Cours de C - IR1
2007-2008

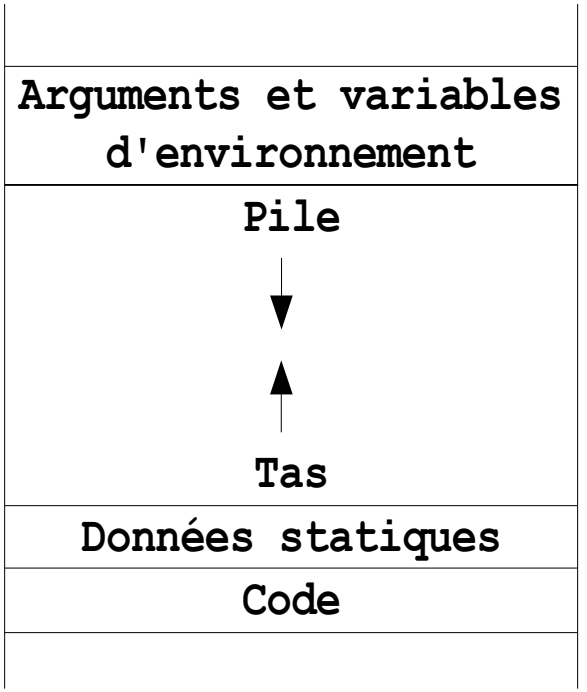
Types, entrées/sorties de base et structures de contrôle

Sébastien Paumier

MCours.com

La mémoire

- on manipule toujours de l'espace mémoire
- octet=unité indivisible de 8 bits (0 ou 1)
- mémoire utilisée par un programme:



Types

- type=taille de zone mémoire + interprétation (signé/non signé, nombre entier/flottant)
- dépendent de la machine
  - pointeurs=4/8 octets sur machine 32/64 bits
- seule contrainte du C:

sizeof(char) ≤ sizeof(short) ≤ sizeof(int) ≤ sizeof(long) = sizeof(size_t)

Les types entiers de base

	Taille	Signé	Valeurs
`signed char`	1	oui	[-127;127]
`unsigned char`	1	non	[0;255]
`char`	1	ça dépend du compilateur !	
`(signed) int`	4	oui	[-2 147 483 647;2 147 483 647]
`unsigned int`	4	non	[0;4 294 967 295]

- La norme C contraint les intervalles des types signés entre $-(2^N-1)$ et 2^N-1

- Un compilateur ***peut*** autoriser la valeur -2^N (-128 pour un **`char`**, par exemple)

  ⇒ portabilité compromise (*je vous jure, ça marchait chez moi!*)

Débordement

- aucune vérification

```
unsigned char c=255;
c=c+1;
```

⇒ `c` vaut 0

- Se méfier des boucles infinies

```
void count_down(int n) {
   unsigned int i;
   for (i=n;i>=0;i--) {
      printf("%d\n",i);
   }
}
```

→ gcc n'avertit pas!

Représentations des entiers

- décimale : `1234`
- hexadécimale : `0x4D2` ou `0x4d2`
- octale : `02322`
  - en maths, 00000012=12
  - pas en C
- pas de représention binaire :(

Opérateurs sur les entiers

- addition: `9+4`
- soustraction `9-4`
- multiplication: `9*4`
- quotient de la division entière: `9/4` (2)
- reste de la division entière: `9%4` (1)

i++

- ne peut pas s'appliquer sur une constante
- utilise la valeur courante de `i`, puis l'incrémente de 1

```
int i=2;
```
⇒ `a` vaut 2 et `i` vaut 3

```
int a=i++;
```

- à utiliser avec précaution 

```
int a=(i++*2+i)*(i++);
```
⇒ valeur de `a` ???

i++

- `++i` : idem, mais on incrémente d'abord

```
int i=2;
int a=++i;
```

⇒ `a` et `i` valent 3

- `i--` et `--i` : même chose avec la soustraction

Les types réels

- calcul en virgule flottante

  ⇒ approximations (ne pas utiliser pour des calculs exacts, ex: finances)

- `float` : réel simple précision
- `double` : réel double précision

Représentations des réels

- `12.34` ou `1234.` ou `.1234`
- notation $mantisse \times 10^{exposant}$

  `1723.68` = `1.72368e3` = `17.2368E2`

  `0.015` = `1.5e-2`
- par défaut, les constantes sont `double`

  `245.45f` = `245.45F` ⇒ `float`

  `245.45` = `245.45l` = `245.45L` ⇒ `double`
- environ 7/15 chiffres significatifs pour les `float`/`double`

Problèmes de précision

- se méfier beaucoup des **float**
- se méfier quand même des **double**

```
int main(int argc,char* argv[]) {
    float f=0.f;
    double d=0;
    int i;
    for (i=0;i<1000;i++) {
        f=f+0.1f;
    }
    for (i=0;i<10000000;i++) {
        d=d+0.1;
    }
    printf("f=%f\nd=%f\n",f,d);
    return 0;
}
```

→

```
$>./a.out
f=99.999046
d=999999.999839
```

Problèmes de précision

- **176.768204+0.00001≠176.768214**
- pas toujours de **float** d=√x tel que $d^2=x$

se méfier de l'égalité sur les réels

```
int main(int argv,char* argc[]) {
/* racine(31247) */
float d=sqrt(31247);
float d_plus_e=d+0.00001;
float d_minus_e=d-0.00001;
printf("d-e=%f\t(d-e)^2=%f\n"
       "  d=%f\t    d^2=%f\n"
       "d+e=%f\t(d+e)^2=%f\n"
       ,d_minus_e,d_minus_e*d_minus_e
       ,d,d*d
       ,d_plus_e,d_plus_e*d_plus_e);
return 0;
}
```

```
$>./a.out
d-e=176.768188  (d-e)^2=31246.992457
  d=176.768204      d^2=31246.997852
d+e=176.768219  (d+e)^2=31247.003246
```

Opérations sur les réels

- `+ - *` comme pour les entiers
- `/` donne une division réelle
- conversions automatiques:

```
2 +/-/* 3.5      ⇒   2.0 +/-/*  3.5
6 / 1.5          ⇒   6.0 / 1.5
8.4 / 2          ⇒   8.4 / 2.0
double a=2;      ⇒   double a=2.0;
int a=2.45;      ⇒   arrondi à int a=2;
```

Piège de conversion

- si on veut un résultat réel pour **int/int**, il faut une conversion explicite

```
int main(int argc,char* argv[]) {
   int a=3;
   int b=7;
   int c=10;
   float average1=(a+b+c)/3;
   float average2=(a+b+c)/3.0;
   float div1=a/b;
   float div2=(float)a/b;
   printf("%f\n",average1);
   printf("%f\n",average2);
   printf("%f\n",div1);
   printf("%f\n",div2);
   return 0;
}
```

```
$>./a.out
6.000000
6.666667
0.000000
0.428571
```

Le type caractère

- on interprète les **`char`** comme des codes de caractères
  - pas de problème entre 0 et 127
  - au-delà, dépend de l'encodage du système (à éviter)
- on désigne le code du caractère `x` avec `'x'`
- on peut utiliser les opérateurs entiers
  - exemple: `'a'+1` vaut `'b'` (car les codes sont bien rangés)

Le type caractère

- ne pas confondre caractère et valeur d'un chiffre 

```
/**
 * Returns the value of the given digit,
 * or -1 if the given character is not a digit
 * in [0;9].
 */
int value_of_digit(char digit) {
   if (digit<'0' || digit>'9') return -1;
   return digit-'0';
}
```

Caractères spéciaux

- `'\n'` : saut de ligne
- `'\r'` : retour au début de la ligne
- `'\t'` : tabulation (largeur variable)

\+ `'\\'` et `'\"'` : les caractères `\` et `"`

```
int main(int argc,char* argv[]) {
   printf("aaaa\r");
   printf("bbb\r");
   printf("cc\r");
   printf("d\n");
   printf("xxxxx\t*\\*\n");
   printf("yyy\t*\"*\n");
   return 0;
}
```

→

```
$>./a.out
dcba
xxxxx    *\*
yyy      *"*
```

Caractères accentués

- lettres accentuées > 127
- problème de portabilité et d'encodage
  - ne marche plus si on change de système et/ou de codage:

```
int main(int argc,char* argv[]) {
   printf("à Marne-la-Vallée\n");
   return 0;
}
```

→

```
$>./a.out
Ó Marne-la-VallÚe
```

Les chaînes de caractères

- pas un type, mais une convention de stockage:

  tableau de caractères, dont le dernier est `'\0'`

- délimitée par des guillemets

```
char* msg="Welcome";
```

- variantes:

```
char msg[]="Welcome";
char msg[]={'W','e','l','c','o','m','e','\0'};
```

Les chaînes de caractères

- chaîne vide `""` = tableau dont la première case contient `'\0'`
- longueur = caractères avant `'\0'`
  - `"abcd"` a une longueur de 4
- concaténation automatique des chaînes constantes

```
int main(int argc,char* argv[]) {
   printf("aaaa\n");
   printf("bbbb\n");
   printf("cccc\n");
   return 0;
}
```

→

```
int main(int argc,char* argv[]) {
   printf("aaaa\n"
          "bbbb\n"
          "cccc\n");
   return 0;
}
```

Les chaînes de caractères

- accès au n^{ème} caractère:

```
char* s="wxyz";
char c=s[2];
```

⇒ `c` vaut `'y'` car on commence à l'indice 0

- attention au débordement

```
int main(int argc,char* argv[]) {
    char* s="Hello";
    printf("%c\n",s[4657]);
    return 0;
}
```

```
$>./a.out
Segmentation fault
```

Les chaînes de caractères

- on ne peut modifier ni les chaînes statiques, ni les arguments de **main**

```c
int main(int argc,char* argv[]) {
    char* s="wxyz";
    s[2]='u';
    printf("%s\n",s);
    return 0;
}
```

```c
int main(int argc,char* argv[]) {
    argv[0][0]='*';
    printf("%s\n",argv[0]);
    return 0;
}
```

```
$>./a.out
Segmentation fault
```

Les variables

- identificateurs: [_a-zA-Z][_a-zA-Z0-9]*
  (ce sera vrai aussi pour les noms de fonctions)

  `hello_89 __tmp` ~~**`2tree`**~~
- éviter ceux qui ressemblent à des mots-clés: **`new`**, **`class`**...

```
$>gcc -Wall -ansi test.c
$>g++ -Wall -ansi test.c
test.c: Dans function « int main(int, char**) »:
test.c:5: erreur d'analyse syntaxique avant « new »
test.c:6: erreur d'analyse syntaxique avantle jeton « - »
```

```
int main(int argc,char* argv[]) {
    int old=7;
    int new=7;
    printf("%d\n",new-old);
    return 0;
}
```

Les variables

- doivent être déclarées avant d'être utilisées
- déclaration: `int a;`
- avec initialisation: `int a=3;`
- déclarations multiples: `float f1=.3,f2;`
- attention:

```
int main(int argc,char* argv[]) {
    char* a="hello",b="you";
    printf("%s %s\n",a,b);
    return 0;
}
```

```
$>gcc -Wall -ansi test.c
test.c: Dans la fonction « main »:
test.c:4: AVERTISSEMENT: initialisation transforme en
entier un pointeur sans transtypage
```

Portée d'une variable

- visible dans le bloc

```
int main(int argc,char* argv[]) {
    int i;
    for (i=0;i<10;i++) {
        int a=i;
    }
    printf("a=%d\n",a);
    return 0;
}
```

- dans une fonction: variable locale
- hors d'une fonction: variable globale

MCours.com

Variables globales

- utilisables dans tout le fichier
- plutôt à éviter:
  - nuit à la modularité du code
  - pas thread-safe

```
int result;

void sum(int a,int b) {
result=a+b;
}

int main(int argc,char* argv[]) {
   sum(4,7);
   printf("%d\n",result);
   return 0;
}
```

Masquage de noms

- un identificateur peut parfois en masquer un autre

```c
float sum(float a,float b) {
    return a+b;
}

int main(int argc,char* argv[]) {
    float argc=2.3;
    float sum=4.5+argc;
    printf("%f\n",sum);
    return 0;
}
```

```c
int main(int argc,char* argv[]) {
    int argc=2;
    float argc=4.5;
    printf("%f\n",argc);
    return 0;
}
```

```
$>gcc -Wall -ansi variables.c
variables.c: Dans la fonction « main »:
variables.c:9: AVERTISSEMENT: déclaration de « argc » cache un paramètre
```

Initialisation

- toujours initialiser les variables

```
int sum(int n) {
    int i,x=0;
    while (i<n) {
        x=x+i;
        i++;
    }
    return x;
}
```

→ le résultat dépend de la valeur par défaut de `i`

- bien les initialiser

```
int maximum(int a,int b,int c) {
    int max=0;
    if (a>max) max=a;
    if (b>max) max=b;
    if (c>max) max=c;
    return max;
}
```

→ `maximum(-4,-10,-5)` vaut 0

Affectation

- `=` : opérateur, ≠ instruction

```
int b=3;

int a=(b=b+2)+4;
```

⇒ `a`=9 et `b`=5

- pratique pour tester des retours de fonction

```
int main(int argc,char* argv[]) {
   char* s="Christian";
   int n;
   if ((n=first_vowel_index(s))!=-1) {
      printf("first vowel=%c\n",s[n]);
   }
   return 0;
}
```

Effets de bord

- modification d'une valeur pendant l'évaluation d'une expression
- dangereux 
- à n'utiliser que dans des cas simples:

```
char c=s[i++];
```

```
if ((c=fgetc(f))!=EOF) {
      /*
        ...
      */
   }
```

Conversion

- `unsigned char c=713;`

  ⇒ `AVERTISSEMENT: grand entier implicitement tronqué pour un type non signé`

  ⇒ `c`=201

- explication en binaire:

```
713 = 1011001001
201 =   11001001
```

Conversion

- on évite les warnings avec des cast:

  `variable=(type)expression`

- il faut être sûr de ce qu'on fait 

```
int main(int argc,char* argv[]) {
    char* s="Christian";
    int n;
    if ((n=first_vowel(s))!=-1) {
        char c=(char)n;
        printf("first vowel=%c\n",c);
    }
    return 0;
}
```

Les constantes

- définies avec la commande:

```
#define identificateur valeur
```

- exemples:

```
#define PI 3.14
#define SIZE_MAX 1024
#define YES 'y'
#define PROMPT "$>"
```

Les constantes

- remplacement brut de chaînes par le préprocesseur
  - sauf dans les identificateurs et les chaînes
- risque d'erreur

```
#define A 143

int main(int argc,char* argv[]) {
    int A=12;
    printf("%d\n",A);
    return 0;
}
```

```
$>gcc -Wall -ansi define.c
define.c: Dans la fonction « main »:
define.c:6: erreur d'analyse syntaxique avant la constante numérique
```

Les constantes

- toute constante numérique non triviale doit être définie!
- évite les modifications multiples

```
void foo(int a1[],int a2[]) {
    int i;
    for (int i=0;i<40;i++) {
        a1[i]=a2[i];
    }
    for (int i=2;i<40;i++) {
        a1[i]=a1[i]+a1[i-2];
    }
}
```

```
#define N 40

void foo(int a1[],int a2[]) {
    int i;
    for (int i=0;i<N;i++) {
        a1[i]=a2[i];
    }
    for (int i=2;i<N;i++) {
        a1[i]=a1[i]+a1[i-2];
    }
}
```

Les constantes

- commentaire si nécessaire
  - pour expliquer la constante
  - pour éviter des modifications malheureuses

```
/**
 * Error codes for I/O functions.
 * They all must be negative.
 */
#define FILE_NOT_FOUND -1
#define READ_FORBIDDEN -2
#define WRITE_FORBIDDEN -3
#define DISK_FULL -4


/* Checksum of the copyright logo file */
#define CHECKSUM 0x7D3ED97F
```

printf

- fonction d'affichage (`stdio.h`)

```
printf(chaîne de format,arguments);
```

- variables indiquées avec:

`%d` : entier `%f` : réel

`%c` : caractère `%s` : chaîne de caractères

\+ `%%` : le caractère `'%'`

- exemple:

```
printf("moyenne(%d,%d)=%f\n",i,j,moy);
```

- voir `man printf` pour les autres options

printf

- il doit y avoir autant de variables que de `%...`
- ne pas afficher une chaîne directement 

```
int main(int argc,char* argv[]) {
    char* insult="*!%s#\n";
    printf(insult);
    return 0;
}
```

```
$>./a.out
*!*!%s#
#
```

printf

- affichage bufferisé
  - quand il y a un `'\n'`
  - quand le buffer est plein

```
int main(int argc,char* argv[]) {
    char* msg="Computing...";
    printf("%s",msg);
    int i;
    float f=0;
    for (i=0;i<200000000;i++) {
        f=f+0.1;
    }
    printf("\nf=%f\n",f);
    return 0;
}
```

→

```
$>./a.out
3 secondes d'attente
Computing...
f=2097152.000000
```

Messages d'erreur

- utiliser la sortie d'erreur avec **`fprintf(stderr,"...",...);`**
- même fonctionnement que **`printf`**

```
int sum(int t[],int size) {
    int i,sum=0;
    if (size<0) {
        fprintf(stderr,"Invalid size: %d\n",size);
    }
    for (i=0;i<size;i++) {
        sum=sum+t[i];
    }
    return sum;
}
```

scanf

- fonction de saisie au clavier (`stdio.h`)
- ressemble à `printf`, mais:
  - que des variables, pas de constantes
  - `&` devant les variables, sauf les chaînes
  - la chaîne de format n'est pas affichée

```
int a,b; char s[64];
scanf("%d %d %s",&a,&b,s);
```

scanf

- espaces ignorés : `"%d %d"` ⇔ `"%d%d"`
- saisie bufferisée par ligne:

```
#define N 8

void get_int_array(int t[]) {
    int i;
    for (i=0;i<N;i++) {
        scanf("%d",&t[i]);
    }
    printf("Done.\n");
}
```

```
$>./a.out
1 2 3 4 5 6 7 8
Done.
```

MCours.com

scanf

- les chaînes sont délimitées par les espaces, les tabulations et les sauts de ligne

```
int main(int argc,char* argv[]) {
    int a;
    char s1[32];
    char s2[32];
    scanf("%d %s %s",&a,s1,s2);
    printf("%d %s %s\n",a,s1,s2);
    return 0;
}
```

→

```
$>./a.out
 5  hello 32    abc
5 hello 32
```

scanf

- **scanf** renvoie le nombre de variables saisies correctement
- en cas d'erreur:

```
int main(int argc,char* argv[]) {
    int a,b,c,n;
    n=scanf("%d %d %d",&a,&b,&c);
    printf("%d %d %d %d\n",n,a,b,c);
    return 0;
}
```

```
$>./a.out
4 8 hello
2 4 8 4198592
```

scanf

- lecture de caractères
  - attention à ce qui reste dans le buffer

```c
#define NO 0
#define YES 1

int yes_or_no() {
    char c;
    do {
        printf("y/n ? ");
        scanf("%c",&c);
    } while (c!='y' && c!='n');
    if (c=='y') return YES;
    return NO;
}
```



```
$>./a.out
y/n ? u
y/n ? y/n ? y
```

if...else...

`if (condition) instruction1 else instruction2`

ou `if (condition) instruction`

- opérateurs de comparaisons:
  - `a<b` `a<=b` `a>b` `a>=b` `a!=b`
  - `a==b` à ne pas confondre avec `a=b` 
- opérateurs sur les entiers et les réels
- ne marchent pas sur les chaînes 

Portée du else

- en cas d'ambiguïté, le **else** se rapporte au **if** le plus proche
- éviter les mauvaises surprises en utilisant des blocs

```
if (a==0)
   if (b==1)
      /* ... */
   else
      /* ... */
```

```
if (a==0) {
   if (b==1) {
      /* ... */
   } else {
      /* ... */
   }
}
```

Erreur classique

- pas de point-virgule après la condition

```
int odd(int n) {
    if (n%2==1);
        return 1;
    return 0;
}

int main(int argc,char* argv[]) {
    int i;
    for (i=0;i<6;i++) {
        printf("odd(%d)=%d\n",i,odd(i));
    }
    return 0;
}
```

```
$>./a.out
odd(0)=1
odd(1)=1
odd(2)=1
odd(3)=1
odd(4)=1
odd(5)=1
odd(6)=1
```

Opérateurs logiques

- convention C: 0=faux ≠0=vrai
- a&&b a||b !a

```
int main(int argc,char* argv[]) {
   printf("4 AND 6\t= %d\n"
          "4 AND 0\t= %d\n"
          "4 OR 6\t= %d\n"
          "0 OR 0\t= %d\n"
          "NOT 17\t= %d\n"
          "NOT 0\t= %d\n",
          4&&6, 4&&0,
          4||6, 0||0,
          !17, !0);
   return 0;
}
```

```
$>./a.out
4 AND 6 = 1
4 AND 0 = 0
4 OR 6  = 1
0 OR 0  = 0
NOT 17  = 0
NOT 0   = 1
```

Opérateurs logiques

- exemple:

```
int is_letter(char c) {
    if ((c>='a' && c<='z')
        || (c>='A' && c<='Z')) {
       return 1;
    }
    return 0;
}
```

Négation

- penser aux formules de Morgan:

$$\overline{A} \wedge \overline{B} = \overline{A \vee B} \qquad \overline{A} \vee \overline{B} = \overline{A \wedge B}$$

```
while ((c=fgetc(f))!=EOF &&
       !(c==' ' || c=='\t'
       || c=='\n' || c=='\r')) {
    /*
    ...
    */
}
```

```
while ((c=fgetc(f))!=EOF
       && c!=' '  && c!='\t'
       && c!='\n' && c!='\r') {
    /*
    ...
    */
}
```

Conditions simplifiées

- condition ⇔ expression
- `if (condition)...` = `if (expression!=0)...`
- écriture simplifiée:

```
if (value!=0)... = if (value)...
if (value==0)... = if (!value)...
```

Évaluation paresseuse

- opérateurs **&&** et **||** paresseux
- évaluation de gauche à droite
- s'arrête dès que possible
  - **(a=0 && b=c)** ⇒ b n'est jamais affecté
  - **(a=1 || b=c)** ⇒ b n'est jamais affecté

```
/* Returns 1 if the given string is not NULL and starts with a
 * character that is not a digit; 0 otherwise. */
int no_digit_prefix(char* s) {
   if (s!=NULL && (s[0]<'0' || s[0]>'9'))
      return 1;
   return 0;
}
```

Priorités

- du plus au moins prioritaire:

```
()
--   ++   !   - (moins unaire)
*    /    %
+    -
<    <=   >   >=
==   !=
&&   ||
=
```

- dans le doute, on parenthèse

Fautes de style à éviter

```
if (condition) {
   /* bloc vide */
} else {
   /*
    ...
    */
}
```



```
if (!condition) {
   /*
    ...
    */
}
```

```
if (a!=valeur_speciale) {
   s=a;
} else {
   s=valeur_speciale;
}
```



```
s=a;
```

MCours.com

Tests inutiles

```
if (n>0) {
    /* ... */
}
if (n<0) {
    /* ... */
}
if (n==0) {
    /* ... */
}
```

1) on évite de faire des tests dont on est sûr qu'ils seront faux

```
if (n>0) {
    /* ... */
}
else if (n<0) {
    /* ... */
}
else if (n==0) {
    /* ... */
}
```

2) on évite de faire des tests dont on est sûr qu'ils seront vrais

```
if (n>0) {
    /* ... */
}
else if (n<0) {
    /* ... */
}
else {
    /* ... */
}
```

Expression conditionnelle

- `if (condition) a=exp1; else a=exp2;`

⇔ `a=(condition)?exp1:exp2;`

- pratique pour des petits réglages:

```
int main(int argc,char* argv[]) {
    char* s="supercalifragilisticexpialidocious";
    int n=count_vowels(s);
    printf("%d vowel%s\n",n,(n>1)?"s":"");
    return 0;
}
```

Boucle for

- **`for (instr1;condition;instr2) instr3`** :
  - faire **`instr1`**
  - tant que **`condition`** est vraie, faire
    - **`instr3`**
    - **`instr2`**

classique:

```
int i;
for (i=0;i<10;i++) {
   printf("i=%d\n",i);
}
```

variante avec double initialisation:

```
int i,n;
for (i=0,n=get_max();i<n;i++) {
   printf("i=%d\n",i);
}
```

Quand la boucle est vide

- mettre un bloc vide { }

```
/**
 * Returns the last position of the given integer
 * in the given array, or -1 if not found.
 */
int find_n(int t[],int n) {
   int i;
   for (i=N-1;(i>=0) && (t[i]!=n);i--) {}
   return i;
}
```

Quand la condition est vide

- la condition vide est toujours vraie
- `for (instr1;;instr2) {...}`
  = boucle infinie

```
/**
 * Returns the sum of integers read from
 * the standard input. The function stops
 * when there are no more integers to read.
 */
int scan_and_sum() {
   int sum=0;
   int n;
   for (;;) {
      if (scanf("%d",&n)!=1) return sum;
      sum=sum+n;
   }
}
```

Attention à la condition

- la condition est évaluée à chaque tour!
- attention aux complexités cachées 

```
void spell(char* s) {
   int i;
   for (i=0;i<strlen(s);i++) {
      printf("s[%d]=%c\n",i,s[i]);
   }
}
```

→ complexité: *longueur(s)*^{2}

```
void spell(char* s) {
   int i,n;
   for (i=0,n=strlen(s);i<n;i++) {
      printf("s[%d]=%c\n",i,s[i]);
   }
}
```

→ complexité: *longueur(s)*

Boucle while

- **`while (condition) instruction`**
- **`while`** vs **`for`** : question de lisibilité

```c
int scan_and_sum() {
    int sum=0;
    int n;
    for (;1==scanf("%d",&n);) {
        sum=sum+n;
    }
    return sum;
}
```

```c
int scan_and_sum() {
    int sum=0;
    int n;
    while (1==scanf("%d",&n)) {
        sum=sum+n;
    }
    return sum;
}
```

Boucle do...while...

- **`do instruction while (condition);`**
- à utiliser quand on doit passer au moins une fois dans la boucle

```
int scan_positive_integer() {
    int n=-1;
    char foo;
    do {
        if (1!=scanf("%d",&n)) {
            scanf("%c",&foo);
        }
    } while (n<0);
    return n;
}
```

→

```
$>./a.out
dff d-34 T_56op
56
```

switch

- choix entre plusieurs constantes entières:

```
switch(expression) {
    case const1: instruction break;
    case const2: instruction break;
    case const3: instruction break;
    ...
    default: instruction
}
```

switch

- plus élégant que plein de **`if...else...`**
- possibilité d'optimisation par le compilateur
- ne pas oublier le **`default`**:

```
int get_base(char choice) {
    int base;
    switch (choice) {
        case 'b': base=2; break;
        case 'o': base=8; break;
        case 'd': base=10; break;
        case 'h': base=16; break;
    }
    return base;
}
```



```
$>./a.out
B
4572656
```

switch

- sans break, l'exécution continue 

```
void print_note(char us_note) {
   switch(us_note) {
      case 'a': printf("la\n");
      case 'b': printf("si\n");
      case 'c': printf("do\n");
      case 'd': printf("re\n");
      case 'e': printf("mi\n");
      case 'f': printf("fa\n");
      case 'g': printf("sol\n");
      default: fprintf(stderr,"Error\n");
   }
}
```



```
$>./a.out
d
re
mi
fa
sol
Error
```

switch

- possibilité de factoriser des valeurs

```
/**
 * Prints the latin name of the given
 * note in US notation.
 */
void print_note(char us_note) {
   switch(us_note) {
      case 'a': case 'A': printf("la\n"); break;
      case 'b': case 'B': printf("si\n"); break;
      case 'c': case 'C': printf("do\n"); break;
      case 'd': case 'D': printf("re\n"); break;
      case 'e': case 'E': printf("mi\n"); break;
      case 'f': case 'F': printf("fa\n"); break;
      case 'g': case 'G': printf("sol\n"); break;
      default: fprintf(stderr,"Error: '%c' is not a US note\n",us_note);
   }
}
```

break

- peut servir à sortir d'une boucle:
  - dans certains cas d'erreur
  - si le résultat d'un calcul est connu avant la fin

```
int add_products(int t[],int t2[]) {
   int i,p=1,p2=1;
   for (i=0;i<N;i++) {
      p=p*t[i];
      if (p==0) break;
   }
   for (i=0;i<N;i++) {
      p2=p2*t2[i];
      if (p2==0) break;
   }
   return p+p2;
}
```

break

- à utiliser avec parcimonie
- à éviter en cas de boucles imbriquées
- à éviter si on peut quitter la fonction

```
int product(int m[][N]) {
    int i,j,p=1;
    for (i=0;i<N;i++) {
        for (j=0;j<N;j++) {
            p=p*m[i][j];
            if (p==0) return 0;
        }
    }
    return p;
}
```

continue

- sert à sauter un tour de boucle
- évite un niveau d'indentation
  ⇒ lisibilité (surtout si le **`else`** est gros)

```
void print_free_seats() {
   int i,j;
   for (i=1;i<=N_ROWS;i++) {
      if (i==13) {
         /* No seat #13 because of stupid
          * superstitious people */
         continue;
      }
      for (j=0;j<N_SEATS;j++) {
         if (seat[i][j]) printf("%2d%c ",i,j+'A');
         else printf("    ");
         if (j==2) printf("  ");
      }
      printf("\n");
   }
}
```

```
$>./a.out
 1A  1B  1C    1D  1E  1F
 2A  2B  2C    2D  2E  2F
 3A  3B  3C        3E  3F
 4A      4C    4D  4E  4F
 5A  5B  5C    5D  5E  5F
 6A  6B  6C    6D
 7A  7B  7C    7D  7E  7F
 8A  8B  8C    8D  8E  8F
 9A  9B  9C    9D  9E  9F
10A 10B 10C   10D 10E 10F
11A     11C   11D 11E
    12B 12C   12D 12E 12F
14A 14B 14C       14E 14F
```