet de faire référence à n'importe quel octet de la mémoire
- Permet d'accéder à la valeur d'une variable ou d'un objet présent à cette adresse
- L'objet dont l'adresse est repérée par un pointeur s'appelle **objet pointé** c'est souvent un objet crée dynamiquement
  - Si p est le pointeur qui le désigne, cet objet est accessible par \*p
- Ecriture niveau algorithmique :
  - On définit un pointeur par ^
  - On utilise @ ou & pour obtenir l'adresse d'une variable existante

# Question pointeur

- La déclaration d'un pointeur implique-t-elle de connaître le type des objets qu'il peut pointer ?

- Non : même taille pour le pointeur donc a priori pas besoin de connaître le type de l'objet pointé

- Oui : il faut savoir le nombre de cases mémoires qu'il faut considérer pour récupérer l'objet pointé



# Exercice pointeurs sur machine 32 bits

```
int i=3;  
int* pi;  
pi=&i;
```

```
char c='c';  
char* pc;  
pc=&c;
```

Objet	Adresse mémoire	valeur
i	4831836000	
pi		
c		
pc		

Remplissez ce tableau



# Solution

```
int i=3;  
int* pi;  
pi=&i;
```

```
char c='c';  
char* pc;  
pc=&c;
```

Objet	Adresse mémoire	valeur
i	4831836000	3
pi	4831836004	4831836000
c	4831836008	'c'
pc	4831836009	4831836008

[https://www.rocq.inria.fr/secret/Anne.Canteaut/COURS\\_C/chapitre3.html](https://www.rocq.inria.fr/secret/Anne.Canteaut/COURS_C/chapitre3.html)

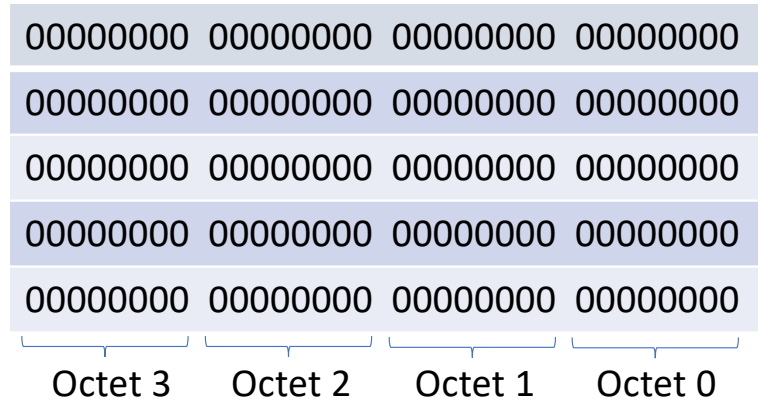


# Mémoire RAM

```
1 #include <stdio.h>
2
3 char C = 'a';
4 int A = 256;
5 int B = 129;
```

Ecrivez le code permettant  
d'obtenir cet affichage.

Adresse de c = 0x16d68f22f  
Code ASCII du caractère a = 97  
Adresse de a = 0x16d68f228  
Adresse de b = 0x16d68f224



Adresse n°116

Adresse n°112

Adresse n°108

Adresse n°104

Adresse n°100

Exemple sur 32 bits

Adresse n°116 : B

Adresse n°112

Adresse n°108

Adresse n°104 : A

Adresse n°100 : C

Chaque nouvel objet commence toujours sur une frontière de mot. Seul les objets de types char (1 octet) ou éventuellement short (2 octets) peuvent posséder des adresses intermédiaires d'octets.



# Exercice

```
1  #include <stdio.h>
2
3  char C = 'a';
4  int A = 256;
5  int B = 129;
```

Ecrivez le code permettant  
d'obtenir cet affichage.

Adresse de c = 0x16d68f22f  
Code ASCII du caractère a = 97  
Adresse de a = 0x16d68f228  
Adresse de b = 0x16d68f224

```
printf("Adresse de c = %p\n", &c);
printf("Code ASCII du caractère %c = %i\n", c, c);
printf("Adresse de a = %p\n", &a);
printf("Adresse de b = %p\n", &b);
return 0;
```



# Résumé type primitif : les pointeurs en C

- Obtenir et afficher l'adresse mémoire d'une variable via l'opérateur de référence '&'

```
char a,b;  
a = 'A';  
b = 'B';  
printf("%d\n",a); //65  
printf("%p\n",&a); //l'adresse de a 0x16dc6f23f
```

- Le type pointeur sur, récupérer l'objet pointé, l'opérateur d'indirection '\*'

```
char a = 'A';  
char *pt_c; //pt_c est un pointeur sur un char  
pt_c = &a; //pt_c contient l'adresse de a  
printf("%c\n",*pt_c); //'A'  
printf("%p\n",pt_c); //l'adresse de a  
*pt_c = 'B';  
printf("%c\n",*pt_c); //le contenu pointé par pt_c  
printf("%c\n",a); //a a également été modifié
```

A  
0x16b3cb22f  
B  
B



# Pointeurs et allocation dynamique de la mémoire

Testez ce code, ou mieux devinez et expliquer ce qui va être affiché

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

int main(int argc, char const *argv[]){
    int *pt_int=NULL;
    *pt_int = 10;
    printf("%p\n",pt_int);
    printf("%d\n",*pt_int);
    return 0;
}
```

On essaie d'écrire dans une zone non allouée de la mémoire on a une « Segmentation fault »



# Allocation dynamique de la mémoire : malloc

## Libération de la mémoire : free

- Il faut affecter l'espace mémoire nécessaire à l'accueil d'une valeur via les fonctions d'allocation dynamique

```
void* malloc(size_t taille);
```

Il faut **caster** la valeur de retour pour correspondre à ce qui est attendu  
En cas d'échec NULL est retourné.

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

```
int main(int argc, char const *argv[]){
    int *pt_int=NULL;
    //Allocation dynamique d'un int
    pt_int = (int *) malloc(sizeof(int));
    printf("%p\n",pt_int);
    //Essai d'affectation après allocation
    *pt_int = 20;
    printf("%i\n",*pt_int);
    free(pt_int);
    return 0;
}
```

Quand vous écrivez une allocation vous **DEVEZ** écrire la **libération** de la mémoire allouée.



# Fonctions d'allocation dynamique de mémoire

```
void* malloc(size_t taille);
```

Pour memory allocation

Alloue un bloc de mémoire de taille totale 'taille' octets sans modifier les valeurs dans la zone de mémoire.

```
void* calloc(size_t elementCount, size_t elementSize);
```

Pour contiguous allocation

Alloue un bloc de mémoire d'un nombre elementCount éléments consécutifs chacun de taille elementSize octets et met tous les octets à 0.

```
void* realloc( void * pointer, size_t memorySize );
```

Pour re-allocation

Réallocation d'un bloc de mémoire dans le tas. Si l'espace libre suivant le bloc à réallouer pointé par 'pointer' est suffisamment grand, le bloc est simplement agrandi. Si l'espace libre n'est pas suffisant, un nouveau bloc de mémoire sera alloué, le contenu de la zone d'origine recopié dans la nouvelle zone et le bloc mémoire d'origine sera libéré automatiquement.


```
void free(void *ptr);
```

Pour la libération de l'espace mémoire alloué

# Résumé fonctions d'allocation/libération mémoire

## Malloc()

```
int* ptr = (int*) malloc ( 5* sizeof ( int ));
```


ptr =  → A large 20 bytes memory block is dynamically allocated to ptr

← 20 bytes of memory →

4 bytes

## Calloc()

```
int* ptr = (int*) calloc ( 5, sizeof ( int ));
```


ptr =  → 5 blocks of 4 bytes each is dynamically allocated to ptr

← 20 bytes of memory →

4 bytes

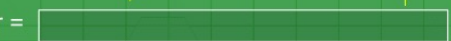
## Realloc()

```
int* ptr = (int*) malloc ( 5* sizeof ( int ));
```

ptr =  → A large 20 bytes memory block is dynamically allocated to ptr

← 20 bytes of memory →

```
ptr = realloc ( ptr, 10* sizeof( int ));
```


ptr =  → The size of ptr is changed from 20 bytes to 40 bytes dynamically

← 40 bytes of memory →

4 bytes

## Free()


```
int* ptr = (int*) calloc ( 5, sizeof ( int ));
```

ptr =  → 5 blocks of 4 bytes each is dynamically allocated to ptr

← 20 bytes of memory →

operation on ptr

free( ptr )

 → The memory of ptr is released

4 bytes

Dynamic Memory Allocation in C using malloc(), calloc(), free() and realloc()

<https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/>