Programmation en C

Table des matières

I. Syntaxe

Mathématiques de base et langage C

Mathématique	Langage C
$x \times 5$	`x*5`
$a = b$ (test)	`a == b`
$a \neq b$	`a != b`
$a \leq b$	`a <= b`
$a \geq b$	`a >= b`
$a \leq b \leq c$	`a <= b && b <= c`
$a < b < c$	`a < b  && b < c`
$a = b = c$	`a == b && b == c`
VRAI	`1`
FAUX	`0`
ET	`&&`
OU	`||`
$a < b$ **ET** $b < c$	`a < b && b < c`
$a = b$ **ET** $b = c$	`a == b && b == c`
$a = b$ **OU** $b = c$	`a == b || b == c`
$33 + \frac{5 \times x + 40 \times y}{3}$	`33+(5*x+40*y)/3`
3 est-il divisible par 2 ?	`3 % 2 == 0`

Exemples à connaître

Fonction et appel de fonction

```
float f(float x)
{
    return x+1;
}

int main()
{
    float x = f(1);
    printf("%f \n", x);
    return 0;
}
```

Fonction en appelant d'autres

```
float g(float x)
{
    return x+1;
}

float h(float x)
{
    return x*x;
}

float f(float x)
{
   return g(x) + h(x);
}
```

Condition si / sinon

```
int f(char c)
{
    if(c=='a')
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
```

Condition si / sinon si / sinon

```
#include <stdio.h>

void e(double x)
{
    if(x<0)
    {
```

```
        printf("negatif. \n");
    }
    else if (x==0)
    {
        printf("zero. \n");
    }
    else
    {
        printf("positif. \n");
    }
}

int main()
{
    e(1);
    return 0;
}
```

Encadrement

```
int encadrement(double x, double a, double b)
{
    if (a <= x && x <= b)
        return 1; // Vrai
    else
        return 0; // Faux
}
```

Utilisation d'un « ou » logique

```
int toto(char x, char a, char b)
{
    if (x == a || x == b)
        return 1; // Vrai
    else
        return 0; // Faux
}
```

Différence entre fonction et procédure

```
#include <stdio.h>

int toto(int x, int y)
{
    return x+y;
}

int main()
{
    printf("%d", toto(1,2));
    return 0;
}
```

```
#include <stdio.h>

void toto(int x, int y)
{
```

```
    printf("%d", x+y);
}

int main()
{
    toto(1, 2);
    return 0;
}
```

Notion de pseudo-code

```
fonction toto(x)
    si x=="toto" alors
        retourner VRAI
    sinon:
        retourner FAUX
    fin si
```

```
int toto(char *s)
{
    if (strcmp(s, "toto") == 0)
       return 1;
    else
        return 0;
}
```

Boucle « pour » : afficher les entiers de *0* à *n-1*

```
procédure afficher(n)
    pour i = 0, ..., n-1 alors
        affichage(i)
    fin pour
```

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

Boucle « tant que » : afficher les entiers de *0* à *n-1*

```
procédure afficher(n)
    i = 0
    tant que i <= n-1 faire
        affichage(i)
        i = i+1
    fin tant que
```

```
void afficher(int n)
{
    int i=0;
    while(i<n)
    {
        printf("%d ", i);
        i++;
    }
}
```

Affichage des éléments d'un tableau

```
#include <stdio.h>

void afficher(int *tab, int n)
{
    int i=0;
    for (i; i<n; i++)
        printf("%d ", tab[i]);
}

int main()
{
    int t[4] = {1, 0, -1, 3};
    afficher(t, 4);
    return 0;
```

```
}
```

Somme des éléments d'un tableau

```
int somme(int *tab, int n)
{
    int i=0, res=0;
    for (i; i<n; i++)
        res += tab[i];
    return res;
}
```

Affichage simultané de deux tableaux

```
void afficher2(int *tab1, int *tab2, int n)
{
    int i=0;
    for (i; i<n; i++)
        printf("%d %d \n", tab1[i], tab2[i]);
}
```

Somme de deux tableaux terme à terme dans un troisième

```
#include <stdio.h>

void somme2(int *tab1, int *tab2, int *tab3, int n)
{
    int i=0, res=0;
    for (i; i<n; i++)
        tab3[i] = tab1[i] + tab2[i];
}


int main()
{
    int t1[4] = {1, 0, -1, 3};
    int t2[4] = {2, 4, 6, 8};
    int t3[4];
    somme2(t1, t2, t3, 4);
    afficher(t3, 4);
    return 0;
}
```

Affichage du résultat :

```
3 4 5 11
```

Afficher le k-ième élément d'un tableau

```
char k_ieme(char *tab, int n, int j)
{
    // index bien défini ?
    if (0<=j && j<=n-1)
        return tab[j];
    return 1; // le "1" indique un probleme
}
```

Maximum des éléments d'un tableau

```
double maxi(double *tab, int n)
{
    int i;
    double res=tab[0]; // probleme si le tableau est vide
    for (i=0; i<n; i++)
        if (tab[i] > res)
            res = tab[i];
    return res;
}
```

Indice d'un élément maximum d'un tableau

```
int i_maxi(double *tab, int n)
{
    int i;
    double maxi=tab[0]; // probleme si le tableau est vide
    int i_maxi = 0;
    for (i=0; i<n; i++)
        if (tab[i] > maxi)
        {
            maxi = tab[i];
            i_maxi = i;
        }
    return i_maxi;
}
```

Afficher les caractères d'une chaîne de caractères

```
#include <stdio.h>

void afficher(char *s)
{
    int i=0;
    while (s[i] != '\0')
    {
        printf("%c", s[i]);
        i++;
    }
}

int main()
{
    char *m = "toto";
    afficher(m);
    return 0;
}
```

On obtient l'affichage : « toto ».
Le codage de la chaîne `m` en mémoire s'effectue comme suit :

't'	'o'	't'	'o'	'\0'

Symbole mathématique : somme ∑

$\sum_{i=0}^{n-1} i = 0+1+2+3+...+(n-1)$	`int somme(int n)`
	`{`
	`  int i;`
	`  int res = 0;`
	`  for (i=0; i<n; i++)`
	`  res += i;`
	`  return res;`
	`}`

$\sum_{i=0}^{n-1} \cos(i) = \cos(0)+ \cos(1)+ \cos(2)+ ...+ \cos(n-1)$	`double somme_cos(int n)`
	`{`
	`  int i;`
	`  double res = 0;`
	`  for (i=0; i<n; i++)`
	`  res += cos(i);`
	`  return res;`
	`}`

Symbole mathématique : produit ∏

$\prod_{i=1}^{n-1} i = 1\times 2\times 3\times ...\times (n-1)$	`int produit(int n)`
	`{`
	`  int i;`
	`  int res=1;`
	`  for(i=1; i<n; i++)`
	`  res *= i;`
	`  return res;`
	`}`

$\prod_{i=0}^{n-1} e^{i} = e^{0}\times e^{1}\times e^{2}\times ...\times e^{n-1}$	`double produit_exp(int n)`
	`{`
	`  int i;`
	`  double res = 1;`
	`  for (i=0; i<n; i++)`
	`  res *= exp(i);`
	`  return res;`
	`}`

II. Mémoire

Notion de pointeur

1 - Exemples

Exemple 1

```
void main()
{
    float x = 1.1;
    float *pt = &x;      // le pointeur pt pointe sur x

    printf("x=%.1f ; pt=%d ; *pt=%.1f \n", x, pt, *pt);
      ~~> x=1.1 ; pt=2686748 ; *pt=1.1
}
```

Exemple 2

```
void main()
{
    float x = 1.1;
    float *pt = &x;

    printf("x=%.1f ; pt=%d ; *pt=%.1f \n", x, pt, *pt);
      ~~> x=1.1 ; pt=2686748 ; *pt=1.1
    printf("&x=%d ; pt=%d ; &pt=%d ", &x, pt, &pt);
      ~~> &x=2686748 ; pt=2686748 ; &pt=2686744
}
```

Exemple 3

```
void main()
{
    float x = 1.1;
    float *pt = &x;
    int *pt2 = &pt;

    printf("x=%.1f ; pt=%d ; *pt=%.1f \n", x, pt, *pt);
      ~~> x=1.1 ; pt=2686748 ; *pt=1.1
    printf("&x=%d ; pt=%d ; &pt=%d \n", &x, pt, &pt);
      ~~> &x=2686748 ; pt=2686748 ; &pt=2686744
    printf("&pt2=%d ; pt2=%d ; *pt2=%d \n", &pt2, pt2, *pt2);
      ~~> &pt2=2686740 ; pt2=2686744 ; *pt2=2686748

}
```

Notes sur l'exemple 3

```
    float *pt = &x;      la valeur de pt est l'adresse de x
                         → pt pointe sur x

    int *pt2 = &pt;      la valeur de pt2 est l'adresse de pt
                         → pt2 pointe sur pt
```

```
*pt vaut la valeur de la variable pointée par pt
    vaut la valeur de x
    vaut 1.1

*pt2 vaut la valeur de la variable pointée par pt2
     vaut la valeur de pt
     vaut 2686748
```

Espace mémoire relatif aux exemples

Nom	Adresse	valeur
x	2686748	1.1
pt	2686744	2686748
pt2	2686740	2686744

2 – Définitions

Opérateur d'adresse.

- L'opérateur d'adresse & permet d'accéder à l'adresse d'une variable, i.e. la zone de la mémoire où la variable est stockée.

Pointeur.

- Un pointeur `pt` est une variable qui a pour valeur l'adresse d'une autre variable `x`.
- On dit que `pt` pointe sur `x`.
- L'opérateur d'indirection * permet d'accéder directement à la valeur de la variable pointée. Ainsi, si `pt` est un pointeur vers un flottant `x`, `*pt` désigne la valeur de `x`.

Propriété.

- L'adresse d'une variable est un entier, quelque soit le type de cette variable.
- Conséquence : la valeur d'un pointeur est un entier, quelque soit le type de la variable sur laquelle il pointe.

Remarque.

- Même si la valeur d'un pointeur est toujours un entier, le type d'un pointeur dépend du type de la variable vers lequel il pointe.

Passage des paramètres – 1

```
void afficher(float a, float b)
{
    printf("a=%.1f ; b=%.1f \n", a, b);
    printf("&a=%d ; &b=%d \n", &a, &b);
}

void main()
{
    float a=1.1, b=2.2;

    // a (main) et b (main)
    printf("a=%.1f ; b=%.1f \n", a, b);
      ~~> a=1.1 ; b=2.2
    printf("&a=%d ; &b=%d \n", &a, &b);
      ~~> &a=2686748 ; &b=2686744

    // a (afficher) et b (afficher)
    afficher(a, b);
      ~~> a=1.1 ; b=2.2
      ~~> &a=2686704 ; &b=2686708
}
```

Espace mémoire relatif

Nom	Adresse	valeur
a (main)	2686748	1.1
b (main)	2686744	2.2
b (afficher)	2686708	1.1
a (afficher)	2686704	2.2

Passage des paramètres – 2

```
void afficher(float *pt1, float *pt2)
{
    printf("pt1=%d ; pt2=%d \n", pt1, pt2);
    printf("*pt1=%.1f ; *pt2=%.1f \n", *pt1, *pt2);
    printf("&pt1=%d ; &pt2=%d \n", &pt1, &pt2);
}

void main()
{
    float a=1.1, b=2.2;
    float *pt1, *pt2;
    pt1=&a;
    pt2=&b;

    // pt1 (main) et pt2 (main)
    printf("pt1=%d ; pt2=%d \n", pt1, pt2);
      ~~> pt1=2686748 ; pt2=2686744
    printf("*pt1=%.1f ; *pt2=%.1f \n", *pt1, *pt2);
      ~~> *pt1=1.1 ; *pt2=2.2
    printf("&pt1=%d ; &pt2=%d \n", &pt1, &pt2);
      ~~> &pt1=2686740 ; &pt2=2686736

    afficher(pt1, pt2);
      // pt1 (afficher) et pt2 (afficher)
      ~~> pt1=2686748 ; pt2=2686744
      ~~> *pt1=1.1 ; *pt2=2.2
      ~~> &pt1=2686704 ; &pt2=2686708
}
```

Espace mémoire relatif

Nom	Adresse	valeur
a (main)	2686748	1.1
b (main)	2686744	2.2
pt1 (main)	2686740	2686748
pt2 (main)	2686736	2686744
pt2 (afficher)	2686708	2686744
pt1 (afficher)	2686704	2686748

Passage des paramètres – 3

```
void echange_ko(float a, float b) {
    float tmp = a;
    a = b;
    b = tmp;
    printf("a=%.1f b=%.1f \n", a, b);
    printf("&a=%d &b=%d \n", &a, &b);
}
void echange_ok(float *pt1, float *pt2) {
    float tmp = *pt1;
    *pt1 = *pt2; // a = *pt2 car pt1 pointe vers a
    *pt2 = tmp; // b = tmp car pt2 pointe vers b

    printf("pt1=%d pt2=%d\n", pt1, pt2);
    printf("*pt1=%.1f *pt2=%.1f\n", *pt1, *pt2);
    printf("&pt1=%d &pt2=%d\n", &pt1, &pt2);
}
void main() {
    float a=1.1, b=2.2;
    float *pt1, *pt2;
    pt1=&a;
    pt2=&b;

    printf("a=%.1f b=%.1f \n", a, b);
        ~~> a=1.1 b=2.2
    printf("&a=%d &b=%d \n", &a, &b);
        ~~> &a=2686748  &b=2686744

    echange_ko(a, b);
        ~~> a=2.2 b=1.1
        ~~> &a=2686704  &b=2686708
    printf("a=%.1f b=%.1f \n", a, b);
        ~~> a=1.1 b=2.2

    printf("pt1=%d pt2=%d\n", pt1, pt2);
        ~~> pt1=2686748  pt2= 2686744
    printf("*pt1=%.1f *pt2=%.1f\n", *pt1, *pt2);
        ~~> a=1.1 b=2.2
    printf("&pt1=%d &pt2=%d\n", &pt1, &pt2);
        ~~> &pt1=2686740  &pt2= 2686736
```

Espace mémoire relatif

Nom	Adresse	Valeur
a (main)	2686748	1.1
b (main)	2686744	2.2
pt1 (main)	2686740	2686748
pt2 (main)	2686736	2686744
b (echange_ko)	2686708	1.1
pt2 (echange_ok)	2686708	2686744
a (echange_ko)	2686704	2.2
pt1 (echange_ok)	2686704	2686748

```
    echange_ok(pt1, pt2);
         ~~> pt1=2686748  pt2= 2686744
         ~~> *pt1=2.2 *pt2=1.1
         ~~> &pt1=2686704  &pt2= 2686708

    printf("a=%.1f b=%.1f \n", a, b);

         ~~> a=2.2 b=1.1

}
```

Passage des paramètres – 4

```
void echange_ko(float a, float b) {
    float tmp = a;
    a = b;
    b = tmp;
    printf("a=%.1f b=%.1f \n", a, b);
    printf("&a=%d &b=%d \n", &a, &b);
}
void echange_ok(float *pt1, float *pt2) {
    float tmp = *pt1;
    *pt1 = *pt2; // a = *pt2 car pt1 pointe vers a
    *pt2 = tmp; // b = tmp car pt2 pointe vers b

    printf("pt1=%d pt2=%d\n", pt1, pt2);
    printf("*pt1=%.1f *pt2=%.1f\n", *pt1, *pt2);
    printf("&pt1=%d &pt2=%d\n", &pt1, &pt2);
}
void main() {
    float a=1.1, b=2.2;
    float *pt1, *pt2;
    pt1=&a;
    pt2=&b;

    printf("a=%.1f b=%.1f \n", a, b);
        ~~> a=1.1 b=2.2
    printf("&a=%d &b=%d \n", &a, &b);
        ~~> &a=2686748  &b=2686744

    echange_ko(a, b);
        ~~> a=2.2 b=1.1
        ~~> &a=2686704  &b=2686708
    printf("a=%.1f b=%.1f \n", a, b);
        ~~> a=1.1 b=2.2

    printf("pt1=%d pt2=%d\n", pt1, pt2);
        ~~> pt1=2686748  pt2= 2686744
    printf("*pt1=%.1f *pt2=%.1f\n", *pt1, *pt2);
        ~~> a=1.1 b=2.2
    printf("&pt1=%d &pt2=%d\n", &pt1, &pt2);
        ~~> &pt1=2686740  &pt2= 2686736
```

Espace mémoire relatif

Nom	Adresse	Valeur
a (main)	2686748	1.1
b (main)	2686744	2.2
pt1 (main)	2686740	2686748
pt2 (main)	2686736	2686744
b (echange_ko)	2686708	1.1
pt2 (echange_ok)	2686708	2686744
a (echange_ko)	2686704	2.2
pt1 (echange_ok)	2686704	2686748

```
    echange_ok(pt1, pt2);
         ~~> pt1=2686748  pt2= 2686744
         ~~> *pt1=2.2 *pt2=1.1
         ~~> &pt1=2686704  &pt2= 2686708

    printf("a=%.1f b=%.1f \n", a, b);

         ~~> a=2.2 b=1.1

}
```

Tableaux

Equivalences

Pour un certain tableau `tab`, et un indice `i` correctement défini, on a les équivalences suivantes :

- `&tab[i] = tab+i`
- `tab[i] = *(tab+i)`

- `tab` est un pointeur constant (non modifiable) dont la valeur est l'adresse du premier élément du tableau.
- `tab+1` est un pointeur constant (non modifiable) dont la valeur est l'adresse du 2-ième élément du tableau.
- `tab+i` est un pointeur constant (non modifiable) dont la valeur est l'adresse du *i*-ième élément du tableau.

Exemple 1 : tableau d'entier (type `int`)

```
void main() {
    int t[5] = {10, 11, 12, 13, 14};
    int i;
    for (i=0; i<5; i++)
        printf("%d  @ %d\n", t[i], &t[i]);
        ~~> 10  @ 2686716
        ~~> 11  @ 2686720
        ~~> 12  @ 2686724
        ~~> 13  @ 2686728
        ~~> 14  @ 2686732

    for (i=0; i<5; i++)
        printf("%d  @ %d\n", *(t+i), t+i);
        ~~> 10  @ 2686716
        ~~> 11  @ 2686720
        ~~> 12  @ 2686724
        ~~> 13  @ 2686728
        ~~> 14  @ 2686732
}
```

Espace mémoire relatif

Nom	Adressse	Valeur
t[0]	2686716	10
t[1]	2686720	11
t[2]	2686724	12
t[3]	2686728	13
t[4]	2686732	14

Nom	Adressse	Valeur
t[0]	**2686716**	10
	2686717	
	2686718	
	2686719	
t[1]	**2686720**	11
	2686721	
	2686722	
	2686723	
t[2]	**2686724**	12
	2686725	
	2686726	
	2686727	
t[3]	**2686728**	13
	2686729	
	2686730	
	2686731	
t[4]	**2686732**	14
	2686733	
	2686734	
	2686735	

Exemple 2 : tableau de flottant (type `double`)

```
double t[5] = {0.0, 0.1, 0.2, 0.3, 0.4};
int i;
for (i=0; i<5; i++)
    printf("%.1f  @ %d\n", t[i], &t[i]);
     ~~> 0.0  @ 2686696
     ~~> 0.1  @ 2686704
     ~~> 0.2  @ 2686712
     ~~> 0.3  @ 2686720
     ~~> 0.4  @ 2686728

printf("\n");
for (i=0; i<5; i++)
    printf("%.1f  @ %d\n", *(t+i), t+i);
     ~~> 0.0  @ 2686696
     ~~> 0.1  @ 2686704
     ~~> 0.2  @ 2686712
     ~~> 0.3  @ 2686720
     ~~> 0.4  @ 2686728
```

Espace mémoire relatif

Nom	Adressse	Valeur
t[0]	2686696	0.0
t[1]	2686704	0.1
t[2]	2686712	0.2
t[3]	2686720	0.3
t[4]	2686728	0.4

Nom	Adressse	Valeur
t[0]	**2686696**	0.0
	2686697	
	2686698	
	2686699	
	2686700	
	2686701	
	2686702	
	2686703	
t[1]	**2686704**	0.1
	2686705	
	2686706	
	2686707	
	2686708	
	2686709	
	2686710	
	2686711	
t[2]	**2686712**	0.2
	2686713	
	2686714	
	2686715	
	2686716	
	2686717	
	2686718	

	2686719	
t[3]	**2686720**	0.3
	2686721	
	2686722	
	2686723	
	2686724	
	2686725	
	2686726	
	2686727	
t[4]	**2686728**	0.4
	2686729	
	2686730	
	2686731	
	2686732	
	2686733	
	2686734	
	2686735	

Allocation dynamique de la mémoire

- **Allocation statique :** réservation d'espace lors de la compilation du programme.
- **Allocation dynamique :** réservation d'espace lors de l'exécution du programme.

Fonctions pour l'allocation dynamique

Les fonctions pour l'allocation dynamique se situent dans l'en-tête **stdlib.h**.

Nous nous intéressons dans ce cours à :

- `malloc()`
- `free()`

Fonction **`malloc`** :

- Signature : `void * malloc (size_t t)`
- Demande au système d'exploitation un bloc mémoire de taille `t` ; et retourne un pointeur vers l'addresse du bloc alloué ; s'il se produit une erreur, la valeur **NULL** est retournée.
- L'allocation par `malloc` s'effectue dans une zone de la mémoire appelée : **tas**.
- Le pointeur retourné est générique (de type : `void *`) ; donc peut être converti implicitement en un pointeur non générique.
- Une application doit toujours contrôler le résultat d'une allocation avant d'utiliser la zone mémoire en question.
- La connaissance du type de pointeur intervient dans les calculs arithmétiques portant sur ce pointeur.

Procédure **`free()`** :

- Signature : `void free(void *pt)`
- Libère un emplacement préalablement alloué.
- A toute instruction de type `malloc` doit être associée une instruction de type `free`.
- Bonne pratique : `free(pt); pt=NULL;`

- Attention aux fuites mémoires :
  - = parties de la mémoire demeurant réservés sans avoir été désalloués ;
  - Schéma pouvant mener à une fuite mémoire :
    - Allocation d'une partie de la memoire ;
    - Perte de l'adresse de cette partie de la mémoire ;
    - Conséquence : cette partie de la mémoire devient inaccessible pour un usage ultérieur .
  - Exemple de fuite mémoire :

```
void fuite_memoire(int n, int k)
{
    int i;
    int *p;
    for (i=0; i<n; i++)
    {
        p=malloc(sizeof(int)*k);
        // on écrase la valeur précédente de p par une nouvelle
    }
    free(p);
}
```

Composition de la mémoire

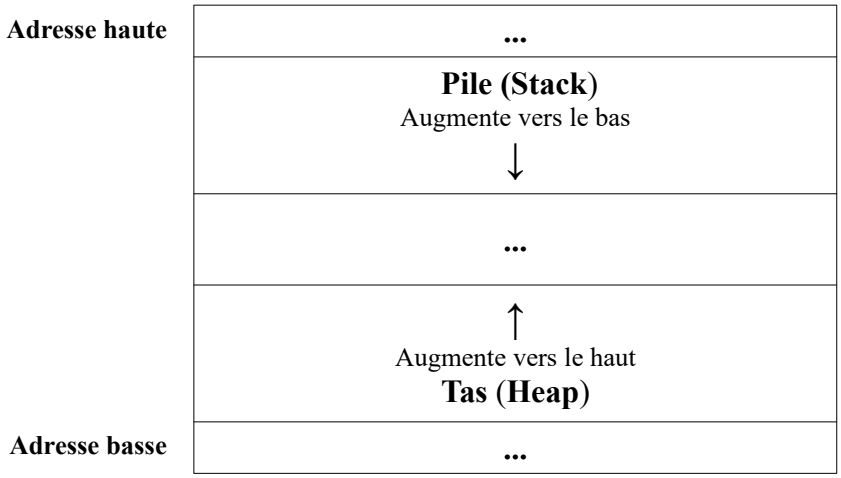
Première approximation

- Suites d'octets (octet = bloc de 8 bits) ;
- Chaque octet est repéré par une adresse, qui est un nombre entier.

Adresse	Octet
0	0 1 1 1 0 0 0 1
1	0 0 0 1 0 1 0 1
2	1 1 0 1 0 0 1 0
3	0 1 1 1 0 1 0 1
...	

Deuxième approximation

- Pile :
  - Stockage des variables locales ;
  - Passage d'arguments aux fonctions ;
  - … ;
  - Structure LIFO (*Last In First Out*) ~ pile d'assiettes ;
- Tas :
  - Stockage des données allouées dynamiquement.



III. Documentation technique

Spécificateurs de format pour `printf`

SYMBOLE	*TYPE*	*IMPRESSION COMME*
`%d ou %i`	**`int`**	entier relatif
`%u`	**`int`**	entier naturel (unsigned)
`%o`	**`int`**	entier exprimé en octal
`%x`	**`int`**	entier exprimé en hexadécimal
`%c`	**`int`**	caractère
`%f`	**`double`**	rationnel en notation décimale
`%e`	**`double`**	rationnel en notation scientifique
`%s`	**`char*`**	chaîne de caractères

Langage C – Impression de caractères spéciaux

`\a` : ***a****lert*, bip sonore ;
`\b` : ***b****ackspace*, espace arrière ;
`\f` : ***f****ormfeed*, saut de page ;
`\n` : ***n****ewline*, saut de ligne ;
`\r` : *carriage* ***r****eturn*, retour chariot en début de ligne ;
`\t` : *horizontal* ***t****ab*, tabulation horizontale ;
`\v` : ***v****ertical tab*, tabulation verticale ;
`\\` : *backslash* ;
`\'` : simple quote ;
`\"` : double quote ;
`\?` : point d'interrogation ;

Programmation en C – Passage d'arguments à un programme

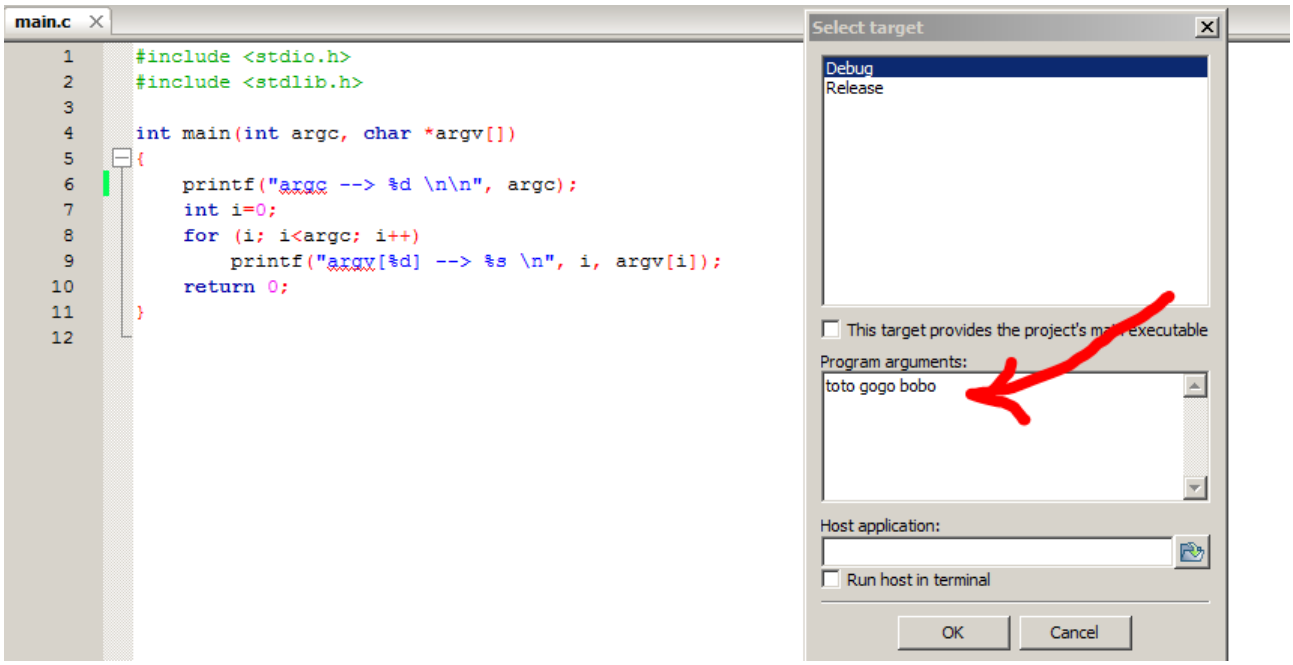
- `argc` est le nombre de paramètres effectivement passé au programme. Ce nombre est toujours au moins égal à 1 car le premier paramètre est toujours le nom de l'exécutable.
- `argv` est un tableau de chaine de caractères contenant les paramètres effectivement passés au programme.

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

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

Sous Code::Blocks :

« Project » → « Set programs' argument... »



Après compilation et exécution :

```
argc --> 4
```

```
argv[0] --> C:\Users\dupont\test\bine\Debug\test.exe
argv[1] --> toto
argv[2] --> gogo
argv[3] --> bobo
```