Une page de C...

① Introduction au langage C.

② Le C++ sous MS DOS®.

- ⇨ Articulation
- ⇨ Le C c'est parti !
  - ❶ Définition symbolique.
  - ❷ La compilation conditionnelle.
  - ❸ L'incorporation de fichiers header.
  - ❹ Démarquage des commentaires.
  - ❺ Création de son premier source.
  - ❻ Les types de données et prototypes C.
  - ❼ Les opérateurs et structure algorithmique standard.
  - ❽ Quelques prototypes à connaître.
  - ❾ Ce que nous n'avons pas vu.

③ Bibliographie.

Cette documentation, écrite en 1994, est libre de droit, merci simplement de respecter son auteur. Vous pouvez retrouver toutes mes documentations et tous mes projets sur mon référentiel GitHub à l'adresse : https://santeroc.github.io.

Introduction au langage C :

Le langage C à été inventé en 1978 par Brian W. Kernighan et Dennis M. Ritchie. La lettre C est souvent épelée pour Compiler. Très répandu de nos jours le C exploite les normes Américaine internationales dite ANSI.
Très proche du langage de la machine (l'assembleur) le C est utilisé sur de nombreuses stations de travaille utilisant des systèmes d'exploitation tel que UNIX®, WINDOWS®, MS DOS®, etc...
Le C existe sous un grand nombre de version pour ne citer que ceux-ci :
Microsoft C++, Borland C++,Turbo C ou C++, Metaware C++, Visual C ,Zortech C++,etc...
Tous ces langages emploient une syntaxe commune mais des prototypes* différents.

C, C++ Y a-t-il une différence ?

Comme vous l'avez sûrement constaté, les langages précédemment interpellés font rapport d'un "++" après le C. Le C++ est une extension du C : il comprend la syntaxe et le vocabulaire standard du C mais aussi une notion récente qui est la programmation orienté OBJET (POO en Français ou OOP en Anglais !).
Le C est un langage fonctionnel et le C++ est un langage fonctionnel ou orienté objet. Dans un programme C on peut trouver du C (bien évidemment) mais aussi de l'assembleur car la plupart des compilateurs d'aujourd'hui supporte le mélange.

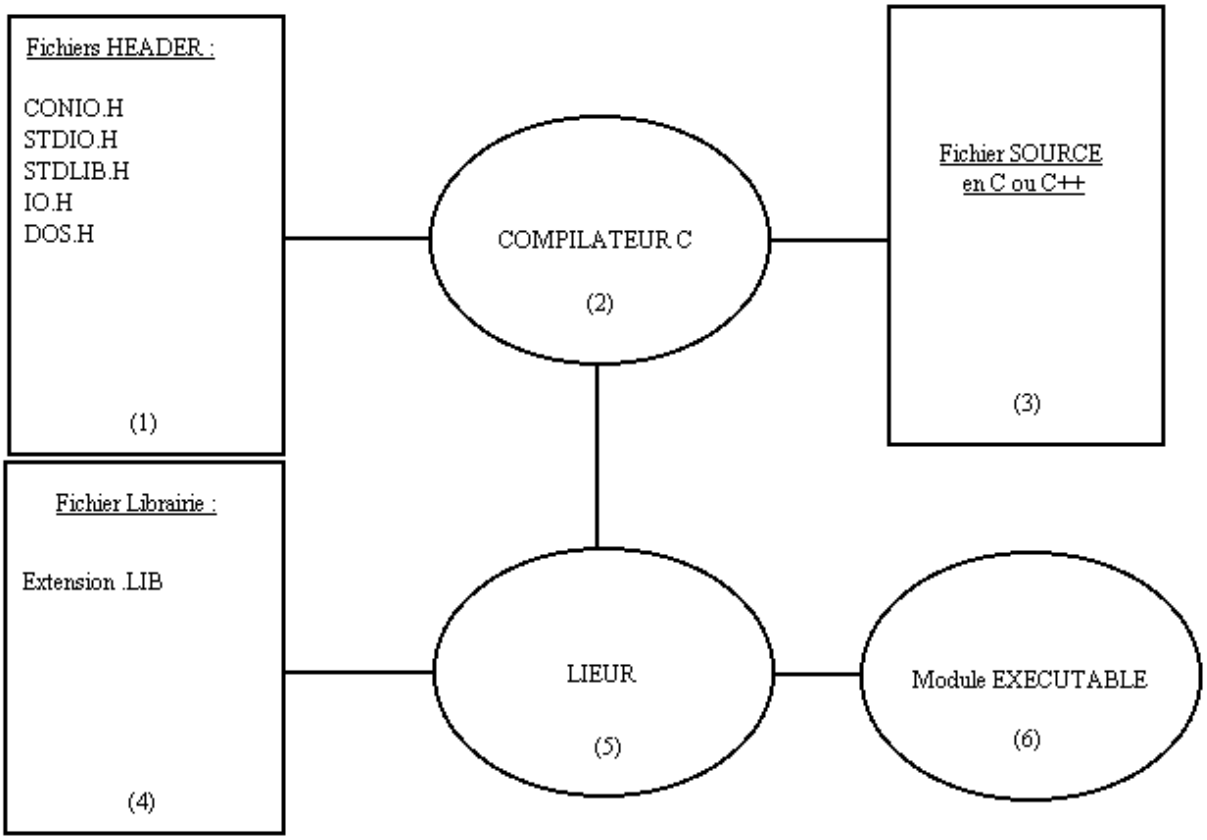
Nous allons étudier ensemble pendant un court moment le C++ de Borland sous MS DOS.
Pour aller plus loin en C++ une bibliographie vous est proposée à la fin de ce chapitre.

(*) Nous verrons cette notion un peu plus tard pour l'instant dites vous qu'il ne s'agit que de mots réservés.

Le C++ sous MS DOS :

Le C est un langage bâtit autour de 4 points vitaux :
Un Compilateur, un lieur, des fichiers d'entête (header file) et des libraires.

⇨Articulation :



Le fonctionnement est simple : Tout d'abord le compilateur examine le source C que vous lui avez demandé de compiler. Dès lors il exécute 2 phases importantes : *Pré compilation* puis *Compilation.* Dans sa première phase le compilateur résout toutes les conditions de compilation, puis s'il n'a pas trouvé d'erreur il entre dans la phase de compilation même. C'est à ce moment que les fichiers Header sont compilés à leurs tours pour créer finalement un fichier **Objet.** Ce fichier **Objet** servira lors du "liage" par le *LIEUR* avec l'aide des librairies à constituer un module finale exécutable par l'utilisateur.
Donc nous pouvons maintenant parler du contenu des fichiers **Header** et **Librairie** : Les fichiers Header contiennent des *PROTOTYPES* C et/ou C++ et les fichiers Librairie contiennent les *ROUTINES ASSEMBLEURS* De chaque PROTOTYPE.

⇨Le C c'est parti :
Tout d'abord nous allons étudier quelques instructions du Pré processeur :
Il faut savoir que toutes les instructions pré processeur commence par une dièse (symbole " # ") et que toutes les instructions en C/C++ sont écrites en minuscule (en effet le compilateur distingue les MAJ/MIN et n'accepte pas une instruction écrite en **majuscule** SI elle **n'existe pas** en majuscule !!!).
Par exemple la directive #include existe mais la directive #INCLUDE n'existe PAS !

I) DEFINITION SYMBOLIQUE :
La directive #define permet d'associer à un nom quelque chose.
Elle est construite de la façon suivante : #define <nom> [ESPACE] <phrase>
Par exemple :
#define entier int
Dans cette exemple le compilateur associe au mot "entier" le mot "int" et lorsque le mot "entier" sera lu quelque part dans votre source il le **REMPLACERA** systématiquement par le mot int.

d'autres exemples :
#define compteur 3 (Remplace compteur par 3 partout dans le source).
#define toto ta=8; (Remplace toto par ta=8; partout dans le source).

NOTE: Il est possible de fixer une fonction par un #define :
#define nbmax(a,b) (((a) > (b)) ? (a):(b))
Crée une fonction nommée nbmax qui reçoit a et b et qui renvoie le plus grand des deux (en réalité vous pourriez comprendre cette phrase comme une insertion des caractères **>** ***?*** 'recopie de l'élément a' ***:*** 'recopie de l'élément b' entre a et b).
Si le compilateur lit :
q=nbmax(38,27);
Alors il traduira en C :
q=((38) > (27) ? (38) : (27));
La création de fonction par define reste très complexe pour des débutants en C et pour démarrer d'un court intérêt.

II) LA COMPILATION CONDITIONNEL :

On peut compiler des parties de codes ou d'autres selon la présence ou non d'une définition C (écrire #define OK dit au pré processeur que OK existe et donc vaut 1, réciproquement pas de #define OK ou #define OK 0 informe le pré processeur que OK n'existe pas ou tout simplement qu'il vaut zéro).

#ifdef <définition> [ou] #ifndef <définition>
 [ce qu'il faut faire le cas échéant.]
 #else [ce qu'il ne faut pas faire.]
#endif [fin de condition]

#ifdef Comme son nom l'indique est vrai si la définition est vraie.
#ifndef [#if not def] est vrai si la définition est fausse.
exemple :

```
#define toto                    // Cette ligne dit que toto est défini, elle est
                                //  facultative.
#ifdef toto
  #include<stdio.h>
  int fnf1(int a)
 {
   return(a);
 }
#else
 #include<conio.h>
 int fnf1(int b)
 {
   return(b+1);
 }
#endif
```

Dans cet exemple en plaçant ou en enlevant #define toto vous déciderez de laquelle des deux fonctions (fnf1) sera utilisée plus tard.
On peut donc inclure entre ces deux directives (#ifdef, #ifndef) soit du code C, soit d'autres instructions pré processeur (ici #include) soit les deux !

III) L'INCORPORATION DE FICHIERS HEADER :

Comme vous l'avez sûrement déjà lu #include est la directive qui lie un fichier header (extension .h) à votre source C. Les bibliothèques C/C++ sont très fournit en fonctionnalité diverse mais vous pouvez décider de créer vos propres librairies en créant des fichiers HEADER (Si vous souhaitez ajouter de nouvelles fonctionnalités par exemple).
Syntaxe :
#include<librairie standard C/C++>
les " < " " > " sont obligatoires !
Vous pouvez aussi écrire :
#include "chemin\...\votre librairie .h"
Cette fois-ci avec des doubles quotte.

IV) DEMARQUAGE DES COMMENTAIRES EN C/C++ :

En C on note les commentaires dans un source a l'aide des marqueurs " /* " et " */ " :
/* Commentaires */
[ou]
/* Commentaires
Commentaires
    Commentaires
Commentaires */
Cette notation est correcte en C et en C++.
En C++ uniquement on dispose d'un "tueur de ligne" bien plus pratique :
.....Code C.... // Commentaire
...Ligne suivante Code C...
Le marqueur " // " Place tout ce qui suit jusqu'à la ligne suivante en commentaire. Ainsi si on veut tuer deux lignes on note :
...Code C... // Commentaire
...Ligne suivante Code C... // Commentaire
Tout ce qui se trouve avant le " // " et après le début de la ligne (ici ce qui est entre les points de suspension) n'est pas en commentaire.
Si on veut que tout la ligne entière soit un commentaire alors on écrira :
...Code C...
// ...Ligne de code en commentaire...
...Ligne suivante Code C...

V) CREATION DE SON PREMIER SOURCE EN C/C++ :

Un source C a comme tout source de langage structuré une STRUCTURE que voici :

```
#include<conio.h>                  // PRIMO : Les Fichiers d'entêtés.
#include<stdio.h>                  // Ici conio (Console), stdio (standard IO),
#include<alloc.h>                  //  alloc (allocation mémoire).

#define conteur 48                 // Définitions.
#define carré(a,b) ((a) * (b))     //

double nb1;                        // Variables Globales (OVER SCOPE).
int z;                             //
char c,d,e;                        //

void fonction1(void)               // Définitions des fonctions ou prototypes
{
  printf("Bonsoir !");             // Corps d'une fonction.
}

void main()                  // LA FONCTION PRINCIPALE.
{
   int a,b,w;                      // Variables Locales (UNDER SCOPE).
   fonction1();                    // Corps de la fonction principale.
...
}
```

A noter que la fonction main est obligatoire dans tout source C/C++ sauf dans un fichier header.
Son prototype est le suivant :
En C :

```
 main();
```

En C/C++ :

```
void main();
```

[ou]

```
int main();
```

Jusqu'ici nous avons beaucoup parlé de prototype sans en avoir donné une explication. Donc nous allons enfin éclairer votre lanterne !...

VI) LES TYPES DE DONNEES ET PROTOTYPES C/C++ :

Un prototype est la description d'une fonction. Il décrit ce que reçoit et ce que renvoie la fonction.
Il est composé d'un type suivi d'un nom arbitraire d'une paire de parenthèse "( )", et entre les deux on trouve encore des types puis :
* Soit un point virgule (ce qui indique que le corps de la fonction est ailleurs),
* Soit le corps de la fonction.

Exemple : void mafonction(int a,char b,long c);
int une_autre_fonction(char *c,long d);
En C K&R on écrivait les protypes ainsi :
exemple : void mafonction(a,b,c)
int a; char b; long c;
suivit du corps de la fonction. Cette notation est aujourd'hui obsolète !

⇒ Les types et modifieurs de type C/C++ :
Les types C sont les suivant :

TYPE :	FORMAT :
int	I2, signé -32768, 32767
short	I2, pareille que int
char	1 octet signé -128, +127
long	I4, signé -2 147 483 648, 2 147 483 647
double	I8, 1.7E-38, 3.4E+38 précisions 7 digits
long double	I10, 3.4E-4932, 1.1E+4932 précisions 19 digits
float	I4 format IEEEE standard.

Modifieur de types :
unsigned : élimine le signe d'un type (Il est fortement recommandé de cocher dans ses options de compilation la case [X] unsigned char pour obtenir des caractères de 0 à 255 !!!! TRES IMPORTANT !!!). Cela évite de toujours marquer unsigned char !
P.S. : Tous les programmes d'exemples sont compilés avec cette option active.
void : C'est le vide C.
far : Implique la donnée dans le FAR HEAP (généralement utilisé avec des pointeurs).

const : Défini un type constant.

Exemple de définition de variables C :

```
int a;                  // Déclare un entier a,
int b,c,d;              // Déclare 3 entiers b,c et d;
char q=21,c='d'; // Déclare 2 caractères q qui prend la valeur 21 et c qui
                        prend la valeur ASCII équivalente à la lettre 'd'.
unsigned long z=24L; // Déclare un I4 non signé de valeur 24 NOTEZ la
                          // présence du L après le 24.
short r=0x1A;           // Déclare un I2 de valeur 1A hexadécimal.
const float e=2.70;     // Déclare un flottant constant de valeur 2.7.
```

Les pointeurs C :

On peut déclarer un pointeur sur n'importe quel type en plaçant devant une *.

exemple :

```
int *b;
char *c;
char far *w;
void far *q;     // Seul type void valide avec void *g !
```

Note : la place de ces pointeurs n'est pas allouée ! Il pointe donc n'importe où en mémoire centrale. On peut allouer pour chacun d'eux une place respective.

Exemple d'un programme C que vous pouvez compiler (sans intérêt sauf si vous pouvez lire les valeurs de a et s dans un débugueur Tapez dans l'IDE <F8>,<ALT> + <W>, <W>, <INS>, <a>,<ENTREE>, <INS>, <s>, <ENTREE>,<F8> pour progresser...):

```
#include<alloc.h>      // A ne jamais oublier.
#include<string.h>     // Pour les manips...
void main()
{
  int *a; // Je déclare un pointeur sur UN ou DES entier(s).
  char *s; // Je déclare un pointeur sur UN ou DES caractère(s).
// Si on avait voulu éviter un malloc avec s on aurait pu écrire :
// char s[20];   // Qui définie s comme un tableau de 20 caractères.

  a=(int*)malloc(1*sizeof(int)); // a (son adresse) pointe maintenant sur une
                                 // zone de 2 octets (1 int = un I2) qui peut
                                 // recevoir une valeur.
```

```
  *a=45;                          // En adresse a j'inscrit dans la mémoire la
                                  // valeur 45 stockée dans l'ordre inverse
                                  // (Standard I2).

  s=(char*)malloc(20*sizeof(char)); // J'alloue un espace de 20 caractères et
                                                  // je fais pointer s dessus.
  *s='c';   // Equivaut à s[0]='c';
  s++;      // J'ajoute 1 à s donc s ne pointe plus sur la même adresse.
// NOTE TRES IMPORTANTE : il ne faut jamais perdre l'adresse de ce que //
l'on a alloué !
// Selon le type de donnée l'incrémentation d'un pointeur peut s'effectuer sur //
plusieurs octets !
 a++;  // Fais progresser l'adresse de a de 2 octets !!! De plus a ne pointe plus
// sur 45 mais de nouveau n'importe où (enfin presque puisque l'on sait
// encore où on est à peu près !).
a--,s--; // Notez la présence de la virgule : Si les opérations successives sont
// des affectations on peut les séparer d'une virgule.
// (++ et -- incrémente ou décrémente une valeur au cas où quelqu'un n'aurait
// pas compris).
 strcpy(s,"SALUT"); // Je copie SALUT dans s notez la différence
                                  // simple/double quotte ici double.
 s++;  // Je regarde ce qui se passe,
 s--;    //...
}
```

A noter : En C/C++ les chaines de caractères sont aux formats ASCIIZ strings. C'est à dire qu'elles se finissent par un 0 final appelé NULL TERMINATOR (Aucun rapport de près ou de loin avec le film !).

Aller sur strcpy dans le programme et appuyez sur <CTRL> <F1> regardez le prototype de strcpy :

```
 char *strcpy(char *s1,const char *s2);
```

Le char * devant strcpy indique que strcpy renvoie un pointeur de type caractère. Notez qu'en C les arguments renvoyés par une fonction ne nécessitent pas toujours une affectation : nous aurions pu écrire :

```
s=strcpy(s,"SALUT"); // Pour les plus téméraires.
```

Puis comme indiqué ci-dessus suit le nom de la fonction ici strcpy (pour string copy) et enfin la liste des arguments entre parenthèse.

De même voyez-vous même le prototype de malloc qui est le suivant :
void *malloc(size_t size);
Cette fonction renvoie un pointeur sur du vide le (int*) et (char*) entre le = et le malloc sont des transtypages. En effet la fonction malloc alloue de la mémoire mais elle ne sait pas pour qui, puisque a est un entier (type int) on doit informer le compilateur du bon type de l'objet concerné.
Cette fonction reçoit un élément nommé size de type size_t.
Ce dernier est un type qui qualifie une quantité de mémoire.
Sachez que les types C/C++ ne sont pas unique on peut créer ses propres types de données voyez l'aide sur typedef. Il ne faut pas se fier à n'importe quoi, dans la parenthèse des éléments reçut figure toujours le type de l'objet suivit du nom de l'objet associé à ce type.
On peut imaginer une fonction glubglub comme suit :

zom *glubglub(gag *wg,rzog a,flipflup *zr);
Cette fonction revoit un pointeur sur un élément de type zom, et reçoit pointeur sur un type gag nommé wg, un élément de type rzog nommé a et un pointeur de type fliflup nommé zr. C'est aussi simple que ça !
Ne paniquez jamais devant un prototype et entraînez vous à en lire des plus variés.
La seule difficulté réside dans le type void puisqu'il peut remplacer n'importe le quel des types existants.
Exemple :
int copybuffer(void *champs,unsigned n);
Cette fonction peut recevoir tout pointeur quelque soit le type de la donnée.
Lorsque le type d'une fonction est défini sa longueur est connue, donc dans un tableau d'entier par exemple une incrémentation du pointeur provoquera un ajout de 2 à l'adresse indiquée, pour un pointeur sur long double une incrémentation du pointeur provoquera un ajout de 10 à l'adresse du pointeur !!!
Tout pointeur est un tableau en C :
Si q est un pointeur quelconque on peut écrire indifféremment :
*q=21;
q[0]=21;
ou
*(q+21)=35;
q[21]=35;

A vous de ne pas perdre l'adresse de votre pointeur ou de ne pas sortir de votre tableau car le compilateur ne gère pas les fins de tableau !
Exemple :

```
char q[10]; // Pointeur de type char sur une zone alloué de 10 caractères de 0
            // à 9.

q[34]=55;  // DANGER !
```

ou encore :

```
strcpy(q,"01234567890123456789012345678 9"); // DANGER !!!
```

Transmission de valeurs à une fonction :
Tout élément définie en C à une adresse qui est obtenue par &objet, ou encore &objet[indice].
Exemple d'une fonction qui intervertit 2 valeurs :

```
void inter(int &a,int &b)   // Réception d'adresse.
{
  int z=a;

  a=b,b=z;
}

void main()
{
  int h=35,i=20;

  inter(h,i);  // Transmet les adresses respectives de h et i.
}
```

LOCALITE DES VARIABLES :
Elle est très simple : toute variable déclarée à l'extérieur d'un bloc (entre { et }) est globale elle est dite OVER SCOPE.
Toute variable déclarée dans un bloc est locale à CE bloc elle est dite UNDER SCOPE.
Toute variable déclarée dans un prototype est local à cette fonction sauf pour des transferts d'adresses et de pointeurs.

Exemple :

```
int glo1;              // Globales
char glo2;

void toto(int a,int b)  // a et b reçut de la fonction appelante mais localisée.
{
  // glo1 et glo2 sont définis ici.
  // a et b son définis ici uniquement.
  int c,d;             // c et d sont définies entre ces deux crochets ( { et } ).
}

void main()
{
 //  glo1 et glo2 sont définis ici aussi.

  int c,d;           // Totalement différent et qui n'ont rien à voir avec la
                     //fonction toto, c et d sont définies dans ce bloc entre les
                     // deux crochets.
}
```

Seul problème : Si une fonction utilise des variables globales redéfinies :
Exemple :

```
int global1;         // Voici un entier global.
void domage(char global1)
{
  // ICI global1 est un char mais n'affecte en rien le globale global1 (int).
}

void c_est_bete()
{
  long global1;  // LA ENCORE global1 est un long et n'affecte en rien
                 // global1 (int).
}
```

RENVOIE D'UNE VALEUR :
toute fonction dont le type de retour n'est pas vide doit retourner une valeur.
La valeur reçut peut-être perçut dans une variable ou laissé vacante.
Exemple :
Renvoie d'un élément simple :

```
int renvoie1() // Le () équivaut à (void).
{
  return(1);  // Renvoyer 1;
}
```

Renvoie d'un pointeur :

```
char *renvoie_salut()
{
   char coucou[]={"SALUT"};  // le [] indique au compilateur C de trouver
                             // tout seul la taille de l'élément concerné.
  return(coucou);
}
```

TRANSTYPAGE :
Transtyper signifie changer de type.
On peut transtyper tous les types y compris ceux que vous avez définit.
On peut décider de changer de type à n'importe quel moment.
Le transtypage s'effectue par exemple pour l'appelle de la fonction malloc (cf. ci-dessus).
On est obligé de transtyper les types void.

```
p=(char*)malloc(10*sizeof(char)); // (char*) est un transtype !
```

Exemple :

```
void main()
{
   char c=255,q;
   int z,w=45;

   z=(int)c;          // Le (int) est un transtype.
   q=(char) w;      // Le (char) est aussi un  transtype.
}
```

Exemple plus flagrant :

```
void main()
{
  char *p;
  int z;
  p=(char*)&z;  // Je pointe sur le premier octet de l'I2.
  z=45;          // Codé en mémoire : 45-00
  p[1]=1;        //  Je rajoute une centaine : 45-01.
}
```

VII) LES OPERATEURS ET STRUCTURE ALGORITHMIQUE STANDARD:

Le C connaît un grand nombre d'opérateur et les utilise d'une façon très barbare pour les débutants. C'est de loin la partie la plus effrayante du langage C. Voyez ceci par exemple :

```
// a est un entier :
a=12,a+=3*2; for(int z=3;z!=41;z+=5,a-=3+z); // Pas facile non ?
```

a) Les opérateurs :

+ Addition,
- Soustraction,
* Multiplication,
/ Division,
% Modulo entier,
++ Incrémentation,
-- Décrémentation,
>> Décalage vers la droite,
<< Décalage vers la gauche,
! La négation (NOT),
& ET opératoire,
| OU opératoire,
= Affectation.

== (Deux signes égales) Comparateur d'égalité,
> Comparateur de supériorité,
< Comparateur d'infériorité,
!= Différence ou incompatibilité,
&& ET logique,
|| OU logique.

b) Les structures standard :

Le SI ALORS SINON :

if(condition) [alors bloc ou instruction]
else [sinon bloc ou instruction]

Le POUR a DE n A l AU PAS DE s :

for(type a;a<=l,a+=s) [bloc ou instruction]
le type de a est généralement numérique.
FORMAT général du POUR en C/C++ :
for(type variable;condition;mode d'incrémentation) [bloc / instruction]
La variable, la condition et le mode d'incrémentation peuvent ne rien avoir en commun !

Le TANT QUE :

while(condition) [bloc ou instruction]

Le REPETER TANT QUE :

do [bloc ou instruction] while(condition);

Le SELON:

```
switch(variable)
{
  case [élément de type]: instructions; ... instructions; break;
   case ...: ... break;
  default: instructions; ... instruction; break;
}
```

c) Délimiteur et règle de calcule :

Il faut tout d'abord savoir distinguer un caractère d'une chaine de caractères :

3 Un caractère est délimité par une paire de simple quotte : 'carac'.
Son interprétation par le compilateur est équivalente au type char c.

3 Une chaine de caractère est délimitée par une paire de double quotte : "chaine".
Son interprétation par le compilateur est équivalente au type char *s.

Explication par l'exemple :

```
int a=5,b=6,c=7,d=9,e,f;   // Quelques définitions.

f=a+b;       // f  reçoit la somme de a et de b (5+6=11).
f+=c;        // f  reçoit f plus c (11+7=18).
f/=2;        // f  reçoit f divisé par 2 (18/2=9).
f*=b+a;    // f reçoit f*(b+a) soit (9*(6+5)=9*11=99).
b++,c--;   // b reçoit b-1 et c reçoit c-1.
f&=0x0A  // f  reçoit f AND 0A hexa <=> f=f & 0x0A; .

if(!a) clrscr();                    // Si a est égale à zéro alors clrscr();
 else  printf("Bonjour !");       //  Sinon printf("Bonjour");

if(a=10 && c=11)  // Si a égale 10 ET c égale 11 alors exécuter le bloc d'
{                          // instruction que voici :
  printf("Fin anormal de pgm !");
  exit(1);
}
 else b-=a,c+=d;  // Sinon b=b-a et c=c+d.
```

Autres exemples :

```
int idx=0;
char s[10]={"1234567890"};

while(idx<9) s[idx++]+=1;  // Tant que idx<9
                                  //    s[idx]=s[idx]+1
                                  //     idx=idx+1
                                  // Fin TQ.

idx=-1;                           // idx égale moins 1.
while(idx<=9) s[++idx]++; // Tant que idx<=9
                                 //      idx=idx+1
                                 //      s[idx]=s[idx]+1
                                 //  Fin TQ.
idx=0;
for(int r=0;r<=100;r++) idx+=r; // Pour r de 0 à 100
                                       //    idx=idx+r
                                       // Fpour.
```

NOTE IMPORTANTE : Il est important de constater que r entier est défini à partir de cette ligne dans le bloc courant.

```
for(idx=0;idx<=10;idx+=2)    // Pour idx de 0 à 10 au pas de 2
{                            //     r=r-idx.
  r-=idx;                    //     Afficher 'Valeur de r : ' r.
  printf("Valeur de r : %i .",r); // Fpour.
}

do  {
        printf("\r\nFrappez sur la touche échap");
       if(!(c=getch()) c=0,getch();
     } while(c!='\x1B');
// Répéter
//     Retour à la ligne
//    Afficher 'Frappez sur la touche échap'
//     c=touche tapez au clavier.
// Jusqu'à ce que c = touche ESC.

c=getch();
switch(c)
{
  case 'a': printf("Vous avez tapé a."); break;
  case 'b': c++; clrscr(); printf("Au revoir !"); break;
  case 'c':
  case 'd': printf("Vous avez tapez c ou d."); break;
  default: printf("Je n'ai pas compris !"); break;
}
//  c=touche tapez au clavier
// Selon c
//     Cas c = 'a' : Afficher 'Vous avez tapez a.'
//     Cas c = 'b' : c=c+1
//                       Effacer l'écran (clrscr();)
//                        Afficher 'Au revoir !'
//     Cas c = 'c' ou c = 'd': Afficher 'Vous avez tapez c ou d.'
//     Autre cas : Afficher 'Je n'ai pas compris !'
// Fin Selon.
```

VIII) Quelques prototypes à connaître :

Le langage C comporte un grand nombre de librairies qui sont extensibles. En fonction des marques utilisées le nom des fonctions varies.
Pour savoir quel vocabulaire apprendre il vous faut connaître deux choses importantes :

- à Sur quelle station travaillez-vous ?
- à Quelle marque de compilateur utilisez-vous ou souhaitez-vous utilisez ?

Sachez seulement qu'il existe forcément un livre qui traite du C ou du C++ dans le contexte que vous recherchez !

Pour le moment nous allons voir quelques fonctions (prototypes) du C++ de BORLAND® sous MS DOS®.

Allocation de mémoire : cf. prototype malloc, calloc...
Libération d'un bloc de mémoire alloué par une fonction ci-dessus : free...
Routine de conversion : atof, atol, atoi, itoa...
Conversion et classification de caractère : isalpha, isdigit, tolower...
Manipulation sur les fichiers : open, close, write, read, chdir, remove,...
Routine d'entrée/sorite et flux de données : Il est fortement conseillé de se reporter à une documentation plus complète. Sinon il existe : getch, kbhit, putch, scanf, printf,...
Le langage C est très concentré sur les routines d'E/S.
Appelle au système : int86, int86x, FP_OFF, FP_SEG, MK_FP, keep, biosdisk, bioskeey, biosequip, harderr...
L'écran et le graphique :

En mode texte : textattr((couleur de fond<<4)|couleur du stylo), cprintf, gotoxy, wherex, wherey, window, insline, clreol, clrscr...
En mode graphique : grâce à l'interface BGI : chercher dans le fichier header <graphics.h> accessible depuis l'IDE...

L'accès aux prototypes des fonctions précédemment citées est possible depuis l'IDE avec des commentaires et des explications complémentaires.
Il suffit de taper le nom de la fonction ou de la librairie à examiner dans l'IDE, de se placer sur une lettre du mot et d'appuyer simultanément sur <CTRL> + <F1>. Sinon se reporter à la bibliographie.

IX) Ce que nous n'avons pas vu :

Les classes C++, la généricité avec template, les structures, les unions, les énumérations, La définition de type avec typedef, les ORP .-> -> *. *.->,
Les niveaux de déférences ou tableau de tableaux (type **t, type *t[]).
Surcharge fonctions/opérateurs.
Mots réservés particuliers new delete _export extern asm continue...
Les options avancer : notion de Memory models.
La directive #pragma.
Les fonctions inline et les pointeurs de fonctions.
Les contrôles de flux cin cout.
Création de ses propres fichiers header, conception de librairies.
Fonctionnement de l'IDE (plate-forme Borland).

Toutes les marques précédemment citées sont des marques déposées de :
AT&T BELL Laboratories, MICROSOFT corporation, BORLAND international, METAWARE Inc, ZORTECH Inc.

Bibliographie :

The Waite Group's Turbo C++ Bible de Naba Barkakati chez SAMS.
Toutes les fonctions du Turbo C++ de Borland détaillé par librairie avec une table des portabilités entre les différents C/C++ (Microsoft, Visual, Unix,...).
Très explicite, Très bien.

Borland C++ 3.1 Object-Oriented Programming de Marco Cantù et Steve Tendon chez BANTAM COMPUTER BOOKS.
Etude rapide des fonctionnalités et des structures du C++ de Borland avec une approche des objets C++ et de l'interface SAA (TURBO VISION) plus une courte initiation à l'environnement MS WINDOWS et ses objets.

An Introduction to Object-Oriented Programming and C++ de Richard S. Wiener et Lewis J. Pinson.
Un support de court très avancé sur la programmation orienté objet en C++.
Très riche en explication, très explicites avec des exercices corrigés.