CHARLES CARNUS
LYCÉE TECHNIQUE
ENSEIGNEMENT SUPÉRIEUR

Langage C

Compilation

Prof : **Kamal Boudjelaba**

12 septembre 2022

B T S S N

LTP Charles Carnus
Avenue de Saint Pierre - Bourran - 12000 RODEZ, France

IR–EC

Table des matières

Liste des figures

Liste des tableaux

1. Les structures

Une structure est une suite finie d'objets de types différents. Contrairement aux tableaux, les différents éléments d'une structure n'occupent pas nécessairement des zones contiguës en mémoire. Chaque élément de la structure, appelé membre ou champ, est désigné par un identificateur.
On distingue la déclaration d'un modèle de structure de celle d'un objet de type structure correspondant à un modèle donné. La déclaration d'un modèle de structure dont l'identificateur est `modele` suit la syntaxe suivante :

```
struct modele
{
  type-1 membre-1;
  type-2 membre-2;
  ...
  type-n membre-n;
};
```

Pour déclarer un objet de type structure correspondant au modèle précédent, on utilise la syntaxe : `struct modele objet;`
Ou bien, si le modèle n'a pas été déclaré au préalable :

```
struct modele
{
  type-1 membre-1;
  type-2 membre-2;
  ...
  type-n membre-n;
} objet;
```

On accède aux différents membres d'une structure grâce à l'opérateur membre de structure, noté ".". Le i-ème membre de objet est désigné par l'expression `objet.membre-i`
On peut effectuer sur le i-ème membre de la structure toutes les opérations valides sur des données de type `type-i`.
Par exemple, le programme suivant définit la structure complexe, composée de deux champs de type double ; il calcule la norme d'un nombre complexe.

```
#include <stdio.h>
#include <math.h>
struct complexe
{
  double reelle;
  double imaginaire;
};

int main()
{
    struct complexe z= {3. , 4.};
    double norme;
    norme = sqrt(z.reelle * z.reelle + z.imaginaire * z.imaginaire);
    printf("norme de (%f + i %f) = %f \n",z.reelle,z.imaginaire,norme);
    return 0;
}
```

2. Les énumérations

Les énumérations permettent de définir un type par la liste des valeurs qu'il peut prendre. Un objet de type énumération est défini par le mot-clef `enum` et un identificateur de modèle, suivis de la liste des valeurs que peut prendre cet objet : `enum modele {constante-1, constante-2,...,constante-n;`
En réalité, les objets de type `enum` sont représentés comme des `int`. Les valeurs possibles constante-1, constante-2,...,constante-n sont codées par des entiers de 0 à $n-1$. Par exemple, le type `enum booleen` défini dans le programme suivant associe l'entier 0 à la valeur "faux" et l'entier 1 à la valeur "vrai".

```
#include <stdio.h>

int main()
{
    enum booleen {faux, vrai};
    enum booleen b;
    b = vrai;
    printf("b = %d\n",b);
}
```

3. Compilation

3.1 A partir de l'IDE

Lancement de la phase de compilation

Pour compiler le fichier source, utilisez l'icône Build. Si vous n'avez pas d'erreur, vous pouvez passer à la phase suivante. Sinon, corrigez les erreurs ...

Exécution du programme

L'exécution n'est possible que si la compilation a été faite sans erreurs.

Les messages d'erreurs de compilation permettent de corriger les fautes de syntaxe. Corrigez les erreurs et relancez jusqu'à obtenir une compilation sans erreurs. Les warnings sont de simples avertissements et n'empêchent pas l'exécution. Faites attention néanmoins à ces warnings.
Le lancement de l'exécution se fait par l'icône Run

Remarque : l'icône Build and Run (compiler et exécuter) lance la compilation et l'exécution par la suite sauf en cas d'erreur de compilation bien sûr.

Dans la section Build log, l'IDE affiche quelques messages en bas de l'IDE. Et si tout va bien, une console apparaît avec le résultat de l'execution du programme.

3.2 Les étapes de la compilation

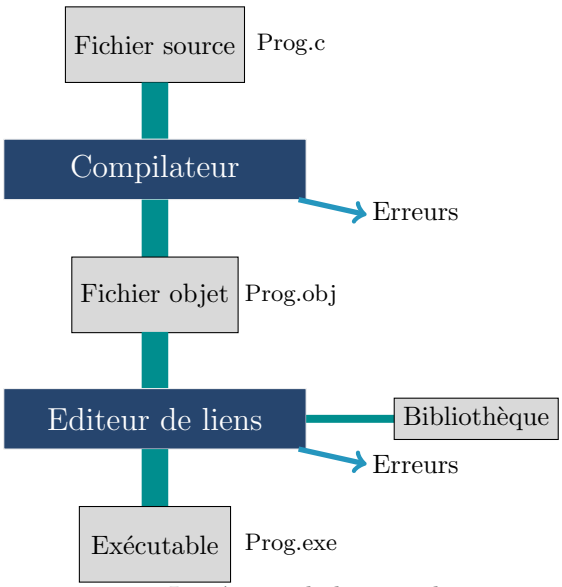

Figure 1. Les étapes de la compilation

La compilation consiste en une série d'étapes de transformation du code source en du code machine exécutable sur un processeur cible.
Le langage C fait partie des langages compilés : le fichier exécutable est produit à partir de fichiers sources par un compilateur.
La compilation passe par différentes phases :

Le préprocessing : Le compilateur analyse le langage source afin de vérifier la syntaxe et de générer un code source brut (s'il y a des erreurs de syntaxe, le compilateur est incapable de générer le fichier objet). Les commentaires sont enlevés et les directives de compilation commençant par # sont d'abord traités pour obtenir le code source brut.

La compilation en fichier objet : Les fichiers de code source brut sont transformés en un fichier dit objet, c'est-à-dire un fichier contenant du code machine ainsi que toutes les informations nécessaires pour l'étape suivante (édition de liens).

L'édition de liens : L'éditeur de liens (linker) s'occupe d'assembler les fichiers objet en une entité exécutable et doit pour ce faire résoudre toutes les adresses non encore résolues. C'est à dire que lorsqu'il fait appel dans le fichier objet à des fonctions ou des variables externes, l'éditeur de liens recherche les objets ou bibliothèques concernés et génère l'exécutable (Il se produit une erreur lorsque l'éditeur de liens ne trouve pas ces références).

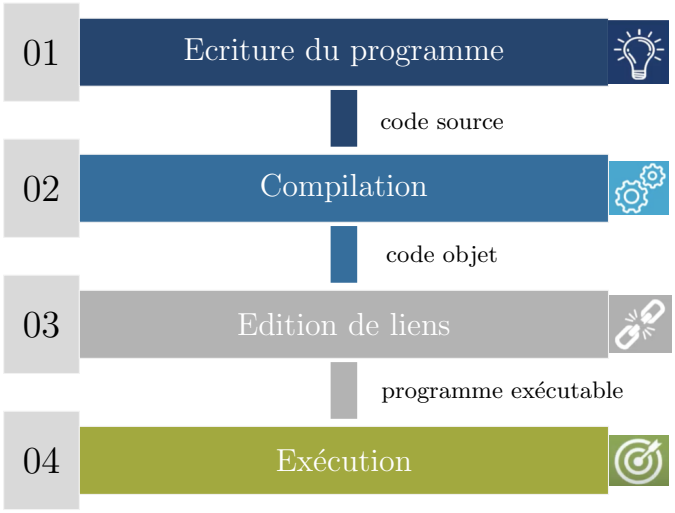

Figure 2. Les étapes pour exécuter un programme

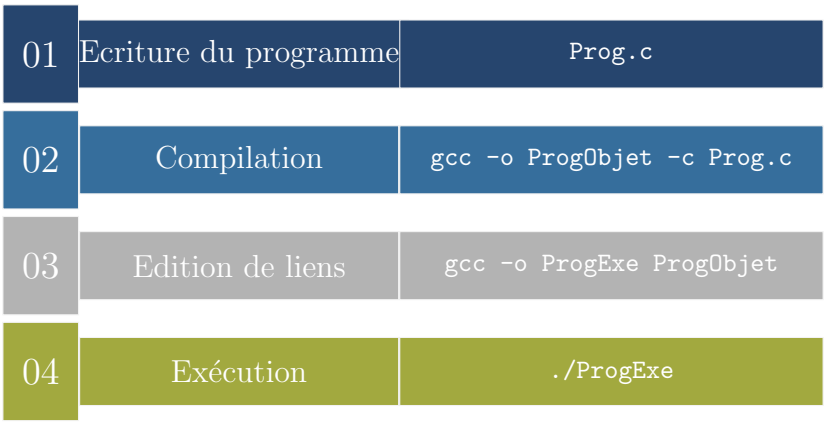

Figure 3. Les commandes pour exécuter un programme

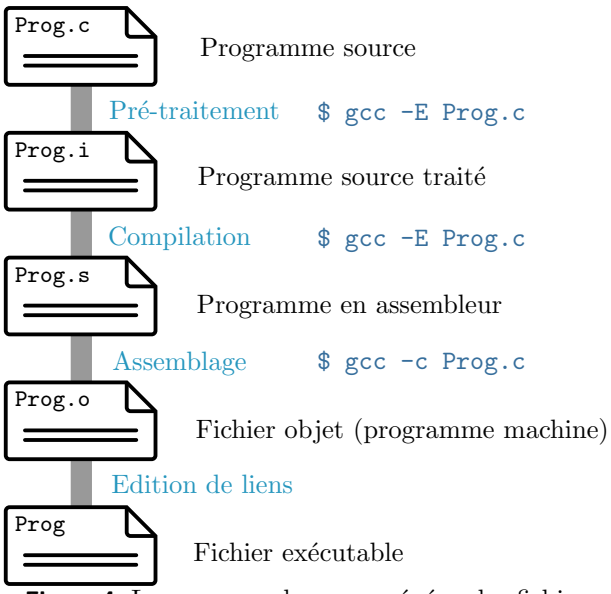

Figure 4. Les commandes pour générer les fichiers

Table 1. Méthode 1 - Commandes pour exécuter un programme en C

Commande[1]	Type du fichier généré	Nom du fichier généré
`gcc Prog.c -o Prog`	Exécutable	Prog
`./Prog`	Pour exécuter le programme	

[1] Les commandes doivent être réalisées dans l'ordre.

Table 2. Méthode 2 - Commandes pour exécuter un programme en C

Commande[2]	Type du fichier généré	Nom du fichier généré
`gcc -o ProgObjet -c Prog.c`	Objet	ProgObjet
`gcc -o ProgExe ProgObjet`	Exécutable	ProgExe
`./ProgExe`	Pour exécuter le programme	

[2] Les commandes doivent être réalisées dans l'ordre.

Table 3. Méthode 3 - Commandes pour exécuter un programme en C

Commande[3]	Type du fichier généré	Nom du fichier généré
`gcc -save-temps Prog.c -o ProgExe`	PS* traité	Prog.i
	Assembleur	Prog.s
	Objet	Prog.o
	Exécutable	ProgExe
`./ProgExe`	Pour exécuter le programme	

[3] Les commandes doivent être réalisées dans l'ordre.

* PS : Programme Source.

Table 4. Méthode 4 - Commandes pour exécuter un programme en C

Commande[4]	Type du fichier généré	Nom du fichier généré
`gcc -E Prog.c`	Affiche le programme source traité	
`gcc -c Prog.c`	Objet	Prog.o
`gcc -o Prog Prog.o`	Exécutable	Prog
`./Prog`	Pour exécuter le programme	

[4] Les commandes doivent être réalisées dans l'ordre.

Remarque 3.1 :

Pour l'exécution des programmes C++, il suffit de remplacer `gcc` par `g++`

Le contenu du dossier

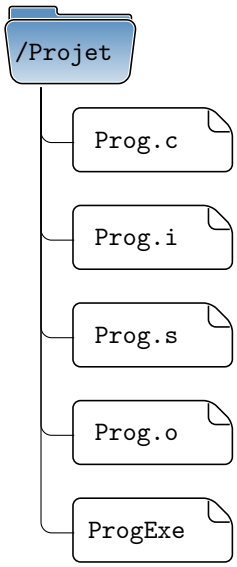

Figure 5. Fichiers générés en utilisant les commandes du tableau 3

Les fichiers générés

Contenu des fichiers générés en utilisant les commandes du tableau 3 :

Programme source : Prog.c Taille = 170 octets

```c
#include <stdio.h>
#define J 5

// Fonction main
int main()
{
    // Déclaration des variables
    int i = 10, k;
    k = i+J;
    printf("k = %d\n", k);
    return 0;
}
```

Programme source traité : Prog.i Taille = 23 ko

```
# 1 "Prog.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 363 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "Prog.c" 2
# 1 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/stdio.h" 1 3 4
.
.
500 lignes plus bas
.
.
extern int __vsnprintf_chk (char * restrict, size_t, int, size_t,
       const char * restrict, va_list);
# 408 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/stdio.h" 2 3 4
# 2 "Prog.c" 2



int main()
{

    int i = 10, k;
    k = i+5;
    printf("k = %d\n", k);
    return 0;
}
```

Programme en assembleur : Prog.s Taille = 474 octets

```
.section __TEXT,__text,regular,pure_instructions
.build_version macos, 10, 15, 4 sdk_version 10, 15, 4
.globl _main                    ## -- Begin function main
.p2align 4, 0x90
_main:                                  ## @main
.cfi_startproc
## %bb.0:
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset %rbp, -16
movq %rsp, %rbp
.cfi_def_cfa_register %rbp
subq $16, %rsp
movl $0, -4(%rbp)
movl $10, -8(%rbp)
movl -8(%rbp), %eax
addl $5, %eax
movl %eax, -12(%rbp)
movl -12(%rbp), %esi
leaq L_.str(%rip), %rdi
movb $0, %al
callq _printf
xorl %ecx, %ecx
movl %eax, -16(%rbp)         ## 4-byte Spill
movl %ecx, %eax
addq $16, %rsp
popq %rbp
retq
.cfi_endproc
                                        ## -- End function
.section __TEXT,__cstring,cstring_literals
L_.str:                                 ## @.str
.asciz "k = %d\n"


.subsections_via_symbols
```

Programme objet : Prog.o Taille = 792 octets

```
cffa edfe 0700 0001 0300 0000 0100 0000
0400 0000 0802 0000 0020 0000 0000 0000
1900 0000 8801 0000 0000 0000 0000 0000
.
.
40 lignes plus bas
.
.
0000 0000 0100 0006 0100 0000 0f01 0000
0000 0000 0000 0000 0700 0000 0100 0000
0000 0000 0000 0000 005f 6d61 696e 005f
7072 696e 7466 0000
```

Programme Exécutable : ProgExe.out Taille = 13 ko

```
cffa edfe 0700 0001 0300 0000 0200 0000
1000 0000 5805 0000 8500 2000 0000 0000
1900 0000 4800 0000 5f5f 5041 4745 5a45
524f 0000 0000 0000 0000 0000 0000 0000
.
.
800 lignes plus bas
.
.
0300 0000 2000 5f5f 6d68 5f65 7865 6375
7465 5f68 6561 6465 7200 5f6d 6169 6e00
5f70 7269 6e74 6600 6479 6c64 5f73 7475
625f 6269 6e64 6572 005f 5f64 796c 645f
7072 6976 6174 6500 0000 0000
```

3.3 Organisation d'un projet : Exemple

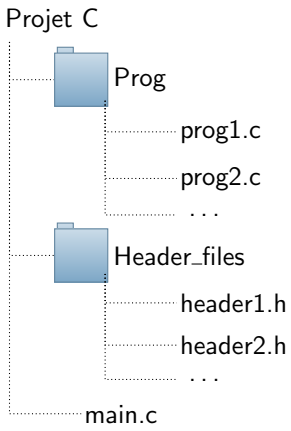

Figure 6. Contenu du répertoire `Projet C`

4. La fonction `main`

La fonction principale `main` est une fonction comme les autres. On peut la considérer de type `void`, ce qui est toléré par le compilateur. Toutefois l'écriture `main()` provoque un message d'avertissement :
`warning: type specifier missing, defaults to 'int'`

En fait, la fonction main est de type `int`. Elle doit retourner un entier dont la valeur est transmise à l'environnement d'exécution. Cet entier indique si le programme s'est ou non déroulé sans erreur.

— La valeur de retour 0 correspond à une terminaison correcte,

— toute valeur de retour non nulle correspond à une terminaison sur une erreur.

La fonction main peut également posséder des paramètres formels. En effet, un programme C ou C++ peut recevoir une liste d'arguments au lancement de son exécution. La ligne de commande qui sert à lancer le programme est, dans ce cas, composée du nom du fichier exécutable suivi par des paramètres. La fonction main reçoit tous ces éléments de la part de l'interpréteur de commandes.

La fonction main possède deux paramètres formels, appelés par convention **`argc`** (argument count) et **`argv`** (argument vector (value)).

- **`argc`** est une variable de type `int` dont la valeur est égale au nombre de mots composant la ligne de commande (y compris le nom de l'exécutable). Elle est donc égale au nombre de paramètres effectifs de la fonction +1.
- **`argv`** est un tableau de chaînes de caractères correspondant chacune à un mot de la ligne de commande. Le premier élément argv[0] contient donc le nom de la commande (du fichier exécutable), le second **`argv[1]`** contient le premier paramètre ...

Le second prototype valide de la fonction **`main`** est donc : `int main (int argc, char *argv[])`

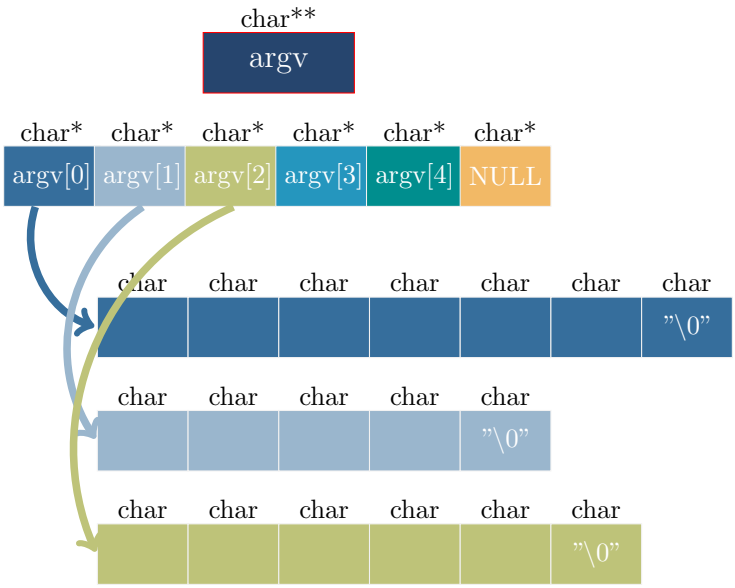

Figure 7. Les arguments de la fonction `main`

Exemple 4.1

Le programme suivant calcule le produit de deux entiers, entrés en arguments de l'exécutable :

```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    int a, b;
    if (argc != 3)
    {
        printf("\nErreur : nombre invalide d'arguments");
        printf("\nUsage: %s int int\n",argv[0]);
        return(EXIT_FAILURE);
    }
    a = atoi(argv[1]);
    b = atoi(argv[2]);
    printf("\nLe produit de %d par %d vaut : %d\n", a, b, a * b);
    return(EXIT_SUCCESS);
}
```

Dans le terminal, taper (après compilation) : `./NomProExe 15 10`
La console affiche : `La somme de 15 et 10 vaut : 25`

Exemple 4.2

Cet exemple affiche les arguments de la fonction `main`

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

int main(int argc, char *argv[])
{
int a, b;
  if (argc != 3)
    {
      printf("\nErreur : nombre invalide d'arguments");
      printf("\nUsage: %s int int\n",argv[0]);
      return(EXIT_FAILURE);
    }
  a = atoi(argv[1]);
  b = atoi(argv[2]);
  printf("\nLa somme de %d et %d vaut : %d\n", a, b, a + b);
  int i;
    for (i=0; i < argc; i++)
        printf("Argument %d : %s\n", i+1, argv[i]);
  return(EXIT_SUCCESS);
}
```

La console affiche : (après la commande `./NomProExe 15 10`)

```
La somme de 15 et 10 vaut : 25
Argument 1 : ./NomProExe
Argument 2 : 15
Argument 3 : 10
```

5. Les directives au préprocesseur

Le préprocesseur est un programme exécuté lors de la première phase de la compilation. Il effectue des modifications textuelles sur le fichier source à partir de directives. Les différentes directives au préprocesseur, introduites par le caractère `#`, ont pour but :

— l'incorporation de fichiers source (`#include`)

— la définition de constantes symboliques et de macros (`#define`)

— la compilation conditionnelle (`#if, #ifdef,` ...)

5.1 La directive `#include`

Elle permet d'incorporer dans le fichier source, le texte figurant dans un autre fichier. Ce dernier peut être un fichier en-tête de la librairie standard (`stdio.h, math.h, ...`) ou n'importe quel autre fichier. La directive `#include` possède deux syntaxes voisines :

#include <nom-de-fichier> Recherche le fichier mentionné dans un ou plusieurs répertoires systèmes définis par l'implémentation (par exemple, /usr/include/).

#include "nom-de-fichier" Recherche le fichier dans le répertoire courant (celui où se trouve le fichier source). On peut spécifier d'autres répertoires à l'aide de l'option `-I` du compilateur.

La première syntaxe est généralement utilisée pour les fichiers en-tête de la librairie standard, tandis que la seconde est plutôt destinée aux fichiers créés par l'utilisateur.

5.2 La directive `#define`

La directive `#define` permet de définir :

— Des constantes symboliques

— Des macros avec paramètres

Remarque 5.1 :

La directive `#undef` supprime une macro (constante) définie.

Définition des constantes symboliques

La directive `#define nom reste-de-la-ligne` demande au préprocesseur de substituer toute occurence de `nom` par la chaîne de caractères `reste-de-la-ligne` dans la suite du fichier source. Son utilité principale est de donner un nom parlant à une constante, qui pourra être aisément modifiée.

```
#define NB_LIGNES 10
#define NB_COLONNES 33
#define TAILLE_MATRICE NB_LIGNES*NB_COLONNES
```

Définition des macros

Une macro avec paramètres se définit de la manière suivante :

```
#define nom(liste-de-paramètres) corps-de-la-macro
```

Par exemple :

```
#define MAX(a,b) (a>b?a:b)
```

Le processeur remplacera dans la suite du code toutes les occurences du type `MAX(x,y)` où x et y sont des symboles quelconques par `(x>y?x:y)`
Une macro a donc une syntaxe similaire à celle d'une fonction, mais son emploi permet en général d'obtenir de meilleures performances en temps d'exécution.
La distinction entre une définition de constante symbolique et celle d'une macro avec paramètres se fait sur le caractère qui suit immédiatement le nom de la macro : si ce caractère est une parenthèse ouvrante, c'est une macro avec paramètres, sinon c'est une constante symbolique. Il ne faut donc jamais mettre d'espace entre le nom de la macro et la parenthèse ouvrante.
Il faut toujours garder à l'esprit que le préprocesseur n'effectue que des remplacements de chaînes de caractères.

En particulier, il est conseillé de toujours mettre entre parenthèses le corps de la macro et les paramètres formels qui y sont utilisés. Par exemple, si l'on écrit sans parenthèses : `#define CARRE(a) a*a` le préprocessuer remplacera `CARRE(a+b)` par $a+b*a+b$ et non par $(a+b)*(a+b)$. De même `!CARRE(x)` sera remplacé par $!x*x$ et non par $!(x*x)$.

5.3 La compilation conditionnelle

La compilation conditionnelle a pour but d'incorporer ou d'exclure des parties du code source dans le texte qui sera généré par le préprocesseur. Elle permet d'adapter le programme au matériel ou à l'environnement sur lequel il s'exécute, ou d'introduire dans le programme des instructions de débogage.
Les directives de compilation conditionnelle se répartissent en deux catégories, suivant le type de condition invoquée :

— la valeur d'une expression

— l'existence ou l'inexistence de symboles

Condition liée à la valeur d'une expression

```
#if condition-1
  partie-du-programme-1
#elif condition-2
  partie-du-programme-2
  ...
#elif condition-n
  partie-du-programme-n
#else
  partie-du-programme-N
#endif
```

Le nombre de `#elif` est quelconque et le `#else` est facultatif. Chaque condition-i doit être une expression constante. Une seule partie-du-programme sera compilée : celle qui correspond à la première condition-i non nulle, ou bien la partie-du-programme-N si toutes les conditions sont nulles.
Par exemple, on peut écrire :

```
#define PROCESSEUR ALPHA
#if PROCESSEUR == ALPHA
  taille_long = 64;
#elif PROCESSEUR == PC
  taille_long = 32;
#endif
```

Condition liée à l'existence d'un symbole

```
#ifdef symbole
  partie-du-programme-1
#else condition-2
  partie-du-programme-2
#endif
```

Si symbole est défini au moment où l'on rencontre la directive `#ifdef`, alors partie-du- programme-1 sera compilée et partie-du-programme-2 sera ignorée. Dans le cas contraire, c'est partie-du-programme-2 qui sera compilée. La directive `#else` est évidemment facultative. De façon similaire, on peut tester la non-existence d'un symbole par :

```
#ifndef symbole
  partie-du-programme-1
#else condition-2
  partie-du-programme-2
#endif
```

Ce type de directive est utile pour rajouter des instructions destinées au débogage du programme :

```
#define DEBUG
  ....
#ifdef DEBUG
  for (i = 0; i < N; i++)
    printf("%d\n",i);
#endif /* DEBUG */
```

Il suffit alors de supprimer la directive `#define DEBUG` pour que les instructions liées au débogage ne soient pas compilées. Cette dernière directive peut être remplacée par l'option de compilation -Dsymbole, qui permet de définir

un symbole.

6. Découper le programme en plusieurs fichiers

Le C (C++) permet de découper le programme en plusieurs fichiers source. Chaque fichier contient une ou plusieurs fonctions. On peut ensuite inclure les fichiers dont on a besoin dans différents projets.
Les fichiers d'en-tête contiennent les déclarations des types et fonctions que l'on souhaite créer.
Un programme écrit en C se compose généralement de plusieurs fichiers-sources. Il y a deux sortes de fichiers-sources :

— ceux qui contiennent effectivement des instructions ; leur nom possède l'extension `.c`,

— ceux qui ne contiennent que des déclarations ; leur nom possède l'extension `.h` (header ou en-tête).

Un fichier `.h` sert à regrouper des déclarations qui sont communes à plusieurs fichiers `.c`, et permet une compilation correcte de ceux-ci. Dans un fichier `.c` on prévoit l'inclusion automatique des fichiers `.h` qui lui sont nécessaires, grâce aux directives de compilation *#include*.
En supposant que le fichier à inclure s'appelle **entete.h**, on écrira *#include <entete.h>* s'il s'agit d'un fichier de la bibliothèque standard du C, ou *#include "entete.h"* s'il s'agit d'un fichier écrit par nous-mêmes.

Exemple 6.1

```c
/***** Prog.c original *****/
#include <stdio.h>
#define J 5

// Fonction main
int main()
{
    // Déclaration des variables
    int i = 10, k;
    k = i+J;
    printf("k = %d\n", k);
    return 0;
}
```

```c
/********** Prog.c **********/
#include <stdio.h>
#define J 5

// Fonction main
int main()
{
    // Déclaration des variables
    int i = 10, k;
    k = i+J;
    printf("k = %d\n", k);
    return 0;
#include "entete.h"
```

```c
/********** entete.h **********/
}
```

Exemple 6.2

```c
/****** main.c ******/
#include <stdio.h>
#include "test.h"

int main()
{
    int const x(5);
    printf("Voici un calcul : %d", x+fonction());

    return 0;
}
```

```c
/****** test.c ******/
#include "test.h"

int fonction()
{
    return 12;
}
```

```c
/****** test.hpp ******/
#ifndef TEST_H
#define TEST_H

int fonction();

#endif
```

```
Générer les fichiers objet : gcc -c test.c main.c
Lier les fichiers :          gcc test.o main.o -o mon_programme.out
Exécuter :                   ./mon_programme.out

Ou
En une seule étape : gcc test.c main.c -o programme.out
Exécuter :           ./programme.out
```

7. Exemples

Exemple 1

```c
#include <stdio.h>
int main()
{
  int JOUR, MOIS, ANNEE, RECU;
  printf("Introduisez la date (JOUR, MOIS, ANNéE) : ");
  RECU=scanf("%i %i %i", &JOUR, &MOIS, &ANNEE);
  // RECU=scanf("%i-%i-%i", &JOUR, &MOIS, &ANNEE);
  printf("\ndonnées reçues : %i\njour  : %i\nmois  :  %i\nannee : %i\n", RECU, JOUR, MOIS, ANNEE);
  return 0;
}
```

Exemple 2

Un programme qui affiche les codes ASCII des lettres et des chiffres :

```c
#include <stdio.h>
int main()
{
    char Compteur;
    int a;

  for (Compteur='A';Compteur<='Z';Compteur++)
  printf("Caractère = %c  code = %d code hexa = %x\n",Compteur,Compteur,Compteur);

  for (Compteur='0';Compteur<='9';Compteur++)
  printf("Caractère = %c  code = %d code hexa = %x\n",Compteur,Compteur,Compteur);

    for (a=0;a<=255;a++)
  printf("Caractère = %c  code = %d code hexa = %x\n",a,a,a);
}
```

Exemple 3

Instruction `break` dans une boucle `for`.

— Tester le programme et conclure.

```
#include <stdio.h>

int main()
{
    int i;
    for (i = 0; i < 5; i++)
    {
        printf("i = %d\n",i);
        if (i == 3)
            break;
    }
    printf("valeur de i à la sortie de la boucle = %d\n",i);
    return 0;
}
```

Exemple 4

Instruction `continue` dans une boucle `for`.

— Tester le programme et conclure.

```
#include <stdio.h>

int main()
{
    int i;
    for (i = 0; i < 5; i++)
    {
        if (i == 3)
            continue;
        printf("i = %d\n",i);
    }
    printf("valeur de i à la sortie de la boucle = %d\n",i);
    return 0;
}
```

Exemple 5

— Tester le programme et conclure.

```
#include <stdio.h>
#define M 2
#define N 3

int tab[M][N] = {{1, 2, 3}, {4, 5, 6}};

int main()
{
    int i, j;
    for (i = 0 ; i < M; i++)
    {
        for (j = 0; j < N; j++)
            printf("tab[%d][%d]=%d\n",i,j,tab[i][j]);
    }
    return 0;
}
```

Exemple 6

— Tester le programme et conclure.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
```

```
    int i, j;
    char c;
    for(i=0, j=0;(c=getchar()!='-');i++)
    /* EOF est un caractère prédéfini marquant la */
    {
        if((c==' ')|| (c=='\t')|| (c=='\n')) continue;
        j++;
    }
    return(EXIT_SUCCESS);
}
```

Exemple 7

— Tester le programme et conclure.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int i, j;
    char c;
    for(i=0,j=0;(c=getchar()!='*');i++)
    {
        if(c=='\n') break;
        if(c==' ') continue;
        if(c=='\t') continue;
        j++;
    }
    return(EXIT_SUCCESS);
}
```

Exemple 8

— Tester le programme et conclure.

```
#include <stdio.h>
int main()
{
 /* Déclarations */
 char CH1[50], CH2[50]; /* chaînes */

 /* Saisie des données */
 printf("Entrez la première chaîne : ");
 gets(CH1);
 printf("Entrez la deuxième chaîne : ");
 gets(CH2);
 printf("%s\n", CH1);
 printf("%s\n", CH2);
 return 0;
}
```

Exemple 9

— Tester le programme et conclure.

```
#include <stdio.h>
int main()
{
    int i,j;
    for(i=1, j=20; i < j; i++, j-=5)
    {
        printf("%d \t %d \n",i,j);
    }
 return 0;
}
```

8. Exercices

Exercice 1

Ecrire un programme qui demande à l'utilisateur de taper un réel et qui affiche "Mention Bien" si le réel est entre 14 et 17 bornes incluses, "Sans Mention" sinon.

Exercice 2

Ecrire un programme qui calcule pour une valeur X donnée de type float, la valeur numérique d'un polynôme de degré n :
$P(X) = A_n X^n + A_{n-1} X^{n-1} + \cdots + A_1 X + A_0$
Les valeurs des coefficients $A_n, \ldots, A_0$ seront entrées au clavier et mémorisées dans un tableau A de type float et de dimension $n+1$ (utiliser la fonction `pow()` pour le calcul).

Exercice 3

Ecrire un programme qui transfère un tableau M à deux dimensions L et C (dimensions maximales : 10 lignes et 10 colonnes) dans un tableau V à une dimension $L * C$.

Exercice 4

Ecrire un programme qui effectue la multiplication de deux matrices A et B. Le résultat de la multiplication sera sauvegarder dans une matrice C qui sera ensuite affichée.

Exercice 5

- Calculer la racine carrée X d'un nombre réel positif A par approximations successives en utilisant la relation de récurrence suivante : $X_{J+1} = \dfrac{\left(X_J + \dfrac{A}{X_J}\right)}{2}$ et $X_1 = A$.
  La précision du calcul J est à entrer par l'utilisateur.
- S'assurer lors de l'introduction des données que la valeur pour A est un réel positif et que J est un entier naturel positif (max. 50).
- Afficher toutes les approximations calculées :
  - La 1ère approximation de la racine carrée de ...est ...
  - La 2ème approximation de la racine carrée de ...est ...
  - La 3ème approximation de la racine carrée de ...est ...
  - ...

Exercice 6

Ecrire un programme qui lit un texte TXT (max. 200 caractères) et qui enlève toutes les apparitions du caractère 'e' en tassant (comprimant) les éléments restants. Les modifications se feront dans la même variable TXT.

Exercice 7

Ecrire un programme qui à partir d'une somme d'argent lue au clavier, donne le nombre minimal de pièces de 2 euros, de 1 euro, de 50 centimes, 20 centimes, 10 centimes, 5 centimes, 2 centimes et 1 centime qui compose cette somme.

Exercice 8

Ecrire un programme qui affiche la valeur du n-ième bit d'une valeur entière ; n et cette valeur sont saisies au clavier

Exercice 9

Ecrire un programme qui affiche en hexadécimal la valeur des 4 octets composant une valeur entière, saisie au clavier.

Exercice 10

Ecrire un programme qui demande à l'utilisateur d'entrer un premier nombre, de choisir l'opération à réaliser ('+' pour addition, '-' pour la soustraction, . . .), puis d'entrer le deuxième nombre et d'afficher le résultat de l'opération. Le demande à l'utilisateur s'il veut faire une autre opération. Cette fois-ci, le programme doit être réalisé en le découpant en fonctions.
Ecrire les fonctions suivantes :

- saisir_operande : elle demande de saisir un opérande (un nombre flottant), elle effectue la saisie et elle retourne la valeur saisie
- saisir_operateur : elle demande de saisir un opérateur (un caractère), elle effectue la saisie et elle retourne l'opérateur saisi
- afficher_resultat : elle effectue le calcul et elle affiche le résultat ; elle a en paramètres les deux opérandes et l'opérateur ; elle affiche un message d'erreur dans le cas de la division par zéro ou si l'opérateur est inconnu
- continuer : elle demande si on veut faire une nouvelle opération (o ou O pour oui), elle saisit la réponse et elle retourne 1 si oui et 0 si non

Ecrire le programme principal qui réalise la calculette en utilisant ces fonctions.