if - else

En combinant plusieurs structures **if - else** en une expression nous obtenons une structure qui est très courante pour prendre des décisions entre plusieurs alternatives:

if (<expr1>)

<bloc1>

else if (<expr2>)

<bloc2>

else if (<expr3>)

<bloc3>

else if (<exprN>)

<blocN>

else <blocN+1>

Les expressions <expr1> ... <exprN> sont évaluées du haut vers le bas jusqu'à ce que l'une d'elles soit différente de zéro. Le bloc d'instructions y lié est alors exécuté et le traitement de la commande est terminé.

L’opérateur conditionnel

<expr1> ? <expr2> : <expr3>

** Si <expr1> fournit une valeur différente de zéro, alors la valeur de <expr2> est fournie comme résultat.*

** Si <expr1> fournit la valeur zéro, alors la valeur de <expr3> est fournie comme résultat.*

Exemple:

La suite d'instructions:

```
if (A>B)
    MAX=A;
else
    MAX=B;
```

peut être remplacée par:

```
MAX = (A > B) ? A : B;
```

La structure while

while (<**expression**>)

<**bloc d'instructions**>

** Tant que l'<expression> fournit une valeur différente de zéro, le <bloc d'instructions> est exécuté.*

** Si l'<expression> fournit la valeur zéro, l'exécution continue avec l'instruction qui suit le bloc d'instructions.*

** Le <bloc d'instructions> est exécuté zéro ou plusieurs fois.*

La partie <**expression**> peut désigner:

une variable d'un type numérique,

une expression fournissant un résultat numérique.

La partie <**bloc d'instructions**> peut désigner:

un (vrai) bloc d'instructions compris entre accolades,

une seule instruction terminée par un point-virgule.

La structure do – while

do

<**bloc d'instructions**>

while (<**expression**>);

Le <bloc d'instructions> est exécuté au moins une fois et aussi longtemps que l'<expression> fournit une valeur différente de zéro.

En pratique, la structure **do - while** n'est pas si fréquente que **while**; mais dans certains cas, elle fournit une solution plus élégante.

Exemple:

```
float N;
do
{
   printf("Introduisez un nombre entre 1 et 10 :");
   scanf("%f", &N);
} while (N<1 || N>10);
```

La structure for

for (<**expr1**> ; <**expr2**> ; <**expr3**>)

<**bloc d'instructions**>

est équivalent à:

<**expr1**>;

while (<**expr2**>)

{

<**bloc d'instructions**>

<**expr3**>;

}

<expr1> est évaluée une fois avant le passage de la boucle. Elle est utilisée pour initialiser les données de la boucle.

<expr2> est évaluée avant chaque passage de la boucle. Elle est utilisée pour décider si la boucle est répétée ou non.

<expr3> est évaluée à la fin de chaque passage de la boucle. Elle est utilisée pour réinitialiser les données de la boucle.

Choix de la structure répétitive

* Si le bloc d'instructions ne doit pas être exécuté si la condition est fausse, alors utilisez **while** ou **for**.

* Si le bloc d'instructions doit être exécuté au moins une fois, alors utilisez **do - while**.

* Si le nombre d'exécutions du bloc d'instructions dépend d'une ou de plusieurs variables qui sont modifiées à la fin de chaque répétition, alors utilisez **for**.

* Si le bloc d'instructions doit être exécuté aussi longtemps qu'une condition extérieure est vraie (par exemple aussi longtemps qu'il y a des données dans le fichier d'entrée), alors utilisez **while**.

Le choix entre **for** et **while** n'est souvent qu'une question de préférence ou d'habitudes.

Fonctions

- Les fonctions sont les briques de base d'un programme C. Elles nous permettent de découper des programmes en parties plus petites donc plus faciles à relire et à mettre au point.

- Une fonction est un sous-programme qui effectue un travail bien précis, à laquelle on passe (généralement) des données et qui retourne (le plus souvent) une valeur.

- Quand une fonction est appelée, l'exécution du programme est transférée à la première instruction de cette fonction. L'exécution de la fonction se termine après l'instruction *return* ou après la dernière instruction du bloc de la fonction.

- Quand la fonction appelée a terminé, l'exécution dans la fonction appelante, reprend à l'endroit de l'expression où l'appel avait été fait. La valeur retournée par la fonction peut être alors utilisée comme opérande dans cette expression.

L'instruction return

Cette instruction:

- permet de renvoyer la valeur précisée en argument à l'instruction appelante,
- provoque une sortie immédiate du bloc principal de la fonction.

Exemples:

return 12; /* la valeur entière est retournée */

return ; /* la valeur aléatoire est retournée (rien pour une fonction *void*) */

return (a > b) ? a : b ; /* la valeur d'une expression est retournée */

L'appel d'une fonction

L'appel d'une fonction est réalisé en signifiant le nom de la fonction suivi de la liste des paramètres réels.

Exemples:

b = **theta ('a' , 180 , v)** ;

pi2 = 2 * **pi()** ;

cls() ;

Passage des paramètres

Quand on invoque une fonction, on doit généralement lui passer des valeurs. Du **coté appelant**, les valeurs passées sont appelées paramètres réels ou paramètres effectifs ou **arguments**.

Par exemple, dans l'appel suivant, deux arguments, une variable (s1) et une constante chaîne de caractères, sont passés :

strcpy(s1 , "Le langage C");

Du **coté appelé**, les variables spécifiées sont appelées paramètres formels ou **paramètres**.

Le mode de transmission est effectué par valeur: la fonction travaille donc sur une copie des paramètres réels.

En compilation classique, l'ordre d'évaluation des paramètres n'est pas défini en langage C. Ainsi dans l'instruction suivante :

toto (++i , tab[i]);

est-ce i ou i incrémenté de 1 qui est l'index de tab ?

La fonction main

Les arguments de la ligne de commande qui lance l'exécution d'un programme sont fournis au programme sous la forme d'un tableau de pointeurs. Pour pouvoir les utiliser, il faut déclarer la fonction *main* de la façon suivante:

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

argc contient le nombre d'arguments de la ligne de commande,

argv est un tableau de pointeurs sur les arguments de la ligne de commande.

Exemple:

```
/* affiche les arguments de la ligne de commande, version tableau */
        int main(int argc, char *argv[])
        {       int i;
                for ( i = 0 ; i < argc ; i++ )
                        printf("%s\n", argv[i]);
        }
```

Déclaration de variables externes

But: déclarer une variable ou une fonction définie dans un autre fichier (on dira aussi "importer").

Visibilité: selon la déclaration (locale ou globale)

Mot clé: **extern**

Durée de vie: le programme

Remarques:

- Possible pour les fonctions non définies en **static**;

- Possible pour les variables globales non définies en **static**.

Exemples de liaison interne/externe:

extern int ma_valeur_int;

extern float ma_valeur_float;

extern void f(**int**);

Les règles de portée

Fonction

visibilité: de la *déclaration* à la *fin du fichier*;

accessible: dans d'autres modules avec **extern** SAUF si déclarée en **static**;

durée de vie: programme.

Variable dans un bloc

visibilité: de la *déclaration* à la *fin du bloc*;

initialisation: aucune par défaut;

durée de vie: bloc (auto).

Variable globale

visibilité: de la *déclaration* à la *fin du fichier*;

accessible: dans d'autres modules avec **extern** SAUF si déclarée en **static**;

initialisation: 0 par défaut;

durée de vie: programme.

```
/* Utilisation des variables statiques */
# include <stdio.h>
void trystat (void); /* le prototype de la fonction trystat*/

main (void)
{
        int i;
        for ( i = 1 ; i <= 3 ; i++ )
        {
                printf("Appel %d \n", i);
                trystat();
                printf("Appel %d \n", i);
                trystat();
        }
        return 0;
}

void trystat (void)
{
        int auto_v = 1;
        static int stat_v = 1;

        printf("auto_v = %d   stat_v = %d \n",  auto_v++,   stat_v++);
}
```

```
/* Utilisation des variables */
# include <stdio.h>
void f (void); /* le prototype de la fonction f*/
void f1 (void); /* le prototype de la fonction f1*/
int var = 7; /* la variable globale initialisée avec la valeur 7*/
main ()
{
        int var = 10; /* la variable locale, qui cache la variable globale var */
        printf("var = %d \n", var ++); /* imprime var = 10 */
        {
                int var =200; /* la variable locale cachant la variable locale var */
                printf("var = %d \n", var ++); /* imprime var = 200 */
        }
        printf("var = %d \n", var ++); /* imprime var = 11 */
        f(); printf("var = %d \n", var ++); /* imprime var = 12 */
        f1(); printf("var = %d \n", var ++); /* imprime var = 13 */
        f1(); printf("var = %d \n", var ++); /* imprime var = 14 */
}
void f (void)
{
        int var = 77; /* encore une variable locale */
        printf("var = %d \n", var ++); /* imprime var = 77 */
}
void f1 (void)
{
        printf("var = %d \n", var ++); /* imprime var = 7 ou 8 (la valeur globale) */
}
```

Le préprocesseur

C'est la première phase d'un processus de compilation. Elle est lancée automatiquement par la commande *cc*. Le préprocesseur est chargé d'effectuer certaines actions préliminaires avant la compilation.

Son activité est uniquement une transformation de texte afin de produire un nouveau fichier. Les principales **tâches** du **préprocesseur** sont:

l'élimination des commentaires,

l'inclusion de fichier source,

la substitution de texte,

la définition de macros,

la compilation conditionnelle.

Une ligne commençant par le caractère # (**dièse**) permet de définir une directive au préprocesseur.

Le préprocesseur peut être lancé indépendamment en exécutant le programme */lib/cpp* sur un fichier texte pouvant être autre chose qu'un programme C.