Langage C
IN4R11

I. Syntaxe du langage.

Pourquoi le langage C ?

Langage impératif, déclaratif :

Très diffusé : ressources importantes.

Utilisé depuis longtemps : fiable, documenté, normalisé

Portable : nombreux compilateurs

Sert de base à : JAVA, C++, C#

3

Historique du langage C :

Inspiré par les langages :

BCPL (Basic Combined Programming Language) 1967 (M.Richards)

B (amélioration de BCPL) 1970 par K.Thompson 1970

le langage B était fruste et non portable

1971 : problème de B avec le PDP-11

1972 : première version de C écrite en assembleur (B. Kernighan/D. Ritchie)

1973 : écriture d'un compilateur C portable (A. Snyder)

4

Historique du langage C :

1975 : écriture de PCC (Portable C Compiler)

1987 : début de normalisation par l'IEEE

1989 : norme ANSI X3-159 (d'où le nom C ANSI)

depuis : apparitions de langages basés sur le C, souple et puissant

5

Présentation du langage C

Langage de bas-niveau : accès à des données que manipulent les ordinateurs

Conçu pour écrire des programmes de système d'exploitation : 90% du noyau UNIX est écrit en C

Compilateur C écrit en C !

Suffisamment général pour des applications variées : scientifiques, accès aux bases de données

Premier langage permettant la création de modules

6

I. Syntaxe du langage C

Un langage manipule des données :

le langage C a des données dites typées :

les valeurs manipulées sont physiquement des 0 et des 1, mais elles sont regroupées : significations différentes

- nombres entiers
- nombres à virgule
- caractères

7

I.1 Les types de base (1)

Le type **char** :

codé sur 1 octet, 255 valeurs différentes

intervalle : de -128 à +127 ou de 0 à 255

utilisé pour les caractères ou pour les 'petits' nombres entiers

le type **int** :

codage variable en taille, généralement 4 octets

intervalle : de -2^31 à 2^31-1

utilisé pour les nombres entiers

8

I.1 Les types de base (2)

Le type **float** : représentation de nombres en virgule flottante (dits nombres réels mais limitations !)

codé sur 4 octets selon la norme IEEE-754

forme : mantisse, exposant (notation scientifique)

intervalle : de -10^{38} à -10^{-38} et de 10^{-38} à 10^{38}

le type **double** : extension du type **float**, codé sur 8 octets

intervalle : de -10^{308} à -10^{-308} et de 10^{-308} à 10^{308}

pour ces types, le signe est compris dans la représentation

9

I.1 Les types de base (3)

Les modificateurs : à mettre devant le type

signed : types *int* et *char* : indique que les valeurs peuvent être positives ou négatives : cas par défaut

unsigned : types *int* et *char* : valeurs positives seulement

short : type *int* : force le stockage sur 2 octets

long : type *int* : force le stockage sur 4 octets

type *double* : stockage sur 10 octets (précision étendue)

10

I.1 Les types de base (4)

Les types entiers : résumé

nom du type	taille en octets	intervalle
(signed) char	1	[-128;+127]
unsigned char	1	[0;255]
(signed) short int	2	[-32768;+32767]
unsigned short int	2	[0;65535]
(signed) int	2 ou 4, selon la machine	
unsigned int	2 ou 4, selon la machine	
(signed) long int	4	[≈–2.10^6; ≈ 2.10^6]
unsigned long int	4	[0; ≈+4.10^6]

11

I.1 Les types de base (5)

Les types flottants : résumé

Nom du type	taille en octets	intervalle
float	4	[-10^{38};+10^{38}]
double	8	[-10^{308};+10^{308}]
long double	10	[-10^{4932};+10^{4932}]

Les valeurs maximales et minimales des types entiers sont,en général, définies dans la librairie standard `limits.h`.
La fonction `sizeof(`*`expression`*`)` : retourne la taille en octets pour stocker *`expression`*.

12

I.1 Les types de base (5)

Le type **void** : type particulier, signifiant 'rien'.

Utile pour écrire les procédures (vu en détail plus tard).

13

I.2 Syntaxe et premiers pas (1)

Mots réservés : ont une signification pour le langage

- certaines instructions
- noms et extensions de type (déjà abordés)

Autres noms utilisés :

- identificateurs (noms des variables et fonctions, vus plus tard)
- constantes

ne pas utiliser les mots réservés pour les identificateurs !

14

I.2 Syntaxe et premiers pas (2)

Variables :

Cases ou cellules contenant une **valeur**

accessibles par leur **nom**

contiennent une valeur d'un certain **type**

physiquement situées dans la mémoire vive (RAM) de l'ordinateur, à un certain emplacement appelée **adresse**

on peut les : lire (consulter)

écrire (modifier)

15

I.2 Syntaxe et premiers pas (2)

Déclaration de variable

Toute valeur susceptible d'être modifiée (de varier) doit être stockée dans une variable.

Pour manipuler une variable, on est obligé de préciser ce qu'elle est pour que le langage puisse la traiter correctement.

C'est le rôle d'une **déclaration de variable** :

présentation des éléments indispensables au compilateur :

- le nom de la variable
- le type de la variable

•les variables ne sont pas initialisée par défaut à la déclaration => valeur aléatoire si utilisation avant initialisation.

16

I.2 Syntaxe et premiers pas (3)

- **Déclaration d'une variable**
  - **nom:**
    - Unique pour chaque variable
    - Commence toujours par une lettre
    - Différenciation minuscule-majuscule
  - **type**:
    - Conditionne le format de la variable en mémoire
    - Peut être soit un type standard ou un type utilisateur
  - **valeur:**
    - Peut évoluer pendant l'exécution
    - initialisation grâce à l'opérateur d'affectation

17

I.3 Instructions(1)

Instructions simples

Ordre donné à l'ordinateur et qui provoque une action : affichage, saisie, ou calcul ..

une instruction peut être un mot réservé (mais pas forcément)

au niveau syntaxique :

une ligne ne doit comporter qu'une instruction (lisibilité)

une instruction doit toujours être suivie d'un point-virgule ;

non-respect de cette règle : ERREUR DE SYNTAXE

18

I.3 Instructions(2)

Instruction composée .

C'est un **ensemble de plusieurs instructions** (chacune d'elles est simple ou composée)

au niveau syntaxique :

une instruction composée est encadrée par des accolades { }

chacune des instructions qui la composent doit être correcte au niveau syntaxique

on appelle aussi une instruction composée un **bloc d'instructions**

19

I.4 Expressions(1)

Une expression est une valeur qui peut être calculée par l'ordinateur, qui contient des termes et des opérateurs reliés par une syntaxe précise.

Plus simplement : une suite de symboles qui est comprise par le langage et qui donne un résultat.

Déjà connu : les expressions mathématiques qui donnent une valeur

1+2×4-(5/6) est une expression mathématique valide et calculable, qui vaut : 49/6

!21+ × ×2 n'est pas une expression, bien que constituée de symboles tous connus : ne respecte pas une syntaxe donnée.

20

I.4 Expressions(2)

En C même principe : règles de syntaxe et de priorités pour calculer la valeur d'une expression.

De plus, une expression a un type : sa valeur calculée sera utilisée par le langage, donc doit avoir un type.

Termes possibles dans une expression :

les variables (puisqu'elles ont un type et une valeur),

des constantes, appels de fonctions (vu plus tard)

Opérateurs : nous en verrons tout un ensemble.

21

I.4 Expressions(3)

Constantes entières : nombre sans virgule, précédé ou non du signe -

exemples : 3 0 824 -12000

constantes caractère : les caractères sont des valeurs entières, on peut leur attribuer une valeur comme pour les entiers ci-dessus, mais aussi directement un caractère en l'encadrant de simples guillemets : la valeur sera alors le code ASCII du caractère

exemples : 'a' 23 'H' ';'

constantes à virgule : un nombre écrit sous la forme classique ou scientifique (avec exposant) : partie entière.partie décimaleEexposant

exemples : 3.14 0.0001 1.27E+32 65634.23476432E-5

22

I.4 Expressions(4)

Opérateurs

Opérateurs:

Opérateurs mathématiques et priorités : +, -, *, /

* et / prioritaires sur + et -

modulo : %

a%*b* donne le reste de la division entière de *a* par *b*.

ajout de parenthèses possible pour gérer lever les ambiguïtés dans les expressions.

23

I.4 Expressions(5)

affectation

Opérateur =

cet opérateur a un rôle particulier.

À sa **gauche**, on doit trouver **une variable obligatoirement** (et pas une expression complexe)

À sa **droite**, on trouve une **expression**.

Rôle : calcule l'expression à droite de l'opérateur = et donne cette valeur à la variable à gauche de cet opérateur.

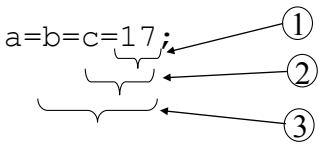
Peut sembler évident, mais attention !

24

I.4 Expressions(6)

affectation

Expression calculée (ou évaluée) de droite à gauche :



étape①: évaluation de 17, rangement dans la variable c;
étape② : c=17 est une expression qui vaut 17, rangée dans b par : b=c=17;
étape③ : b=c=17 est une expression qui vaut 17, rangée dans a par a=b=c=17;
étape 4 : a=b=c=17 est une expression qui vaut 17, qui n'est pas rangée.

25

I.4 Expressions(7)

combinaison d'opérateurs

Combinaison d'opérateurs mathématiques et d'affectation :

permet d'appliquer un opérateur mathématique à une variable puis d'affecter le résultat de cette opération à la variable.

Opérateurs :
addition et affectation : +=
soustraction et affectation : -=
multiplication et affectation : *=
division et affectation: /=

emploi par l'exemple :

x += 4; signifie x=x+4;

e *= d+5.1; signifie e=e*(d+5.1);

y /= x; signifie y =y/x;

et ainsi de suite

26

I.4 Expressions(8)

opérateurs binaires

Opérateurs logiques (binaires) : permettent de manipuler des entiers au niveau du bit.

Ils s'appliquent aux entiers de toute longueur: short, int ou long, signés ou non.

&	ET logique	|	ou inclusif
^	OU exclusif	~	complément à 1
<<	décalage à gauche	>>	décalage à droite

27

I.4 Expressions(9)

Conversion implicite de type

Les expression peuvent contenir des termes qui sont de type différents : le **compilateur** accepte ces expressions mais effectue des conversions de type pour les valeurs.

Cette opération s'appelle **cast** ou **transtypage**.
Exemple :

```
int a,b;
double x;
a=5;
x=3.1415926;
b=a+x; /* l'expression est affectee a un int ! */

warning C4244: '=' : conversion from 'double ' to 'int ',
possible loss of data
```

28

I.4 Expressions(10)

Conversion explicite de type

Solution : effectuer soit même les transtypages pour être certain du type des valeurs !

Pour transtyper une expression d'un type en un autre, il suffit de noter entre parenthèses le type souhaité devant l'expression :

(type_nouveau) expression.

Soit i un entier : **(double)i** donne la même valeur mais en type double.

soit x un double : **(int)x** convertira x en int, en supprimant les chiffres après la virgule.

29

I.5 Tests

exemple simple : saisie en contrôlant la valeur entrée

une valeur à saisir doit absolument être supérieure à une borne *b_inf*.

algorithme :

saisir la valeur

tant que la valeur est < *b_inf*, recommencer la saisie.

30

I.5 Tests
Conditions

`if` sert à réaliser un test selon une **condition logique**

cette condition sera évaluée et aura une valeur de vérité : ***vraie*** ou ***fausse***. C'est selon cette valeur, vraie ou fausse, que l'on peut choisir de faire certaines instructions ou non.

Utilisation d'opérateurs de comparaison :

2 valeurs sont elles-égales ? ⇒ réponse de type vrai ou faux

1 valeur est-elle supérieure (au sens strict ou au sens large) à une autre ? ⇒ réponse de type vrai ou faux

31

I.5 Tests
Conditions

Opérateurs de comparaison :

Syntaxe	emploi	vrai si
==	val1 == val2	val1 est égal à val2
!=	val1 != val2	val1 est différent de val2
<	val1 < val2	val1 strictement inférieur à val2
<=	val1 <= val2	val1 inférieur ou égal à val2
>	val1 > val2	val1 strictement supérieur à val2
>=	val1 >= val2	val1 supérieur ou égal à val2

Attention à l'opérateur d'égalité ==, il ne faut pas le confondre avec l'opérateur d'affectation =

32

I.5 Tests

Opérateurs Logiques

Tables de vérité des opérateurs logiques. Soient A et B deux conditions, qui peuvent prendre les valeurs VRAI ou FAUX.

- L'expression A ET B a pour valeur VRAI ssi A et B ont pour valeur VRAI en même temps, sinon A ET B vaut FAUX.
- L'expression A OU B a pour valeur VRAI si A est VRAI ou si B est VRAI.
- L'expression NON A a pour valeur VRAI si A a pour valeur FAUX, et vaut FAUX si A a pour valeur VRAI.

Traduction en C :

ET : &&
OU : || (pour obtenir 2 barres verticales, AltGr 6 sur le clavier)
NON : !

33

I.5 Tests

Opérateurs Logiques

Remarque sur le parenthèsage des expressions logiques :

&& est prioritaire sur **||.**

Il faut systématiquement utiliser des parenthèses (les plus prioritaires) pour construire les expressions.

Ne pas avoir à connaître par cœur les priorités des opérateurs;

Rendre les expressions lisibles.

Mettre des parenthèses autour de chaque condition.

34

I.5 Tests

Syntaxe de l'instruction if

Réaliser un test simple : si une condition est vraie alors faire une instruction.

```
if (condition) instruction; /* cas d'une instruction
simple */
```

ou

```
if (condition) { instructions } /* cas d'un bloc
d'instruction */
```

si la condition est fausse, l'instruction (ou le bloc) ne sera pas effectuée.

35

I.5 Tests

L'alternative : if …..else

Syntaxe :

```
if (condition)
{
      instructions; /* si la condition est vraie */
}
else
{
      autres instructions; /* si elle est fausse */
}
```

36

I.5 Tests

Alternative multiple

On peut enchaîner les if…else if …else au lieu de les imbriquer

```
if (delta < 0.)
{
       /* traitement du cas sans solutions */
}
else if (delta > 0.)
{
       /* traitement du cas à 2 solutions */
}
else
{
              /* traitement du cas à une solution */
}
```

37

I.5 Tests

Instruction de sélection: switch

Autre instruction pour agir selon la valeur d'une expression entière (de type *char* ou *int*) : **switch** … **case**.

Selon la valeur de l'expression, on liste les actions à faire : presque équivalent à une suite de **if…else if…else if**.

Syntaxe :

```
switch(expression)
       {
              case valeur_1 : instructions;

              case valeur_2 : instructions;

              default : instructions;
       }
```

38

I.5 Tests
Instruction de sélection: switch

À la suite de chaque **case** : ensemble d'instructions sans la notation utilisée pour les blocs {}.

Le mot réservé **default** indique les instructions effectuées lorsque la valeur de l'expression ne correspond à aucune valeur listée par les **case**.

Attention aux comportement : quand un **case** est effectué, tous les **case** suivants ainsi que le **default** seront aussi effectués !

Il faut alors utiliser **break**, instruction de rupture de séquence, pour sortir du **switch**.

39

I.5 Tests
Instruction de sélection: switch

Exemple 1 : utilisation de **break**

```
switch(carac1+carac2)
{
        case 1 :  printf("message numéro 1\n");
                  break;

        case 2 : printf("message numéro 2\n");
                  break;

        case 5 : printf("message numéro 5");
                 break;

        default : printf("message par défaut\n");
                 break;
}
```

*Rupture de séquence : sortie du **switch***

Résultat : affichage du message 1 seul.

40

I.6 Les itérations

boucle while ( )

syntaxe :

while (condition) instruction;

ou

while (condition) {bloc d'instruction}

La condition est une condition de répétition : tant que cette condition sera vraie, on continuera la boucle.

on teste la condition puis exécute les instructions, puis. Si elle est vraie, on recommence les instructions, sinon, on sort de la boucle.

41

I.6 Les itérations

do …… while ( );

syntaxe :

do instruction; **while** (condition);

ou

do {bloc d'instruction} **while** (condition);

comme pour le while, la condition est une condition de répétition : tant que cette condition sera vraie, on continuera la boucle.

Différence : l'instruction ou le bloc d'instructions est situé avant le test de la condition, et cet ordre est respecté lors de l'exécution du programme :

on fait les instructions, puis on teste la condition. Si elle est vraie, on recommence les instructions, sinon, on sort de la boucle.

42

I.6 Les itérations

do …… while ( );

Avec **do…while**, on fait au moins une fois les instructions du corps de la boucle

Avec **while**, on peut ne pas faire les instructions de la boucle, si la condition est fausse dès le premier test !.

Exemple d'applications : saisie sécurisée d'une valeur.

Pour tester si une valeur est correcte, il fait d'abord la saisir !

Si cette valeur est incorrecte, on la ressaisira : on utilise alors une boucle do…while, puisqu'il faut faire les instructions (ici la saisie) AVANT de faire le test de la condition !

43

I.6 Les itérations

boucle for

Dernier type de boucle : la boucle for, qui sert essentiellement à répéter une ou des instructions un nombre de fois déterminé.

Conceptuellement très proche de la boucle while.

La boucle **for** intègre naturellement la notion de compteur : une variable est utilisée pour compter le nombre de fois où cette boucle doit être effectuée. C'est, de préférence, une variable entière.

Présentation simple (simpliste) de la boucle for dans un premier temps, uniquement avec un compteur entier.

En fait beaucoup plus puissante.

44

I.6 Les itérations

boucle for

La plupart du temps, on considère que la boucle for est utilisée ainsi :

• **for(initialisations du compteur; condition ; incrémentation du compteur) {bloc d'instructions}**

En C le **;** indique une instruction vide !

```
for(;;)
 {
 ....
 }
```

Boucle infinie!

45

I.7 Entrés-Sorties

***printf* .**

- Les affichages : utilisation de ***printf***

printf permet de réaliser un affichage à l'écran.

Capable d'afficher les valeurs des variables (pas leur nom !)

syntaxe générale :

***printf*(" chaine de ctrl ", arg_1, arg_2, ….., arg_n);**

arg_i : nom de la variable i

46

I.7 Entrés-Sorties

printf .

" chaine de ctrl " :

C'est un texte contenant des mots ou caractères à afficher : caractères normaux (lettres) et spéciaux (mise en page)

basé sur le codage ASCII : codes pour quelques caractères spéciaux

\n : passage à la ligne

\a : signal d'alerte bip

\t :tabulation horizontale

\v : tabulation verticale

**** : caractère d'échappement (affiche l'antislash \)

\" : caractère d'échappement (affiche les guillemets ")

\b : retour arrière (BS = BackSpace)

un texte doit se trouver entre double guillemets **" "**

47

I.7 Entrés-Sorties

printf .

Utiliser un code de format dans la chaîne pour préciser que l'on veut afficher une valeur, le type de la valeur est précisé.

Code	type de la valeur à afficher
%d	entier signé
%c	caractère
%u	entier non signé
%e	nombre à virgule - exposant
%f	nombre à virgule float
%lf	nombre à virgule double
%g	nombre à virgule : meilleure
%s	chaîne de caractères

48

I.7 Entrés-Sorties

scanf .

Saisie de valeurs : emploi de ***scanf***

comme printf, scanf est une fonction qui permet de faire des saisies, en utilisant aussi des formats pour interpréter ce qui est saisi.

Emploi légèrement différent : pas de texte, seulement une chaîne avec des formats.

***scanf*(" chaine de ctrl ", arg_1);**

arg_1 : nom de la variable
saisie d'une valeur ⇔ un scanf

possibilité de saisir plusieurs valeurs dans un scanf : emploi pas évident, source de confusion.

49

I.7 Entrés-Sorties

scanf .

Saisie de valeurs : emploi de ***scanf***

scanf s'emploie de la manière suivante :

scanf(chaine_de_format, adresse_de_la_variable);

différent de printf : variables, ici adresses.

Faire précéder la variable dont on veut saisir la valeur de l'opérateur **&**.

Les codes pour les formats sont les mêmes que pour printf (%d, %u, %f,…)

50

I.7 Entrés-Sorties

Exemples de saisies avec scanf : différents types

```
#include <stdio.h>

void main()
{
       unsigned int monAbsolue;
       char unSeul;
       double nbDecimal;
       int x;

       /* 4 saisies séparées pour les 4 valeurs */

       scanf("%u",&monAbsolue);
       scanf("%c",&unSeul);
       scanf("%lf",&nbDecimal);
       scanf("%d",&x);

       printf("%u %c %lf %d\n", monAbsolue, unSeul,
       nbDecimal, x);
}
```

51

I.7 Fonctions

Dans un programme : certaines opérations (ou séquences d'opérations) peuvent se répéter plusieurs fois : affichage de tableau, saisie, ou même calculs (cosinus () qui peutre être appelée plusieurs fois dans un programme

Notion de **sous-programmes** : parties de programme que l'on peut utiliser quand on en a besoin en les **appelant**. Ce sont des **fonctions**.

52

I.7 Fonctions.

Une fonction est un regroupement d'instructions qui effectue une certaine tâche.

Une fonction à une syntaxe et un mode de fonctionnement particuliers.

Elle a ses propres variables (ce sont des variables dites 'locales'); qui n'existent que pour la fonction.

En fait, un programme est composé de :

un programme dit principal, là ou débutent les instructions;

un ou des sous-programmes ou fonctions, qui seront appelées (mises en œuvre) par le programme principal ou par d'autres fonctions.

53

I.7 Fonctions.

Utilisation et écriture

Le **prototype** d'une fonction est l'ensemble des informations permettant de savoir **comment utiliser (appeler)** une fonction.

Un prototype ne décrit pas comment la fonction transforme les entrées en sorties, et n'est pas un appel à la fonction.

Syntaxe en C : un prototype s'écrit de la manière suivante

type_retourné nom(type entrée$_1$, type entrée$_2$,…,type entrée$_n$**);**

54

I.7 Fonctions

Suite de l'exemple mathématique :

voici le prototype de la fonction de calcul

```
float       fonc_calc(int, char );
```

on le lit de la manière suivante :

la fonction dont le **nom** est *fonc_calc* a besoin de deux valeurs:

- la 1ère de type **int**,
- la 2ème de type char

et elle donne une valeur de type ***float*** en **retour**.

un **prototype** est aussi appelé **déclaration** de fonction.

55

I.8 Fonctions

Après le mode d'emploi, il faut maintenant écrire les instructions qui composent la fonction : comment, à partir des entrées, obtenir la sortie ou valeur de retour.

Il s'agit de la **définition** de la fonction. On doit rappeler les valeurs d'entrée, le nom et le type de la valeur de retour. La syntaxe d'une définition de fonction est la suivante :

```
type_de_retour  nom(type_entrée_1  nom_entrée1, type_entrée_2  nom_entrée2,…)
{
        déclarations des variables utilisées par la fonction;
        instructions de la fonction;
        instruction_de_retour;
}
```

Deux parties : entête (première ligne) et corps (le reste) 56

I.8 Fonctions

Instruction de retour :

la valeur est calculée dans la fonction, il faut indiquer qu'il s'agit de la valeur de retour :

instruction **return**

indique que la fonction est terminée et indique la valeur de retour.

Syntaxe : return valeur_de_retour;

On indique le type de la valeur de retour dans le prototype et l'entête : la valeur retournée doit être du même type.

57

I.8 Fonctions

Le prototype :

```
float moyenne(char, char, char);
```

et la définition :

```
float moyenne(char a, char b, char c)
{
       float resultat;
       resultat=(float)(a+b+c)/3.0;
       return(resultat);
}
```

ou encore :

```
float moyenne(char a, char b, char c)
{
       return((float)(a+b+c)/3.0);
}
```

58

I.8 Fonctions
Procédures

Certaines fonctions n'ont pas de pas de valeur de retour, sont aussi appelées **procédures**.

On utilise alors le type **void** (rien) pour la valeur de retour. Il faudra tout de même utiliser `return;` pour terminer la fonction.

59

I.8 Fonctions
Appels

Le programme appelant une fonction doit juste savoir comment utiliser la fonction, et non pas comment elle fonctionne : doit connaître le prototype de la fonction à appeler.

Le **prototype** de la fonction à appeler doit être situé **avant** le programme appelant.
La **définition** par contre, peut être située avant ou après, c'est le compilateur qui fera le lien.

L'appel d'une fonction est une demande pour utiliser la fonction: on doit donner des valeurs aux entrées pour que la fonction les traite, et on récupère la valeur de retour de la fonction.

60

I.8 Fonctions

Appels

Un appel de fonction est traité comme une instruction dans le programme appelant.

Syntaxe d'un appel de fonction :
soit le prototype suivant::

type_retour nom_fonction(type param$_1$, type param$_2$, ... type param$_2$);

type variable = **nom_fonction(**valeur arg$_1$,valeur arg$_2$,.valeur arg$_n$**)**;

2 effets : récupération de la valeur de retour; appel à la fonction.

61

I.8 Fonctions

Posistionnement

Où positionner les prototypes, appels et définitions dans un programme ?
Voici la structure type d'un programme utilisant des fonctions :

inclusions de fichiers d'entête (comme stdio.h)
définition des constantes par #define
} *(vu plus tard)*

prototypes de fonctions

programme principal (***main*** contient les *appels de fonctions*)

définition des fonctions

62

I.8 Fonctions

Positionnement

Exemple avec un programme utilisant un calcul de moyenne de 3 valeurs :

```
#include <stdio.h>
float moyenne(char, char, char);  <-------------  prototype

void main()
{
        char x1, x2, x3;
        float moy;

        /* saisie des valeurs de x1, x2, x3 */

        moy = moyenne(x1, x2, x3);  <-------------  appel

        printf("la moyenne est : %f\n",moy);
}
```

63

II. Retour sur les Variables

Classe d'une variable

- **La classe d'une variable peut être :**
  - **auto**
  - **register**
  - **static**
  - **extern**

64

II. Retour sur les Variables
Classe d'une variable

1. Classe automatic ou auto

- **La classe** auto (par défaut) : la variable est créée à l'entrée du bloc (dans la pile) et libérée automatiquement à sa sortie

C'est une variable temporaire

65

II. Retour sur les Variables
Classe d'une variable

2. Classe register

- **register :** la variable est créée, possède la même durée de vie et visibilité qu'une classe auto, mais sera placée dans un registre du processeur.

variable temporaire

66

II. Retour sur les Variables
Classe d'une variable

3. Classe static

variable stockée dans la même zone que le code machine du programme.

Sa durée de vie= durée de vie du programme

Elle est locale du bloc ou elle est déclarée.

variable permanente

67

II. Retour sur les Variables
Classe d'une variable

4. Classe extern

- Une variable précédée du mot-clé **extern** siginifie que cette variable est définie dans un autre fichier.
- Une variable commune à plusieurs fichiers devra donc être déclarée:
  - de manière classique dans le fichier où elle est définie
  - et **extern** dans les autres.

68

III Structure d'un programme C

69

1. Le C : un langage compilé

L'ordinateur (le micro-processeur) ne comprend que le langage machine, constitué d'une suite de 0 et de 1.

Traduction des ordres donnés dans d'autres langages (PASCAL, C, Basic,...)

traduction au fur et à mesure : **interprétation**

traduction de tout un programme : **compilation**

compilateur : traduit un langage de programmation en langage machine

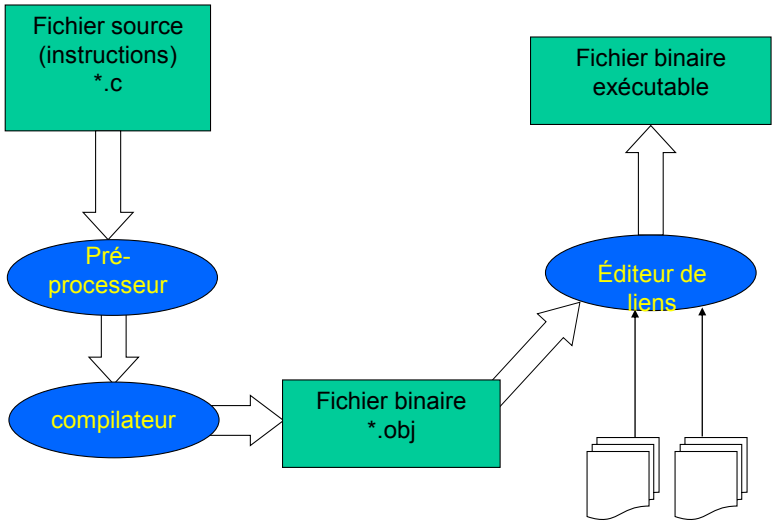
Principe de la compilation

Qu'est ce qu'un programme C?

C'est un texte écrit avec un éditeur de texte, respectant une certaine syntaxe et stocké sous forme d'un ou plusieurs fichiers (généralement avec l'extension .c).

Les instructions du langage C sont obligatoirement encapsulées dans des fonctions

Etapes d'exécution d'un programme



Etapes d'exécution d'un programme

Étapes à réaliser :

- écrire le fichier .c (environnement dédié) → **fichier .c**
- exécuter le compilateur → **fichier .o**
- si tout se passe bien, exécuter l'édition des liens → **fichier .exe**
- si tout se passe bien, on obtient un fichier exécutable

plusieurs sources d'erreur possibles :

compilation/édition des liens/exécution

2. Structure d'un programme C

- Un programme C se présente de la façon suivante :
  - [*directives au préprocesseur*]
  - [*déclarations de variables externes*]
  - [*fonctions*]

    main()

    {*déclarations de variables internes*

    instructions

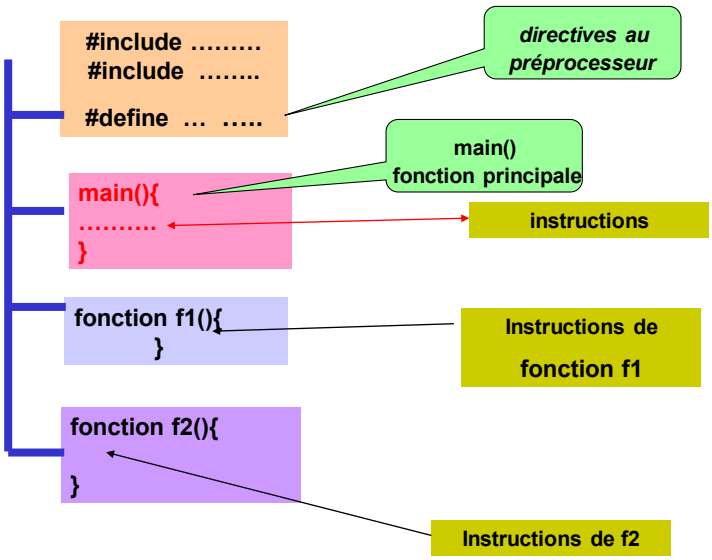
    }
  - [*fonctions*]

Ce qui est entre [..] est optionnel

Ce qui est en rouge est obligatoire

74

2.1 Structure Programme C



75

2.2 Fonctions

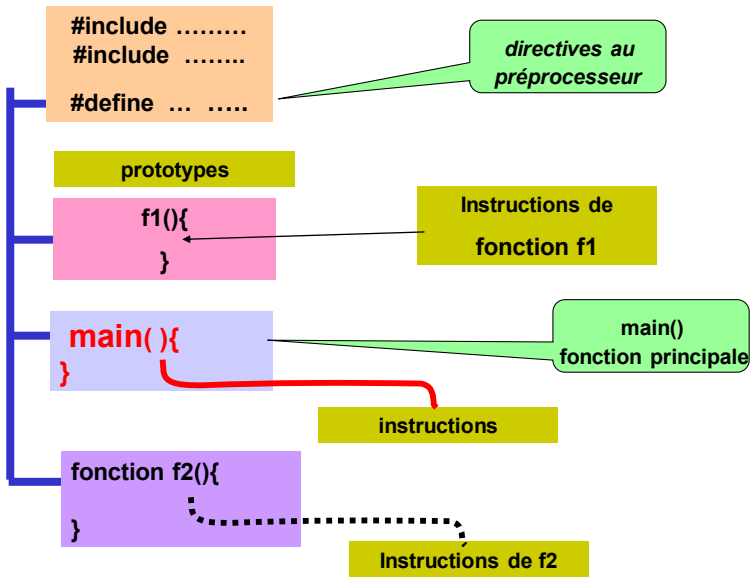
- Fonctions (procédures)

type ma_fonction (*paramètres ...*)
{*déclarations de variables internes*
instructions
return ..
}

Les fonctions peuvent être placées indifféremment avant ou après la fonction ***main***.

76

Structure Programme C



77

2.3 Pré-processeur

- Le pré-processeur (ou pré-compilateur) 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, comment par le caractère : #

78

2.3 Pré-processeur

- **Rôle des *directives* au préprocesseur:**
- **#include : insertion de fichiers**
- **#define : la définition de constantes et de macros**
- **#if, #ifdef ..: la compilation conditionnelle**

79

2.3.1 #include

- **#include** <nom_fichier.h>

  Cette directive permet d'insérer un fichier, qui sera cherché dans bibliothèque du C

- **#include "nomfichier"**

  permet d'insérer à cet endroit un fichier, comme s'il était écrit ici, en le cherchant dans le répertoire où se trouve le fichier source

80

2.3.1 #include

Exemple:

```
# include <stdio.h>
# include <string.h>
# include  <math.h>
# include "monpgm.h"

 int main(){
 printf("Apprentis GO! ");
return 0;
}
```

bibliothèque C

Programme utilisateur

81

2.3.1 #include

Exemple:

```
 # include <stdio.h>
# include ”mon-fichier-1”
# include ”mon-fichier-2”

  int main()
  {
  printf(’’Apprentis GO!’’);
  return 0;
  }
```

Fichiers de l'utilisateur

82

2.3.2 La directive #define

- **La directive #define permet de définir :**
  - **des constantes symboliques,**
  - **des macros avec paramètres.**

83

2.3.2.1 La directive #define

- **Constantes symboliques**

  Syntaxe:

  #define ***nom_variable1*** ***valeur1***

  chaque occurrence du nom_variable par la valeur valeur1.

84

2.3.2.1 La directive #define

- Constantes symboliques

  Exemples

  #define nom_variable1 valeur1

  #define PI 3.1415926

  #define begin {

  #define end }

  #define TAILLE 100

85

2.3.2.2 La directive #define

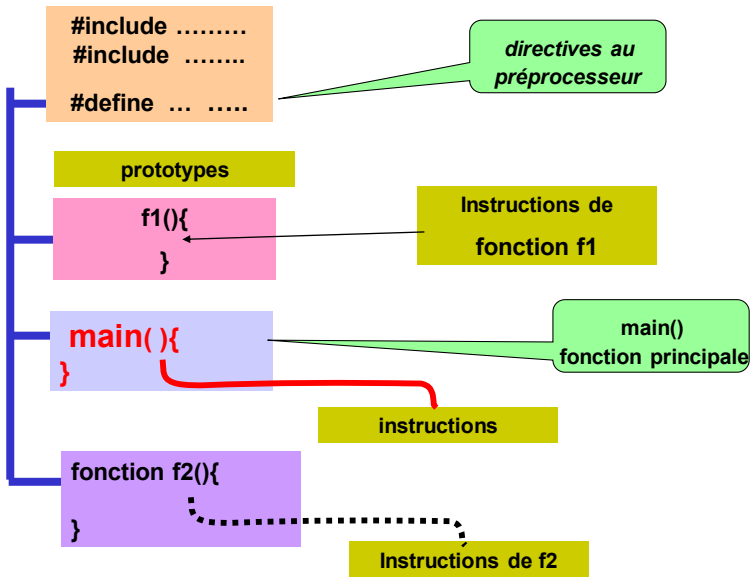
macros avec paramètres

#define nom_variable1 valeur1

chaque occurrence du nom_variable par la valeur valeur1.

86

Structure Programme C



87

Structure d 'un programme

Le makefile sous Linux

- Le programme utilisé pour exécuter des « Makefile » est le programme *make*, intégré à tous les systèmes Unix.
- Les options de la commande ***make*** sont les suivantes :
- ***–k :*** permet de continuer la compilation après la première erreur trouvée. Ce qui permet de repérer toutes les erreurs dès la première compilation.
- ***-f <fichier> :*** permet d'utiliser un fichier « makefile » à utiliser pour la compilation. Par défaut l'exécution de make recherche un fichier makefile ou Makefile.
- ***-h :*** aide sur le programme make.
- Pour ajouter une ligne de commentaire, il suffit d'utiliser le caractère «#».

88

Structure d 'un programme
Le makefile sous Linux

```
mon_programme :     principal.o outil.o outillage.o
                    gcc -o mon_programme principal.o outil.o outillage.o
principal.o   :     principal.c principal.h
                    gcc -c principal.c
outil.o       :     outil.c outil.h
                    gcc - c outil.c
outillage.o   :     outillage.c outillage.h
                    gcc -c outillage.c
```

89

Structure d 'un programme
Fichier Makefile sous Linux

- Un fichier makefile se présente comme suit :
- ***cible* : *dépendances***
- ***règles***
- Les dépendances représentent les fichiers dont on a besoin pour créer la cible.
- Les règles sont en fait l'opération de compilation pour créer la cible : ici gcc –c … ou gcc –o …

90

Exemple de Makefile

prod: produit.o main.o
gcc -o prod produit.o main.o
main.o: main.c produit.h
gcc -c main.c
produit.o: produit.c produit.h
gcc -c produit.c

Les fichiers objet (.o):
- main.o : dépend du fichier source main.c et du fichier en-tête produit.h
- produit.o : dépend du fichier source prod.c et du fichier en-tête produit.h

Le fichier executable:
- prod : depend des fichiers main.co et produit.o,

Ils sont obtenus en effectuant:

- la compilation de ces fichiers sources sans édition de liens : **gcc -c**

- Le fichier exécutable **prod** est obtenu en effectuant l'édition de liens **gcc -o** des fichiers **produit.o et main.o**.

91

II .Types structurés

69

Tableaux

Certains problèmes nécessitent beaucoup de variables du même type.

Exemple : relevé de températures matin et soir dans 10 villes pour 10 jours : 200 valeurs à stocker : déclarer 200 variables ?

Utiliser un **tableau** regroupant ces 200 valeurs : ensemble de 'cases' (de cellules mémoire) que l'on repère avec leur **numéro** ou **indice**.

Pour l'exemple précédent : une variable tableau : un groupement de 200 cases plutôt que 200 variables.

Toutes les cases ont le même type : on fera un regroupement de **char**, d'**int**, de **double**,…

un tableau est une variable : il faut la nommer.

93

Déclaration de tableaux

Syntaxe : utiliser des crochets entre lesquels on indique le nombre d'éléments (de variables) dans le regroupement après le nom de variable.

```
type    nom_du_tableau[nombre_d_éléments];
```

exemples de déclarations :

tableau nommé tab contenant 50 valeurs de type char :

```
char    tab[50];
```

tableau nommé abcd contenant 8 valeurs de type double :

```
double abcd[8];
```

tableau nommé releveTemp contenant 200 entiers longs :

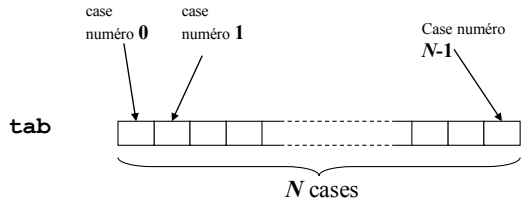
```
long int releveTemp[200];
```

94

Accès aux éléments

Accès aux éléments : par le numéro de la case (indice).

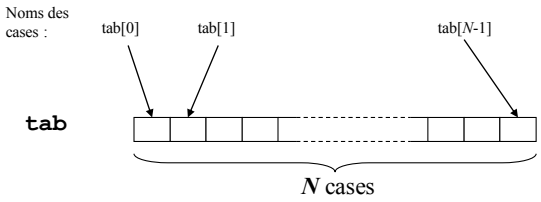
Soit *N* le nombre cases (*N* constant) : les indices vont de 0 à *N*-1



95

Accès aux éléments

Chaque case est accessible par un nom : le nom du tableau suivi de l'indice noté entre crochets. Dans une case : une variable.



96

Accès aux éléments

Exemple : remplissage de quelques cases d'un tableau et affichage

```
#include <stdio.h>

void main()
{
        char tablo[10];

        tablo[0] = 'x';
        tablo[1] = 'F';
        …
        tablo[9] = 'D';

        printf("%c\n",tablo[5]);
}
```

97

Initialisation d'un tableau

Lister les valeurs des éléments lors de la déclaration (impossible après)

syntaxe :

type nom_du_tableau**[**nb_elements**]**={ elt_0, elt_1,…,elt_i };

initialise les éléments du tableau avec les valeurs fournies. Si le nombre de valeurs fournies est inférieur au nombre d'éléments du tableau, les cases restantes sont initialisées à 0 (quel que soit le type).

98

Copie de tableaux

Exemple on aimerait copier un tableau (nommé tabS) vers un autre (nommé tabD).

Ne pas faire tabD=tabS !

On doit toujours travailler avec les éléments individuels : la copie doit séffectuer élément par éléments.

tabD[i]=tabS[i]

idem pour les comparaisons.

99

Structures

Qu'est-ce qu'une structure

rôle et déclaration de types composés

utilisation de **typedef**

variables de types composé : déclaration, emploi, pointeurs

tableaux et structures

Structures

Le mot réservé **`struct`**, permet de définir un nouveau type .

Syntaxe

struct nom_du_type_composé

{

liste des variables regroupées avec leur type;

};

il ne s'agit pas d'une déclaration de variable !

Indique juste comment est composé le nouveau ***moule***

Structures

Exemple : le type s_eleve sera déclaré de la manière suivante:

```
struct s_eleve
{
        char nom[40];
        char prenoms[25];
        unsigned int age;
        char adresse[40];
        float note;
};
```

le nouveau type s'appelle : **struct s_eleve** qui contient des **champs**.

utilisable comme n'importe quel autre type, on peut déclarer des variables de ce nouveau type et les manipuler.

Structures

Exemple de déclaration de variable dans un programme :

```
struct s_eleve
{
        char nom[40];
        char prenoms[25];
        unsigned int age;
        char adresse[40];
        float note;
};

void main()
{
        struct s_eleve  bonEleve;
        /* bonEleve est une variable de type s_eleve */

        /* suite du programme */
}
```

Structures

Emploi du mot réservé **typedef** :

contrairement à ce que l'on pourrait croire d'après le nom de ce mot réservé, **typedef** ne sert pas à définir un type. C'est le mot réservé struct qui a ce rôle.

Le type composé s'écrit ***struct nom_du_type*** :

peu pratique

pas cohérent avec les autres types simples, un seul mot :

possibilité de donner un nom plus court en utilisant un 'raccourci' analogue au #define : c'est le rôle de **typedef.**

Structures

définition de type sans typedef : on écrit :

```
struct s_complex
{
        float re;
        float im;
};
```

avec typedef :

```
typedef struct s_complex
{
        float re;
        float im;
}complex;
```

maintenant, `complex` est un nouveau type utilisable (presque) comme n'importe quel autre type. On peut continuer à écrire struct s_complex.

Structures

Manipulation des variables dont le type est composé.

Déclarer une variable d'un type composé : utilisation de plusieurs valeurs, pas d'une seule.

Chaque variable ou **champ** dans le type composé, porte un **nom** grâce auquel on peut accéder au champ.

C'est ce **nom** qui va être utile pour manipuler individuellement les champs.

Avec une variable dont le type est composé, on ne travaille qu'avec les champs individuels.

Structures

Parmi les champs que l'on trouve dans un type composé, nous avons pour l'instant vu que l'on utilisait des types de base du C :

on peut utiliser d'autres types composés, à condition qu'ils soient définis avant. Accès par l'application successive des opérateurs '.'

exemple : définit un type ***s_date*** utilisé dans un type ***s_eleve***.

```
typedef struct s_date
{
        unsigned int jour, mois, annee;
}t_date;
typedef struct s_eleve
{
        char nom[20];
        char prenom[20];
        t_date dateNaiss; /*utilisable car déjà défini*/
} t_eleve;
```

Structures imbriquées

Déclaration d'une variable de type t_eleve et initialisation de sa date de naissance :

```
void main()
{
        t_eleve eleve;

        eleve.dateNaiss.jour=5;
        eleve.dateNaiss.mois=12;
        eleve.dateNaiss.annee=1999;

        /* allocations dynamiques */
        /* pour les champs nom et */
        /* prenom de la variable  */
}
```