Nathanaël COTTIN

Pratique du langage C

Introduction
Notions fondamentales
Concepts avancés

Nathanaël Cottin

Nathanaël COTTIN

Avant-propos

- Ce document explique les principaux points délicats du langage C
- Il n'a pas pour vocation d'introduire l'ensemble des éléments du langage C, notamment syntaxiques, supposés connus
- Il aborde des concepts d'utilisation évoluée du langage C

Plan général

- Partie 1 : Principaux types de données
- Partie 2 : Variables et pointeurs
- Partie 3 : Les tableaux
- Partie 4 : Passage de paramètres aux fonctions
- Annexes : Concepts évolués du langage

Nathanaël COTTIN

Partie 1
Principaux types de données

Types élémentaires
Types évolués
Types dérivés

Nathanaël COTTIN

Types élémentaires

Nathanaël COTTIN

- Type entier : « long », « int », « short »
- Type caractère : « char »
- Types réel : « long double », « double » et « float »
- Type booléen : « int », sachant que la valeur logique « faux » s'exprime par 0 et « vrai » par toute autre valeur (généralement 1)
- Types dérivés (« unsigned », etc.)

Espace mémoire occupé

Nathanaël COTTIN

Type élémentaire	Taille (en octets)
long	4
int	2 ou 4
short	2
char	1
long double	10
double	8
float	4

Types évolués : tableaux (1/2)

- Tableaux :
  - Données stockées en mémoire de manière contiguë
  - Nom attribué au tableau = pointeur sur son premier élément
- Exemple :

```
int tab[] = {11, 22, 33, 44, 55};
```

Types évolués : tableaux (2/2)

- Chaîne de caractères : tableau de « char » terminé par le caractère spécial « '\0' »
- Exemple :

```
char maChaine[] = "Hello world!";
```

Types dérivés (1/2)

- Type énuméré :

  enum bool {FALSE,TRUE,FAUX=0,VRAI};

  Les identificateurs de la liste énumérée sont des constantes entières commençant par défaut à 0 et incrémentées

  La valeur entière d'un identificateur peut être spécifiée. Dans ce cas, les valeurs des identifiants suivants la prennent pour référence

  Ainsi : FALSE = 0, TRUE = 1 FAUX = 0 et VRAI = 1
  Donc : FALSE = FAUX = 0 et TRUE = VRAI = 1

- Structure :

  struct personne {char *nom; char *prenom;};

Nathanaël COTTIN

Types dérivés (2/2)

- Union :

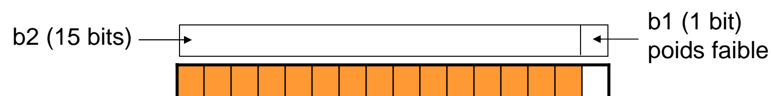
  union nombre {int nbInt; long nbLong;};

  La quantité de mémoire réservée lors de l'allocation(automatique ou dynamique) correspond à l'élément le plus gourmand (« long » dans cet exemple)

- Champs de bits : (type entier en mémoire)

  struct bits {int b1:1; int b2:15;};

b2 (15 bits)



b1 (1 bit)
poids faible

Nathanaël COTTIN

Nathanaël COTTIN

Partie 2
Variables et pointeurs

Relations

Usage

Nathanaël COTTIN

Définitions

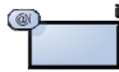
- Une déclaration de variable réserve un certain nombre de blocs en mémoire selon le type de cette variable. Le premier bloc mémoire occupé est supposé réservé à l'adresse « @ »

- Un pointeur occupe une taille fixe (type entier) en mémoire. Il référence une variable (ou NULL)

Relations variable / pointeur

Nathanaël COTTIN

- Déclaration d'une variable entière : « int i; »

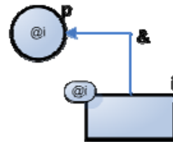


« @i » désigne l'adresse mémoire de la variable « i »

- Déclaration d'un pointeur sur un entier : « int *p; »

« @p » désigne l'adresse mémoire du pointeur « p »

- « p » référence la variable « i » : « p = &i; »

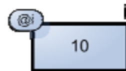


« & » est un opérateur de déréférencement

Assignation d'une valeur

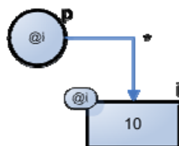
Nathanaël COTTIN

- Assignation directe : « i = 10; »



- Assignation indirecte : « *p = 10; »

« * » permet de se rendre à l'adresse pointée par « p »



Le symbole « * » est utilisé à la fois pour déclarer une variable de type pointeur et pour accéder à la valeur pointée

Nathanaël COTTIN

Partie 3
Les tableaux

Définition

Déclaration

Accès aux éléments

Définition d'un tableau

Nathanaël COTTIN

- Tableau :
  - Données stockées en mémoire de manière contiguë
  - Nom attribué au tableau = pointeur sur son premier élément :
    - *Constant* en cas d'allocation automatique
      Exemple : « int tab[10]; », « int tab[] = {1, 2, 3}; »
    - V*ariable* en cas d'allocation dynamique
      Exemple : « int *tab = (int *)malloc(10 * sizeof(int)); »

Déclaration d’un tableau

Nathanaël COTTIN

- Avec allocation automatique :

```
char str[] = "azerty";
char str[] = {'a', 'z', 'e', 'r', 't', 'y','\0'};
```

- Avec allocation dynamique :

```
char str[] = (char *)calloc(7 * sizeof(char));
```

  - Le tableau n’est pas initialisé
  - Le dernier élément est réservé pour ‘\0’
  - Utilisation de « free » impérative

Accès aux éléments d’un tableau

Nathanaël COTTIN

- Premier élément accessible via l’indice 0
- Tableau de 10 entiers : « int tab[10]; » :
  - Initialiser le 1er élément à la valeur 5 :

```
tab[0] = 5;
*tab = 5;
```

« tab » est un pointeur sur son premier élément

  - Récupérer la valeur du 8ème élément :

```
int v = tab[7];
int v = *(tab + 7);
```

En effet, « tab + 7 » ⇔ « &tab[7] » car « tab » est un pointeur sur « tab[0] »

Partie 4
Les fonctions

Définition

Passage des arguments

Nathanaël COTTIN

Définition

- Une fonction est un sous-programme réalisant une partie du traitement demandé
- Une fonction renvoie ou non une valeur
- On appelle procédure une fonction qui ne retourne aucun résultat (« void »)
- L’exécution peut dépendre de paramètres donnés lors de l’appel de la fonction

Nathanaël COTTIN

Appel d'une fonction

Nathanaël COTTIN

- L'adresse de retour du programme est empilée
- Les variables passées en argument lors de l'appel d'une fonction sont empilées (par copie)
- Un saut inconditionnel est effectué pour exécuter le code associé à la fonction
- L'appel de la fonction dépile les paramètres dont elle a besoin
- Le code est exécuté
- L'adresse de retour est dépilée
- Le résultat (si présent) est empilé
- Le programme se poursuit à l'adresse de retour

Passage des arguments

Nathanaël COTTIN

- Passage par valeur : passage direct, aucune modification possible de la valeur
- Passage par pointeur : passage indirect, possibilité de modifier la variable pointée
- Passage par référence : cas particulier de passage par pointeur (indirection)

Les variables passées en argument sont empilées avant d'exécuter le code associé à la fonction : une copie (lue dans la pile) est donc utilisée par la fonction

➔ Ce qui explique que le passage par valeur ne modifie pas la variable du programme appelant

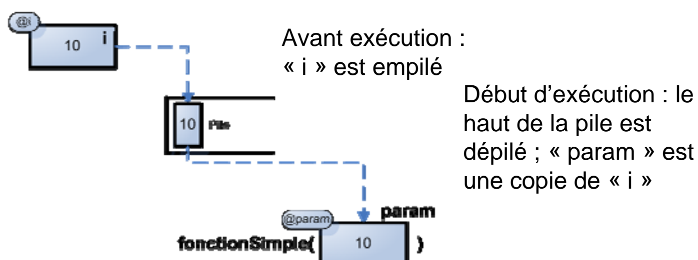
Passage par valeur

- Déclaration :

  void fonctionSimple(int param) {…}

- Appel :

  fonctionSimple(i);



Avant exécution : « i » est empilé

Début d'exécution : le haut de la pile est dépilé ; « param » est une copie de « i »

Nathanaël COTTIN

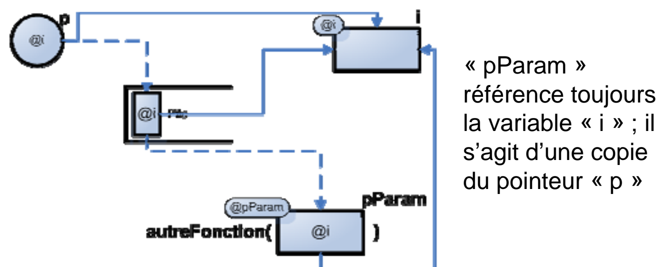
Passage par pointeur

- Déclaration :

  void autreFonction(int *pParam) {…}

- Appel :

  autreFonction(p);



« pParam » référence toujours la variable « i » ; il s'agit d'une copie du pointeur « p »

Nathanaël COTTIN

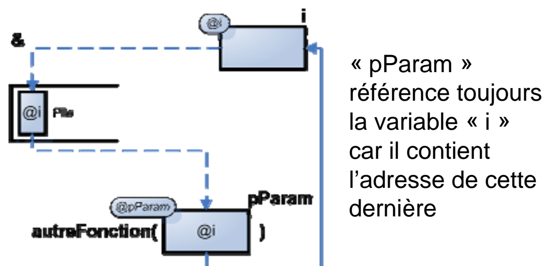
Passage par référence

- Déclaration :

  void autreFonction(int *pParam) {…}

- Appel :

  autreFonction(&i);



« pParam » référence toujours la variable « i » car il contient l'adresse de cette dernière

Nathanaël COTTIN

Modification des arguments

- Quel que soit le type de passage du paramètre (valeur, pointeur ou référence), une copie est utilisée par la fonction appelée (utilisation de la pile)
- La variable passée en argument par valeur ne peut être modifiée directement
- Il faut passer par une référence à cette variable en utilisant « & » ou en déclarant un pointeur sur cette variable

Nathanaël COTTIN

Exemple : échange de valeurs

Nathanaël COTTIN

- Déclaration incorrecte :

```
int echange(int a, int b) {…}
```

L'appel de cette fonction induit une recopie des variables passées en paramètres. Par conséquent les valeurs initiales restent inchangées suite à l'exécution de la fonction (cf « fonctionSimple »)

- Déclaration correcte :

```
int echange(int *a, int *b) {
  int tmp = *b;
  *b = *a; *a = tmp;
}
```

Types évolués et dérivés en argument

Nathanaël COTTIN

- Le principe d'utilisation de la pile reste valable
- Les types évolués non gérés explicitement à l'aide d'allocation dynamique sont copiés en intégralité dans la pile avant l'appel de fonction
- Les données accessibles par référence dans la variable d'origine demeurent accessibles (i.e. les pointeurs internes permettent de modifier les variables pointées (sauf si déclaration constante « const » – Cf annexes section 1)

Nathanaël COTTIN

Annexes
Concepts évolués du langage

Arguments constants

Pointeurs vers fonctions

Divers

Nathanaël COTTIN

Annexe 1
Arguments constants

Utilisation du mot-clé « const »

Nathanaël COTTIN

- Permet de limiter l'accès à une variable ou un pointeur
- Le contenu de la variable (valeur ou adresse mémoire si pointeur) ne peut être modifié : on parle de lecture seule (« read only »)
- Une référence obtenue à l'aide de « & » est implicitement déclarée « const » (Cf « autreFonction(&i) »)

Utilisation légale des constantes

Nathanaël COTTIN

- Déclaration :

  const double pi = 3.141592654;
- Instructions légales :
  - Passage de « pi » en tant que paramètre d'une fonction
  - Assignation de la valeur d'une variable « i » :

    double i = pi;
  - Déréférencement dans pointeur constant :

    const double *p = π

Utilisation non autorisée

Nathanaël COTTIN

- Instructions illégales :
  - Changement de valeur :
    pi = 3.1416;
  - Déréférencement dans pointeur non constant :
    int *p = π → const int *p = π

- Attention :
  - Instructions valides pour le compilateur !
  - Il s'agit plus de conventions d'écriture

Fonction à argument constant

Nathanaël COTTIN

- Utile en cas d'argument de type pointeur
- Déclaration :

```
void testConst(const int *pParam) {…}
```

- Corps de la fonction :

```
void testConst(const int *pParam) {
  *pParam = 1;          /* Erreur de compilation */
}
```

Contrairement à la déclaration de variables, le compilateur refuse de compiler l'instruction d'assignation de la valeur au paramètre constant

Nathanaël COTTIN

Annexe 2
Les pointeurs vers fonctions

Définition

Nathanaël COTTIN

- Nom de fonction = pointeur constant sur la fonction
- Sa valeur correspond à l'adresse mémoire de la première instruction de la fonction
- Il est possible de déclarer un pointeur (variable) destiné à référencer une fonction

Déclaration

Nathanaël COTTIN

- Exemple :

```
int (*pFonction)(const char *);
```

- L'emploi des premières parenthèses est obligatoire :

```
int *pFonction(const char *);
```

est une déclaration de prototype de fonction retournant un pointeur sur un entier et non un pointeur sur une fonction

Appel

Nathanaël COTTIN

- Exemple (suite) :

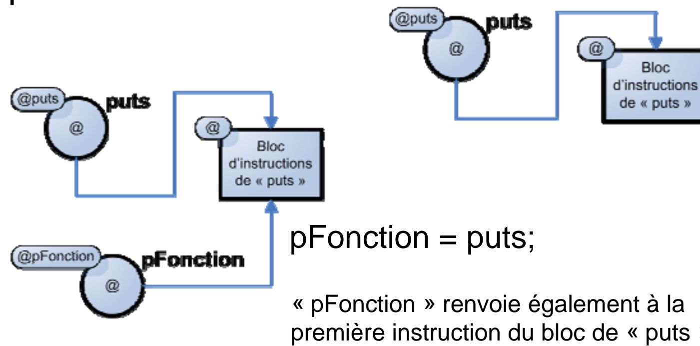
```
pFonction = puts;
(*pFonction)("Appel de la fonction puts");
```

- La signature de la fonction utilisée (il s'agit ici de « puts ») doit correspondre à la déclaration du pointeur sur fonction
- Les parenthèses sont également obligatoires

Exécution mémoire

Nathanaël COTTIN

- La fonction « puts » est enregistrée (comme toute autre fonction) à une adresse en mémoire correspondant à la première instruction de son corps



pFonction = puts;

« pFonction » renvoie également à la première instruction du bloc de « puts »

www.ncottin.net

Annexe 3
Divers

Nathanaël COTTIN

Fonctions « inline »

- Valable pour les fonctions courtes
- Le compilateur n'effectue pas d'appel aux fonctions « inline » mais recopie directement leur code
- Exemple :

```
inline int plusGrandOuEgal(int a, int b) {
  return a >= b;
}
```

Nathanaël COTTIN

Fonctions à arguments variables (1/2)

- Au minimum un argument obligatoire
- Autres arguments facultatifs
- Exemple : affichage d'entiers

```
#include <stdio.h>
#include <stdarg.h>

void afficherEntiers(int val, …) {
   int arg = val;
   va_list argList;              /* Liste des arguments facultatifs */
   va_start(argList, val);       /* Initialisation de la liste */
   while (arg != 0) {
          printf("%d ", arg);
          arg = va_arg(argList, int);   /* Argument suivant (et type) */
   }
   va_end(argList);              /* Clôture de la liste des arguments */
}
```

Nathanaël COTTIN

Fonctions à arguments variables (2/2)

- Appel de cette fonction :

```
void main() {
   afficherEntiers(1, 2, 3, 4, 5, 0);
}
```


Nathanaël COTTIN