



Cours de C

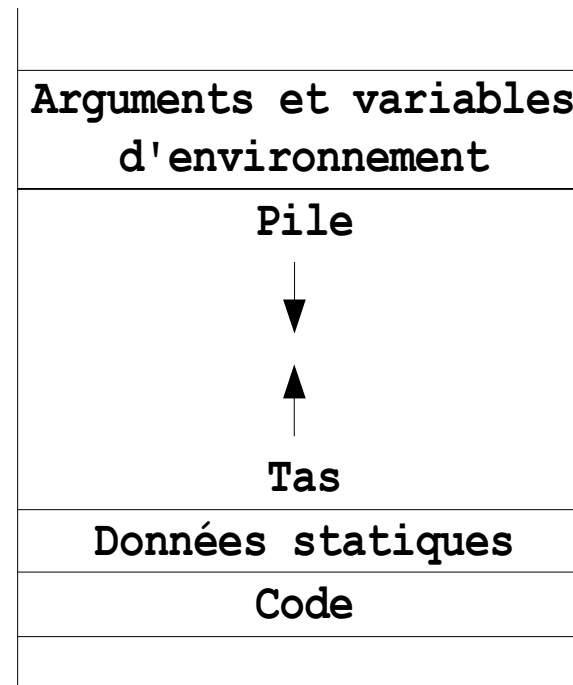
Types, variables, constantes

Sébastien Paumier



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
<code>signed char</code>	1	oui	[-127;127]
<code>unsigned char</code>	1	non	[0;255]
<code>char</code>	1	ça dépend du compilateur !	
<code>(signed) int</code>	4	oui	[-2 147 483 647;2 147 483 647]
<code>unsigned int</code>	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 représentation 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;` \Rightarrow `a` vaut 2 et `i` vaut 3

`int a=i++;`

- à utiliser avec précaution 

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



i++

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

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

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

- `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* $\times 10^{\text{exposant}}$
 - `1723.68 = 1.72368e3 = 17.2368E2`
 - `0.015 = 1.5e-2`
- par défaut, les constantes sont **double**
 - `245.45f = 245.45F \Rightarrow float`
 - `245.45 = 245.45l = 245.45L \Rightarrow 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<100000000;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

- $176.768204 + 0.00001 \neq 176.768214$
- pas toujours de `float` $d = \sqrt{x}$ 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 \Rightarrow 2.0 $+/-/*$ 3.5

6 $/$ 1.5 \Rightarrow 6.0 $/$ 1.5

8.4 $/$ 2 \Rightarrow 8.4 $/$ 2.0

`double a=2;` \Rightarrow `double a=2.0;`

`int a=2.45;` \Rightarrow 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` ⚡

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

```
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`...


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

```
$>gcc -Wall -ansi test.c  
$>g++ -Wall -ansi test.c
```

```
test.c: Dans fonction « int main(int, char**) »:  
test.c:5: erreur d'analyse syntaxique avant « new »  
test.c:6: erreur d'analyse syntaxique avant le jeton « - »
```



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



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

```
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;  
}
```

```
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
```