

Programmation impérative

Le langage C

F. Pellegrini
ENSEIRB

Ce document est copiable et distribuable librement et gratuitement à la condition expresse que son contenu ne soit modifié en aucune façon, et en particulier que le nom de son auteur et de son institution d'origine continuent à y figurer, de même que le présent texte.

© 2004-2010 F. Pellegrini

1

Pourquoi tant de langages ?

- Les ordinateurs ne comprennent que le langage machine
 - Trop élémentaire, trop long à programmer
- Nécessité de construire des programmes à un niveau d'abstraction plus élevé
 - Meilleure expressivité et généricité
 - Moins de risques d'erreurs
- Il y a autant de langages que de besoins différents et de domaines d'applications...

Historique du langage C

- Inventé aux Bell Labs / ATT en 1970
- Conçu pour être le langage de programmation d'Unix, premier système d'exploitation écrit dans un langage autre qu'un langage machine
- Diffusé grâce à Unix
- Popularisé par sa concision, son expressivité et son efficacité
- Disponible actuellement sur quasiment toutes les plate-formes

Caractéristiques (1)

- Langage impératif
 - Le programmeur spécifie explicitement l'enchaînement des instructions devant être exécutées :
 - Fais ceci, puis cela
 - Fais ceci, si cela est vrai
 - Fais ceci, tant de fois
 - Fais ceci, tant que cela est vrai
 - Famille comprenant aussi le BASIC, le FORTRAN, le Pascal, etc...

Caractéristiques (2)

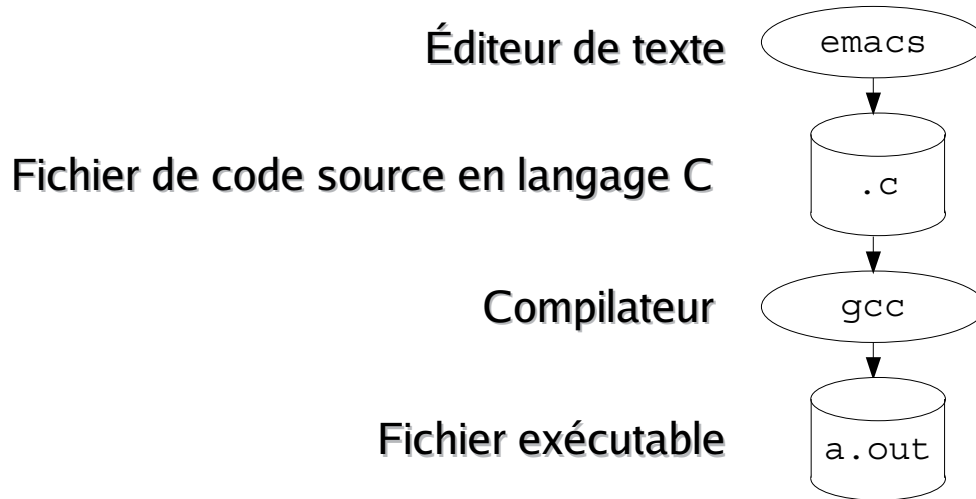
- Langage de haut niveau
 - Programmation structurée
 - Organisation des données (regroupement structurel)
 - Organisation des traitements (fonctions)
 - Possibilité de programmer « façon objet »
- Langage de bas niveau
 - Conçu pour être facilement traduit en langage machine
 - Gestion de la mémoire « à la main »
 - Pas de gestion des exceptions

Compilation (1)

- C est un langage compilé
 - Le programmeur écrit son programme sous la forme d'un code source, contenu dans un ou plusieurs fichiers texte d'extension « .c »
 - Un programme appelé compilateur (habituellement nommé `cc`, ou `gcc`) vérifie la syntaxe du code source et le traduit en code objet, compris par le processeur
 - Le programme en code objet ainsi obtenu peut alors être exécuté sur la machine

Compilation (2)

- Schéma simplifié :



Compilation (3)

- Exemple de compilation simple

```
#include <stdio.h>
main ()
{
    printf ("Bonjour!\n");
}
```

bonjour.c

```
% gcc bonjour.c
% a.out
Bonjour!
%
```

Compilation (4)

- En plus des erreurs, le compilateur peut émettre des avertissements lorsqu'il détecte des incohérences dans l'écriture du code
 - Le compilateur est votre allié, pas votre ennemi !
- Toujours utiliser un compilateur à son niveau d'alerte maximale
 - Un programme correct doit compiler sans avertissements (nécessaire mais pas suffisant...)
 - Utiliser plusieurs compilateurs différents est un plus

```
% gcc -Wall bonjour.c
```

Structure d'un programme (1)

- Doit obligatoirement contenir une fonction principale « `main ()` », qui est exécutée lorsque le programme est lancé

```
main ()  
{  
}
```

rien.c

- La présentation du code n'est pas importante pour le compilateur, mais pour le programmeur

```
main ( ) {  
}
```

rien.c

Structure d'un programme (2)

- Un programme est la spécification d'un processus de traitement d'informations
- Un programme impératif spécifie précisément les traitements devant être réalisés, sous la forme de suites d'instructions élémentaires
- Ces instructions opèrent sur les valeurs numériques contenues dans des variables nommées

Instructions (1)

- Une instruction peut être :
 - Une instruction simple
 - Un bloc d'instructions
- Une instruction simple
 - Se termine toujours par un point-virgule « ; »
- Un bloc d'instructions définit un ensemble d'instructions s'exécutant en séquence
 - Encadré par des accolades « { ... } »
 - Pas de point-virgule après l'accolade fermante

Instructions (2)

- Une instruction simple peut être
 - Une affectation de variable
 - Un test
 - Une boucle
 - etc.

Commentaires (1)

- Les commentaires sont non seulement utiles, mais nécessaires à la compréhension d'un programme
 - Pensez à ceux qui vous suivront !
- Ils doivent donner une information de niveau d'abstraction plus élevé que le code qu'ils commentent
 - Sinon c'est de la paraphrase inutile...

Commentaires (2)

- On aura en général trois types de commentaires
 - Commentaires d'en-tête de fichier
 - Explique ce que contient le fichier
 - Commentaires d'en-tête de fonction ou de structure de donnée (aussi appelé « commentaire de bloc »)
 - Explique la fonction et l'usage du fragment de code qui suit
 - Commentaires de lignes
 - Déclaration de variable
 - Début de boucle
 - Astuces, etc.

Commentaires (3)

- Forme « standard » : `/* ... */`
 - Le commentaire finit au premier « `*/` » rencontré

```
/* Ce commentaire, qui a un style de commentaire  
** de bloc, s'étend sur plusieurs lignes.  
*/
```

```
a = a + 1;      /* Ceci est un commentaire de ligne */  
b = b - 1;      /* Et ceci en est un autre          */
```

- Pas de commentaires imbriqués !

```
a = a + 1;      /* Ce commentaire /* imbriqué */ ne marche pas */
```

Commentaires (4)

- Forme « C++ » : // ...
 - Valides jusqu'à la fin de la ligne
 - Ne pas utiliser pour du code C standard

```
prht = prht - remise;      /* Commentaire de ligne standard */  
ptva = prht * 0.196;      // Commentaire à la C++ ; bof...
```

- **Attention à bien fermer vos commentaires !**

```
pttc = prht + ptva;      /* Aie, danger, la ligne suivante  
total = total + pttc;    /* Sera ignorée car il manque le : */
```

- Les éditeurs de texte modernes (comme emacs) affichent les commentaires en couleur, ce qui permet de repérer ce type d'erreurs

Variables (1)

- Les variables servent à stocker les données manipulées par le programme
- Les variables sont nommées par des identificateurs alphanumériques

Variables (2)

- Format des identificateurs :
 - La première lettre est comprise dans l'ensemble [a-zA-Z_] (mais éviter le « _ » en début d'identificateur, réservé aux variables du système)
 - Les autres lettres sont comprises dans l'ensemble [a-zA-Z0-9_]
 - Différence entre majuscules et minuscules
 - 31 caractères significatifs selon la norme

✓ Brol	✓ bro1_2_brol	✗ 1bro12trop
✓ bRol	✓ bro1_	✗ 123
✓ bro1123	★ _bro1	✗ 123_45

Variables (3)

- Toutes les variables sont typées
 - Types simples
 - Types structurés
 - Références (pointeurs) de types
- On ne pourra mettre dans une variable d'un type donné une valeur d'un type incompatible
 - C est un langage (assez) fortement typé

Types simples

- Deux familles principales
 - Types entiers :
 - `char` : type caractère, correspondant au stockage d'un unique caractère
 - `int`, `long`, `short` : types numériques servant au stockage des nombres entiers signés
 - Versions « `unsigned` » pour les entiers non signés
 - Types flottants (« types à virgule flottante ») :
 - `float` : nombres à virgule flottante en simple précision
 - `double` : nombres à virgule flottante en double précision

Définition de variables

- Au début d'un bloc d'instructions
 - Après une accolade ouvrante : « `{` »
- On spécifie le type, puis le nom de la variable
 - Autant de fois que nécessaire
- On peut mettre plusieurs noms, séparés par une virgule : « `,` »

```
{
int          i;                /* i : variable entière signée */
int          j, k;            /* j et k : idem */
unsigned long l;              /* l : entier long non signé */
double       d, e;            /* d, e : flottants double préc. */
float        f;                /* f : flottant simple précision */
char         c;                /* c : caractère */
}
```

Affectation de variables (1)

- On donne une valeur à une variable au moyen de l'instruction d'affectation « = »

```
i = 2;  
j = 4356;  
d = 7.43;  
e = 1.25e3;  
e = 18.6;  
c = 'A';
```

- Ce n'est pas une égalité au sens mathématique
 - Son sens est : prends la valeur à droite du « = » et mets-la dans la variable qui est à sa gauche.
 - On le lit : « reçoit »

Affectation de variables (2)

- Il est possible de déclarer et d'initialiser une variable en même temps

```
int      i = 2;  
double  d = 7.43;  
char    c = 'A';  
int      j = 8, k = 42;
```

- Ce n'est pas souhaitable, car définition et initialisation sont deux actions différentes
 - Elles n'ont pas lieu au même moment dans la chaîne de compilation et d'exécution
 - Elles nécessitent donc des commentaires différents

Constantes entières (1)

- Structure générale
 - Signe, optionnel : « + », « - »
 - Suite de chiffres
 - Marqueurs de type, optionnels :
 - « U » : constante non signée
 - « L » : constante de type long

Constantes entières (2)

- Constantes décimales
 - Notation en base dix
 - Suite de chiffres décimaux : [0-9]
 - Pas de « 0 » non significatif en début de nombre

```
0
+0
42
+42
-42
123456
-21474836478
4294967295U
4294967295UL
```

Constantes entières (3)

- Constantes octales
 - Notation en base huit (3 bits par chiffre)
 - Commencent par un zéro
 - Suite de chiffres octals : [0-7]

```
0           ≡ 0
05          ≡ 5 × 80 = 5
-017       ≡ -(1 × 81 + 7 × 80) = -15
+0460L     ≡ 4 × 82 + 6 × 81 + 0 × 80 = 304L
```

- Ne mettez pas de « 0 » devant les constantes entières pour faire joli...

Constantes entières (4)

- Constantes hexadécimales
 - Notation en base seize (4 bits par chiffre)
 - Commencent par « 0x » ou « 0X »
 - Suite de chiffres hexadécimaux : [0-9A-F]

0	0000	4	0100	8	1000	C	1100
1	0001	5	0101	9	1001	D	1101
2	0010	6	0110	A	1010	E	1110
3	0011	7	0111	B	1011	F	1111

```
+0x0       ≡ 0
-0X712     ≡ -(7 × 162 + 1 × 161 + 2 × 160) = -1810
0xA3DUL   ≡ +(10 × 162 + 3 × 161 + 13 × 160) = 2621UL
```

Constantes caractères

- Unique caractère, encadré par des apostrophes « ' » (pas des guillemets)
- Caractère d'échappement : la barre inversée « \ » (appelée « *antislash* » en Anglais)

\0	Caractère nul	\"	Guillemet	\t	Tabulation
\\	Antislash	\r	Retour chariot	\b	Retour arrière
\'	Apostrophe	\n	Nouvelle ligne	\a	Bip terminal

```
'A'           '\''
' '          '\n'
'\0'
```

Domaines des types entiers

- Domaines des types entiers pour une architecture 32 bits

Type	Bits	Valeur minimale	Valeur maximale
char	8	-128	127
unsigned char	8	0	255
int	32	-2147483648	2147483647
unsigned int	32	0	4294967295
short	16	-32768	32767
unsigned short	16	0	65535
long	32	-2147483648	2147483647
unsigned long	32	0	4294967295
long long *	64	-9223372036854775808	9223372036854775807
unsigned long long *	64	0	18446744073709551615

* Norme C99

- Le type `int` est toujours de la taille du mot machine
- Voir le fichier `/usr/include/limits.h`

Constantes à virgule flottante

- Structure générale
 - Signe, optionnel : « + », « - »
 - Suite de chiffres et point décimal « . »
 - Suite de chiffres, optionnelle
 - Exposant « e » ou « E » et exposant entier décimal, optionnels
 - Marqueur de type, optionnel :
 - « F » : constante en simple précision

```
-12.34  
+12.F  
-12.34e-56
```

Expressions

- Une expression est une construction du langage à laquelle est associée un type et une valeur
- Elle correspond le plus souvent à un calcul faisant intervenir des constantes, des variables, et des opérateurs

Opérateurs arithmétiques (1)

- Ces opérateurs servent à la construction d'expressions numériques

Type	Opérateur	Entier	Flottant
Addition	+	✓	✓
Soustraction	-	✓	✓
Multiplication	*	✓	✓
Division	/	✓	✓
Modulo	%	✓	✗

- Les parenthèses permettent d'outrepasser les priorités relatives par défaut des opérateurs

Opérateurs arithmétiques (2)

- La division entière renvoie le quotient de la division euclidienne, arrondi vers 0

```
i = 7 / 3;           /* i vaut 2 */
i = -7 / 3;         /* i vaut -2 */
```

- Le modulo est le reste de la division entière. C'est la différence entre le nombre et le produit du diviseur par le résultat de la division entière

```
j = 7 % 3;          /* j vaut 1, car 7 - (2 * 3) vaut 1 */
j = -7 % 3;         /* j vaut -1, car -7 - ((-2) * 3) vaut -1 */
```

Conversion de type (1)

- Lorsque les deux opérandes d'un opérateur sont de types différents mais comparables, il y a, avant l'évaluation proprement dite, conversion implicite du type le plus faible vers le type le plus fort

`char < short ≤ int ≤ long < float < double`

- Les constantes entières sont de type `int`
- Les constantes flottantes sont de type `double`

```
'A' + 1    ⇒ int
sh * 12.34 ⇒ double
12 / 23.F  ⇒ float
```

Conversion de type (2)

- Lors d'une affectation, le type du membre droit est converti dans le type du membre gauche
- Si le type de destination est plus faible que le type du membre droit :
 - Le résultat peut être indéfini pour les types flottants, si le nombre ne peut être représenté
 - Il y a troncature pour les types entiers

```
char c; int i; double d;
c = 'A' + 1;
i = i * 12.34;
c = 0;          /* À éviter ! Utiliser plutôt : c = '\0' */
d = 0;          /* Pareil : utiliser plutôt : d = 0.0 */
```

Conversion de type (3)

- L'opérateur parenthésé de conversion de type (aussi appelé « *type cast* ») permet les conversions explicites
 - Permet de spécifier explicitement les conversions au compilateur
 - Supprime les avertissements du compilateur lors de conversions vers des types plus faibles

```
int    diviseur;
int    reste;
double nombre;
...
reste = (int) nombre % diviseur;
```

Affectations combinées

- Comme de nombreuses instructions arithmétiques sont des modifications du contenu de variables, il existe des raccourcis spécifiques

`var = var op expr; ⇔ var op= expr;`

- On trouve ces raccourcis pour tous les opérateurs arithmétiques binaires : « += », « -= », « *= », « /= », « %= »

```
i = 3; i += 12;    /* i = 15          */
i = 3; i -= 7;    /* i = -4 (i de type int) */
i = 3; i *= 4;    /* i = 12           */
i = 3; i /= 2;    /* i = 1 (division entière) */
i = 13; i %= 5;   /* i = 3            */
```

Incrémentation et décrémentation (1)

- Comme de nombreuses instructions arithmétiques mettent en jeu des incrémentations et décréments, des raccourcis spécifiques existent

```
var += 1;    ⇔    var ++;
var -= 1;    ⇔    var --;
```

- Ces instructions sont également des expressions qui ont une valeur et peuvent être utilisées dans une expression englobante

```
i = 3;
a = i ++;    /* L'expression "i ++" a un type et une valeur */
```

Incrémentation et décrémentation (2)

- Selon que les opérateurs sont positionnés avant ou après la variable sur laquelle ils opèrent, on lit la valeur de la variable avant ou après l'opération, et on a soit une pré-, soit une post- incrémentation (ou décrémentation)

```
var ++;    ⇒    Post-incrémentation
++ var;    ⇒    Pré-incrémentation
```

```
i = 3; a = i ++;    /* Post-incrémentation : a = 3 et i = 4 */
i = 3; a = ++ i;    /* Pré-incrémentation : a = 4 et i = 4 */
i = 3; a = i --;    /* Post-décrémentation : a = 3 et i = 2 */
i = 3; a = -- i;    /* Pré-décrémentation : a = 2 et i = 2 */
```

Affectations et expressions

- Les affectations sont par elles-mêmes des expressions, car elles ont :
 - Un type : le type du membre droit
 - Une valeur : la valeur du membre droit
- On peut donc les utiliser comme termes d'expressions englobantes
 - Attention à la lisibilité !

```
i = j = k = 1;          /* Usage courant */
i = 2 * j += k *= 3;    /* À éviter... */
```

Précédence des opérateurs (1)

- Tableau partiel des opérateurs arithmétiques déjà abordés

Opérateur	Signification	Associativité
()	Parenthésage	G → D
-	Négation unaire	D → G
++ --	Inc/decrémentation	D → G
* / %	Mult/div/modulo	G → D
+ -	Addition/soustraction	G → D
= += -= *= /= %=	Affectation	D → G

Fonction d'écriture printf (1)

- Sert à afficher du texte
- Affiche la chaîne de caractères passée en paramètre, entre guillemets « " »
- Les séquences d'échappement valables pour les caractères s'appliquent également

```
printf ("Je dis \"Bonjour!\"\n");  
printf ("Avec l'apostrophe\n");  
printf ("Avec l\'apostrophe\n"); /* Le \"'\" n'est pas nécessaire */
```

Fonction d'écriture printf (2)

- La chaîne peut indiquer comment afficher d'autres paramètres, au moyen des séquences d'échappement « % »

%%	« % »
%c	Caractère (char)
%d	Entier (int)
%ld	Entier long (long)
%f	Flottant (float)
%lf	Flottant double précision (double)

```
printf ("La valeur de i est %d\n", i);  
printf ("%lf est plus petit que %lf\n", d1, d2);  
printf ("Le caractère que vous avez tapé est \'%c'\n", c);
```

Fonction de lecture scanf (1)

- Fonction inverse de `printf`
- Sert à analyser du texte tapé et à le convertir en valeurs placées dans les variables passées en paramètres, selon la chaîne de format passée en premier paramètre
 - Pas d'espaces dans la chaîne de format
- On doit mettre un « & » avant les noms des variables de types simples (référence)

```
int i;  
printf ("Entrez votre valeur : ");  
scanf ("%d", &i);
```

Structures de contrôle

- Servent à orienter l'exécution du programme en fonction de la valeur courante d'une expression
 - Exécution conditionnelle d'un fragment de code si une certaine condition est vérifiée
 - Exécution répétitive d'un fragment de code tant qu'une certaine condition est vérifiée
- Les conditions de contrôle sont des expressions logiques booléennes

Expressions booléennes (1)

- Une expression booléenne peut être une comparaison entre deux valeurs, au moyen des opérateurs :
 - « < » : strictement inférieur
 - « <= » : inférieur ou égal
 - « == » : égal (attention : « = » est l'affectation !)
 - « != » : différent
 - « >= » : supérieur ou égal
 - « > » : strictement supérieur

```
a >= b  
a != b  
a < b
```

Expressions booléennes (2)

- Elle peut aussi être une combinaison logique entre deux expressions booléennes, au moyen des opérateurs :
 - « || » : ou bien
 - « && » : et aussi
 - « ! » : non pas

```
(a > b) || (a > c) /* a n'est pas le plus petit des trois */  
((an % 4) == 0) && ! (((an % 100) == 0) && ((an % 400) != 0))  
/* an est bissextile */
```

- La deuxième clause n'est pas toujours évaluée
 - Attention aux effets de bord !

```
(a ++ > b) || (a > c --) /* Si (a > b) est vrai, pas de c-- */
```

Instruction `if ... else` (1)

- **Format**
 - `if (expression) instruction1`
 - `if (expression) instruction1 else instruction2`
- **Si l'expression est vraie, l'instruction1 est exécutée, sinon l'instruction2 est exécutée, si elle existe**

```
if (((an % 4) != 0) || (((an % 100) == 0) && ((an % 400) != 0)))
    nbjours = 365;                               /* Si année pas bissextile */
else {
    nbjours = 366;
    bissext = 1;
}
```

Instruction `if ... else` (2)

- **Lorsqu'on a plusieurs cas à tester, on peut enchaîner les `if ... else`**

```
if (n < 10)
    printf ("%d est plus petit que 10.\n", n);
else {
    /* Accolades pour la lisibilité, pas nécessaires */
    if (n > 20)
        printf ("%d est plus grand que 20.\n", n);
}
```

- **Si les conditions sont de même nature, on peut aligner les `else` sur le premier `if`**

```
if (n < 10)
    printf ("%d est plus petit que 10.\n", n);
else if (n > 20)
    printf ("%d est plus grand que 20.\n", n);
```

Instruction `if ... else` (3)

- Un `else` se rapporte toujours au dernier `if` rencontré

```
if (n < 10)
  if (n < 5)
    printf ("%d est plus petit que 5.\n", n);
else
    /* Mauvaise indentation et bogue : 4 < n < 10 ! */
    printf ("%d est plus grand que 9.\n", n);
```

- Des accolades sont nécessaires pour associer un `else` à un `if` antérieur

```
if (n < 10) { /* Ces accolades sont nécessaires */
  if (n < 5)
    printf ("%d est plus petit que 5.\n", n);
}
else /* Ici on a bien n >= 10 ! */
  printf ("%d est plus grand que 9.\n", n);
```

Opérateur ternaire `... ? ... : ...` (1)

- Dans de nombreux cas, un test sert juste à positionner le contenu d'une variable

```
if (a > b)
  max = a;
else
  max = b;

max = b;
if (a > b)
  max = a;
```

- Plusieurs problèmes :
 - Redondance de code source
 - Lisibilité, risques d'erreurs
 - Redondance de code objet
 - Affectations multiples

Opérateur ternaire ... ? ... : ... (2)

- L'opérateur ternaire « ? : » renvoie une valeur différente selon la vérité de l'expression qui le précède
 - (expression) ? expression1 : expression2

```
max = (a > b) ? a : b;
```

- Si l'expression conditionnelle est vraie, la valeur renvoyée par l'opérateur est la valeur d'expression1, sinon c'est celle d'expression2
 - Seule une des expressions est évaluée, comme pour toutes les autres instructions conditionnelles

Instruction switch ... case

- L'instruction switch ... case sert à traiter des choix multiples en fonction de la valeur d'une expression entière

```
printf ("Entrez votre choix : ");
scanf ("%c", &c);
switch (c) {
    case 'A' : /* Évaluation d'une expression entière */
        ... /* Clause de choix sur une constante */
        break; /* Instructions exécutées jusqu'au break */
    case 'q' : /* Plusieurs clauses pour le même bloc */
    case 'Q' : /* d'instructions : continue si pas break */
        ... /* Instructions exécutées jusqu'au break */
        break;
    default : /* Clause optionnelle "voiture balai" */
        printf ("Choix \"%c\" invalide.\n", c);
        break;
}
```

Instruction while

- **Format**
 - while (expression) instruction
- **Tant que l'expression est vraie, on exécute l'instruction**
 - L'instruction doit modifier l'expression, sinon on a une boucle infinie

syracuse.c

```
...
printf ("Entrez le nombre de départ : ");
scanf ("%d", &n);
while (n > 1) {
    n = ((n % 2) == 0) ? (n / 2) : (3 * n + 1);
    printf ("%d\n", n);
}
```

Instruction for (1)

- **Format**
 - for (instruction1; expression; instruction2)
instruction3
- L'instruction1 est d'abord exécutée (initialisation). Puis, tant que l'expression est vraie, on exécute l'instruction3 (corps de boucle), puis l'instruction2 (itérateur)

```
printf ("Je compte jusqu'à 10 :\n");
for (i = 1; i <= 10; i++) /* Pour i allant de 1 à 10 inclus */
    printf ("%d\n", i);
```

Instruction `for` (2)

- L'instruction `for` est sémantiquement équivalente à l'instruction `while`

```
for (i = 0; i < 10; i ++)  
    instruction;  
  
i = 0;  
while (i < 10) {  
    instruction;  
    i ++;  
}
```

- Elle apporte cependant une lisibilité supplémentaire
 - Lien conceptuel entre l'initialisateur et l'itérateur
 - Proximité visuelle entre l'initialisateur et l'itérateur

Instruction `for` (3)

- On peut utiliser l'opérateur virgule « , » pour exécuter plusieurs instructions simples dans l'initialisateur ou l'itérateur d'une boucle

```
int    i, f, n;  
  
/* Calcul de n!, en C un peu « extrême », sur une ligne */  
for (i = n, f = 1; i > 1; f *= i, i --) ; /* Boucle à corps vide */  
  
/* La même, un peu moins violente, sur deux lignes */  
for (i = n, f = 1; i > 1; i --) /* On voit mieux l'itérateur */  
    f *= i; /* Le but de la boucle est plus visible */
```

- L'opérateur virgule sert peu autrement...

```
a = (b ++, c); /* La valeur de a est celle de c. Moche... */
```

Instruction `do ... while`

- **Format**
 - `do instruction while (expression)`
- L'instruction est d'abord exécutée, puis l'expression est évaluée. Si elle est vraie, on reboucle sur l'exécution de l'instruction
 - À la différence de la boucle `while`, l'instruction est toujours exécutée au moins une fois

```
do {  
    printf ("Entrez un nombre entre 1 et 10 : ");  
    scanf ("%d", &n);  
} while ((n < 1) || (n > 10));
```

Expressions booléennes (3)

- Une expression booléenne est une fonction logique qui ne retourne que les valeurs VRAI et FAUX
- En C, il n'existe pas de type booléen propre
 - Type `bool` introduit dans la norme C99 seulement
- Le type booléen est émulé par le type `int`, avec les conventions suivantes :
 - FAUX : valeur 0
 - VRAI : toutes les autres valeurs non nulles

Expressions booléennes (4)

- On peut donc :
 - Stocker le résultat d'une expression booléenne dans une valeur entière
 - Utiliser le résultat d'une expression entière comme valeur de condition

```
i = ((a > b) && (a > c)); /* i vaut 0 ou 1 avec gcc */  
  
if (diviseur) /* diviseur est une variable entière */  
    nombre /= diviseur;
```

- Cette fonctionnalité est dangereuse...

```
a = 0;  
b = 0;  
if (a = b) /* Bogue : « == » et pas « = » */  
    printf ("a et b sont égaux.\n");
```

Instruction break (1)

- L'instruction `break` sert à quitter la boucle la plus interne, ou bien le `case`, dans lesquels elle se trouve
 - On ne sort que d'un seul niveau
 - Ne concerne pas les `if ... else`

```
s = 0;  
for (i = 0; i < 10; i++) { /* Lecture d'au plus 10 entiers */  
    scanf ("%d", &n);  
  
    if (n <= 0) /* Si entier non strictement positif */  
        break; /* On quitte la boucle */  
  
    s += n;  
}
```

Instruction break (2)

- L'instruction `break` peut être utilisée pour sortir d'une boucle déclarée comme boucle infinie

```
while (1) {                                /* Boucle infinie ; comme for ( ; ; ) { */
    scanf ("%c", &c);
    c = toupper (c);

    if (c == 'Q')                          /* Si c vaut la valeur de sortie */
        break;                             /* On sort de la boucle infinie */
    switch (c) {
        case 'A' :
            ...
            break;                          /* Ce break là ne sort que du switch */
            ...
    }
}
```

Instruction continue (1)

- L'instruction `continue` sert à sauter l'itération courante de la boucle la plus interne dans laquelle elle se trouve, et à passer à l'itération suivante
 - Évite le recours au `goto`

```
s = 0;
for (i = 0; i < 10; i ++) {
    scanf ("%d", &n);

    if (n <= 0)                             /* Si entier non strictement positif */
        continue;                          /* On passe au suivant */

    s += n;
}
```

Instruction continue (2)

- L'instruction continue permet de se dérouter sur une condition exceptionnelle sans avoir à indenter le reste du corps d'une boucle

```
for (...; ...; ...) {  
    ...  
    if (condition) {                /* Cas particulier      */  
        ...                        /* Traitement occasionnel */  
    }  
}
```

```
for (...; ...; ...) {  
    ...  
    if (! condition)                /* Cas exceptionnel */  
        continue;  
    ...                            /* Traitement habituel */  
}
```

Sous-programmes et fonctions (1)

- Les sous-programmes servent à factoriser du code et à améliorer sa lisibilité
- Lorsque le même fragment de code doit être exécuté plusieurs fois, il est naturel d'isoler ce fragment et de lui donner un nom, pour y faire référence lorsqu'on veut l'exécuter
- Lorsque, à un endroit du programme, on veut exécuter les instructions du sous-programme, on spécifie le nom du sous-programme considéré : on « appelle » le sous-programme

Sous-programmes et fonctions (2)

- La plupart des sous-programmes servent à calculer un résultat à partir des valeurs de certaines variables du programme appelant
 - Ils se comportent comme des fonctions mathématiques
- On doit donc spécifier
 - La liste et le type des paramètres que l'on peut passer à la fonction
 - Le type de la valeur qu'elle renvoie

Fonctions (1)

- En C, il n'y a pas de différence entre fonction et sous-programme
 - Tous sont définis sous la forme de fonctions censées renvoyer une valeur d'un type spécifié
 - Un sous-programme est une fonction qui ne renvoie rien (type `void`)

Fonctions (2)

- Une fonction est un bloc d'instructions ayant :
 - Un type pour les valeurs qu'elle retourne
 - Un nom
 - Une liste de paramètres typés, entre parenthèses

```
int factorielle (int n) { ... }
double puissance (double valeur, double exposant) { ... }

double                /* Type de retour                */
puissance (          /* Nom de la fonction, mis en évidence                */
double      valeur, /* Liste des paramètres, déclarés comme                */
double      exposant) /* Des variables et commentés par ligne                */
{                    /* Accolade de début de bloc fonction                */
    ...              /* Indentation du code dans le bloc                    */
}
```

Fonctions (3)

- Les paramètres d'une fonction sont vus comme des variables locales au bloc de code de la fonction
- On peut définir d'autres variables dans le bloc

```
void                /* Pas de valeur de retour */
compte_a_rebours (
int      n)        /* Paramètre de la fonction, vu comme local */
{
    int      i;    /* Variable locale à la fonction */

    printf ("Attention, compte à rebours :\n");
    for (i = n; i >= 0; i --)
        printf ("%d...\n", i);
    printf ("Et c'est parti !\n");
}
```

Instruction `return` (1)

- On spécifie la valeur que renvoie une fonction au moyen de l'instruction `return`
 - Valeur de même type que le type de retour déclaré de la fonction

```
int          /* Type de retour : int */
factorielle (
int          n)
{
    int      f;          /* Variables locales dont on a besoin */
    int      i;

    for (f = 1, i = 2; i <= n; i ++ )
        f *= i;
    return (f);          /* Renvoi de la valeur, de type int aussi */
}                       /* Les parenthèses, c'est juste plus joli */
```

Instruction `return` (2)

- L'instruction `return` permet la terminaison anticipée de la fonction
 - Peut exister en plusieurs exemplaires

```
int
factorielle (
int          n)
{
    int      f;
    int      i;

    if (n < 0)          /* Si paramètre non conforme */
        return (-1);    /* Renvoie un code d'erreur */
    for (f = 1, i = 2; i <= n; i ++ )
        f *= i;
    return (f);          /* Renvoie la valeur attendue */
}
```

Instruction return (3)

- Pour les fonctions ne retournant rien, on utilise l'instruction `return` sans argument

```
void                               /* Pas de valeur de retour */
compte_a_rebours (
int    n)
{
    int    i;

    if (n <= 0) {
        printf ("La fusée est déjà partie...\n");
        return;                /* Retour anticipé, sans argument */
    }
    printf ("Attention, compte à rebours :\n");
    for (i = n; i >= 0; i --)
        printf ("%d...\n", i);
    printf ("Et c'est parti !\n");
    return;                    /* Optionnel, car fin de fonction termine */
}
```

Appel de fonction

- On appelle une fonction en donnant son nom, suivi de la valeur des paramètres de l'appel, dans l'ordre de leur définition
- Le nom de fonction avec ses arguments est une expression typée utilisable normalement

```
main ()
{
    int    i;
    int    j;

    i = factorielle (7);
    printf ("La factorielle de 7 est %d\n", i);
    j = 5;
    printf ("La factorielle de %d est %d\n", j, factorielle (j));
}
```

Opérateurs bit à bit

- Les types entiers peuvent être vus comme des vecteurs de bits de tailles fixes (8, 16, 32, 64)
 - char c = 'A'; c =

0	1	0	0	0	0	0	1
---	---	---	---	---	---	---	---
- On peut y effectuer des opérations booléennes opérant en parallèle sur chacun des bits
 - Opérations booléennes « et », « ou », « ou exclusif », « inversion »
 - Décalages

Opérateur « et » : &

- Réalise un « et » booléen bit à bit

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

a =

0	1	1	0	1	0	1	1
---	---	---	---	---	---	---	---

b =

0	0	1	1	1	0	0	1
---	---	---	---	---	---	---	---

a & b =

0	0	1	0	1	0	0	1
---	---	---	---	---	---	---	---

- Ne pas confondre « & » et « && »

```

a = 0xC5;          /* a = 11000101 */
b = 0x57;          /* b = 01010111 */
x1 = a & b;        /* x1 = 01000101 */
x2 = a && b;        /* x2 = 00000001 */
a &= 0x07;        /* Garde les 3 bits faibles (a % 8) */
    
```

Opérateur « ou » : |

- Réalise un « ou » booléen bit à bit

A	B	A B
0	0	0
0	1	1
1	0	1
1	1	1

a = 0 1 1 0 1 0 1 1

b = 0 0 1 1 1 0 0 1

a | b = 0 1 1 1 1 0 1 1

- Ne pas confondre « | » et « || »

```

a = 0xC5;          /* a = 11000101 */
b = 0x57;          /* b = 01010111 */
x1 = a | b;        /* x1 = 11010111 */
x2 = a || b;       /* x2 = 00000001 */
a |= 0x04;         /* Met à 1 le troisième bit */
    
```

Opérateur « ou exclusif » : ^

- Réalise un « ou exclusif » (x-or) booléen bit à bit

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

a = 0 1 1 0 1 0 1 1

b = 0 0 1 1 1 0 0 1

a ^ b = 0 1 0 1 0 0 1 0

- Identifie les bits qui diffèrent

```

a = 0xC5;          /* a = 11000101 */
b = 0x57;          /* b = 01010111 */
x = a ^ b;         /* x = 10010010 */
    
```

Opérateur « non » : ~

- Réalise une complémentation booléenne bit à bit

A	~A
0	1
1	0

a = 0 1 1 0 1 0 1 1

~a = 1 0 0 1 0 1 0 0

```
a = 0xC5; /* a = 11000101 */
a &= ~0x04; /* Met à zéro le troisième bit */
```

Opérateurs de décalage : << et >>

- Réalisent un décalage du mot binaire vers la gauche ou la droite, en insérant des 0 dans les positions laissées libres

a = 0 1 1 0 1 0 1 1

a << 2 = 1 0 1 0 1 1 0 0

a >> 3 = 0 0 0 0 1 1 0 1

```
scanf ("%d", &n);
for (i = 1, b = 0; i != 0; i <<= 1) { /* Pour tous les bits */
    if ((i & n) != 0) /* Si le bit est à 1 */
        b++; /* On en compte un de plus */
}
printf ("Le nombre %d contient %d bits à 1\n", n, b);
```

Variables références (1)

- Le plus souvent, on sait exactement à quelle variable va s'appliquer un traitement
 - Noms de variables « en dur » dans le programme
- Dans certains cas, on voudrait pouvoir faire varier dynamiquement le nom de la variable à laquelle appliquer un traitement donné
 - Pas possible de changer le programme en cours d'exécution, car il est compilé
 - Nécessité de variables servant à « identifier », à « référencer », d'autres variables

Variables références (2)

- On déclare une variable référence à une variable d'un type donné au moyen de l'opérateur d'indirection « * »

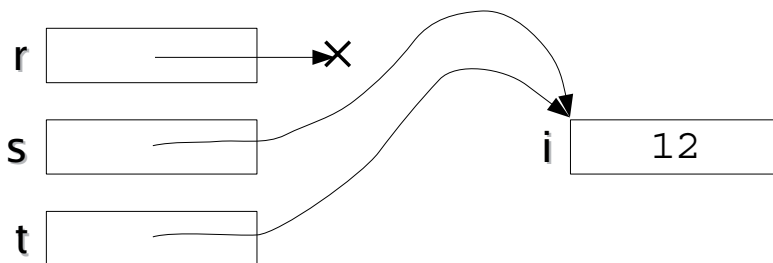
```
int    i;  
int *  r;    /* Référence sur une variable entière */  
int *  s, * t; /* Autres références à des variables entières */
```

- On peut lire la déclaration « type * var » de deux façons :
 - var référence une variable de type type
 - *var est de type type

Variables références (3)

- On initialise une variable référence soit à NULL (elle ne référence rien), soit avec l'opérateur de référence « & », soit avec une autre variable de référence de même type

```
i = 12;
r = NULL;           /* Référence nulle      */
s = &i;            /* Référence sur i     */
t = s;             /* Autre référence sur i */
```



Variables références (4)

- Pour accéder au contenu d'une variable référence, on utilise également l'opérateur d'indirection « * »

```
int    a;
int    b;
int *  maxptr;

a = 5;
b = 17;
maxptr = (a > b) ? &a : &b;    /* Obtient la référence du max */

(*maxptr) ++;                /* Incrémente le max de 1 */

printf ("Le maximum incrémenté est %d\n", *maxptr);
```

Variables références (5)

- On peut tout à fait avoir des références de références...

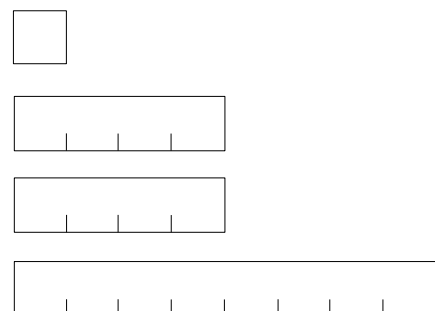
```
int    a;
int *  p;
int ** q;

a = 5;
p = &a;
q = &p;
**q = 7;           /* Maintenant a vaut 7 */
```



Occupation mémoire

- À toute variable créée est associée une zone de la mémoire, servant à stocker le contenu de cette variable
- La taille de cette zone mémoire dépend du type de la variable considérée
 - char : 1 octet
 - int : 2, 4 ou 8 octets (ici : 4)
 - float : 4 octets
 - double : 8 octets
 - etc.



Opérateur sizeof ()

- L'opérateur `sizeof` donne la taille en octets du type ou de la variable « passée en paramètre »
 - Ce n'est pas une fonction normale
 - Évalué et remplacé par la constante entière correspondante lors de la compilation

```
printf ("Sur mon système, un \"int\" fait %d octets\n",
        sizeof (int));          /* Taille d'un type */

double d;
printf ("Sur mon système, un \"double\" fait %d octets\n",
        sizeof (d));          /* Taille d'une variable du type */
```

Références et pointeurs (1)

- Tout mot (octet) de la mémoire est identifié par un numéro unique : son adresse mémoire
- On peut donc identifier toute zone mémoire servant au stockage d'une variable par :
 - Son adresse de début
 - Sa taille (dépend du type de la variable)



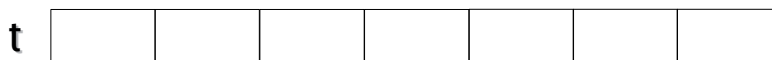
Références et pointeurs (2)

- En termes d'implémentation, en C, une variable référence contient l'adresse de la variable référencée
 - La référence est en fait un pointeur sur l'adresse mémoire de début de la variable
- Les pointeurs C sont plus permissifs que les références
 - Possibilité de positionnement à des valeurs arbitraires non nécessairement valides
 - Arithmétique sur les pointeurs

Tableaux (1)

- Un tableau est une collection de variables de même type que l'on peut accéder individuellement par leur indice dans le tableau
- On déclare la taille du tableau entre « [] »

```
int t[7]; /* t est un tableau de 7 entiers */
```



Tableaux (2)

- On accède à un élément d'un tableau en donnant l'indice de l'élément entre « [] »
 - En C, les indices commencent à 0, pas à 1

t

t[0]	t[1]	t[2]	t[3]	t[4]	t[5]	t[6]
------	------	------	------	------	------	------

- Aucun contrôle de débordement sur les indices !

```
int t[7]; /* Tableau de 7 éléments entiers */
int a; /* Entier en général placé après */

for (i = 0; i < 7; i++) /* Initialise tout le tableau */
    t[i] = i;
for (i = 0; i < 7; i += 3) /* Pour tous les multiples de 3 */
    scanf ("%d", &t[i]); /* Écrase avec la valeur lue */
scanf ("%d", &a);
t[7] = 10; /* On écrit en dehors du tableau ! */
```

Tableaux (3)

- On peut initialiser le contenu d'un tableau lors de sa déclaration
 - Liste de constantes du type adéquat, entre accolades (constantes seulement, car initialisé à la compilation)
 - Nombre d'éléments comptés par le compilateur

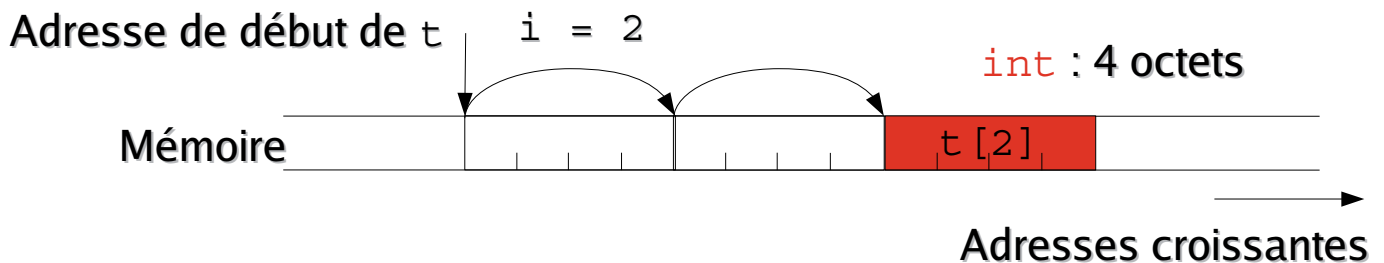
```
int t[7] = { 2, 4, 7, 5, 6, 3, 1 };
int t[] = { 2, 4, 7, 5, 6, 3, 1 }; /* Le compilateur comptera */
```

- La liste peut n'être que partielle

```
int t[7] = { 2, 4, 7, 5 }; /* 7 places prises, 4 initialisées */
```

Tableaux (4)

- Si t est de type « tableau de type », $t[i]$ est donc le contenu de la zone mémoire :
 - Commençant à l'octet d'adresse $(t + i * \text{sizeof}(\text{type}))$
 - De taille $\text{sizeof}(\text{type})$

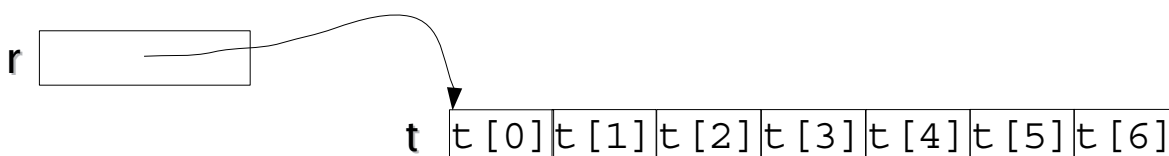


Tableaux et pointeurs (1)

- Un tableau est une référence sur une zone mémoire de variables contiguës de même type
 - On peut utiliser un nom de tableau comme référence

```
int    t[7];           /* Tableau de 7 entiers    */
int *  r;             /* Pointeur sur un entier */

r = t;               /* t est en fait de type (int *) */
a = t[0];           /* Lecture du premier élément du tableau */
b = *r;             /* Comme r pointe sur t, on lit aussi t[0] */
c = *t;             /* Comme t est un (int *), ceci marche aussi */
t = r;             /* Ne marche pas ! t est une constante... */
```

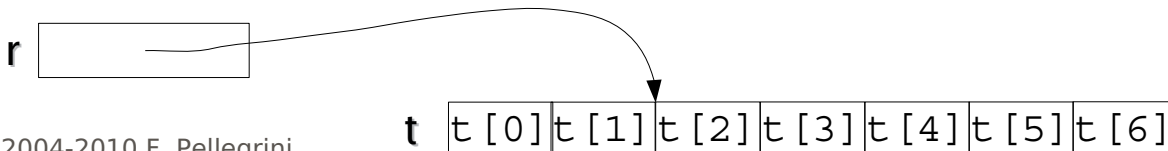


Tableaux et pointeurs (2)

- Tableaux et pointeurs étant de même type, on peut également utiliser la notation « [] » à partir de variables de types pointeurs
 - « [i] » veut donc dire : contenu de la i^{ème} case de type adéquat, comptée à partir de l'adresse du pointeur

```
int    t[7];           /* Tableau de 7 entiers */
int *  r;             /* Pointeur sur un entier */

r = &t[2];           /* On pointe sur la troisième case de t */
r[1] = 12;          /* On met 12 dans r[1], c'est-à-dire t[3] */
```



95

Tableaux et pointeurs (3)

- Un « pointeur de » et un « pointeur sur un tableau de » sont syntaxiquement de même type
 - Le « sur un tableau » est une information d'usage
- C'est l'algorithme (et donc le programmeur) qui définit s'il est légal d'accéder aux cases voisines de la case mémoire référencée par un pointeur donné à un instant donné
 - Le compilateur n'a en pratique aucun moyen d'inférer cette information de l'analyse du code
 - Les problèmes surviennent à l'exécution...

Arithmétique des pointeurs (1)

- À partir d'un pointeur, on peut créer un pointeur de même type mais pointant à i cases en amont ou en aval

```
q = &p[i]; /* i indifféremment positif ou négatif */
```

- On peut simplifier cette expression en définissant une arithmétique des pointeurs
 - $p + i \Leftrightarrow \&p[i]$
 - $p - i \Leftrightarrow \&p[-i]$

```
int t[7]; /* Tableau de 7 entiers */
int * r; /* Pointeur sur un entier */

r = t + 2; /* Identique à l'exemple précédent */
```

Arithmétique des pointeurs (2)

- L'arithmétique des pointeurs permet aussi d'exprimer la différence entre deux pointeurs de même type
 - $p - q \Leftrightarrow$ quotient de la division par la taille du type de la différence entre les adresses des pointeurs p et q

```
int t[7]; /* Tableau de 7 entiers */
int * r; /* Pointeur sur un entier */
int i;

r = t + 2; /* Identique à l'exemple précédent */
i = r - t; /* Différence entre deux pointeurs */
```

- $q + (p - q)$ peut être différent de p si pas alignés

Arithmétique des pointeurs (3)

- Les opérateurs de comparaison « < », « <= », « == », « != », « >= » et « > » s'appliquent également aux pointeurs
 - Les variables doivent être de même type
 - Pour que le résultat ait un sens, elles doivent avoir été construites par décalage à partir d'un ancêtre commun
- Comparer q et p est équivalent à tester le signe de (q - p)
 - $(q > p) \Leftrightarrow ((q - p) > 0)$

Arithmétique des pointeurs (4)

- Tous les opérateurs arithmétiques d'addition et de soustraction d'entiers existent également pour les pointeurs
 - « += », « -= », « ++ », « -- »

```
int    t[7];
int *  p;
int    i;

for (i = 0; i < 7; i++)      /* Boucle sur les indices      */
    scanf ("%d", &t[i]);    /* Ou bien : scanf ("%d", t + i) */

for (p = t; p < t + 7; p++) /* Boucle sur références de cases */
    scanf ("%d", p);
```

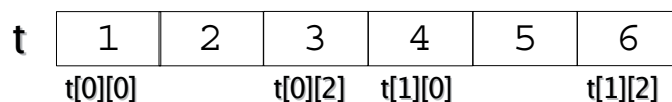
Tableaux multidimensionnels (1)

- Il est possible de déclarer des tableaux multidimensionnels en spécifiant plusieurs paires de « [] »

```
int t[7][3]; /* Tableau de 7 lignes de 3 colonnes */
```

- Les données sont stockées par ligne majeure, c'est-à-dire que dans $t[n][m]$ la case $t[i+1][0]$ est stockée juste après la case $t[i][m-1]$

```
int t[2][3] = { { 1, 2, 3 }, { 4, 5, 6 } };
```

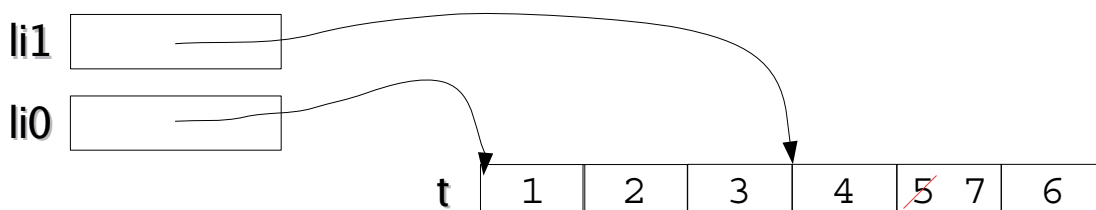


Tableaux multidimensionnels (2)

- Le compilateur effectue les calculs d'adresses de façon à accéder à la bonne case du (sous-)tableau stocké de façon linéaire

```
int t[2][3] = { { 1, 2, 3 }, { 4, 5, 6 } };
int * li0; /* Pointeur sur la première ligne */
int * li1; /* Pointeur sur la deuxième ligne */

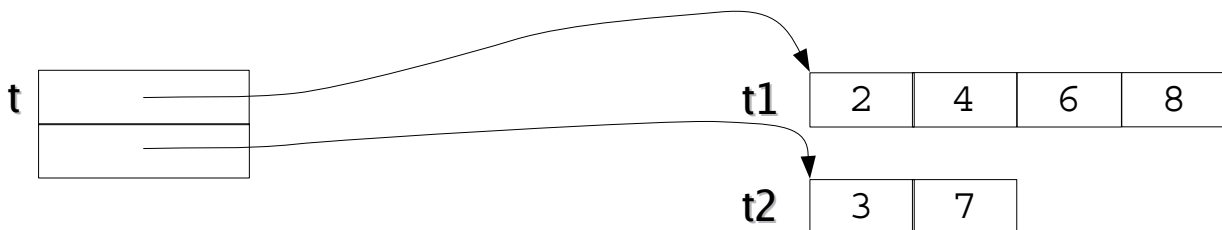
li1 = t[1]; /* t de type (int **), t[1] de type (int *) */
li0 = t[0]; /* Même valeur d'adresse, pas même type ! */
li1[1] = 7; /* Écriture tout à fait correcte */
t[0][4] = 7; /* Pas de vérification de débordement ! */
```



Tableaux multidimensionnels (3)

- On peut émuler des tableaux multidimensionnels au moyen de pointeurs de pointeurs

```
int * t[2]; /* t est de type (int **) */
int t1[4] = { 2, 4, 6, 8 };
int t2[2] = { 3, 7 };
t[0] = t1;
t[1] = t2;
a = t[0][3]; /* a = 8 */
b = t[1][0]; /* b = 3 */
c = t[0][4]; /* En dehors des limites ! */
```



Constantes chaînes de caractères (1)

- Ensemble de constantes caractères accolées, encadrées par des guillemets « " »
 - Même utilisation du caractère d'échappement « \ »

\0	Caractère nul	\"	Guillemet	\t	Tabulation
\\	Antislash	\r	Retour chariot	\b	Retour arrière
\'	Apostrophe	\n	Nouvelle ligne	\a	Bip terminal

```
"Je dis \"Bonjour\" à tout le monde\n"
```

Constantes chaînes de caractères (2)

- Possibilité de chaînes multi-lignes avec « \ » placé en fin de ligne

```
"Ceci est une chaîne de caractères \r  
multi-\r  
lignes"
```

- Possibilité d'agglutiner plusieurs chaînes en une seule

```
"Ceci est une autre chaîne" " de caractères"  
" en morceaux.\n"
```

Chaînes de caractères (1)

- Il n'existe pas de type chaîne spécifique
- Une chaîne de caractères est un tableau unidimensionnel de caractères
 - Le nom de la chaîne fait référence à l'adresse de début du premier caractère de la chaîne
- Une chaîne de caractères bien formée est toujours terminée par un caractère nul '\0'
 - Indique où le contenu utile de la chaîne finit
 - Ne pas oublier de compter et réserver sa place !

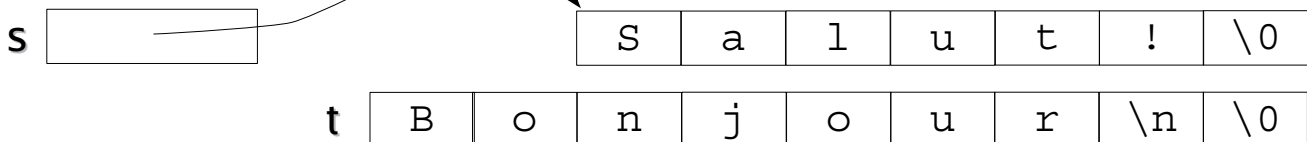
Chaînes de caractères (2)

- On peut déclarer et initialiser une chaîne de caractères comme on déclare un tableau de caractères

```
char t[] = { 'B', 'o', 'n', 'j', 'o', 'u', 'r', '\n', '\0' };
```

Il est préférable d'utiliser les constantes chaînes

```
char t[] = "Bonjour\n" ; /* Tableau modifiable en mémoire */
char * s = "Salut!"; /* Pointeur sur une constante */
t[1] = 'Z'; /* Légal car t est modifiable */
s[1] = 'Z'; /* Segfault car zone constante */
s = t; /* Légal car s est une variable */
s[1] = 'A'; /* Légal car t[1] est modifiable */
```



Affichage de chaînes

- Directement dans `printf` si chaîne simple
 - Code « `%s` » pour afficher des chaînes
- La chaîne est affichée jusqu'au premier « `\0` » rencontré

```
char t[] = "Bonjour\n";
char * s = "Salut!";
char * r = "%sEt aussi %s\n";
```

```
printf ("Bonjour\n"); /* Le compilat. crée une chaîne constante */
printf (t); /* Attention : pas de "%" dans la chaîne ! */
printf ("%s", t); /* Le plus sûr si on n'est pas sûr de t */
printf (r, t, s); /* Possible mais délicat */
```

Entrée de chaînes

- On utilise `scanf`
 - Code « `%s` » aussi
 - Pas besoin du « `&` » car une chaîne est déjà une référence sur une zone mémoire
 - Limiter la taille avec la syntaxe « `%numss` » pour éviter les débordements de tampons

```
char str[10];          /* Une chaîne de 10 caractères */
int a;

printf ("Entrez votre nom : ");
scanf ("%s", str);    /* Pas de « & » car str est une référence */

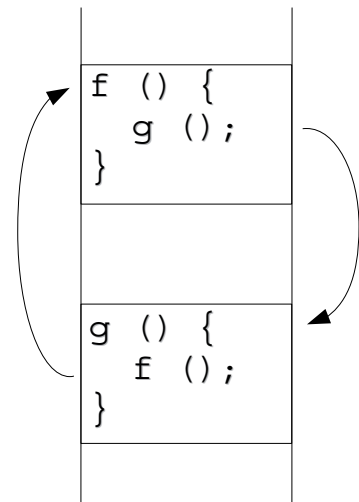
scanf ("%9s", str);  /* On limite à 1 de moins (pour \0) */
```

Fonctions de manipulation de chaînes

- `strlen ()` : renvoie la taille courante d'une chaîne (pas la taille maximale du tableau)
- `strcpy ()` : copie d'une chaîne source vers un tableau de caractères destination
- `strcat ()` : ajout d'une chaîne source à la fin d'une chaîne destination
- `strchr ()` : recherche de la première occurrence d'un caractère dans une chaîne
- etc.

Prototypage (1)

- Pour générer l'appel à une fonction et vérifier le typage de ses paramètres, le compilateur doit déjà connaître l'existence de la fonction et son typage
 - Il suffit que la fonction appelée soit définie avant la fonction appelante dans le fichier source
 - Pas toujours faisable, dans le cas de fonctions s'appelant mutuellement



Prototypage (2)

- Pour résoudre le problème, il faut pouvoir déclarer une fonction avant de la définir
 - Déclaration d'un prototype de la fonction
- Un prototype est la déclaration d'une fonction sans le bloc d'instructions qui la suit

```
int factorielle (int nombre);    /* Prototype de la fonction */

main ()
{
    i = factorielle (7);        /* Vérification à l'utilisation */
}

int
factorielle ( ...              /* Définition effective */
```

Prototypage (3)

- Utiliser de façon systématique le prototypage possède de nombreux avantages
 - Séparation de la spécification d'une fonction de son implémentation
 - Programmation par contrat
 - Possibilité de compilation séparée
 - Facilité de travail en équipe
 - Double vérification de cohérence
 - Entre le prototype et les instances d'appel
 - Entre le prototype et la définition de la fonction

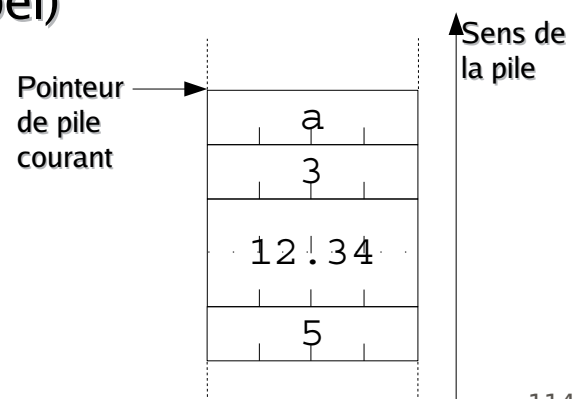
Mécanisme d'appel d'une fonction (1)

- Lorsqu'une fonction est appelée, il se produit les actions suivantes :
 - Empilement des valeurs des paramètres d'appel dans la pile de programme, en ordre inverse
 - Empilement de l'adresse de retour (adresse du code situé juste après l'appel)
 - Appel à la fonction

```
int    i = 3;
int    j = 5;
double d = 12.34;

...
f (i, d, j);
... ←
```

a : adresse de retour après le code d'appel

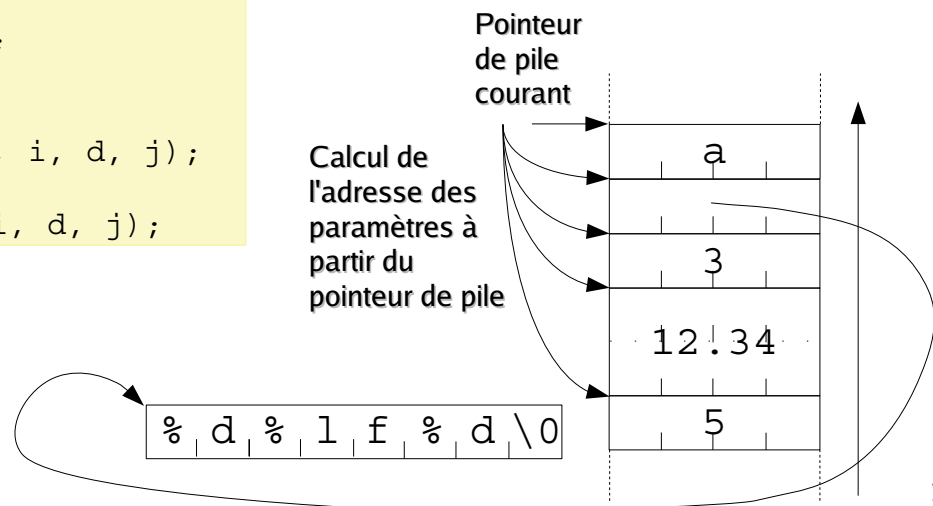


Mécanisme d'appel d'une fonction (2)

- Placer les paramètres dans l'ordre inverse permet de gérer les fonctions à nombre de paramètres variables, comme `printf`

```
int    i = 3;
int    j = 5;
double d = 12.34;

...
printf ("%d%lf%d", i, d, j);
...
printf ("%d%lf", i, d, j);
```

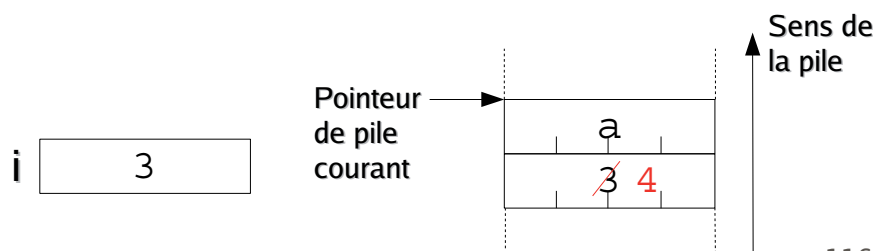


Mécanisme d'appel d'une fonction (3)

- Le passage des paramètres dans la pile s'effectue par valeur et pas par référence
 - C'est la valeur du paramètre qui est passée dans la pile, pas son adresse (sauf pour les chaînes)
 - Si on modifie un paramètre, c'est sa copie qui est modifiée, pas lui-même

```
f (int j) {
    j = 4;
}

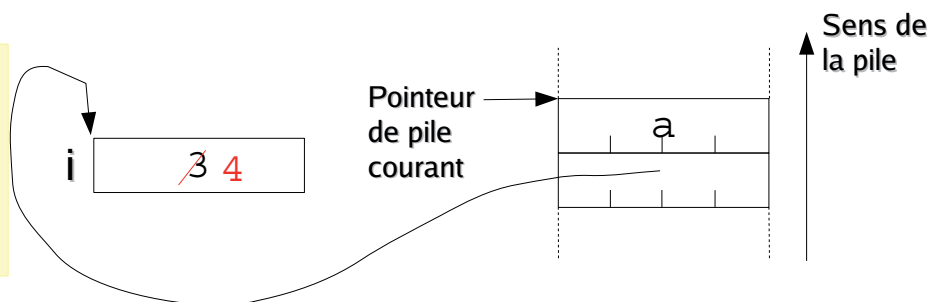
...
i = 3;
f (i);
```



Mécanisme d'appel d'une fonction (4)

- Si l'on veut pouvoir modifier la valeur d'un paramètre au delà de l'exécution de la fonction, il faut passer son adresse
 - Émulation du passage par référence
 - Exemple : `scanf`

```
f (int * jptr) {
    *jptr = 4;
}
...
i = 3;
f (&i);
```



La fonction `main` (1)

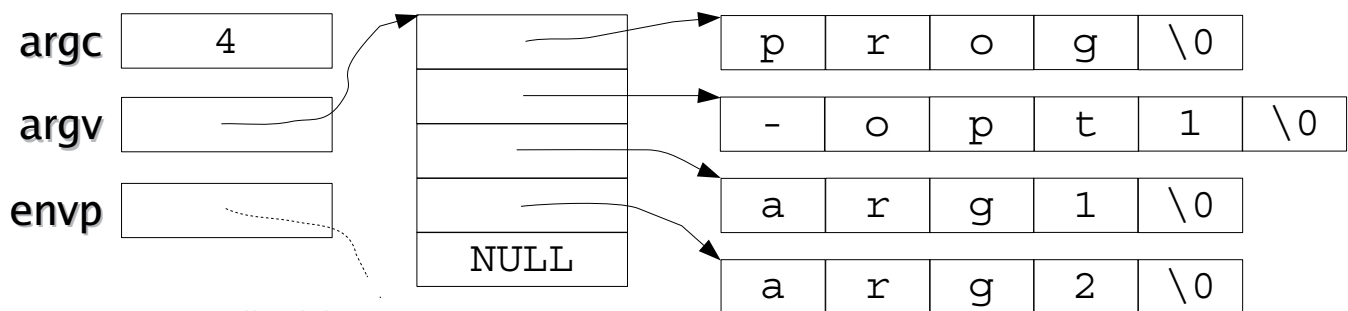
- La fonction `main` retourne un `int`
 - Code de retour du programme, renvoyé au processus appelant (en général l'interpréteur de commandes)
- La fonction `main` possède trois arguments
 - `argc`, entier spécifiant le nombre de paramètres passés à la commande, y compris son propre nom
 - `argv`, tableau de pointeurs de chaînes de caractères listant les arguments de la commande
 - `envp`, tableau de pointeurs de chaînes de caractères listant les variables d'environnement connues

La fonction main (2)

- Format complet de la fonction main

```
int
main (
int          argc,    /* Nombre d'arguments y compris le nom    */
char *      argv[],  /* Tableau des chaînes des arguments      */
char *      envp[]) /* Tableau des chaînes des variables d'env. */
{
    ...
}
```

```
% prog -opt1 arg1 arg2
```



© 2004-2010 F. Pellegrini

119

La fonction main (3)

- Exemple d'usage des paramètres de main

```
int
main (
int          argc,    /* Nombre d'arguments y compris le nom    */
char *      argv[],  /* Tableau des chaînes des arguments      */
char *      envp[]) /* Tableau des chaînes des variables d'env. */
{
    int      i;

    for (i = 0; i < argc; i++)
        printf ("Argument %d : \"%s\"\n", i, argv[i]);

    for (i = 0; envp[i] != NULL; i++)
        printf ("Environnement %d : \"%s\"\n", i, envp[i]);

    return (0);
}
```

© 2004-2010 F. Pellegrini

120

La fonction `main` (4)

- Les petits programmes se servent rarement des variables d'environnement
 - On ne spécifie pas `envp` dans la définition de `main`
 - Pas de problème car c'est le dernier argument, stocké au fond de la pile d'appel
- Format usuel de la fonction `main`

```
int
main (
int          argc,      /* Nombre d'arguments y compris le nom */
char *      argv[])   /* Tableau des chaînes des arguments */
{
    ...
}
```

Visibilité des variables (1)

- Une variable déclarée en dehors du corps d'une fonction est visible de toutes les fonctions du fichier dans lequel elle est déclarée (variable globale au fichier)

```
int i;          /* Variable visible dans tout le fichier */

f ()
{
    i = 4;      /* On la voit et on l'utilise ici */
}

main ()
{
    i = 3;      /* On la voit et on l'utilise ici */
    f ();
    printf ("%d", i);
}
```

Visibilité des variables (2)

- Une variable déclarée à l'intérieur du corps d'une fonction ou d'un bloc d'instructions n'est visible qu'à l'intérieur du corps du bloc

```
f () {  
    int i;           /* Une première variable locale i */  
    i = 4;  
}  
g () {  
    int i;           /* Une autre, différente de la première */  
    i = 6;  
}  
main ()  
{  
    f ();           /* Ici, on ne voit ni l'une, ni l'autre */  
    g ();  
}
```

Visibilité des variables (3)

- Lorsqu'une variable locale a le même nom qu'une variable globale, celle-ci est masquée à l'intérieur du bloc où la variable locale est définie

```
int i;              /* Variable globale au fichier */  
  
f () {  
    int i;          /* Variable locale de même nom, qui la masque */  
    i = 4;          /* Ici, on utilise donc la variable locale */  
}  
  
main ()  
{  
    i = 3;          /* Ici, c'est la variable globale que l'on voit */  
    f ();  
    printf ("%d\n", i); /* On affiche donc "3" */  
}
```

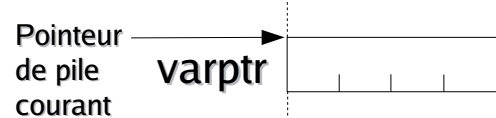
Portée des variables (1)

- Une variable locale n'existe qu'à l'intérieur du bloc dans lequel elle est définie
 - L'espace pour la variable est alloué dans la pile lors de l'entrée dans le bloc, et libéré à la sortie du bloc
 - En dehors du bloc, cet espace mémoire peut être utilisé pour stocker d'autres variables locales d'autres blocs
- Il ne faut pas référencer une variable locale à l'extérieur du contexte de pile où elle est définie
 - Le compilateur peut émettre un avertissement lorsqu'on renvoie l'adresse d'une variable locale

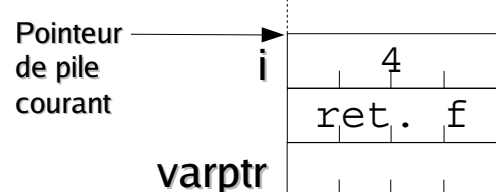
Portée des variables (2)

```
void
g ()
{
    int j;
    j = 5;
}
int *
f ()
{
    int i;
    i = 4;
    return (&i); /* Pas bon */
}
int
main ()
{
    int * varptr;
    varptr = f ();
    g ();
    printf ("%d\n", *varptr);
}
```

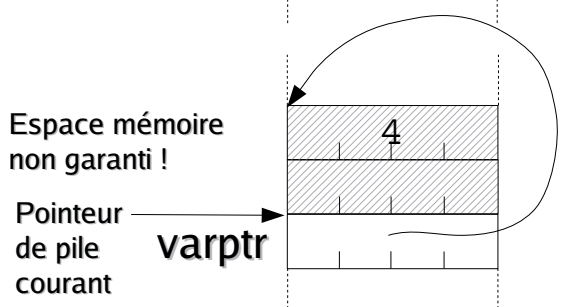
Avant l'appel de f :



À l'intérieur de f :



Après l'appel de f :



Portée des variables (3)

- Une variable locale doit donc être initialisée à chaque entrée dans le bloc qui la contient
 - Le compilateur peut émettre un avertissement si ce n'est pas le cas

```
int
f () {
    int i = 1;      /* Variable allouée et initialisée à l'exécution */
    return (i ++); /* L'incrémentaion sera perdue au retour de f */
}
main ()
{
    printf ("f : %d fois\n", f ()); /* Affiche "1" */
    printf ("f : %d fois\n", f ()); /* Affiche "1" aussi ! */
}
```

Portée des variables (4)

- Le mot clé `static`, utilisé pour qualifier une variable locale, étend sa portée à la durée du programme
 - Allouée dans le tas et pas dans la pile
 - Visibilité toujours limitée au bloc où elle est définie

```
int
f () {
    static int i = 1; /* Allouée et initialisée à la compilation */
    return (i ++);   /* L'incrémentaion sera conservée */
}
main ()
{
    printf ("f : %d fois\n", f ()); /* Affiche "1" */
    printf ("f : %d fois\n", f ()); /* Affiche "2" */
}
```

Portée des variables (5)

```
void
g ()
{
    int    j;
    j = 5;
}
int *
f ()
{
    static int    i;    /* i est déclarée dans le tas et pas la pile */
    i = 4;
    return (&i);      /* Plus de problème à écrire cela */
}
int
main ()
{
    int *    varptr;
    varptr = f ();
    g ();
    printf ("%d\n", *varptr);
}
```

Portée des variables (6)

- Pour une variable statique, l'initialisation à la déclaration a un sens particulier

```
int
f () {
    static int    i = 1;    /* Allouée et initialisée à la compilation */
    return (i ++);        /* L'incrémentatation sera conservée */
}
int
g () {
    static int    i;        /* Allouée à la compilation */
    i = 1;            /* Initialisée à l'exécution */
    return (i ++);        /* L'incrémentatation ne sera pas conservée */
}
int
h () {
    int    i = 1;        /* Allouée et initialisée à l'exécution */
    return (i ++);        /* L'incrémentatation ne sera pas conservée */
}
```

Allocation dynamique

- On a souvent besoin d'utiliser de la mémoire qu'on ne peut pas réserver à l'avance parce qu'on n'en connaît pas la taille à la compilation
 - Données lues à partir d'un fichier, du clavier, etc.
- Nécessité de pouvoir réserver dynamiquement de la mémoire dans le tas à l'exécution
- Le tas contient donc de la mémoire
 - Réservée et initialisée à la compilation
 - Réservée à la compilation mais non initialisée
 - Non réservée (tout le reste, jusqu'à la pile)

La fonction `malloc` (1)

- Permet de réserver un certain nombre d'octets dans le tas
 - On passe en paramètre le nombre d'octets souhaités (utiliser l'opérateur `sizeof` pour calculer la taille à partir du nombre d'objets)
 - La fonction renvoie un pointeur sur le début de la zone allouée, ou `NULL` si plus de mémoire

```
int    tabnbr;           /* Nombre d'éléments dans le tableau */
int *  tabptr;          /* Pointeur sur le tableau des éléments */
printf ("Entrez le nombre d'éléments : ");
scanf ("%d", &tabnbr); /* Lecture du nombre d'éléments */
tabptr = malloc (tabnbr * sizeof (int)); /* Allocation */
for (i = 0; i < tabnbr; i++) /* Chargement des éléments */
    scanf ("%d", &tabptr[i]); /* Ou aussi : « tabptr + i » */
```

La fonction `malloc` (2)

- Il est important de tester la valeur de retour de la fonction `malloc` pour détecter au plus tôt les problèmes
 - Test avec traitement d'erreur dans le cadre du déroulement du programme
 - Assertion avec la macro `assert`

```
if ((tabptr = malloc (tabnbr * sizeof (int)) == NULL) {  
    ...                               /* Gestion de l'erreur */  
}
```

```
tabptr = malloc (tabnbr * sizeof (int));  
assert (tabptr != NULL);           /* Termine le programme si pas vrai */
```

La fonction `free`

- Permet de libérer une zone mémoire allouée avec `malloc`
 - Ne pas utiliser avec les zones non allouées par `malloc`
 - Ne pas appeler `free` plusieurs fois sur la même zone

```
free (tabptr);
```

- Afin d'éviter les fuites mémoire, il est préférable d'appeler `free` dès que la zone mémoire considérée n'est plus utilisée

Erreurs avec l'allocation dynamique (1)

- L'allocation dynamique est un mécanisme fragile
 - Blocs libres et occupés chaînés par des pointeurs situés avant et après chaque bloc, facilement corrompus en cas d'écriture en dehors des bornes
 - Pas de vérifications de cohérence afin de conserver l'efficacité des routines
- Les erreurs peuvent n'être détectées que longtemps après l'instruction qui les cause
 - *Segmentation fault* dans un `malloc` parce que le `free` précédent s'est fait sur un bloc corrompu

Erreurs avec l'allocation dynamique (2)

- Existence de bibliothèques et d'outils destinés à détecter les erreurs d'accès à la mémoire
 - Bibliothèques de gestion mémoire instrumentées pour détecter les écritures en dehors des bornes
 - Nécessité de compiler spécifiquement avec ces bibliothèques ou positionnement de variables d'environnement pour certaines autres (variable `MALLOC_CHECK_` sous Linux)
 - Outils de vérification dynamique de chaque accès mémoire lors de l'exécution, tels Valgrind ou Purify
 - Purify est cher et nécessite une compilation particulière
 - Valgrind ne fonctionne que sous Linux

Types énumérés (1)

- Une énumération est un sous-ensemble du type `int` auquel est associé un nombre fini de valeurs symboliques

```
enum Feu_ {
    FEU_VERT,
    FEU_ORANGE,
    FEU_ROUGE
} ;
```

/ Nom de l'énumération */*
/ Constantes symboliques */*
/ En majuscules de préférence */*
/ Ici le ";" est obligatoire */*

Types énumérés (2)

- On définit des variables de types énumérés comme des variables de types simples

```
enum Feu_ f1, f2;
```

- On peut aussi définir des variables lors de la définition de l'énumération

```
enum Feu_ {
    FEU_VERT,
    FEU_ORANGE,
    FEU_ROUGE
} f1, f2;
```

/ Voilà pourquoi le ";"... */*

Types énumérés (3)

- On utilise les valeurs symboliques comme des constantes numériques

```
switch (f1) {
    case FEU_VERT :
        je_passe ();
        break;
    case FEU_ORANGE :
        if (trop_pres ()) {
            je_passe ();
            break;
        }
        /* Notez l'absence de "break" ici */
    default :
        /* FEU_ROUGE */
        je_stoppe ();
}

if (f2 == FEU_ORANGE)
    ...
```

Types énumérés (4)

- Par défaut, les constantes sont numérotées consécutivement à partir de 0
- On peut spécifier explicitement leur valeur au moyen de l'opérateur « = »

```
enum Premiers10_ {
    UN = 1,
    DEUX,
    TROIS,
    CINQ = 5,
    SEPT = 7
};
/* Cette constante vaut 1 et pas 0 */
/* On numérote consécutivement, donc 2 */
/* Pareil : celle-là vaut 3 */
/* Celle-ci vaut 5 */
/* Et celle-là vaut 7 */
```

Types structurés

- Les types structurés permettent de regrouper au sein d'une même entité des variables ayant entre elles des liens fonctionnels
 - Un type structuré est défini par la liste des variables qu'il contient, appelées « champs »
- Ils améliorent la concision, la lisibilité et l'évolutivité des programmes
 - Simplifient la définition des variables
 - Simplifient le passage de paramètres
 - Pas de modification des appels si les champs changent

Structure (1)

- Définie par le mot clé `struct`, suivi du nom donné à la structure et de la liste de ses champs, entre accolades

```
struct Point_ {  
    double      x, y, z;    /* Coordonnées spatiales */  
};  
  
struct Planete_ {  
    int         type;      /* Type de planétoïde          */  
    double      m;        /* Masse du planétoïde       */  
    struct Point_ pos;    /* Position du planétoïde    */  
}; /* Ici le ";" est obligatoire */
```

Structure (2)

- On définit les variables de types structurés comme les variables de types simples

```
struct Planete_ p1, p2;
```

- On peut aussi définir des variables lors de la définition de la structure

```
struct Planete_ {  
    int         type;          /* Type de planétoïde      */  
    double      m;            /* Masse du planétoïde    */  
    struct Point_ pos;        /* Position du planétoïde */  
} p1, p2;                    /* Voilà pourquoi le ";"... */
```

Structure (3)

- On accède aux champs d'une structure au moyen de l'opérateur « . »

```
struct Planete_ plandat;  
double          masse;  
  
plandat.m       = 1e30;  
plandat.pos.x   = 12.34;  
plandat.pos.y   = 56.78;  
plandat.pos.z   = 90.12;  
...  
masse = plandat.m;
```

Structure (4)

- On accède aux champs d'une référence de structure au moyen de l'opérateur « -> »

`sptr->champ` \Leftrightarrow `(*sptr).champ`

```
struct Planete_ * planptr;
double          masse;

planptr->m      = 1e30; /* planptr est une référence, donc "->" */
planptr->pos.x  = 12.34; /* planptr->pos est un struct Point_ , */
planptr->pos.y  = 56.78; /* donc on utilise "." */
planptr->pos.z  = 90.12;
...
masse = planptr->m;
```

Structure (5)

- On peut utiliser les variables de type structure comme des variables de types simples

```
struct Planete_ p1, p2;
p1.pos.x = 12.34;
p2       = p1; /* Équivalent à une copie mémoire entre zones */
```

- On peut initialiser des structures en utilisant les accolades pour chaque structure et sous-structure

```
struct Planete_ p1 = { 1, 1e30, { 12.34, 56.78, 90.12 } };
```

Structure (6)

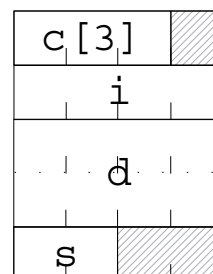
- Pour passer des types structurés en paramètres d'une fonction, il faut mieux utiliser le passage par référence
 - Pas de recopie de l'intégralité des structures dans la pile lors de l'appel

```
void
planeteAffiche (
struct Planete_ *   planptr)
{
    printf ("Type : %d\nMasse : %lf\nPosition : (%lf,%lf,%lf)\n",
            planptr->type, planptr->masse,
            planptr->pos.x, planptr->pos.y, planptr->pos.z);
}
...
planeteAffiche (&plandata);
```

Structure (7)

- Les données de la structure sont placées les unes après les autres en mémoire dans l'ordre dans lesquelles elles sont définies, aux alignements de types près
 - La taille de la structure peut être supérieure à la somme des tailles de ses membres
 - La taille d'une structure peut varier selon la machine

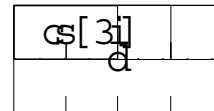
```
struct Brol_ {
    char    c[3];
    int     i;
    double  d;
    short   s;
}
```



Union (1)

- Définie par le mot clé `union`, suivi du nom donné à la structure et de la liste de ses champs, entre accolades
- Analogue à la structure, sauf que les données partagent le même espace mémoire
 - Sert à stocker différents types de données dans le même objet sans consommer d'espace inutile

```
union Brol_ {
    char    c[3];
    int     i;
    double  d;
    short   s;
}
```



Union (2)

```
enum Type_ {
    TYPE_ENTIER,
    TYPE_FLOTTANT
};
struct Valeur_ {
    enum Type_ type;
    union {
        int     i;
        double  d;
    } val;
};
...
switch (valptr->type) { /* valptr est de type struct Valeur_ * */
    case TYPE_ENTIER :
        printf ("%d", valptr->val.i);
        break;
    case TYPE_FLOTTANT :
        printf ("%lf", valptr->val.d);
        break;
}
```

Définition de types (1)

- On définit un nouveau type au moyen du mot clé `typedef`
`typedef ancien_type nouveau_type`
- Permet de dissocier la définition des variables de leur implémentation
 - Typage fonctionnel et non plus matériel

```
typedef float Age;      /* Définition : un Age est un float */

main ()
{
    Age    a;          /* On déclare a comme un Age    */
    a = 18.5F;        /* On utilise a comme un float */
}
```

Définition de types (2)

- `typedef` permet de cacher l'implémentation du nouveau type ainsi défini (type simple, structure, union ou énumération)

```
struct Point_ {          /* Définition et nommage de la structure */
    double    x;
    double    y;
};

typedef struct Point_ Point; /* Définition du type */

main ()
{
    Point    p;          /* Définition d'un point; ne sait pas ce que c'est */
    p.x = 12.34;
    p.y = 45.67;
}
```

Définition de types (3)

- On peut effectuer la définition de la structure et du type en même temps

```
typedef struct Point_ { /* Définition de la structure */
    double    x;
    double    y;
} Point;           /* Nommage du type */

typedef enum Booleen_ {
    FAUX,
    VRAI
} Booleen;

typedef struct Cellule_ {
    struct Cellule_ * precptr; /* "struct" car type pas défini */
    struct Cellule_ * suivptr;
    CelluleDonnees    celldat;
} Cellule;           /* À partir d'ici le type est défini */
```

Préprocesseur

- La phase de compilation proprement dite (cc1) est toujours précédée d'une phase de pré-traitement (« *preprocessing* », cpp)
- Le préprocesseur reçoit en entrée le flot de caractères du fichier source .c et renvoie en sortie un flot de caractères modifié en fonction des directives qu'il y trouve
 - « gcc -E » permet de voir ce que renvoie cpp
- Les directives du préprocesseur commencent par un « # » situé en premier caractère

Directive #include (1)

- La directive #include remplace la ligne qui la contient par le contenu du fichier spécifié
- Deux formes
 - #include <fichier> : inclusion de fichier fourni par le système, en général à partir du répertoire système /usr/include
 - #include "fichier" : inclusion de fichier fourni par l'utilisateur, en général à partir du répertoire courant
 - Les autres répertoires dans lesquels chercher le fichier sont définis au moyen de l'option « -I » du compilateur

Directive #include (2)

- Les fichiers d'en-tête fournis par le système permettent d'interfacer les programmes d'application avec le système d'exploitation
 - Constituent l'API de la bibliothèque libc
 - Respectent maintenant la norme Posix

```
#include <stdio.h> /* Définition de FILE, stdin, printf ... */
#include <stdlib.h> /* Définition de random, malloc, exit ... */
#include <math.h> /* Définition de PI, E, cos, atan ... */
#include <ctype.h> /* Définition de isupper, toupper, isalpha ... */
#include <limits.h> /* Définition des tailles de int, long ... */
#include <stdarg.h> /* Définition des fcts à arguments variables */
```

Inclusion de fichiers (1)

- L'inclusion de fichiers est nécessaire à la programmation par contrat
 - En plus du code binaire des bibliothèques qu'on utilise, il faut disposer des prototypes des fonctions pour que le compilateur puisse générer les appels
 - Ces prototypes doivent être définis dans tout fichier « .c » qui appelle les fonctions
 - Nécessité de les inclure à partir d'un fichier spécifique fourni avec la bibliothèque
- Définition de fichiers d'en-tête (« *headers* ») d'extension « .h »

Inclusion de fichiers (2)

- Les fichiers d'en-tête contiennent :
 - Des définitions de constantes et de macros
 - Des définitions de types
 - Des prototypes de fonctions
- Ils ne doivent pas contenir d'autres inclusions de fichiers d'en-tête
 - Solution de facilité génératrice d'erreurs
 - On doit pouvoir voir quels fichiers « .h » sont utilisés dans un fichier « .c »
 - Réservé aux fichiers système...

Inclusion de fichiers (3)

- Lorsqu'on crée un module « .c », on a en fait besoin de deux fichiers d'en-tête « .h » :
 - Un fichier « .h » interne, contenant les définitions et prototypes nécessaires aux fonctions internes au module
 - Fichier « module_private.h »
 - Un fichier « .h » externe, contenant les définitions et prototypes de l'interface du module vis-à-vis des modules clients
 - Fichier « module.h »
- Pour les modules internes, on les fusionne

Directive #define (1)

- La directive #define permet de définir des macros, avec ou sans paramètres
 - Lorsque le préprocesseur reconnaît le nom d'une macro, il la remplace par sa définition

```
#define PI          3.1415926
#define PISUR4     (PI/4.0)
...
if (f () <= PISUR4) {
    ...
}
```

- Une macro n'est pas une construction du langage C ; il ne faut pas de « ; » à la fin

Directive #define (2)

- Lors de la définition d'une macro avec paramètres, il faut coller la parenthèse ouvrante au nom de la macro
 - Les arguments sont remplacés textuellement
 - Vérifications de syntaxe à la compilation seulement

```
#define MIN(a,b)  (((a)<(b))?(a):(b))  /* a et b pas des variables */  
...  
a = MIN (i, j);  /* Le a ici n'est pas le a de la macro */
```

Directive #define (3)

- Le nommage des macros comme des fonctions est pratique, mais peut être dangereux à cause des effets de bords

```
#define MIN(a,b)  (((a)<(b))?(a):(b))  /* a et b pas des variables */  
...  
a = MIN (b ++, c ++); /* Donne ((b ++)<(c ++))?(b ++):(c ++) */
```

- Les identifiants des macros doivent être entièrement en majuscules pour les différencier des fonctions classiques

Directive #define (4)

- Une définition peut s'étendre sur plusieurs lignes
- Lors du remplacement textuel des arguments, la directive « ## » permet de joindre les deux termes accolés pour faire un seul identifiant

```
#define FONCTIONPLUS(type)      \  
type                            \  
plus_##type (                  \  
type a,                         \  
type b)                          \  
{                                \  
    return (a + b);             \  
}  
...  
FONCTIONPLUS (int)              /* Crée la fonction plus_int () */  
FONCTIONPLUS (float)           /* Crée la fonction plus_float () */
```

Directive #undef (1)

- La directive #undef permet de supprimer une définition de macro
- Une fois la macro supprimée, l'identificateur correspondant à la macro ne fait plus l'objet d'aucun remplacement de la part du préprocesseur

```
#undef FONCTIONPLUS  
...  
FONCTIONPLUS (int)              /* Donne : "FONCTIONPLUS (int)" */  
FONCTIONPLUS (float)           /* Donne : "FONCTIONPLUS (float)" */
```

Directive #ifdef (1)

- La directive #ifdef permet de fournir ou non certaines parties du code au compilateur en fonction de la définition ou non d'une macro
- Format
 - #ifdef ... #endif
 - #ifdef ... #else ... #endif

```
#ifdef TYPE_LONG
typedef long Entier;
#define additionne additionne_long
#else /* TYPE_LONG */          /* On le met pour faciliter la lecture */
typedef int Entier;
#define additionne additionne_int
#endif /* TYPE_LONG */        /* Et ici aussi */
```

Directive #ifdef (2)

- La sélection du #endif ou du #else correspondant à un #ifdef se fait en prenant le premier rencontré

```
#ifdef COND1
#ifdef COND2          /* Ceci ne marche pas... */
#endif /* COND1 */   /* Ce "#endif" va avec "#ifdef COND2" */
...
#endif COND1
#endif /* COND2 */
#endif /* COND1 */
```

Directive #if (1)

- La directive #if remplit la même fonction que la directive #ifdef mais elle se base sur l'évaluation d'une expression logique mêlant macros et constantes numériques

```
#define VERSION 2.0
#define TAILLE_INT      8
...
#if ((VERSION >= 1.3) || (TAILLE_INT == 8))
typedef long  Entier;
...
#endif /* ((VERSION >= 1.3) || (TAILLE_INT == 8)) */
```

Directive #if (2)

- Au sein de l'expression logique évaluée par une directive #if, le mot-clé « defined » permet de tester l'existence d'une macro

#ifdef ↔ #if defined

```
#define VERSION 2.0
#define DEBUG
...
#if ((VERSION > 1.0) && (! defined DEBUG))
#undef assert
#define assert(expr)  ()
#endif /* ((VERSION > 1.0) || (! defined DEBUG)) */
```

Définition par la ligne de commande

- Les macros peuvent être définies ou supprimées :
 - À l'intérieur des fichiers d'en-tête
 - Options de configuration positionnées par le développeur pour une utilisation optimale ou par défaut de son module
 - Au moment de la compilation
 - Options « -Dnom », « -Dnom=valeur », « -Unom » de la ligne de commande de `gcc`, par exemple

Compilation multi-fichiers (1)

- Trois étapes distinctes s'enchaînent au cours de la compilation
 - Pré-traitement du code source (`cpp`)
 - Inclusion de sous-fichiers
 - Remplacement de macros
 - Compilation du code source pré-traité (`cc`)
 - On a un fichier objet pour chaque fichier source compilé
 - Édition de liens (`ld`)
 - Fusion de tous les fichiers objets pour créer le programme binaire exécutable

Compilation multi-fichiers (2)

- Schéma réel

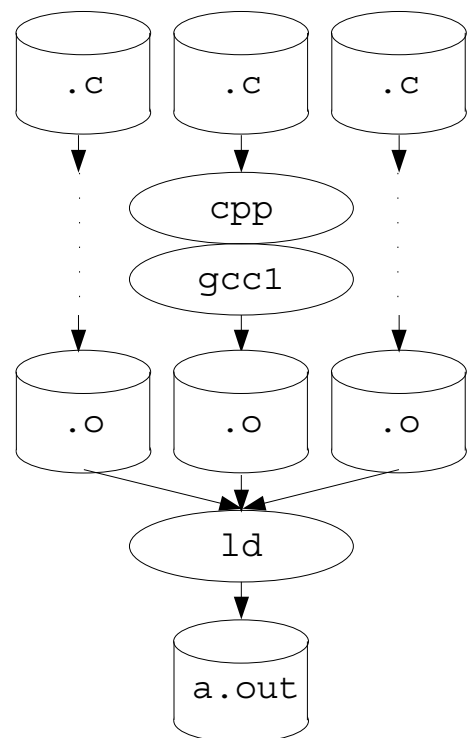
Fichiers de code source en langage C

Préprocesseur
Compilateur proprement dit

Fichiers objet

Éditeur de liens (*linker*)

Fichier exécutable



Compilation multi-fichiers (3)

- On limite la compilation à la seule production d'un fichier objet « .o » au moyen de l'option « -c » du compilateur
- Pour effectuer l'édition de liens, on peut aussi passer par le compilateur, en lui fournissant en entrée la liste des fichiers « .o » à lier

```

% gcc -c brol.c -o brol.o
% gcc -c trol.c -o trol.o
% gcc -c chmol.c -o chmol.o
% gcc brol.o trol.o chmol.o -o brol
  
```

- On peut automatiser la compilation en utilisant un script ou le programme `make`

Compilation multi-fichiers (4)

- Pour les gros projets organisés en sous-répertoires, il faut pouvoir inclure les fichiers d'en-tête et les bibliothèques provenant d'autres répertoires
- L'option « `-I` » du compilateur permet d'augmenter la liste des répertoires dans lesquels chercher les fichiers d'en-tête
- L'option « `-L` » du compilateur permet d'augmenter la liste des répertoires dans lesquels chercher les fichiers de bibliothèques

Variables globales (1)

- Par défaut, une variable déclarée en dehors d'une fonction est visible dans tout le programme
 - C'est une variable globale car sa portée est globale
 - Mais les autres fichiers ne savent pas qu'elle existe
- Il y a conflit à l'édition de liens si deux fichiers « `.c` » possèdent chacun une variable globale de même nom
 - Deux déclarations
 - Mais surtout deux définitions !

VARIABLES GLOBALES PRIVÉES

- Pour empêcher qu'une variable globale d'un fichier ne puisse être vue d'un autre fichier, on la déclare avec le qualificatif « `static` »
 - Évite les risques de conflits de nommage entre variables de modules différents
 - C'est un autre sens de « `static` », différent de celui utilisé pour les variables locales, car les variables globales sont de toute façon toujours allouées dans le tas

```
static int compteur = 0;      /* Compteur global au fichier */
int f () {
    return (compteur ++);
}
```

VARIABLES GLOBALES PARTAGÉES (1)

- Pour que plusieurs modules utilisent la même variable globale, il faut que :
 - Chaque module la déclare, pour que le compilateur puisse la connaître à chaque fois
 - Un seul module la définisse
- La déclaration sans définition d'une variable globale se fait avec le qualificatif « `extern` »

```
int i; /* Déclarée et définie */ brol.c
extern int i; /* Déclarée mais pas définie */ trol.c
...
i = 3; /* C'est bien le "i" de brol.c qu'on modifie */
```

Variables globales partagées (2)

- Toute variable globale doit appartenir à un module, qui offre l'accès à cette variable
 - Déclaration de la variable comme « extern » dans le fichier « .h » du module
 - Définition de la variable (sans « extern ») dans le fichier « .c » du module

```
extern int    i;          /* Déclaration de la variable */          bro1.h
#include "bro1.h"
...
int    i;          /* Définition de la variable déjà déclarée */
#include "bro1.h"
...
i = 3;          /* Utilisation de la variable */          trol.c
```

Récapitulatif sur les variables

- Lieu de définition, portée et visibilité des variables en fonction de leurs caractéristiques de déclaration

Lieu de déclaration	Qualificatif		
	(auto)	static	extern
Fonction	Pile	Tas	X
	Fonction	Programme	
	Fonction	Fonction	
Fichier	Tas	Tas	Doit être définie dans un autre fichier
	Programme	Programme	
	(Programme)	Fichier	Fichier

Fonctions privées

- Pour empêcher qu'une fonction d'un fichier ne puisse être vue d'un autre fichier, on la déclare avec le qualificatif « `static` »
 - Évite les risques de conflits de nommage entre fonctions de modules différents

```
static
int
factorielle2 (
int          n)
{
    return ((n < 2) ? 1 : (n * factorielle2 (n - 1)));
}
int
factorielle (
int          n)
{ return ((n < 0) ? -1 : factorielle2 (n)); }
```

© 2004-2010 F. Pellegrini

179

Manipulation des fichiers (1)

- La manipulation des fichiers s'effectue au moyen de fonctions normalisées de la `libc`
- Fonction d'ouverture d'un fichier
 - Prend un chemin dans le système de fichiers et renvoie un descripteur identifiant de façon unique le fichier ouvert au sein du processus
- Fonctions de lecture/écriture/positionnement
 - Utilisent le descripteur
- Fonction de fermeture
 - Clôt le fichier et inactive le descripteur

© 2004-2010 F. Pellegrini

180

Manipulation des fichiers (2)

- Fonctions de bas niveau
 - Manipulent des descripteurs de fichiers (« *file descriptors* ») de type `int`
 - Fonctions `open`, `close`, `read`, `write`, `lseek` ...
 - Non bufferisées : 1 appel = 1 appel système
- Fonctions de haut niveau
 - Manipulent des pointeurs de flots de type `FILE` *
 - Fonctions : `fopen`, `fclose`, `fread`, `fscanf`, `fwrite`, `fprintf`, `fseek`, `fflush` ...
 - Bufferisées : groupage des appels systèmes

Fonction `fopen` (1)

- Format
 - `fp = fopen("chemin", "mode")`
 - `fp = fopen("chemin", "mode", droits)`
- Le mode peut être :
 - `"r"` : lecture seule d'un fichier préexistant
 - `"r+"` : lecture/écriture d'un fichier préexistant
 - `"w"` : écriture seule d'un fichier nouveau/tronqué
 - `"w+"` : lecture/écriture d'un fichier nouveau/existant
 - `"a"` : ajout : écritures valides à la fin seulement

Fonction `fopen` (2)

- Si le fichier n'existe pas déjà, le champ `droits` spécifie les droits du nouveau fichier
 - Notation octale : 0644, 0600, 0755, etc.
- La fonction `fopen` renvoie un pointeur sur une structure `FILE`, définie dans `<stdio.h>`

```
#include <stdio.h>
...
FILE *    fp;
...
if ((fp = fopen ("../config", "r+")) == NULL) {
    printf ("Impossible d'ouvrir le fichier de configuration\n");
    return (1);
}
```

Fonction `fclose`

- **Format**
 - `fclose (fp)`
- Sert à fermer le flot associé au pointeur
- Toujours fermer un flot dès qu'on le peut
 - Nombre limité de flots ouverts par processus
- Toujours ne le fermer qu'une seule fois

Fonctions fread et fwrite

- **Format**
 - `nbrlu = fread (ptr, taille, nbr, flot)`
 - `nbrécrit = fwrite (ptr, taille, nbr, flot)`
- **Servent à lire et écrire des blocs de taille octets en mémoire à partir de l'adresse ptr**

```
Mastruct * sttab;  
int      stnbr;  
FILE *   fp;  
...  
if (fwrite (sttab, sizeof (Mastruct), stnbr, fp) < stnbr) {  
    printf ("Impossible de sauvegarder le vecteur\n");  
    fclose (fp);  
    return (1);  
}
```

Fonctions fscanf et fprintf

- **Format**
 - `fscanf (fp, "format", argument1, argument2, ...)`
 - `fprintf (fp, "format", argument1, argument2, ...)`
- **Fonctionnent comme scanf et printf, mais opèrent à partir d'un flot quelconque**

```
int  datanbr;          /* Nombre d'éléments      */  
int  datatab[TAILLEMAX]; /* Tableau d'au moins ce nombre */  
...  
if ((fscanf (fp, "%d", &datanbr) < 0) ||  
    (datanbr < 0) ||  
    (datanbr > TAILLEMAX)) {  
    ...  
    for (i = 0; i < datanbr; i ++)  
        if (fscanf (fp, "%d", &datatab[i]) < 0) ...
```

Pointeurs de flots standards

- Trois pointeurs de flots standards existent au démarrage de tout processus
 - `stdin` : flot standard d'entrée (clavier)
`scanf (... ⇔ fscanf (stdin, ...`
 - `stdout` : flot standard de sortie (écran)
`printf (... ⇔ fprintf (stdout, ...`
 - `stderr` : flot standard d'erreur (écran aussi)
- Fermés automatiquement à la terminaison du processus

Conventions de codage (1)

- Un module consiste en la définition d'un ou plusieurs fichiers « `.c` » possédant tous le même nom de base, représentatif du service rendu par le module
 - Par exemple pour un module de gestion de matrices, on pourra avoir : `matrice.c`,
`matrice_es.c`, `matrice_addition.c`,
`matrice_verifie.c`, etc.
- À chaque module doit correspondre un fichier d'en-tête externe donnant l'API du module
 - Par exemple : `matrice.h`

Conventions de codage (2)

- Lorsque, dans un module, on définit un type, les fonctions manipulant ce type doivent être préfixées par le nom du type suivi du type de la méthode appliquée
 - Exemple : pour le type `Matrice` dérivant de la structure `struct Matrice_`, on aura les fonctions `matriceInit`, `matriceLit`, `matriceEcrit`, `matriceVerifie`, etc.
- Il est préférable que ces noms de méthodes (`Init`, `Lit`, `Ecrit`, `Verifie`, etc.) soient normalisés pour le plus de types possibles

Conventions de codage (3)

- Les fonctions prennent de préférence comme paramètre une référence sur le type, pour laisser à l'appelant la liberté de choisir le mode de stockage de ses objets

```
f ()
{
    Matrice    m1;           /* Structure allouée dans la pile          */
    Matrice *  m2;           /* Pointeur pour allocation dans le tas    */
    FILE *     fp1, * fp2;
    ...
    matriceLit (&m1, fp1); /* La fonction marche */
    ...
    m2 = malloc (sizeof (Matrice));
    matriceLit (m2, fp2); /* La fonction marche aussi */

    matriceEcrit (m2, stdout); /* Généricité par rapport aux flots */
}
```

Pointeurs de fonctions (1)

- Une fonction, comme tout objet manipulé par un programme, possède une adresse qui l'identifie de façon univoque
 - C'est l'adresse de début de la fonction, spécifiant l'emplacement en mémoire de la première instruction de la fonction
 - L'opération de base sur une fonction n'est pas la lecture/écriture du contenu mais l'exécution

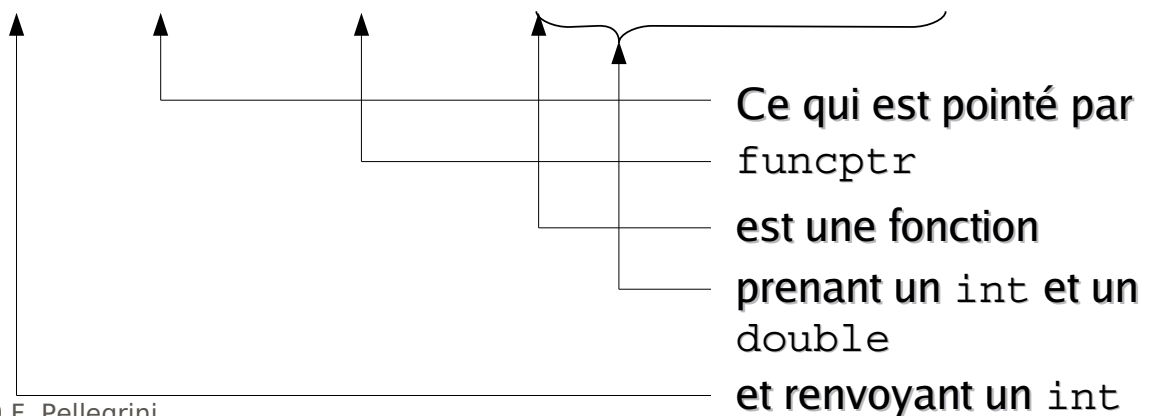
Pointeurs de fonctions (2)

- Analogie tableaux/fonctions
 - Dans le cas des tableaux, $t[i]$ représente la valeur retournée par la lecture de la $i^{\text{ème}}$ case du tableau, dont l'adresse de début est t
 - Dans le cas des fonctions, $f(i)$ représente la valeur retournée par l'exécution, avec comme paramètre i , de la fonction dont l'adresse de début est f
- Dans certains langages, on ne différencie pas l'un de l'autre dans la syntaxe
 - Les parenthèses servent aux deux

Pointeurs de fonctions (3)

- Comme pour les tableaux, on déclare un type pointeur de fonction en spécifiant le type de ce qui est renvoyé, ainsi que le type des arguments, au moyen de l'opérateur « * »

```
int (* funcptr) (int, double);
```



Pointeurs de fonctions (4)

- Les parenthèses autour du « * » sont nécessaires à la définition
 - « int (* funcptr) (int, double); » définit un pointeur sur une fonction renvoyant un int
 - « int * funcptr (int, double); » définit le prototype d'une fonction renvoyant un pointeur d'int
- On lit toujours une déclaration en partant du nom de l'objet déclaré et en allant vers l'extérieur de la déclaration

Pointeurs de fonctions (5)

```
typedef struct OpTable_ { /* Type table d'opérations */
    char        typeval;
    double      (* funcptr) (double, double);
} OpTable;

OpTable optable[] = { { '+',  func_add }, /* Table d'opérations */
                     { '*',  func_mul },
                     { '\0', NULL } }; /* Fin de table */

main (
int     argc,
char *  argv[])
{
    double  v1, v2, r;
    int     i;

    for (i = 0; optable[i].typeval != '\0'; i++) {
        if (optable[i].typeval == argv[2][0]) /* Si opération trouvée */
            printf ("%s %c %s = %lf\n", /* Calcule le résultat */
                argv[1], argv[2][0], argv[3],
                optable[i].funcptr (atof (argv[1]), atof (argv[3])));
    }
}
```

Précédence des opérateurs (2)

Opérateur	Signification	Associativité
()	Parenthésage	G → D
[] -> .	Accès	G → D
! ~ ++ -- + -	Négation, inc/décrément, unaire	D → G
* & sizeof (type)	Dé/référencement, conversion	D → G
* / %	Mult/div/modulo	G → D
+ -	Addition/soustraction	G → D
<< >>	Décalage	G → D
< <= >= > == !=	Comparaison	G → D
& ^	Opérations booléennes	G → D
&&	Opérations logiques	G → D
? :	Opérateur ternaire	D → G
= += -= *= /= %= >>= <<= &= ^= =	Affectation	D → G
,	Suite d'expressions	G → D

Normes du langage C

- Normalisation du langage par l'ANSI
- Norme C78
- Norme C89
 - Nouvelle définition des fonctions (l'actuelle...)
 - Qualificatifs « const », « volatile »
- Norme C99
 - Types « long long », « complex »
 - Qualificatif « restrict »
 - Définitions dans les initialisations de boucles

Qualificatif const

- Indique au compilateur que le contenu de la variable ne doit pas changer
 - Permet de vérifier que c'est le cas
 - Autorise des optimisations pour le compilateur

```
const int i = 12; /* Création de constante sans le préprocesseur */  
  
int f (const int i, char * const p, const int * const q);  
  
int  
f (  
const int i, /* i ne sera pas modifié dans la fonction */  
char * const p, /* p garanti constant mais pas son contenu */  
const int * const q) /* q et son contenu seront constants */  
{  
    *p = 34; /* Écriture valide */  
    *q = 56; /* Produira une erreur à la compilation */  
}
```

Qualificatif volatile

- Indique au compilateur que le contenu de la variable peut être modifié de façon asynchrone
 - Évite les optimisations du compilateur
 - Sert à l'écriture des programmes multi-threadés tels que les pilotes de périphériques

```
volatile int contval = 1; /* Drapeau de continuation */
volatile int termval = 0; /* Drapeau de terminaison */

calcul () {
    while (contval == 1) /* Tant qu'on peut continuer */
        ... /* Effectue le calcul */
    termval = 1; /* Indique qu'on a terminé */
}
...
contval = 0; /* Demande la terminaison du calcul */
while (termval == 0) ; /* Attend la fin pour lire le résultat */
```

Qualificatif restrict

- Indique au compilateur que ce qui est pointé par le pointeur ne pointe jamais au même endroit que d'autres pointeurs restrict
 - Permet les optimisations d'« *anti-aliasing* »
 - Programme faux si utilisé de façon incorrecte !

```
void
vector_add ( /* Ajoute le vecteur source au vecteur destination */
int * restrict dst, /* s et d ne pointent pas sur la même zone */
int * restrict src, /* On peut alors même mettre src en « const » */
int
nbr)
{
    int n;
    int * restrict s;
    int * restrict d;
    for (d = dst, s = src, n = nbr; n > 0; n --)
        *d ++ += *s ++; /* Optimisations par blocs de la boucle */
```

Ancienne définition des fonctions

- Dans l'ancienne norme C :
 - Le type des paramètres ne faisait pas partie de la définition des fonctions
 - Les prototypes n'acceptaient que le type de retour et pas le type des paramètres

```
void    memcpy ();          /* Prototype sans le type des arguments */
int     (* funcptr) ();     /* Pareil pour le pointeur de fonction */

void
memcpy (dst, src, nbr)     /* Pas de types pour les paramètres */
void *  dst;               /* Paramètres définis après la ")" */
void *  src;
int     nbr;
{
    /* Début de la fonction */
    ...
}
```

Bibliothèques (1)

- Lorsqu'on fournit un ensemble de fonctions rendant un ensemble cohérent de services, il serait préférable de grouper les fichiers objets associés sous la forme d'un unique fichier
 - Facilite la manipulation et la mise à jour
- Ce service est rendu par les fichiers de bibliothèque
 - Fichiers servant à archiver des fichiers objet
 - Utilisables par l'éditeur de liens

Bibliothèques (2)

- Deux types de bibliothèques
 - Bibliothèques statiques
 - Format en « lib*.a » (Unix) ou « *.lib » (DOS)
 - Liées à l'exécutable lors de la compilation
 - Augmentent (parfois grandement) la taille des exécutables
 - On n'a plus besoin que de l'exécutable proprement dit
 - Bibliothèques dynamiques
 - Format en « lib*.so » (Unix, « *shared object* ») ou « *.dll » (Windows, « *dynamic loadable library* »)
 - Liées à l'exécutable lors de l'exécution
 - Permettent la mise à jour indépendante des bibliothèques
 - Problème si pas présentes (variable « LD_LIBRARY_PATH »)

Bibliothèques (3)

- On crée les fichiers bibliothèques au moyen de programmes spécifiques
 - Programme « ar » sous Unix
 - Ajoute ou met à jour les fichiers objets de l'archive
 - Il est parfois nécessaire d'utiliser aussi « ranlib »
 - Liste et ordonne les symboles des fichiers objets pour faciliter le travail de l'éditeur de liens

```
% cd src/libbrol
% gcc -c brol.c -o brol.o
% gcc -c brol_io.c -o brol_io.o
% gcc -c brol_check.c -o brol_check.o
% gcc -c brol_compute.c -o brol_compute.o
% ar ruv libbrol.a brol.o brol_io.o brol_check.o brol_compute.o
% ranlib libbrol.a
```

Bibliothèques (4)

- L'option « `-L` » du compilateur/éditeur de liens permet d'augmenter la liste des répertoires dans lesquels chercher les fichiers de bibliothèques
- L'option « `-lxxx` » du compilateur permet d'ajouter le fichier bibliothèque « `libxxx.a` » à la liste des fichiers consultés pour y trouver les symboles manquants, dans l'ordre dans lequel ces fichiers sont listés

```
% cd ../main/  
% gcc main.c -o main -L../libbrol/ -lbrol -lm
```