Tutorial TIGCC 68k
C/ASM

Par Quiche Team

Introduction

Ce tutorial est écrit pour les débutant qui ne connaissent rien à la programmation mais aussi à ceux qui maîtrisent la programmation sur ordinateur et qui veulent apprendre le langage C avec tigcc. Tigcc est le langage C utilisé sur ti-89 et ti-92+. Les premières leçons sont spécialement faites pour les débutants. Les autres peuvent lire en vitesse pour peut-être apprendre de nouvelles petites choses.

Nous allons commencer par vous faire un petit historique. Ce langage avec son interface a été créé par une équipe internationale. Tigcc a été créé au début de l'année 2000. Depuis cette création, la communauté Tigcc devient plus grande et il y a de plus en plus de programmeur de C! Si vous voulez plus d'information, je vous conseille de visiter le site officiel.

Tigcc est un freeware génial car vous pouvez faire vos programmes sans autres softwares. Vous pouvez écrire et compiler vos codes avec le même software. Cela constitue sa grande différence en opposition avec l'asm. Grâce à tigcc, vous n'aurez pas besoin d'écrire votre programme avec notepad puis de le compiler avec un autre software et finalement de revenir sur notepad pour voir vos erreurs. En plus, si vous lancez VTi, vous pouvez tester vos programmes sans arrêter le développement de tigcc. VTi est le second software dont vous aurez besoin. Il est utile et lui aussi est gratuit ! :) Et si vous êtes curieux ou un peu expérience dans la programmation, vous pouvez voir le code source de tigcc : c'est une source ouverte.

VTi est important car vous n'aurez pas de soucis si votre programme comporte des erreurs pouvant endommager votre calculatrice. Vous ne détruirez pas votre ordinateur avec un bug dans votre programme; tous les utilisateurs de Microsoft le savent bien. :)

Nous allons maintenant décrire tigcc. Nous ne vous donnerons pas trop de détails mais seulement le nécessaire pour commencer. Si vous avez déjà une expérience sur le langage C, vous n'aurez pas trop de problèmes pour utiliser tigcc. N'ayez pas peur si vous ne connaissez pas le langage C, nous allons vous donner le plus d'exemples possible pour vous aider. Si vous ne comprenez pas, si vous voulez plus d'explications et/ou si vous ne voyez pas vos erreurs, nous restons à votre disposition et vous pouvez nous envoyer un mail. Surtout n'hésitez pas !!!

Donc tigcc est bien pour faire vos programmes qui se lanceront très vite sur vos calculatrices. Les plus optimisés des programmes basics sont misérables face à un programme évolué de tigcc. Maintenant, il y a aussi l'asm. Ces programmes en assembleur sont plus rapides que ceux de tigcc. Par contre il est plus facile de faire un bon code avec tigcc. Comme la différence de vitesse est vraiment faible entre les 2 et que tigcc est plus claire et plus simple à comprendre nous avons choisi de vous apprendre ce langage. Malgré notre préférence pour tigcc, l'asm est vraiment bien pour sa rapidité mais il faut qu'il soit bien codé et ça, ce n'est pas

forcément évident!

De plus, vous pouvez faire des programmes pour ti-89, ti-92+et V200. Cela est très simple car il y a une fonction qui vous donne la taille du LCD. Le plus difficile vient de la différence des touches et donc de trouver les bonnes pour tester vos programme sur votre calculatrice.

Pour éviter des leçons ennuyeuses, nous avons rajouté un grand nombre d'exemples sympathiques. Nous allons commencer la description étape par étape avec un maximum d'exemples afin d'aider le plus possible les débutants.

Commençons maintenant ! Et n'oubliez pas que programmer est amusant et le but est de passer du bon temps !! :)

Leçon 1 - Affichage

---

1) Installation de Tigcc

Vous aurez besoin de télécharger Tigcc development environment et VTi.

Avant que l'on vous explique les fonctions, vous devez apprendre à utiliser Tigcc. C'est très simple. Suivez les étapes:
- Lancer Tigcc development et Vti, à moins que vous ne vouliez vraiment utiliser votre calculatrice (mais dans ce cas il ne faut pas avoir peur de l'endommager :-( ). De plus, vous devrez configurer Tigcc pour lui montrer le chemin d'accés à Vti. Afin que Tigcc puisse utiliser directement Vti au lancement d'un code.
- Créez un nouveau projet car votre programme peut être plus complexe qu'un seul fichier!.
- Créez un nouveau fichier C; c'est le but de ce site! :-)
- Et vous pouvez renomer ce fichier pour que se soit plus claire pour vous.

Quand vous créez un nouveau fichier C, le "template wizard" s'affiche sur l'écran et vous devez choisir les options. Nous vous conseillons de prendre les options par défaut sauf pour "optimized call rom".

Premièrement, vous devez choisir une calculatrice. Pour un programme simple, vous ne devriez pas avoir de problème. Il y a des problèmes de touches et tout particulièrement pour les touches de direction et les tailles différentes des écrans mais nous verrons cela plus tard.

Deuxièmement, vous aurez le choix entre avec "kernel" ou sans. Ce sont des librairies nécessaire pour l'asm (encore un avantege que l'asm ne possède pas par rapport à Tigcc :-( ). Nous vous conseillons de prendre l'option sans kernel car sinon il peut y avoir des problèmes de compatibilité. Un programme avec kernel est plus puissant mais le risque de bug est plus grand et pour l'instant nous ne maîtrisons pas cet aspect de la programmation! Après vous pouvez, et nous vous le conseillions de choisir l'option "optimized ROM calls". Comme cela votre programme sera plus court et plus rapide!

Finalement, vous pouvez retournez une valeur dans le TIOS et choisir des options de sorti: "Done" qui est la valeur commune quand vous sortez de votre programme. La dernière option sert à sauvegarder et restaurer l'écran LCD pour qu'à la fin d'un programme, il retrouve son affichage d'avant le lancement du programme.

Vous obtenez:

```
// Donnée souce C
// Créé le 18/05/2002; 22:35:52

#define USE_TI89              // Utilisation pour Ti 89
#define USE_TI92PLUS          // Utilisation pour Ti 92+

#define OPTIMIZE_ROM_CALLS // utilisation de "ROM Call Optimization"

#define SAVE_SCREEN // Sauvegarde/Restaure le contenu de l'écran

#include <tigcclib.h> // Inclu toutes les têtes de lignes

// Fonction Principale
void _main(void)
{
// Placez ici votre code.
}
```

Les directives "#define ..." dépendent des cases sélectionnées. Et #include <tigcclib.h> est un englobement qui inclu toutes les librairies avec les fonctions ou autres définitions.

A chaque fois que le compileur voit "//" alors il ignore la suite. C'est un commentaire et c'est très important pour comprendre les programmes des autres programmeurs et vos anciens programmes. Cela est très utlile surtout pour les codes très long et très complexes. Pour faire un commentaire, il y a une autre méthode. C'est de commancer par "/*" et de finir par "*/".

```
// première exemple de commentaires

/*
deuxième exemple de commentaires
deuxième ligne de commentaires
...
*/
```

top

2) L'exemple classique du "Hello!"

Nous allons voir comment faire pour mettre un texte à l'écran. Sur les Ti 89/92+, il existe 3 types de polices. Il existe différentes manières pour les afficher. Premièrement, nous allons voir la plus commune des fonctions mais pas forcément la plus rapide et la plus petite en taille. La syntaxe est **printf("le texte")**. Vous créez une nouvelle ligne C et vous l'ajoutez à votre code entre { et }. Vous lancez votre code avec le bouton "run" ou avec la touche de raccourcie "F9".


```
// Données sources C
// Créé le 18/05/2002; 22:35:52

#define USE_TI89              // Utilisation pour Ti 89
#define USE_TI92PLUS          // Utilisation pour Ti 92+

#define OPTIMIZE_ROM_CALLS    // Utilisation de ROM Call Optimization

//#define SAVE_SCREEN           // Sauvegarde/Restaure le contenu de
l'écran

#include <tigcclib.h>         // Inclu toutes les têtes de lignes


// Fonction Principale
void _main(void)
{
  clrscr();  // Efface l'écran
  printf("Helo!!!"); // Affiche la chaîne: "Hello!!!"
  ngetchx(); // Attente qu'une touche soit pressée
}
```

Un fichier C banal comporte toujours les sections suivantes:

- Lien de directive
- section d'inclusion
- _main()

void _main(void) { ... } est comme votre programme principal. "(void)" signifie que vous n'avez pas d'argument et "void" sans parenthèse signifie qu'aucune valeur n'est retourné au TiOS quand le programme est fini. “Void _main(void)” est la première procédure qui est mis en marche, nous verrons plus tard et en détail cette notion de procédure. Vous écrivez votre code à l'intérieur des crochets: { }. Ces crochets sont importants et ils montrent le début et la fin de votre programme. Généralement, ils montrent le début et la fin du corps d'une fonction.

clrscr() efface l'écran et met le curseur à la position (0,0). Vous pouvez voir que chaque ligne se termine par";", c'est une des spécificité du langage C. En principe, vous pouvez mettre des arguments à l'intérieur des parenthèses mais pour cette fonction vous mettez rien puisqu'elle n'a pas d'argument.

printf("Hello!!!") affiche la chaîne de caractère: "Hello!!!" à la position du curseur qui est initialisée par la fonction clrscr(). Vous devez comprendre qu'une chaîne est un assemblement de caractère: du texte. J'utilise le mot: "chaîne" plutôt que texte car une chaîne est un type de variable et il est généralement défini dans les langages de programmation. Je vous expliquerez les différents types du langage C dans les prochaines lçons.

Vous devez être attentif avec ces deux fonctions puisque ces fonctions n'existent pas dans le TiOS. En fait, ces fonctions sont créés à partir d'autre fonctions ainsi elles sont moins rapide et tiennent plus de place que les fonctions du Tios. En effet, elles sont décrites dans la Tigcc librairie et il y a un code qui permet de faire fonctionner ces fonctions à partir d'autres fonctions qui sont inclus dans le tios. La taille réelle de ces fonctions est ce code. Ces fonctions sont plus maniables mais vous devez être attentif si vous voulez optimisez votre code.

ngetchx() attend qu'une touche soit appuyez sinon votre programme affiche très rapidement le texte et ensuite le programme s'arrête mais vous ne pouvez pas voir le texte. de plus, cette fonction est capable de donner le nom de la touche appuyée je vous expliquerez cela plus tard.

top

3) L'exemple classique du "Hello!!!" avce les fonctions du TiOS


```
// C Source File
// Created 18/05/2002; 23:43:27

#define USE_TI89              // Produce .89z File
#define USE_TI92PLUS          // Produce .9xz File

#define OPTIMIZE_ROM_CALLS    // Use ROM Call Optimization

#define SAVE_SCREEN           // Save/Restore LCD Contents

#include <tigcclib.h>         // Include All Header Files

// Main Function
void _main(void)
{
  ClrScr();
  DrawStr(0,0,"Hello!!!",A_NORMAL);

  ngetchx();
}
```

ClrScr() efface seulement l'écran LCD. Vous devez faire attention car en langage C , une majuscule est différente d'une minuscule. Par exemple, clrscr() est différent de ClrScr(). Ce qui signifie que ClrScr() conserve la position du curseur et si vous utilisez printf, la chaîne ne s'affichera pas obligatoirement en haut et à gauche de l'écran.

DrawStr(0,0,"Hello!!!",A_NORMAL) affiche la chaîne à la position (0,0). A_NORMAL est la façon d'afficher le texte. Si vous voulez l'afficher à une autre position; par exemple: (x,y), DrawStr(x,y,"Hello!!!",A_NORMAL), x est le nombre de pixel en partant de la gauche et y est le nombre de pixel en partant du haut. Les dfifférentes façons d'afficher sont:
A_NORMAL: le caractère s'additionne sur les pixels déjà présent
A_REVERSE: on a une inversion des pixels par rapport à la destination
A_REPLACE: le caractère remplace la surface de destinatin
A_XOR: les pixels sont affichés ou masqués si la destination et le caractère sont identiques
A_SHADED: le caractère est masqué afin que chaque autre pixel est tourné off alors la destination est additionné

Mais si vous voulez avoir d'autres détail je vous conseille de regarder la documentation de Tigcc, surtout c'est difficile de se souvenir de tous les détails, par exemple, les différentes façons d'affichage et c'est beaucoup plus facile de jeter un coup d'œil dans la documentation que de se rappeler de la syntaxe exacte quand vous programmez. N'oubliez pas que le travail de Zeljko Juric est formidable et ça serait stupide de ne pas le mettre à profit!

ngetchx() vous devrez déjà savoir ce que c'est sinon vous avez besoin de revenir en arrière.

top

4) Les polices de la Ti89 et Ti92+

Il y a trois police différentes sur les Ti89/92+. Vous pouvez sélectionner votre police à l'aide de la fonction**: FontSetSys(short font)**. L'argument de cette fonction est:
- F_6x8, qui signifie que la largeur de la police est 6 pixels et sa hauteur est 8 pixels. Cette police est mono-espacée ce qui veut dire que tous les caractères ont la même largeur. C'est la police par défaut.
- F_8x10, qui signifie que la largeur vaut 8 pixels et la hauteur 10 pixels. Cette police est aussi mono-espacée. La taille de la police est la plus grande de toutes le polices.
- F_4x6, qui signifie que la largeur vaut 4 pixels et la hauteur 6 pixels. Cette police a une largeur variable c'est-à-dire que tous les caractères n'ont pas la même largeur. Par exemple, le 'i' est plus petit en largeur que le 'w'. La taille de cette police est la plus petite.

Si vous changez de police, vous obtiendrez la police par défaut quand vous relancerez le programme. Nous pouvons voir maintenant un exemple pour faciliter la compréhension.


```
// C Source File
// Created 19/05/2002; 13:38:49
```

```
#define USE_TI89                // Produce .89z File
#define USE_TI92PLUS            // Produce .9xz File

#define OPTIMIZE_ROM_CALLS      // Use ROM Call Optimization

#define SAVE_SCREEN             // Save/Restore LCD Contents

#include <tigcclib.h>           // Include All Header Files

// Main Function
void _main(void)
{
  clrscr();

  printf("Here, the default face\n");
  FontSetSys(F_6x8);
  printf("Always the same font\n\n");

  FontSetSys(F_4x6);
  printf("The smallest font\n\n\n");

  FontSetSys(F_8x10);
  printf("The tallest font");

  ngetchx();
}
```

Je pense que vous devrez comprendre ce code. La seul difficulté est '\n' à l'intérieur de la chaîne. Chaque fois, que le compileur voit '\n', il arrête la chaîne et retourne à la ligne pour continuer à écrire la chaîne. Le caractère '\' indique au compileur qu'il y a quelque chose de spéciale. Ainsi si vous voulez afficher le caractère '\', vous êtes contraint de l'écrire deux fois pour que le compileur l'affiche une seul fois. Allez c'est pas trop dur?

Leçon 2 - Variables

---

1) Définition

Nous allons essayer de vous expliquer ce qu'est une variable le plus simplement possible. Une variable est une boîte où vous pouvez stocker des informations: réel, entier, chaîne de

caractères... Mais quand vous sortez de votre programme, ces informations sont effacées. Si vous voulez garder l'information, vous devez utiliser une autre méthode. Nous vous l'expliquerons plus tard. C'est surtout utilisé afin de mémoriser les meilleurs scores mais nous n'en aurons pas besoin pour le moment, nous devons avancer étapes par étapes, la tour Eiffel ne c'est pas construite en un jour :-).

Pour programmer vous devez stocker des informations que vous utiliserez durant tout le code. Par exemple, si vous voulez afficher le score du joueur à la fin du jeu, vous devez compter les points durant tout le jeu..

En langage C, vous devez déclarer ces variables. La déclaration se fait au début du code et ce n'est pas conseiller de les déclarer ailleurs. Cela est spécifique au langage C. Quand vous déclarer une variable, vous devez lui donner un nom et un type. Le nom est utilisé pour repérer l'information stockée sinon la variable n'est pas valable. A noter que vous ne pouvez pas donner à votre variable un nom de fonction appartenant au Tios, sinon cela crée une erreur durant la compilation; par exemple vous ne pouvez pas utiliser le nom clrscr comme nom de variable. Quand vous déclarer une variable, vous devez dire son type au compilateur car il veut savoir combien de place de mémoire il devra utiliser pour cette variable. En langage C et autres langages, vous avez beaucoup de types de variables: réel, flottant,...

Type:	**available values:**
short int	-32768 to 32767
long int	-2147483648 to 2147483647
signed char	-128 to 127 (en fait chaque caractère est un nombre dans le code ascii)
signed int	-32768 to 32767 (avec un signe par défaut) [or -2147483648 to 2147483647 si'-mnoshort' est donné]
signed short int	-32768 to 32767
signed long int	-2147483648 to 2147483647
unsigned char	0 to 255
unsigned int	0 to 65535 [or 0 to 4294967296 si '-mnoshort' est donné]
unsigned short int	0 to 65535
unsigned long int	0 to 4294967296
long long int	-4294967296 to 4294967295

unsigned long long int	0 to 8589934592

En fait, la mémoire du type est calculée en bytes ce qui donne la taille des caractères (pour un short integer "un petit entier", elle est de 2^16 soit 65536, pour un long integer "un grand entier", elle est de 2^32 soit 4294967296, pour les double long integer elle est de 2^33 soit 8589934492). Et un nombre utilise un bit de mémoire, c'est pourquoi les nombres sans signe augmentent les possibilités car le signe prend aussi une place. Par exemple, la mémoire du caractère ressemble à ---- ---- dans une boîte, vous pouvez mettre 0 ou 1 et il y a 8 cases alors vous avez 2*2*2*2*2*2*2*2 = 2^8 = 256 possibilités.

float est un type de décimal (réel) à simple précision.
double est un type de décimal (réel) à double précision alors il y aura une amplitude de 1e-999 à 9.999999999999999e999.

Un autre type de variable est le booléen mais il n'existe pas en langage C. Mais il y a un truc, vous pouvez utiliser 2 constantes: VRAI et FAUX. La valeur VRAI est 1 et FAUX est 0 donc le type de déclaration est integer. Comme cela vous pouvez créer le type booléen C'est fort non? Si vous commencez en programmation, n'ayez pas peur car nous allons vous expliquer à quoi ça sert plus tard.

En théorie, la syntaxe des variables de déclaration est : **type nom=valeur** .
L'initialisation est optimale mais vous devez être attentif car vous ne connaissez pas la valeur de la variable et vous devez l'initialiser plus tard. Quand vous sortez du programme, la valeur peut être modifiée et pour plus de sécurité vous ne devez pas utiliser de variables non initialisées car ces variables ont des valeurs prises au hasard.

'nom' est le nom de la variable. 'valeur' est sa première valeur. '=' est le symbole de l'affectation en langage C.

top

2) Votre premier programme avec variables


```
// Donnée source C
// Créé le 19/05/2002; 16:11:28

#define USE_TI89              // Utilisation pour Ti 89
#define USE_TI92PLUS          // Utilisation pour Ti 92 plus

#define OPTIMIZE_ROM_CALLS // utilisation de "ROM Call Optimization"

#define SAVE_SCREEN // Sauvegarde/Restaure le contenu de l'écran

#include <tigcclib.h> // Inclu toutes les têtes de lignes

// Fonction Principale
```

```
  void _main(void)
  {
  // différente voies de déclaration et d'initialisations des variables
  // le type entier 'integer'
    int a = 0;
    int i = 0, j = 0;
    int variable = 4;
    int bob, count = 17;

  /*
  Le type est un petit entier positif 'positive short integer'. si vous
oubliez
 non signer le compilateur fait une erreur car le nombre est trop grand
  */
    unsigned short int s = 64000, key;
  // le type est une petit entier 'short integer' et il a un signe par
défaut
    short int sign = -30000;
  // le type est un entier positif long 'positive long integer'
    unsigned long int l = 78000;
  // le type est un réel 'float' ou 'real' si vous préférer
    float real = 2108458745, r = 254.1317;


    clrscr(); // efface l'écran mais je pense que vous le savez déjà

  // regarder l'explication dans la leçon 2 variables
    printf ("Display:%d ,est un entier\n", variable);
    printf ("un réel: %E\nun autre réel: %f\n", real, r);

    ngetchx();

    clrscr();
    l = l - 32000;
    i++;
    j--;
    a += 2;
    bob = 2;
    count -= (bob *= 3) - a ;
    printf ("valeur de i:%d\n", i);
    printf ("valeur de j:%d\n", j);
    printf ("valeur de a:%d\n", a);
    printf ("valeur de bob:%d\n", bob);
    printf ("valeur de count:%d\n", count);

    key = ngetchx();

    printf ("la touche que vous avez pressée est: %d", key);
    printf ("valeur de i:%d\nvaleur de j:%d\nvaleur de a:%d\nvaleur de
bob:%d\nvaleur de count:%d\n", i, j, a, bob, count);

    ngetchx();
  }
```

Nous pensons que les commentaires du début de code sont suffisant et que vous pouvez le comprendre. Il est spécialement fait pour montrer les différents types de déclarations.

`printf ("Display:%d ,est un entier\n", variable)` affiche 'Display:4 ,est un entier'. En fait, à chaque fois que le compilateur voit %, il met un drapeau. Après il voit 'd' et

il sait qu'il doit mettre un entier décimal avec un signe comme type de variable (c'est la déclaration de 'd'). Finalement, il trouve le nombre dans les variables : 'variable' et met la valeur sur le drapeau '%'. Si vous voulez afficher '%', vous devez l'écrire 2 fois pour ne pas que le compilateur pense que c'est un drapeau, souvenez vous de la première leçon :-).

`printf ("un réel: %E\nun autre réel: %f\n", real, r)` affiche 'un réel: -4.740082.e+8' (sur la même ligne) et sur la ligne suivante, 'un autre réel: 254.1317'. C'est la même explication sauf que pour 'e' qui est un -réel qui a pour syntaxe [-]d.dddde[sign]ddd, c'est une forme exponentielle et f a pour syntaxe seulement [-]dddd.dddd .

% [{h|l}] type

Type {h|l}	**Explication**
h	Force short integer
l	Force long integer

Type	**Explication**
d, i	entier décimal avec signe
u	entier décimal sans signe
o	entier octadécimal (ce n'est pas une optionANSI, i.e. spécifique aux TI)
b	entier binaire (ce n'est pas une optionANSI, i.e. spécifique aux TI)
x	entier hexadécimal en minuscule
X	entier hexadécimal en majuscule
e	réel, forme [-]*d.dddd***e**[sign]*ddd* (forme exponentielle)
E	comme 'e' mais avec une majuscule pour la lettre pour l'exponentielle
f	réel, forme [-]*dddd.dddd*

g	réel: plus compact que les formes 'e' et 'f': c'est la forme la plus communément utiliser en dialogue réel
G	comme 'g' mais avec une majuscule pour les exposant
r	réel, forme ingénieur (ce n'est pas une optionANSI, i.e. spécifique aux TI)
R	comme 'r' mais avec un lettre comme exposant
y	point réel, mode spécifique (ce n'est pas une optionANSI, i.e. spécifique aux TI)
Y	comme 'y' mais avec une majuscule pour l'exposant
c	caractère
s	Chaîne
p	Pointeur; principalement le même que 'x' mais n'utilise pas les modifications de 'l'
%	non; le caractère '%' est afficher à la place

Il y a d'autres types et si vous êtes particulièrement intéressés par un type, nous vous invitons à regarder la documentation de Tigcc. N'oubliez pas que le travail de Zeljka Juric est génial :-).

l=l-32000 est votre première opération sur variable. Nous pouvons couper la syntaxe pour que se soit plus simple. La valeur de gauche est l et la valeur de droite est l-32000. Après le compilateur s'intéresse à la valeur de droite. Il prend la valeur de la variable: 'l' et soustrait 32000. Donc il fait l-32000=78000-32000=46000. Après, il attribue la valeur 46000 à la variable 'l'. C'est simple non?

i++ veut dire i=i+1. En fait, vous prenez la valeur de 'i' et vous lui ajouté 1 et vous changer la valeur initiale de la variable 'i'. Si vous voulez vous pouvez remplacer i++ par i=1+1. C'est la même chose mais i++ est un raccourci qui est très intéressant quand vous faites de longues pages de code.

j-- veut dire j=j-1, c'est un autre raccourci. Alors que a+=2 veut dire a=a+2, le C est vraiment bien pour ces raccourcis. Il y a aussi d'autres raccourcis: variable opération = valeur veut dire que la valeur variable est égale à elle même subissant l'opération demander par la valeur donnée Les différents opérateurs sont +,-,*,/,% et aussi d'autres que vous pourrez trouver dans la documentation de Tigcc. Le plus dure à comprendre est % qui donne le reste. 'exp1 % exp2' donne le reste de la division de exp1 par exp2. Nous verrons plus tard une utilisation de cette fonction car elle peut être très utile et peut donner des codes plus courts et plus efficaces.

bob=2 est une initialisation de variable. Vous pouvez initialiser vos variables après leur déclaration.

'count -= (bob*=3) – a' est une opération complexe. Nous allons contourner les problèmes. Quand vous voyez la syntaxe, vous notez qu'il y a 2 '=' mais il n'y a pas problème! C'est la magie du langage C. Vous devez comprendre comme le compilateur. Il voit la premier symbole de l'affectation qui est à gauche du premier = car il lit de gauche à droite et de haut en bas. Donc la valeur de gauche est 'count' et la valeur de droite est '(bob*=3)-a'. Ici le compilateur analyse (bob*=3) en premier, c'est comme en mathématique, l'affectation entre parenthèse est prioritaire. Alors 'bob' prend la valeur de droite multipliée par lui-même. Vous pouvez en déduire que 'bob*=3' veut dire bob*3=2*3=6. Alors vous obtenez 'count-=6-a'. Vous pouvez calculer la valeur de droite 6-a=6-2=4 et après rechercher 'count-=4' count=count - 4=17-4=13. C'est génial même si c'est difficile au début mais après avec le temps ce type de calcul vous ferons gagner des lignes de code. Mais si c'est trop dure pour vous, faites à votre propre manière, l'important c'est que ça marche!

Après nous affichons le résultat de quelques variables. Un équivalent d'une partie du code peut être `printf ("valeur de i:%d\nvaleur de j:%d\nvaleur de a:%d\nvaleur de bob:%d\nvaleur de count:%d\n", i, j, a, bob, count)`. Mais cette syntaxe n'est pas énormément plus simple que la syntaxe d'avant. Vous devez choisir selon votre préférence. N'oubliez pas que l'important en programmation est de s'amuser! :-).

La dernière difficulté est `key = ngetchx()`. Cette fonction est utile pour donner un nom à la touche à presser et nous allons utiliser le résultat plus tard dans le code. Le nom de la touche est un caractère ou un nombre:0 à 255. C'est un code ascii. Chaque touche à un code ascii différent et il y a des constantes qui évite de se rappeler ce code. Par exemple, la constante du code ascii de Echap est KEY_ESC et toutes les touches ont une constante et ces constantes sont appelées touches communes. Pour connaître le nom d'une constante allez voir la documentation de Tigcc.

Pour les variables booléennes, il y a un opérateur spécifique: '!'. En fait, une variable booléenne prend 2 valeurs: TRUE=1 et FALSE=0 et que vous mettez cet opérateur avant une variable booléenne, le compilateur remplace les valeurs actuelles les par les autres valeurs. Par exemple:

```
// Données sources C
// créé le 08/06/2002; 16:22:37


#define USE_TI89              // Utilisation pour Ti 89
#define USE_TI92PLUS          // Utilisation pour Ti 92+

#define OPTIMIZE_ROM_CALLS // utilisation de "ROM Call Optimization"

#define SAVE_SCREEN // Sauvegarde/Restaure le contenu de l'écran

#include <tigcclib.h> // Inclu toutes les têtes de lignes

// Fonction Principale

void _main(void)
{
  int b = TRUE; // c'est une variable booléenne
```

```
    b = !b; // la nouvelle valeur est FALSE
    b = !(!b); // la nouvelle valeur est toujours FALSE
}
```

Nous allons vous expliquer la dernière ligne, car nous pensons que vous pouvez comprendre les autres. Nous allons diviser le problème comme quand le compilateur travaille. En fait, le compilateur évalue l'expression entre parenthèse. Donc la valeur de cette expression est TRUE et après il effectue l'opération '!' alors la valeur finale de la variable recalculée est FALSE. C'est pourquoi après l'affectation, la nouvelle valeur de la variable est toujours FALSE.

Leçon 3 - Les conditions

Les conditions vous permettent de choisir une partie du code selon la valeur de quelques variables. C'est très intéressants, par exemple, pour un jeu d'envahisseur spatial, où vous devez effacer des ennemis quand le joueur les tue.

1) La syntaxe des conditions

La syntaxe est:

```
if (expression)
statement;
```

ou plus fréquemment,

```
if (expression) {
statement
}
```

En fait, le compilateur évalue l'expression entre parenthèses et seulement si c'est vrai, le compilateur dirige le code : statement (corps). L'expression évaluée peut faire une comparaison entre la valeur d'une variable et une autre valeur de variable ou un nombre. Cette comparaison est faite avec des symboles :

- '= =' le symbole égale à, vous devez être prudents parce que vous pouvez le confondre avec

'=' : le symbole de l'affectation.
- '>' le symbole supérieur à.
- '<' Signifie inférieur à.
- '<=' Signifie inférieur ou égal à.
- '>=' le symbole supérieur ou égal à.
- '!=' Signifie différent de.

Mais vous ne devez pas mettre un de ces symboles si l'expression est une booléenne, car il prend la valeur vraie ou fausse.
Il y a une autre condition : 'autrement' et voici sa syntaxe :

```
if (expression) {
statement1
} else {
statment2
}
```

En fait, le compilateur vérifie l'expression et si c'est vrai alors il exécute le code de statment1 sinon il exécute le code de statment2. Laissez-nous illustrer ce point. Cet exemple est stupide mais vous pourrez voir comment il travaille.


```
// Données sources C
// Créé le 07/06/2002; 23:49:06

#define USE_TI89              // Utilisation pour Ti 89
#define USE_TI92PLUS          // Utilisation pour Ti 92+

#define OPTIMIZE_ROM_CALLS    // Utilisation de "ROM Call Optimization"

#define SAVE_SCREEN           // Saugarde et restore le contenu de
l'écran

#include <tigcclib.h>         // Inclut toutes les tête de ligne

// Fonction principale
void _main(void)
{
  int a=25,b=35;
  int n=1; // C'est une variable boléenne
  clrscr();

  if (a==25) {
    printf("The condition1 is true\n"); // c'est affiché à l'écran
  }
  if (a==b) {
    printf("The condition2 is true\n"); // ce n'est pas affiché à l'écran
  }
  if (a<b) {
    printf("The condition3 is true\n"); // c'est affiché à l'écran
  }
  if (a>32) {
    printf("The condition4 is true\n"); // ce n'est pas affiché à l'écran
  } else {
    printf("The condition4 is false\n"); // c'est affiché à l'écran
```

```
    }
    if (b!=a) {
      printf("The condition5 is true\n"); // c'est affiché à l'écran
    }
    if (n) {
      printf("The condition6 is true\n"); // c'est affiché à l'écran
    }
    printf("the value of boolean variable is %d\n",n);
    n=!n; /* une variable boléenne prend 2 valeur: VRAI=1 et FAUX=0
             Cette assignation est autant valable pour ces valeurs que pour
d'autres valeurs */
    printf("the new value of boolean variable is %d\n",n);
    if (n) {
      printf("The condition7 is true\n"); // ce n'est pas affiché à l'écran
    }

    ngetchx();

  }
```

---

top

2) Les opérateurs bouléens ou logiques:'||', '&&' et '!'

Leurs significations sont '||' = ou (ou inclusif), '&&' = et, '!' = non. En fait, '&&' et '||' sont des opérateurs binaires et '!' est un opérateur unitaire, autrement dit, un opérateur unitaire s'applique à une expression. Ces opérateurs vous permettent de faire des conditions plus complexes au lieu d'évaluer plusieurs conditions avec des 'si'. Par exemple, nous pouvons voir comment utiliser l'opérateur : '&&' au lieu de deux conditions 'si'.

```
if (condition1) {
if (condition2) {
statement
}
}
```

C'est équivalent de:

```
if ((condition1) && (condition2)) {
statement
}
```

Ces deux codes sont équivalents mais le deuxième est plus clair et il s'écrit plus rapidement. En fait, le compilateur exécute le corps si et seulement si la condition1 est vrai et la condition2 est vrai sinon le compilateur saute jusqu'au signe '}'.

```
if ((condition1) || (condition2)) {
statement
}
```

L'opérateur : '||' signifie que l'une des condition ou les 2 doivent être vraies pour que tout soit vrai.
Même si je ne vous le dis pas, vous pouvez les utiliser 'autrement'. Après le résumé de ces opérateurs, nous verrons un exemple.

```
if (condtion1) operator (condition2) {
statement
```

```
}
```

&&	condition2	
condition1	VRAI	FAUX
VRAI	VRAI	FAUX
FAUX	FAUX	FAUX

||	condition2	
condition1	VRAI	FAUX
VRAI	VRAI	VRAI
FAUX	VRAI	FAUX

expression	VRAI	FAUX
!(expression)	FAUX	VRAI

Comme je le disait avant, FAUX est une constante, qui est égal à 0 et VRAI est une constante, qui n'est pas égal à 0, nous supposons d'habitude que sa valeur est 1.


```
// Données sources C
// Créé le 07/06/2002; 23:49:06

#define USE_TI89              // Utilisation pour Ti 89
#define USE_TI92PLUS          // Utilisation pour Ti 92+

#define OPTIMIZE_ROM_CALLS    // Utilisation de "ROM Call Optimization"

#define SAVE_SCREEN           // Saugarde et restore le contenu de
l'écran

#include <tigcclib.h>         // Inclut toutes les tête de ligne

// Fonction principale
void _main(void)
{

  int a=1, b=2;
  int t=TRUE, f=FALSE; // ce sont des variables boléennes
  clrscr();

  if ((a == 1) & (b <= 5)) {
    printf("the condition1 is true\n"); // c'est affiché à l'écran
```

```
    }
    if ((a >= 0) & (b < 2)) {
      printf("the condition2 is true\n"); // ce n'est pas affiché à l'écran
    } else {
      printf("the condition2 is false\n"); // c'est affiché à l'écran
    }
    if ((a > 17) | t) {
      printf("the condition3 is true\n"); // c'est affiché à l'écran
    }
    if ((a < 13) | t) {
      printf("the condition4 is true\n"); // c'est affiché à l'écran
    }
    if ((a < 13) ^ t) {
      printf("the condition5 is true\n"); // ce n'est pas affiché à l'écran
    }
    if (f ^ t) printf("the condition6 is true\n"); // c'est affiché à
l'écran

    ngetchx();

  }
```

---

top

3) La syntaxe des multi-conditions

Le but de la fonction de 'commutateur' est d'éviter d'écrire plusieurs lignes avec 'si' et cela rend le programme plus clair. La syntaxe est :

```
switch (variable)
{
case value1:
   statment1
   break;
case value2:
   statment2
   break;
case value3:
   statment3
case valueA:
   statment4
case valueB:
case valueC:
case valueD:
   statment5
   break;
default:
   statment6
}
```

c'est équivalent à:

```
if (variable==value1) {
statment1
}
if (variable==value2) {
statment2
}
if (variable==value3) {
statment3
}
if (variable==valueA) {
statment4
}
if ((variable==value3) || (variable==valueA) || (variable==valueB) || (variable==valueC)
|| (variable==valueD) {
statment5
}
if ((variable!=value1) && (variable!=value2) && (variable!=value3) &&
(variable!=valueA)
&& (variable!=valueB) && (variable!=valueC) && (variable!=valueD) {
statment6 // it is the default case
}
```

Je pense que si vous avez compris la fonction 'si', vous devriez comprendre le commutateur: 'switch'. En fait, vous utilisez une variable, qui peut prendre des valeurs très différentes et chaque valeur vous fait un cas. Vous notez que le cas est fini par 'break' et s'il n'y a pas ce mot clef, une partie de ce cas est égale à un autre cas. Le mot clef : 'default' est le cas si la valeur de variable n'est pas égale à aucun cas, c'est le cas par défaut.

top

4) Une autre condition: l'opérateur '?:'

Cette autre condition ressemble à une condition 'if-then-else' sans corps. Le but de cet opérateur est d'affecter une variable selon la condition. l'opérateur: '?' est un opérateur ternaire. La syntaxe est:

nom = condition ? valeur1 : valeur2;

'nom' est le nom de ta variable qui est affecté par la valeur qui sera valeur1 si la condition est vérifiée ou valeur2 si cette condition est fausse. Le type de votre variable et celle des valeurs doivent être les mêmes, c'est préférable pour que ça marche! :-) Simplifions ce blabla par un court exemple.


```
// Données sources C
```

```
  // Créé le 07/06/2002; 23:49:06

  #define USE_TI89              // Utilisation pour Ti 89
  #define USE_TI92PLUS          // Utilisation pour Ti 92+

  #define OPTIMIZE_ROM_CALLS    // Utilisation de "ROM Call Optimization"

  #define MIN_AMS 100           // Compilateur pour AMS 1.00 ou plus

  #define SAVE_SCREEN           // Saugarde et restore le contenu de
l'écran

  #include <tigcclib.h>         // Inclut toutes les tête de ligne

  // Fonction principale
  void _main(void)
  {
    int a = 13, b = 17;
    int min, max;

    clrscr();
    if (a < b)
      min = a;
    else
      min = b;
    printf ("min(%d;%d) = %d\n", a, b, min);

    // Cette voie est un peu plus courte
    if (a > b)
      printf ("max(%d;%d) = %d\n", a, b, a);
    else
      printf ("max(%d;%d) = %d\n", a, b, b);

    printf("\n");

    max = (a < b)? b : a;
    printf ("max(%d;%d) = %d\n", a, b, max);

    printf ("min(%d;%d) = %d\n", a, b, (a < b)? a : b);

    ngetchx();
  }
```

Leçon 4: Les boucles

Les boucles sont une importante de la programmation, elles sont très intéressantes. En effet, le but est de traiter un grand nombre de données et écrire le moins de code possible grâce aux boucles. Souvent l'écriture d'un code sans boucles est impossible. Il y a deux ou trois sortes de boucles. Je dis deux ou trois parce que deux boucles sont souvent très semblables. J'appellerai ces trois boucles : la boucle "pour", la boucle "tant que" et la boucle "jusqu'à ce que". La boucle "jusqu'à ce que" n'existe pas vraiment dans la langue C mais je vous l'expliquerai plus tard.

1) La boucle “for”

Je commence par cette boucle parce que c'est la première boucle que j'ai appris. En effet, avant de commencer avec beaucoup de données, c'est surtout intéressant pour, par exemple, répéter plusieurs fois le même texte à l'écran. Bien sûr, si vous voulez, vous pouvez faire plusieurs fois "copier/coller" mais votre code deviendra trop gros et illisible. Je commence par un court exemple et après j'expliquerai la syntaxe et les buts de cette boucle.


```
// Données sources C
// Créé le 01/11/2002; 21:58:08

#define USE_TI89              // Utilisation pour Ti 89
#define USE_TI92PLUS          // Utilisation pour Ti 92+


#define OPTIMIZE_ROM_CALLS    // Utilisation de "ROM Call Optimization"

#define SAVE_SCREEN           // Saugarde et restore le contenu de
l'écran

#include <tigcclib.h>         // Inclut toutes les tête de ligne


// Fonction principale

void _main(void)
{
  int i;

  clrscr();

  for (i = 1; i < 10; i++ ) {
    printf ("You have to type 9 times anywhich key to exit the
program\n");
    ngetchx();
  }
}
```

Si vous n'utiliseriez pas de boucle, vous devriez écrire ce code :


```
  // Données sources C
  // Créé le 01/11/2002; 21:58:08

  #define USE_TI89              // Utilisation pour Ti 89
  #define USE_TI92PLUS          // Utilisation pour Ti 92+


  #define OPTIMIZE_ROM_CALLS    // Utilisation de "ROM Call Optimization"

  #define SAVE_SCREEN           // Saugarde et restore le contenu de
l'écran

  #include <tigcclib.h>         // Inclut toutes les tête de ligne


  // Fonction principale
  void _main(void)
  {
    clrscr();

    printf ("You have to type 9 times anywhich key to exit the program\n");
    ngetchx();
    printf ("You have to type 9 times anywhich key to exit the program\n");
    ngetchx();
    printf ("You have to type 9 times anywhich key to exit the program\n");
    ngetchx();
    printf ("You have to type 9 times anywhich key to exit the program\n");
    ngetchx();
    printf ("You have to type 9 times anywhich key to exit the program\n");
    ngetchx();
    printf ("You have to type 9 times anywhich key to exit the program\n");
    ngetchx();
    printf ("You have to type 9 times anywhich key to exit the program\n");
    ngetchx();
    printf ("You have to type 9 times anywhich key to exit the program\n");
    ngetchx();
    printf ("You have to type 9 times anywhich key to exit the program\n");
    ngetchx();

  }
```

Vous pouvez voir que le second code a 5 lignes de plus. Cependant, si vous ne savez pas combien de fois vous voulez montrer le même texte, vous devez utiliser une boucle ou bien vous arrêtez de programmer. :-) la syntaxe est :

For (var = initial value; condition; step) {
Statement
}

En fait, premièrement vous affectez votre variable appelée var. Cette affectation est la

valeur initiale de votre variable. Ensuite, j'appellerai souvent cette variable "a" et "z" parce que j'utilise un clavier français et c'est plus facile. Est-ce qu'une bonne ou une mauvaise habitude ? Je ne sais pas; c'est ma façon de programmer. Deuxièmement, vous devez dire au compilateur quand il doit arrêter la boucle et passer au code suivant. C'est pourquoi la syntaxe ressemble à une condition : "var < valeur finie + 1". Nous pouvons traduire cela avec la phrase suivante : "le compilateur continue la boucle jusqu'à ce que la valeur de 'var' soit égale à la valeur finale". Troisièmement, vous devez donner le pas de votre boucle c'est-à-dire vous choisissez les valeurs que votre variable prendra. En fait, cela ressemble à une affectation. L'affectation: "var ++" est généralement utilisé mais vous pouvez aussi utiliser une autre affectation, par exemple, var + = 2 si vous voulez n'utiliser que les chiffres impairs. Ici je montre la partie la plus intéressante des boucles. En effet, rappelez-vous que var est une variable et la variable est très intéressante et peut être utilisée pendant la déclaration de votre boucle afin de traiter avec les données que vous voulez. Pour simplifier, nous pouvons voir un court exemple. Si vous voulez afficher les nombres de 1 à 10. Vous pouvez écrire :

```
clrscr();
printf ("1\n2\n3\n3\n4\n5\n6\n7\n8\n9\n10");
ngetchx();
```

Cependant, ce n'est pas le but de cette partie. C'est pourquoi nous allons utiliser pour cela une boucle simple et particulièrement pour comprendre le but et la puissance des boucles.

```
int a;
clrscr();
for (a = 1; a < 11 ; a++)
printf ("%u\n", a);
ngetchx();
```

Pour 10 nombres, ce n'est pas vraiment plus rapide, mais si vous voulez montrer 100 nombres, il est un obligatoire d'utiliser une boucle ou bien vous êtes têtus et stupides.; :) Vous pouvez voir le code :


```
// Données sources C
// Créé le 01/11/2002; 21:58:08

#define USE_TI89              // Utilisation pour Ti 89
#define USE_TI92PLUS          // Utilisation pour Ti 92+


#define OPTIMIZE_ROM_CALLS    // Utilisation de "ROM Call Optimization"

#define SAVE_SCREEN           // Saugarde et restore le contenu de
l'écran

#include <tigcclib.h>         // Inclut toutes les tête de ligne


// Fonction principale
void _main(void)
```

```
{
  clrscr();
  printf ("1\n2\n3\n3\n4\n5\n6\n7\n8\n9\n10");
  ngetchx();

  int a;
  clrscr();
  for (a = 1; a < 11 ; a++)
    printf ("%u\n", a);
  ngetchx();
}
```

D'autre part, vous devez être prudents parce que vous ne devez pas modifier la variable des boucles dans la boucle de déclaration. C'est très dangereux mais le compilateur ne vous l'interdit pas.

Généralement, quand vous traitez avec un certain nombre de données, vous utilisez des tableaux parce que c'est plus commode. Comme nous n'avons pas encore vu les tableaux , je vous montrerai plus tard un exemple traitant avec des tableaux et des boucles, c'est trop fort!

top

2) La boucle “while”

Je continue avec cette boucle parce que la langue C préfère cette boucle "tant que" et elle est utile pour faire une boucle "jusqu'à ce que". Cet exemple est la même chose que l'exemple précédent qui utilisait la boucle "pour"


```
// Données sources C
// Créé le 01/11/2002; 21:58:08

#define USE_TI89              // Utilisation pour Ti 89
#define USE_TI92PLUS          // Utilisation pour Ti 92+


#define OPTIMIZE_ROM_CALLS    // Utilisation de "ROM Call Optimization"

#define SAVE_SCREEN           // Saugarde et restore le contenu de
l'écran

#include <tigcclib.h>         // Inclut toutes les tête de ligne


// Fonction principale
void _main(void)
{
  int a = 1;

  clrscr();
```

```
  while (a < 11) {
    printf ("%u\n", a);
    a++;
  }
  ngetchx();
}
```

Comme vous pouvez le voir, vous pouvez utiliser une boucle "pour" ou une boucle "tant que" mais vous devez choisir la boucle qui est la plus simple pour votre algorithme. En fait, une est plus lisible mais ce n'est pas vrai pour tout les algorithmes, vous devrez choisir votre manière de programmer. Maintenant, nous pouvons voir la syntaxe :

While (condition) {
statement
}

D'autre part, vous devez être prudents. Vous ne devez pas oublier la première affecattion "a = 1" ou bien la boucle fait n'importe quoi et surtout elle continue et ne s'arrête jamais!!! Cette boucle est plus dangereuse parce que vous pouvez facilement oublier cette affectation. En effet, cette affectation n'appartient pas à la syntaxe de la boucle à la différence de la boucle "pour" . Néanmoins, vous ne devez pas négliger cette boucle à cause de ce petit problème et vous pouvez aussi faire des erreurs avec la boucle "pour". Rappelez-vous que vous pouvez faire plusieurs essais avant la découverte du bon algorithme. Si la boucle ne s'arrête jamais, vous arrêterez facilement l'émulateur et vous recherchez l'erreur, c'est simple, n'est-ce pas ? C'est pourquoi je vous conseille d'utiliser un émulateur plutôt que votre calculette pour faire votre essai; arrêter un émulateur est plus facile que d'enlever les piles quand votre programme fait des siennes. De plus, vous pouvez utiliser la fonction de l'émulateur : "retournez à l'état sauvé" ce qui signifie en anglais: “revert to saved state"; vous ne devrez pas relkancer de nouveau l'émulateur!

Les applications de cette boucle sont les mêmes que n'importe quelles autres boucles. Les grandes différences entre la boucle "pour" et la boucle "tant que" est que vous contrôlez le compteur plus facilement, mais toutes les boucles peuvent faire le même travail vous devrez penser au plus simple, et c'est tout!

Afin d'améliorer votre programme, vous devriez jouer avec ces deux boucles. C'est la meilleure façon de s'améliorer efficacement.

top

3) La boucle “until”

Dans le langage C, il n'y a pas le mot-clé : "until" ce qui signifie "jusqu'à ce que", c'est la raison pourquoi je vous ai dit qu'il n'y a pas vraiment cette boucle dans le C. J'appelle cette boucle 'jusqu'à' mais c'est une erreur parce que la signification de "jusqu'à" est fausse. En fait, seul l'emplacement du test de la condition est différent de la boucle "tant que". Ces deux boucles sont assez voisinesce qui veut dire que vous pouvez changer très facilement de boucle: "tant que" ou "jusqu'a ce que". Un exemple est toujours bienvenu.


```
// Données sources C
// Créé le 01/11/2002; 21:58:08

#define USE_TI89              // Utilisation pour Ti 89
#define USE_TI92PLUS          // Utilisation pour Ti 92+


#define OPTIMIZE_ROM_CALLS    // Utilisation de "ROM Call Optimization"

#define SAVE_SCREEN           // Saugarde et restore le contenu de
l'écran

#include <tigcclib.h>         // Inclut toutes les tête de ligne


// Fonction principale
void _main(void)
{
  int a = 1;

  clrscr();

  do {
    printf ("%u\n", a);
    a++;
  } while (a < 11);

  ngetchx();
}
```

Vous pouvez noter que le test de vérification de fin de boucle est à la fin et pas au début comme pour la boucle "tant que". Dans les autres langages, comme par exemple "le Pascal", quand l'essai est à la fin, les boucles sont appelées boucle "jusqu'à ce que" et le mot-clé : "jusqu'à" est écrit. La syntaxe de cette boucle est :

do {
Statement
} while (condition)

top

4) Quelques mots-clefs utilisés dans les boucles: “break” et “continue”

Il y a deux mots clefs qui appartiennent aux boucles. Ils sont “break” et “continue”. Leur syntaxe est très simple, c'est juste le mot clef qui est suivi du point-virgule: ';' mais comme d'habitude ces mots-clés doivent être dans le corps de la boucle.

“break” est capable d'arrêter la boucle même si la condition d'arrêt de la boucle est fausse et vous pouvez le mettre n'importe où dans le corps de la boucle. Cela dépend du résultat que vous voulez.

“continue” a un effet inversée en comparant avec “break. En effet, “continue” est capable d'arrêter la déclaration de la boucle, la condition de boucle est vérifiée et la boucle continue ou pas suivant si la condition est validée ou non. C'est simple, n'est-ce pas ?;-)

Ces deux méthodes sont très puissantes pour vous simplifier la vie, mais elles sont très dangereuses. Parce que vous pouvez facilement faire une boucle qui ne s'arrête jamais ou s'arrête toujours et le compilateur passe au code suivant.

Laissez-nous vous faire voir un cour exemple, il augmentera votre compréhension.


```
1:       // Données sources C
2:      // Créé le 23/12/2002; 19:38:02
3:
4:      #define USE_TI89              // Utilisation pour Ti 89
5:      #define USE_TI92PLUS          // Utilisation pour Ti 92+
6:
7:      #define OPTIMIZE_ROM_CALLS    // Utilisation de "ROM Call
Optimization"
8:
9:      #define SAVE_SCREEN           // Saugarde et restore le contenu de
l'écran
10:
11:     #include <tigcclib.h>         // Inclut toutes les tête de ligne
12:
13:     // Fonction principale
14:     void _main(void)
15:     {
16:       int go_on = 1; // Cette variable est une boléenne: VRAI=1 &
FAUX=0
17:       int a = 0;
18:
19:       clrscr();
20:
21:       do {
22:         a++;
23:         if (a == 13) {
24:           continue;
25:           break;
26:         }
27:         if (a == 17) {
28:           break;
29:           continue;
30:         }
31:         printf ("%d\n", a);
32:       } while (go_on);
33:
34:       ngetchx();
35:     }
```

Quand vous commencez ce programme, vous pouvez voir que les nombres : '13' et '17' ne sont pas affichés. Je pense que vous pouvez comprendre pourquoi quand vous lisez le code. Vous pouvez remarquer que les lignes : 25 et 29 ne sont pas lu par le compilateur, elles sont inutiles. Vous pouvez aussi remarquer que s'ils n'utilisent pas ces deux nouveaux mots-clés, la boucle ne s'arrête jamais parce que la variable : "go_on" est toujours vrai et ne change pas dans la boucle.

Pour terminer, les boucles sont très puissantes et vous pouvez traiter avec une grande quantité de données sans écrire beaucoup de lignes de code. Comme d'habitude, vous devez jouer autour de ces nouvelles notions et celles que vous avez déjà vues. C'est la meilleure méthode pour améliorer et comprendre les astuces de programmation. Jusqu'à présent nous avons vu les notions principales de programmation qui sont les boucles.

Leçon 5 – Tableaux, pointeurs et chaînes de caractères

Cette partie du langage C est très importante parce que vous ne pouvez pas faire un programme sans utiliser un tableau. En effet les tableaux sont capables d'utiliser un grand nombre de données. Votre programme est plus lisible et aussi plus possible. Les chaînes sont plus spécifiques et vous pouvez utiliser une pseudo-constante pour afficher le texte au lieu d'utiliser une variable de type chaîne. De plus les chaînes sont utiles quand vous voulez faire un jeu. Par exemple, quand vous voulez montrer un score, le nombre de vies, … vous devez utiliser une variable de chaîne. La fonction de "DrawStr" utilise seulement une entrée de caractère et vous ne pouvez pas montrer directement un nombre à la différence de fonction de "printf". D'autre part, les pointeurs sont spécifiques au langage C. Les pointeurs sont très puissants et sont capables de diminuer la mémoire et d'augmenter la performance de votre programme. Je traite avec ces trois notions parce qu'elles sont équivalentes; seule l'écriture change.

1) Tableaux

Un tableau rassemlble plusieurs variables de même type. J'expliquerai comment un tableau travaille plus tard avec les pointeurs. Les tableaux sont très utiles pour traiter une grande quantité de données parce que chaque ligne de votre tableau est une variable et chaque

variable a un nombre au lieu d'un nom. Vous pouvez utiliser une autre variable afin d'utiliser les variables de la ligne de votre tableau. Ainsi, vous pouvez facilement utiliser le tableau dans une boucle et traiter avec une grande quantité de données. Maintenant nous pouvons voir comment nous pouvons utiliser les tableaux.

Le tableau marche comme une variable, vous devez déclarer et affecter un tableau pour être capable de l'utiliser. La syntaxe est :

Déclaration :

Type name [the number of lines]

Affectattion :

Name [the number of the line] = value

En fait, dans le langage C, les tableaux commencent toujours par 0. Bien sûr le dernier rang du tableau est (le numéro de lignes - 1). Maintenant nous allons illustrer cette nouvelle notion par un bref exemple:


```
// Données sources C
// Créé le 07/06/2002; 23:49:06

#define USE_TI89              // Utilisation pour Ti 89
#define USE_TI92PLUS          // Utilisation pour Ti 92+

#define OPTIMIZE_ROM_CALLS    // Utilisation de "ROM Call Optimization"

#define SAVE_SCREEN           // Saugarde et restore le contenu de
l'écran

#include <tigcclib.h>         // Inclut toutes les tête de ligne

// Fonction principale

void _main(void)
{
  int array1[4];
  short array2[17];
  short a = 22;

  array1[0] = 21;
  array1[1] = a;
  array1[2] = array1[0] + 2;
  a = 3;
  array1[a] = 24;


  clrscr();

  for (a = 0; a < 4; a++)
    printf ("%d ", array1[a]);

  for (a = 0; a < 17; a++)
    array2[a] = a;

  ngetchx();
}
```

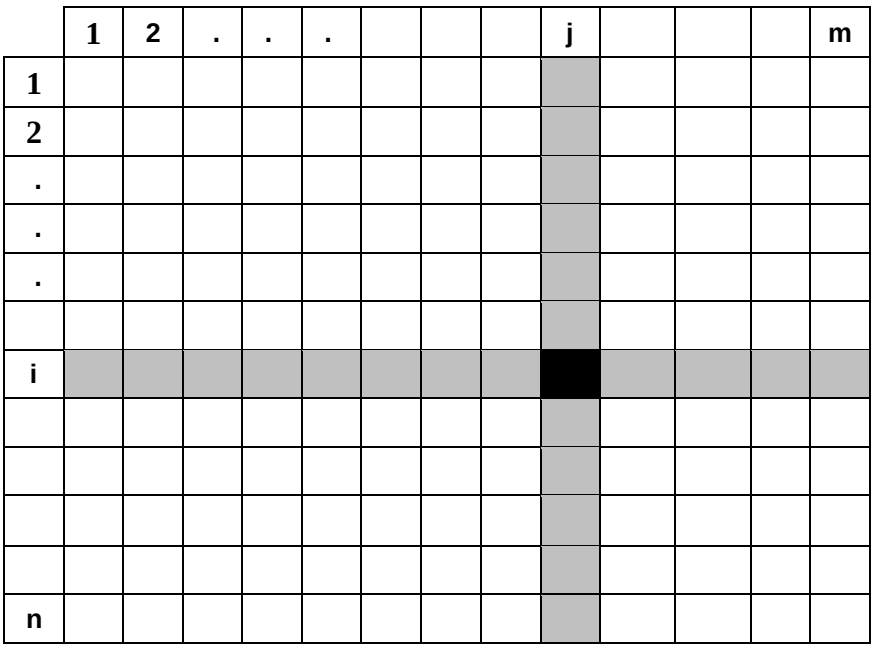
Il y a un autre type de tableaux : la matrice; en fait vous avez des rangées et des colonnes. La syntaxe est :
Déclaration :

Type name[the number of rows] [the number of colons]

affectation :

Name[the number of the row] [the number of the colon] = value

La matrice réunit plusieurs tableaux. J'ai fait un petit graphisme pour mieux visualiser, il est plus facile d'imaginer et comprendre le travail d'une matrice.



Type de matrice [**n**][**m**]

matrice[**i**][**j**] = case noire

Comme je l'ai dis précédemment, les tableaux sont particulièrement utiles pour traiter une grande quantité de données. Laissez-nous vous faire voir un exemple de trie de tableau!


```
// Données sources C
// Créé le 07/06/2002; 23:49:06

#define USE_TI89              // Utilisation pour Ti 89
#define USE_TI92PLUS          // Utilisation pour Ti 92+

#define OPTIMIZE_ROM_CALLS    // Utilisation de "ROM Call Optimization"

#define SAVE_SCREEN           // Saugarde et restore le contenu de
l'écran

#include <tigcclib.h>         // Inclut toutes les tête de ligne

// Fonction principale
void _main(void)
{
  short array[13];
```

```
    short a,z;
    short aux;

    randomize ();
    clrscr();

    printf ("An random array:\n");

    for (a = 0; a < 13; a++) {
      array[a] = random (10); // 0 <= short <= 9
      printf ("%d ", array[a]);
    }

    for (a = 0; a < 12; a++) {
      for (z = a + 1; z < 13; z++) {
        if (array[a] > array[z]) {
          aux = array[z];
          array[z] = array[a];
          array[a] = aux;
        }
      }
    }

    printf("\n\nThe sorted array:\n");
    for (a = 0; a < 13; a++)
      printf ("%d ", array[a]);

    ngetchx();
}
```

Ne vous inquiétez pas avec les nouvelles fonctions : randomize () et random (). Je vous les expliquerai plus tard dans le détail. Pour le moment, vous devez seulement savoir que "random (nombre)" est une fonction qui vous donne un numéro entre 0 et le nombre; plus exactement : 0 < = aléatoire (nombre) < = le nombre-1.

D'autre part, les deux boucles qui sont capables de trier le tableau sont plus complexes à comprendre. Il y a plusieurs façons de trier un tableau mais cette voie est pour moi la mieux pour la vitesse et la taille de mémoire. Afin de comprendre le fonctionnement de tri, vous pouvez regarder les diagrammes suivants:

7	**2**	**1**	**5**	**3**
2	**7**	**1**	**5**	**3**
1	**7**	**2**	**5**	**3**

1	**2**	**7**	**5**	**3**

1	**2**	**3**	**5**	**7**

top

2) Pointeurs

C'est une spécificité du langage C. Les pointeurs sont très puissants et maintenant je vais seulement vous expliquer la base. Dans les leçons suivantes, je vous expliquerai des utilisations plus intéressantes : procédures, fonctions, fichiers, …

Un pointeur est une variable qui a pour but principal de se rappeler de l'adresse d'un secteur. À la différence d'une variable qui se rappelle d'une valeur. Cette adresse est une valeur hexadécimale qui indique un secteur de mémoire. En fait, vous pouvez obtenir la valeur de ce secteur et après vous pouvez utiliser un pointeur comme une variable. De plus, vous pouvez aussi obtenir l'adresse d'une variable, même si la fonction principale d'une variable est sa valeur et non son adresse. La déclaration et l'affectation d'un pointeur ne sont pas plus durs qu'avec une variable. Vous devez seulement vous souvenir que vous travaillez sur l'adresse. Voici un exemple.


```
// Données sources C
// Créé le 07/06/2002; 23:49:06

#define USE_TI89              // Utilisation pour Ti 89
#define USE_TI92PLUS          // Utilisation pour Ti 92+

#define OPTIMIZE_ROM_CALLS    // Utilisation de "ROM Call Optimization"

#define SAVE_SCREEN           // Saugarde et restore le contenu de
l'écran

#include <tigcclib.h>         // Inclut toutes les tête de ligne

// Fonction principale
void _main(void)
{
  int *pointer1;
  int *pointer2;
  int *pointer3;

  int a = 13;
  int z;

  pointer1 = &a;
  pointer2 = pointer1;
  *pointer2 += 4;
  pointer3 = &z;
  z = 17;

  clrscr();
  printf ("pointer1: %d\npointer2: %d\npointer3: %d\n", *pointer1,
*pointer2, *pointer3);
  ngetchx();
}
```

Pour déclarer un pointeur, vous devez définir le type de votre pointeur et son nom comme pour une variable mais vous ajoutez le signe '*' afin d'informer le compilateur. Ce n'est pas plus dur!;-) Après que vous ne devez pas initialise le pointeur avec une valeur, mais avec une adresse de mémoire. Cette adresse est une variable ou un autre pointeur. Quand vous n'utilisez pas le signe '*', cela signifie que vous travaillez sur l'adresse d'un pointeur ou sinon vous travaillez sur la valeur de l'adresse de votre pointeur. Rappelez-vous qu'avec des pointeurs vous travaillez sur l'adresse et vous pouvez obtenir la valeur du secteur demandé.

top

3) Chaînes de caractères

Nous allons voir que plusieurs fonctions travaillent avec les chaînes. En effet, les chaînes sont un type spécifique et vous ne pouvez pas utiliser le symbole habituel de l'affectation , il crée des erreurs qui sont dures à comprendre et il est très facile de faire des erreurs!

La déclaration de votre chaîne dépend de l'utilisation de votre chaîne : variable ou constante. Si vous voulez utiliser seulement une chaîne constante, vous pouvez déclarer sans donner la taille de votre chaîne. Si vous voulez utiliser une chaîne comme une variable, vous devez dire au compilateur la taille de votre chaîne et plus particulièrement vous ne devez pas utiliser le symole de l'affectation: '='. La syntaxe est :

```
char name[] = “data”;
char name[value];
```

avec :

'Name' : le nom de votre chaîne
'Data' : votre chaîne de caractères
'Value' : la taille de votre chaîne

Après avoir initialiser la variable de chaîne, vous devez utiliser une fonction spécifique et pas le symbole de l'affectation! Cette fonction est 'strcpy' (pour copier la chaîne) et voici la syntaxe :

```
strcpy (string1, string2);
```

avec :

'String1' : le nom de la chaîne que vous voulez initialiser
'String2' : le nom de la chaîne que vous affectez à la chaîne : 'string1' ou vous pouvez remplacer 'string2' par "data" (une chaîne constante de caractères)

Si vous voulez additionner deux chaînes, autrement dit, ajouter deux chaînes, vous devriez utiliser la fonction de 'strcat' dont la syntaxe est :

```
strcat (string1, string2);
```

En fait, 'string2' peut être une chaîne constante de caractères. Le résultat est équivalent : ' string1 = string1 + string2 ' mais vous ne pouvez pas utiliser ces symboles spécifiques aux autres types de variables!

Leçon 6 – Procédures et fonctions

Les procédures et les fonctions sont très utiles et intéressantes. Elles sont capables de diminuer la taille de votre code. En fait, chaque fois que vous voulez faire un copier / coller, vous devez vous demander si vous pouvez utiliser une procédure. La grande différence entre une procédure et une fonction est : une procédure exécute seulement une partie de votre code et une fonction exécute une partie de votre code et renvoie une valeur quand la fonction est finie. Maintenant après cette courte introduction, je vais vous expliquer comment elles travaillent et quand vous devrez les utiliser afin d'optimiser votre code.

1) Procédure

En fait, une procédure est capable de séparer une partie de votre code et après vous pourrez exécuter cette partie de code très facilement. En les utilisant vous éviterez les copier / coller et donc allégerez votre code de manière significative. En plus votre code sera plus lisible et donc plus facile à comprendre. Comme je vous l'ai dit dans l'introduction, chaque fois que vous voulez faire un copier / coller, vous devez vous demander si c'est vraiment utile et si une procédure ne serait pas plus commode et efficace.

La syntaxe d'une procédure simple est:

```
void name ( ) {
statement
}
```

'name' est bien sûr le nom de votre procédure. Le mot 'void' signifie que c'est une procédure et pas une fonction ou plutôt la fonction ne renvoie rien : vide. Cette fonction est une procédure; c'est sa définition :-). Le corps ("statement") est la partie de votre code que vous voulez séparer. Et la syntaxe permettant d'appeler une procédure est :

```
name ( );
```

Si vous êtes un bon observateur, vous pouvez voir que 'void _main (void)' ressemble à une procédure. En effet, 'void _main (void)' est une procédure. C'est la procédure principale est sera donc appelée en première par le compilateur. Vous ne pourrez pas supprimer cette procédure ou bien le compilateur ne saura plus par où commencer.

Maintenant nous pouvons voir où placer la procédure dans votre code. Vous avez deux choix. Le plus simple est de l'écrire avant le 'void _main (void)' et de ne pas la déclarer. La deuxième possibilité est d'écrire votre procédure après le 'void _main (void)' mais vous devrez la déclarer avant le 'void _main (void)'. La déclaration est assez simple : '**void name( );**'. En fait, le compilateur lit le code comme un homme qui lit une langue européenne; le compilateur lit de gauche à droite et de haut en bas. C'est la raison pour laquelle le compilateur a déjà vu votre procédure si vous l'avez écrit avant le 'void _main (void)' et c'est pour cette raison que ce n'est pas la peine de la déclarer.

Pour illustrer ce point, j'ai écrit trois codes pour voir les différentes manières d'écrire une procédure dans un programme. Je vous expliquerai la déclaration de variables dans une procédure dans le paragraphe suivant.

Exemple1 sans procédure :


```
// C Source File
// Created 26/02/2003; 17:12:58

#define USE_TI89                // Compile for TI-89
#define USE_TI92PLUS            // Compile for TI-92 Plus
#define USE_V200                // Compile for V200

#define OPTIMIZE_ROM_CALLS      // Use ROM Call Optimization

#define MIN_AMS 100             // Compile for AMS 1.00 or higher

#define SAVE_SCREEN             // Save/Restore LCD Contents

#include <tigcclib.h>           // Include All Header Files

// Main Function
void _main(void)
{
  int a;

  clrscr();

  for (a = 0; a < 10; a++) {
    printf("*");
  }
  printf("\n");
  for (a = 0; a < 10; a++) {
    printf("*");
  }

  ngetchx();
}
```

Exemple2 avec une procédure mais sans déclaration :


```
// C Source File
// Created 26/02/2003; 17:14:28

#define USE_TI89                // Compile for TI-89
#define USE_TI92PLUS            // Compile for TI-92 Plus
#define USE_V200                // Compile for V200

#define OPTIMIZE_ROM_CALLS      // Use ROM Call Optimization
```

```
#define MIN_AMS 100           // Compile for AMS 1.00 or higher

#define SAVE_SCREEN           // Save/Restore LCD Contents

#include <tigcclib.h>         // Include All Header Files

void star() {
  int a;

  for (a = 0; a < 10; a++) {
    printf("*");
  }
}

// Main Function
void _main(void)
{
  clrscr();

  star();
  printf("\n");
  star();

  ngetchx();
}
```

Exemple3 avec une procédure déclarée :


```
// C Source File
// Created 26/02/2003; 17:17:31

#define USE_TI89              // Compile for TI-89
#define USE_TI92PLUS          // Compile for TI-92 Plus
#define USE_V200              // Compile for V200

#define OPTIMIZE_ROM_CALLS    // Use ROM Call Optimization

#define MIN_AMS 100           // Compile for AMS 1.00 or higher

#define SAVE_SCREEN           // Save/Restore LCD Contents

#include <tigcclib.h>         // Include All Header Files

// declaration of your procedure
void star();

// Main Function
void _main(void)
{
  clrscr();

  star();
  printf("\n");
```

```
  star();

  ngetchx();
}

void star() {
  int a;

  for (a = 0; a < 10; a++) {
    printf("*");
  }
}
```

Bien sûr, vous pouvez aussi déclarer votre procédure dans le deuxième exemple mais ce n'est pas une obligation.

Nous avons vu que nous appelons une procédure dans la procédure principale mais vous pouvez aussi appeler une procédure dans une autre procédure. Quand vous utilisez la première façon d'utiliser une procédure, le compilateur doit lire la procédure avant l'appel de celle-ci ou bien il ne la reconnaît pas et il plante! C'est pourquoi la deuxième voie, avec déclaration de procédure, est plus sûre et vous n'êtes pas obligé de faire attention à l'ordre de vos procédures. Je vous conseille cette utilisation car elle est plus rapide et plus simple à comprendre. Plus tard nous verrons comment créer un fichier pour placer toutes les déclarations de procédures afin de rendre le code plus claire! :-)

Quand vous appelez une procédure, le Tios prend beaucoup de temps. Quelques fois, la procédure est très courte et vous ne voulez pas perdre de temps à son activation mais ne voulez pas non plus faire de copier / coller afin de garder un code lisible. Vous devez donc faire une procédure inline. En fait, la procédure inline est exactement comme un copier / coller mais ce n'est pas vous qui travaillez, le compilateur le fait à votre place. Donc vous ne gardez que les avantages sans avoir les inconvénients. C'est de la balle!

La syntaxe des procédures inlines est aussi simple qu'une procédure simple : vous ajoutez seulement le mot ‘inline’dans la première position :

```
inline void name ( ){
statement
}
```

Quand vous déclarez une procédure inline, vous devez aussi ajouter le mot ‘inline’ comme : ‘inline void name ( );’ c'est super facile! :-) Par exemple, le code suivant vous donne à l'écran le même résultat qu'avec les autres exemples.


```
// C Source File
// Created 01/03/2003; 13:38:50

#define USE_TI89              // Compile for TI-89
#define USE_TI92PLUS          // Compile for TI-92 Plus
#define USE_V200              // Compile for V200

#define OPTIMIZE_ROM_CALLS    // Use ROM Call Optimization

#define MIN_AMS 100           // Compile for AMS 1.00 or higher
```

```
#define SAVE_SCREEN             // Save/Restore LCD Contents

#include <tigcclib.h>          // Include All Header Files

inline void star() {
  int a;

  for (a = 0; a < 10; a++) {
    printf("*");
  }
}

// Main Function
void _main(void)
{
  clrscr();

  star();
  printf("\n");
  star();

  ngetchx();
}
```

Bien sûr, si vous voulez quitter votre procédure avant la fin normale, vous devez ajouter : ‘return;’ et vous quitterez la procédure à ce moment là. Il est intéressant quand vous avez des conditions et que selon les événements vous voulez quitter ou non la procédure.

top

2) Les différentes sortes de variables et de constantes

Comme vous pouvez le voir sur les exemples précédents, il y a aussi des déclarations de variables dans les procédures; C'est comme pour la procédure _main. En fait, si la déclaration de votre variable est dans une procédure, cette variable est appelée la variable 'locale' et est disponible uniquement dans cette procédure. Nous pouvons dire : la variable est née avec ‘{‘ et est morte avec ‘}’. Vous disposez de plusieurs méthodes pour déclarer vos variables dans chacune de vos corps de procédures, par exemple, une pour que la déclaration d'une variable ne soit valable que dans un seul corps. Bien sûr, c'est pareil que pour la procédure principale. Ces variables sont appelées des variables 'automatiques' et ce sont les variables par défaut.

Si vous voulez obtenir que les données de votre variable ne soit pas effacées après le symbole ' }', vous devez ajouter ‘static’ dans la déclaration de votre variable ainsi vous n'aurez pas de variable automatique. La syntaxe est :

static type name

Si vous avez déjà compris ce que j'ai expliqué, vous ne pouvez utiliser une variable locale que dans une même procédure ou corps. Cependant, si vous voulez utiliser une variable dans toutes les procédures, vous devez utiliser un autre type de variables. Ce type est appelé 'global'. Une variable globale est utilisable n'importe où donc vous pouvez utiliser cette variable dans toutes les procédures et tous les fichiers mais nous n'avons pas vu ce point encore. La déclaration d'une variable globale est aussi simple qu'une variable locale; seulement la place de la déclaration est différente. Vous devez déclarer une variable globale

juste après le début de votre code : après la section 'include'. Je vous conseille d'utiliser la variable globale le moins possible pour organiser votre code le plus possible.

Je vais vous expliquer quelque chose mais ce n'est pas un conseil et vous devrez éviter de le faire. Comme nous l'avons déjà vu, il y a deux types de variables : locale et globale. En fait, la variable locale est plus importante que la variable globale. Si vous avez ces 2 types de variables et que vous leur donnez à chacune le même nom cela ne posera aucun problème au compilateur car il voit parfaitement la différence entre chacune des variables mais pour vous cela sera totalement incompréhensible et là je vous souhaite bonne chance pour reprendre plus tard votre programme :-(

Il y a deux sortes de constantes : les constantes qui ressemblent à un copier / coller et le constantes qui ressemblent à une variable. La première sorte de constantes est plus facile à utiliser selon moi. La syntaxe est :

#define name value

Souvenez-vous les espaces dans la langage C n'ont aucune signification mais c'est mieux visuellement, non? La place de cette déclaration est en haut de votre code, après la section 'include'. En fait, elle ressemble à la procédure 'inline' : à chaque fois que le compilateur voit le nom de votre constante dans votre code, il remplace le nom de votre constante par sa valeur. Je préfère cette sorte de constante, je pense que c'est plus clair et plus simple. La syntaxe de la deuxième sorte de constantes est :

const type name = value;

La différence avec l'autre constante est que le compilateur alloue une place de mémoire pour cette valeur. En fait, cette constante ressemble à une variable dont vous ne pouvez pas changer la valeur.

Nous allons voir un exemple afin de comprendre comment toutes ces notions marchent.


```
// C Source File
// Created 09/03/2003; 12:09:03

#define USE_TI89              // Compile for TI-89
#define USE_TI92PLUS          // Compile for TI-92 Plus
#define USE_V200              // Compile for V200

#define OPTIMIZE_ROM_CALLS    // Use ROM Call Optimization

#define MIN_AMS 100           // Compile for AMS 1.00 or higher

#define SAVE_SCREEN           // Save/Restore LCD Contents

#include <tigcclib.h>         // Include All Header Files

#define const1          56

int global = 17;

void procedure () {
  int local_procedure = 5;
  int local = 21;

  printf ("local_proc = %d\n", local_procedure);
  printf ("global = %d\n", global);
  printf ("local = %d\n", local);
```

```
    // the compiler doesn't know the 'local_main' variable
    printf ("const1 = %d\n", const1); // this constant is like a global
constant
    // the compiler doesn't know the 'const2' constant
  }

  // Main Function
  void _main(void)
  {
    int local_main = 13;
    int local = 27;
    const int const2 = 54;

    clrscr();

    printf ("Within the 'main' procedure:\n\n");
    printf ("local_main = %d\n", local_main);
    printf ("global = %d\n", global);
    printf ("local = %d\n", local);
    // the compiler doesn't know the 'local_procedure' variable
    printf ("const1 = %d\n", const1); // this constant is like a global
constant
    printf ("const2 = %d\n", const2); // this constant is like a local
constant

    ngetchx();

    clrscr();

    printf ("Within the procedure:\n\n");
    procedure ();

    ngetchx();
  }
```

---

top

3) Paramètres

Les paramètres de procédures sont très utiles et augmentent l'utilisation de procédures. En fait, vous pouvez créer vos propre fonction , comme par exemple pour faire votre fonction 'printf' mais le véritable intérêt est que vous pouvez créer des procédures spécifiques à votre programme. Vous pourrez ensuite utiliser ces procédures dans d'autres programme si ces procédures vous sont utiles. En effet, vous pouvez faire votre propre librairie mais nous verrons cela plus tard! Une procédure parfaite doit être utilisable dans n'importe quel code. Un simple copier / coller doit vous permettre une insertion dans n'importe quel code.

Comme je l'ai dis avant, quand nous utilisons la notion de paramètres, nous devons parler des pointeurs. En effet, il y a deux sortes de paramètres: par référence ou adresse et par valeur. Nous allons commencer par les paramètres par valeur parce qu'ils sont les plus simples à comprendre.

a) Paramètres par valeur

La syntaxe est:

void name (type name1, type name2, …) {
statement
}

et quand nous appelons la procédure ::

name (X1, X2, …);

avec Xi = value, variable, pointeur (avec le symbole étoile '*')

Comme vous pouvez le voir, vous devez déclarer les paramètres comme une variable simple : vous dites au compilateur le type de vos paramètres. Le nom de ces paramètres est donné par leur valeur parce que vous ne donnez à vos paramètres que la valeur des variables ou juste une valeur. En fait, à la fin de votre procédure les paramètres n'ont pas de valeur. Si vous changez la valeur de vos paramètres dans la procédure, les paramètres n'affectent aucune variable. Voici un exemple bref, pour permettre une meilleure compréhension :


```
// C Source File
// Created 12/03/2003; 17:07:41

#define USE_TI89               // Compile for TI-89
#define USE_TI92PLUS           // Compile for TI-92 Plus
#define USE_V200               // Compile for V200

#define OPTIMIZE_ROM_CALLS     // Use ROM Call Optimization

#define MIN_AMS 100            // Compile for AMS 1.00 or higher

#define SAVE_SCREEN            // Save/Restore LCD Contents

#include <tigcclib.h>          // Include All Header Files

#define height            5 // it's a constant to change easily the
height of your object

void star (int number) {
  int a;

  for (a = 0; a < number; a++)
    printf("*");
  printf("\n");
}

void space (int number) {
  int a;

  for (a = 0; a < number; a++)
    printf(" ");
}

// Main Function
void _main(void)
{
```

```
    int i;

    clrscr();

    for (i = 0; i < height; i++) {
      space (height - i);
      star (i * 2);
    }
    for (i = height; i > 0; i--) {
      space (height - i);
      star (i * 2);
    }

    ngetchx();
}
```

Vous pouvez remarquer que si vous n'utilisez pas de procédures, votre code serait plus grand parce que vous devriez faire deux copier / coller. Je pense que vous pouvez comprendre cet exemple sinon aller relire la leçon sur les boucles. :-)

b) Paramètres par adresse ou par référence

La différence est quand vous quittez votre procédure, les paramètres par adresse obtiennent la dernière valeur de la procédure. En fait, la valeur de votre procédure après le lancement de cette procédure dépend du code de votre procédure. En effet, le paramètre ne stocke pas la valeur quand vous appelez une procédure, mais seulement son adresse. C'est la raison pour laquelle que quand vous affectez le paramètre dans une procédure, vous affectez l'adresse et vous changez la valeur de cette adresse. Comme nous travaillons avec l'adresse, nous devons utiliser les pointeurs. En fait, les paramètres par adresse ou par référence sont des pointeurs et quand nous appelons une procédure, nous utilisons un pointeur ou bien nous donnons l'adresse des variables. Avec les paramètres par adresse nous devons utiliser une variable ou un pointeur, nous ne pouvons pas qu'utiliser une valeur. Après cette petite introduction, nous pouvons voir la syntaxe :

```
Void name (type *name1, type *name2, …) {
Statement
}
```

Et la syntaxe pour appeler une procédure :

```
name (X1, X2, …);
```

Avec Xi = adresse ce qui signifie un pointeur sans le symbole étoile : '*' ou une variable avec le symbole de déréférencement : '&'.

Quand vous utilisez le symbole de déréférencement, il doit obtenir l'adresse de votre variable et vous n'avez besoin que de cela pour le paramètre par adresse. Comme vous pouvez le voir dans la syntaxe, il n'y a que des paramètres par adresse mais ce n'est pas une obligation et vous pouvez mettre des paramètres par valeur mais il n'y a pas d'ordre. Pour illustrer ce point comme d'habitude, je vous ai écrit un code afin de comparer les deux paramètres différents. Rappelez-vous que vous devez jouer avec toutes les notions que vous avez déjà appris, c'est la meilleure solution pour améliorer votre savoir! Bon courage! :-)


```
// C Source File
// Created 13/03/2003; 17:16:24

#define USE_TI89               // Compile for TI-89
#define USE_TI92PLUS           // Compile for TI-92 Plus
#define USE_V200               // Compile for V200

#define OPTIMIZE_ROM_CALLS     // Use ROM Call Optimization

#define MIN_AMS 100            // Compile for AMS 1.00 or higher

#define SAVE_SCREEN            // Save/Restore LCD Contents

#include <tigcclib.h>          // Include All Header Files

void addition (int x1, int x2, int *answer) {
  *answer = x1 + x2;
  x1 = 3;
  x2 = 5; // these variables are modified only within this procedure
}

void change (int *x, int *y, int *z) {
  int aux;

  aux = *x;
  *x = *y;
  *y = aux;
  *z = 27;
}


// Main Function

void _main(void)
{
  int a, b, c;
  int *sum;
  c = 13; /* you have to initialize also this variable or else when the
code is running,
             this variable haven't address */
  sum = &c; // the pointer have to initialize for the same explication

  clrscr();

  a = 13;
  b = 4;
  addition (a, b, &c); // you want to give the address of your variable
so you use '&'
  printf ("13 + 4 = %d\n", c);


  addition (17, 4, sum);/* you want to give the address of your pointer
so you don't use
                           //the '*' acronym, it's the definition of a
pointer */
  printf ("\n\n17 + 4 = %d", *sum);
```

```
    ngetchx();
    clrscr();

    printf ("initial value:\n a = %d\n b = %d\n c = %d", a , b, c);
    change (&a, &b, &c);
    /*
    the procedure gets the address of these variables and it modifies the
data of these address
    at the end of procedure, the address is available within the procedure
and the out of
    the procedure
    */
    printf ("\n\nfinal value:\n a = %d\n b = %d\n c = %d", a , b, c);

    ngetchx();
  }
```

top

4) Fonctions

En fait, une procédure est une fonction qui ne renvoie aucune valeur : 'void'. Si vous voulez utiliser une fonction, vous devez dire au compilateur le type de valeur que vous voulez. La syntaxe est :

type name (parameters by value or address) {
statement
}

Bien sûr, l'utilisation des paramètres n'est pas obligatoire. Quand vous voulez appeler une fonction, cela ressemble à la méthode des procédures : vous écrivez dans votre code : ' name (); ' mais comme vous utilisez une procédure, le compilateur doit obtenir une valeur donc vous devez affecter une variable, afficher la valeur à l'écran, … cela se fait comme pour une variable. Laissez-nous illustrer ce point!


```
// C Source File
// Created 09/03/2003; 14:03:53

#define USE_TI89              // Compile for TI-89
#define USE_TI92PLUS          // Compile for TI-92 Plus
#define USE_V200              // Compile for V200

#define OPTIMIZE_ROM_CALLS    // Use ROM Call Optimization

#define MIN_AMS 100           // Compile for AMS 1.00 or higher

#define SAVE_SCREEN           // Save/Restore LCD Contents
```

```c
#include <tigcclib.h>          // Include All Header Files

int addition (int x1, int x2) {
  return (x1 + x2);
}

int multiplication (int x1, int x2) {
  return (x1 * x2);
}

// Main Function
void _main(void)
{
  int aux;

  clrscr();

  aux = addition (13, 8);
  printf ("13 + 8 = %d\n\n17 * 2 = %d", aux, multiplication(17, 2));

  ngetchx();
}
```

top

5) La recursivité

La recursivité ressemble aux boucles mais elle est faite avec des procédures et / ou des fonctions. En fait, dans le code de votre procédure, vous rappelez cette procédure : c'est le même effet que pour une boucle simple. Cela a l'avantage de diminuer votre code mais il y a des inconvénients : premièrement le compilateur stocke tous les appels et par conséquent elle est moins rapide qu'une boucle simple. Deuxièmement elle est plus complexe à comprendre. Mais parfois, vous aurez moins de problèmes en utilisant la recursivité. Nous pouvons voir deux exemples : un pour comprendre comment ça mrche et un autre pour voir son efficacité dans un véritable exemple.

Example 1:


```c
// C Source File
// Created 16/03/2003; 14:39:25

#define USE_TI89               // Compile for TI-89
#define USE_TI92PLUS           // Compile for TI-92 Plus
#define USE_V200               // Compile for V200

#define OPTIMIZE_ROM_CALLS     // Use ROM Call Optimization

#define MIN_AMS 100            // Compile for AMS 1.00 or higher
```

```
  #define SAVE_SCREEN           // Save/Restore LCD Contents

  #include <tigcclib.h>         // Include All Header Files

  #define number              7
    /*
    you must be careful because number < 13 or else the result is huge and
over an unsigned long int
    factorial (n) = n! = n*(n-1)*(n-2)*...*3*3*1 and 0! = 1
    example: 5! = 5*4*3*2*1 = 120
    */

  unsigned long factorial1 (int n) {
    int a;
    unsigned long facto;

    if (n == 0)
      return(1);

    facto = 1;
    for (a = 2; a <= n; a++)
      facto *= a;
    return (facto);
  }

  unsigned long factorial2 (int n) {
    if (n == 0)
      return (1);

    if (n != 1)
      return (n * factorial2 (n - 1));
    else
      return (1);
  }


  // Main Function
  void _main(void)
  {
    clrscr();

    printf ("13! = %lu\n", factorial1(number));
    printf ("13! = %lu", factorial2(number));

    ngetchx();
  }
```

Exemple 2:

Cet exemple explique comment vous pouvez faire un menu et c'est la méthode la plus simple pour faire cela car si vous n'utilisez pas le recursivité, vous utiliserez une quantité de conditions avec de nombreuses boucles et vous obtiendrez la même chose. La vitesse n'est pas importante pour un menu! :-)


```
// C Source File
// Created 16/03/2003; 15:10:58

#define USE_TI89              // Compile for TI-89
#define USE_TI92PLUS          // Compile for TI-92 Plus
#define USE_V200              // Compile for V200

#define OPTIMIZE_ROM_CALLS    // Use ROM Call Optimization

#define MIN_AMS 100           // Compile for AMS 1.00 or higher

#define SAVE_SCREEN           // Save/Restore LCD Contents

#include <tigcclib.h>         // Include All Header Files

// I declare the procedure because I call procedures within procedure
// and so I don't need to sort the procedures.

char menu();

void about();

void options();

void speed();

void difficulty();

// The procedures:

/*
The function 'menu' returns a char because I use the result like a
boolean value.
1 = true
0 = false
I use a char because it's not the worth to use an integer or a short. The
compiler will use more
memory storage and it's unuseful. Even a char takes much memory to store
only 1 or 0!
*/

char menu() {
  int key;

  clrscr();

  printf("F1 - Game\n\n");
  printf("F2 - Options\n\n");
  printf("F3 - About ...\n\n");
  printf("F4 - Exit");

  key = ngetchx();

  switch (key) {
    case KEY_F1:
      return (1);
      break;
    case KEY_F2:
      options();
      break;
```

```
      case KEY_F3:
        about();
        break;
      case KEY_F4:
      case KEY_ESC:
        return (0);
    }

    return (menu());
  }

  void about() {
    clrscr();

    printf ("This menu is created by:\n\n\n    Frédéric RIVAL\n\nMember of
the Quiche Team");
    ngetchx();
  }

  void options() {
    int key;

    clrscr();
    printf("F1 - Dificulty\n\n");
    printf("F2 - Speed\n\n");
    printf("F3 - Return");

    key = ngetchx();

    switch (key) {
      case KEY_F1:
        difficulty();
        break;
      case KEY_F2:
        speed();
        break;
      case KEY_F3:
      case KEY_ESC:
        return;
    }

    options();
  }

  void speed() {
    clrscr();
    printf("Choose your speed");
    ngetchx();
  }

  void difficulty() {
    clrscr();
    printf("Choose your difficulty");
    ngetchx();
  }

  // Main Function
  void _main(void)
  {
    while (menu()) {
      clrscr();
```

```
    printf("Here the game!");
    ngetchx();
  }
}
```
