Ordre du jour :
- 2h de cours (Structures de code - Bonnes pratique de dev)
- Correction de certains points du TP1
- Point sur le versionning
- Début du TP2 (Création des groupes, Présentation du sujet)

Développement informatique

Cours 3 :

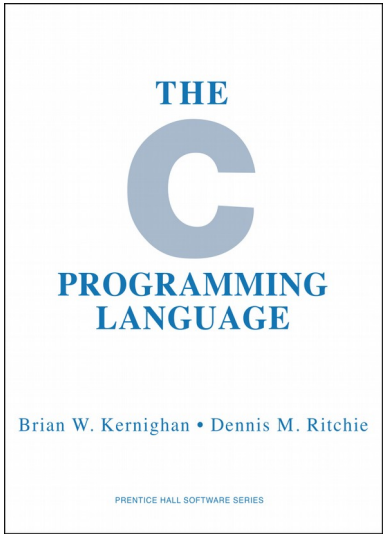
-Structures de contrôle : Illustration en C

-Introduction aux bonnes pratique de génie logiciel

Historique du C/C++

- Inventé au cours de l'année 1972 (Laboratoires Bell), en même temps que UNIX par Dennis Ritchie et Ken Thompson

- En 1978, sortie du livre « The C Programming Language » décrivant le C → La bible des développeur C : le « K&R »



Historique du C/C++

ANSI = American National Standards Institure

- Première normalisation ANSI en 1989 → C89 (C90 norme ISO)

- Seconde norme en 1999 → C99

→ tableau de taille variable

→ Nombres complexes

→ Pointeur restreint

→ Les déclarations mélangées

→ Les fonctions *inline*

→ Syntaxe des commentaires…

- Troisième norme en 2011 (unicode, généricité, threads)

Caractéristiques générales

- Langage impératif et généraliste
- Langage compilé
- Fortement typé
- Langage de programmation de référence pour les OS (*nix, windows)

Philosophie du C :

In keeping with C's free-wheeling, "I assume you know what you're doing" policy, the compiler does not complain if you try to write to elements of an array that do not exist.

Caractéristiques générales

-Ancienneté, standard ouvert

-Facile à apprendre…

-Code, temps d’exécution et occupation mémoire prévisible (→ microcontrôleur, noyau des OS)

-Peu de vérifications du code par le compilateur

-Faible modularité, gestion des exceptions faible

-Portabilité

-Code potentiellement source de grave bug (dépassement de tampon…)

Environnement de compilation

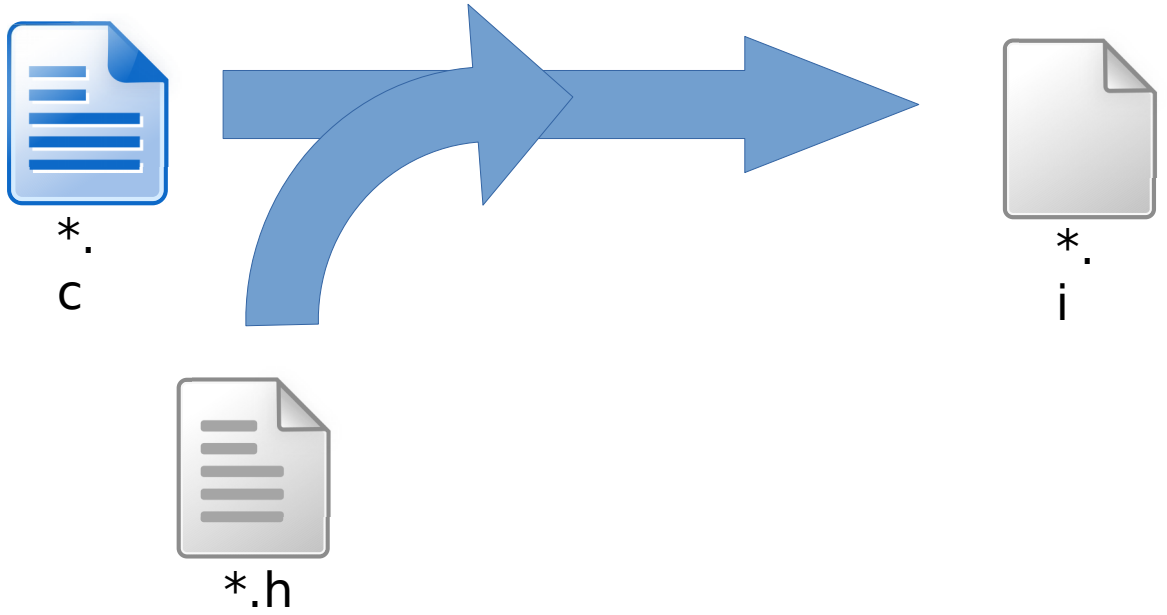
- **Outils de compilation (toolchain)**
  - -Pre processeur
  - -Compilateur
  - -Assembleur
  - -Editeur de lien (linker)
- **Fichier *.h (header) contenant les prototypes**
- **librairies (*.a ; *.lib ; *.so (*nix) ; *.dll.a)**
- **Utilitaires (binutils)**
- **debuggeur, make...**

Structure d'une application C

- Les fichiers *.c (ou *.cpp) contenant le code
- Les fichers *.h contenant le prototype/signature des fonctions
- Un ou plusieurs Makefile

Etapes de la compilation : Pre processeur

→ **Suppression des commentaires (// et /* */)**
→ **Inclusion des fichiers .h (#include)**
→ **Traitement des directives de compilation** (`define elif else endif error if ifdef ifndef include line pragma undef`)
→ **Contrôle la syntaxe du programme.**



► gcc -E main.c > hello.i
►more hello.i

Etapes de la compilation : Compilation en assembleur

→ Le code du langage C est transformé en code assembleur

- ► gcc -S hello.i
- ► more hello.s

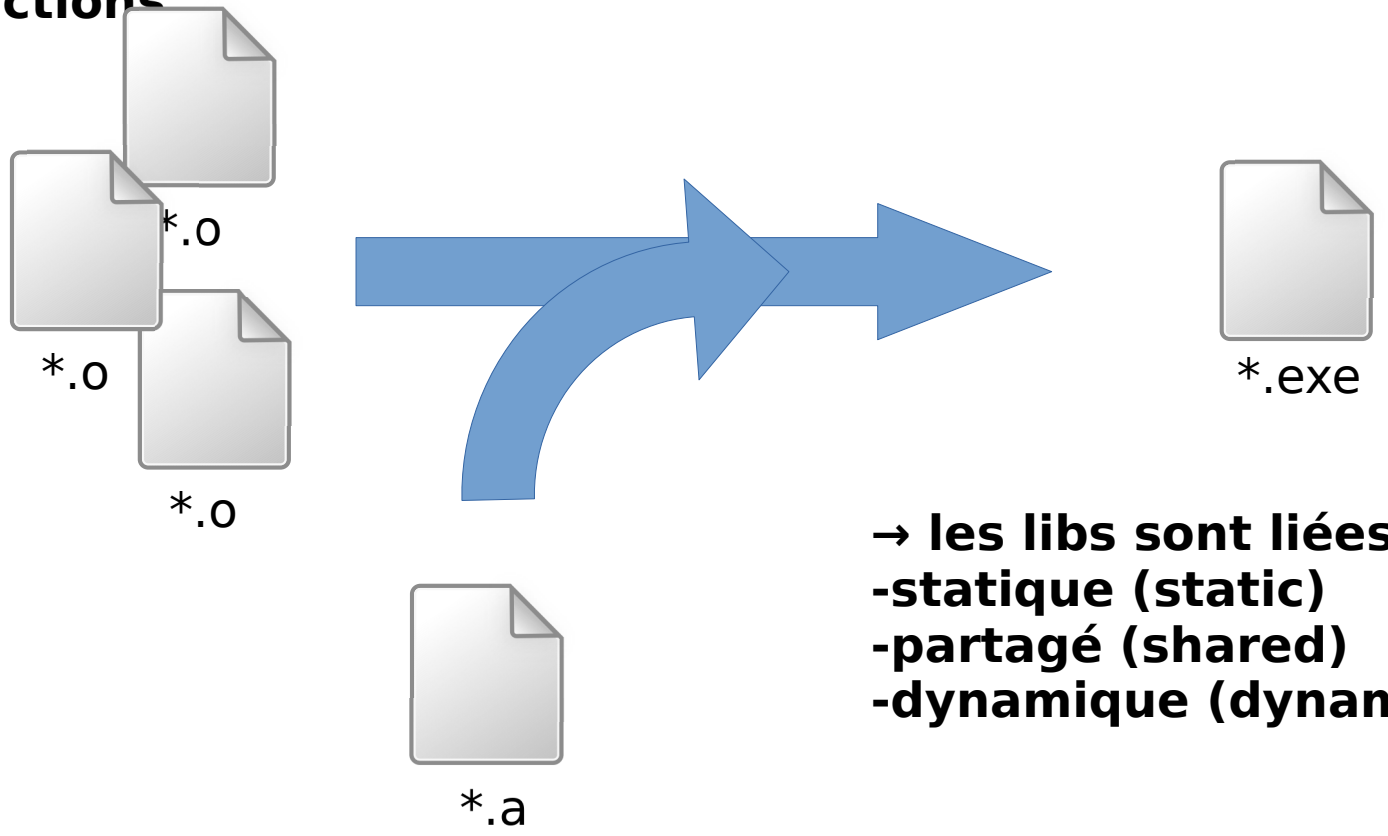
Etapes de la compilation : Compilation en code machine

→ Le code code assembleur est transformé en langage machine (binaire)

- ► gcc -c hello.s
- ► od -x hello.o

Etapes de la compilation : Edition de liens

→ Les codes binaires sont fusionnés dans un seul fichier, et les appels aux fonctions sont substitués par les adresses des fonctions



→ les libs sont liées en :
-statique (static)
-partagé (shared)
-dynamique (dynamic)

- gcc hello.o -o hello.exe
- hello.exe

Rôle du make

- La commande make travaille avec un fichier de configuration (le Makefile)

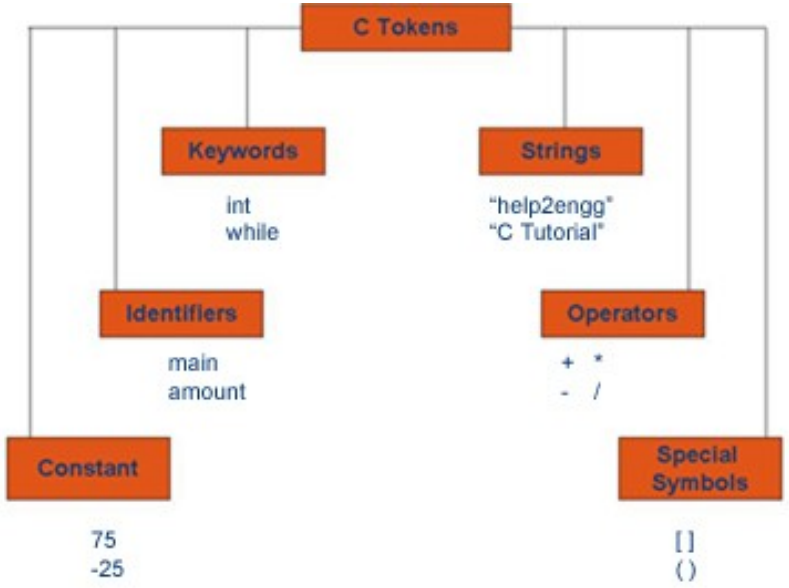
- Il permet entre autre de réaliser les étapes de compilation, dans l'ordre nécessaire, pour l'ensemble des fichiers du projet

- Nombreuses commandes pour générer un Makefile à partir de la structure du projet et en fonction de la plateforme de compilation (cf autoscan, aclocal, autoheader, autoconf, automake, configure)

Élément d'un programme C

Les tokens

- élément de base individuel le plus petit existant en C
- 6 types de tokens :



L'analyse lexicale se trouve tout au début de la chaîne de compilation. C'est la tâche consistant à décomposer une chaîne de caractères en lexèmes, unités ou entités lexicales, aussi appelées tokens en anglais

Élément d'un programme C : mots clefs

- Mots clefs du langage :

-32 en C89

auto, break, case, char, const , continue, default, do, double, else, enum , extern, float, for, goto, if, int, long, register, return, short, signed , sizeof, static, struct, switch, typedef, union, unsigned, void , volatile , while.

-Cinq de plus en C99

_Bool, _Complex, _Imaginary, inline, restrict. **(préfixé avec _ pour la compatibilité, redéfini en bool, complex, imaginary dans <stdbool.h> et <complex.h>)**

Élément d'un programme C : identificateurs

Les identificateurs (nom des variables ou fonctions)

- Un identificateur est composé de lettres non accentuées, de chiffres et du caractère _ dans un ordre quelconque sauf pour le premier caractère qui ne peut pas être un chiffre

- Le C (et beaucoup d'autres langages) est sensible à la casse

- Pas d'espace !!

- Le document de définition du langage C a réservé un certain nombre d'identificateurs . Ils peuvent être utilisés par les implémenteurs (ceux qui écrivent les compilateurs) ou pour des extensions du langage

Élément d'un programme C : identificateurs

Nommage des variables et fonctions :

- reflète sont rôle dans le programme : maxAllowedSpeed, growthRate

Variables :

- Une variable possède un nom qui doit respecter des règles et des conventions
- Une variable possède un type (explicite dans les langages typés) → défini l'occupation mémoire, une valeur maximal, une valeur minimal.
- Une variable possède une valeur → déclaration, initialisation
- Une variable possède une porté (à ne pas confondre avec visibilité, La durée de vie des variables est liée à leur portée, c'est-à-dire à la portion du programme dans laquelle elles sont définies.)
- Cas particulier : les constantes (accessible dans tout le programme, non modifiable

Fonctions :

- Une fonction possède des arguments (paramètres formels, par opposition aux paramètres effectifs)
- Une fonction possède un type de retours (int, float…) ou void pour les fonctions ne retournant rien

=> Définition de la signature (prototype) d'une fonction

Élément d'un programme C : identificateurs

Conventions d'écriture et de nommage : propre à chaque langage :

- Lower Camel Case : « valMax » « userName » « maxTaxRate »

  → notation utilisée en java pour les variables

-Upper Camel Case : « VitesseMax »

  → notation utilisée en java pour les noms de Class

-Hongroise : iAge, fTaux, dSomme

  → C#

-C :
https://users.ece.cmu.edu/~eno/coding/CCodingStandard.html

https://www.doc.ic.ac.uk/lab/cplus/cstyle.html

https://en.wikipedia.org/wiki/Naming_convention_(programming)

Élément d'un programme C : identificateurs

-Conventions pour certains noms de variables

- i, j, k, l sont en général utilisées comme compteur de boucle
- x, y, z pour des coordonnées
- foo, temp, dummy, swap... sont généralement utilisés pour les variables temporaires
- done, failed, active, current, running... pour les variables booléennes,
- isdone, isrunning, is_shuting_down, is_active... pour les fonctions retournant un booléen

=> Importance de respecter les normes de codage :

-Travail en équipe

-outils d'analyse de code (sonar qube , pdm, findbugs...)

Élément d'un programme C : identificateurs

Certains identificateurs sont réservés (exemple en C):

Les identificateurs commençant par	suivis de	Exemples valides	Exemples réservés
"_"	"_A-Z"	_123	_ABC
"is"	"a-z"	is_abc	isabc
"mem"	"a-z"		
"str"	"a-z"		
"to"	"a-z		
"wcs"	"a-z"		
"E"	"A-Z" ou "0-9"	Eabc	E123 EABC
"LC_"	"A-Z"	LC_abc LC_123 LCABC	LC_ABC
"SIG"	"_A-Z"	SIGabc	SIGABC SIG_ABC

string.h

code d'interruptions

Type primitif des variables

Type primitif, taille et valeurs maximales et minimales en C

Écriture complète	Écriture concise, généralement utilisée	Taille, avec les options par défaut	Intervalle de valeurs
`bool`	`bool (boolean)`	variables	true et false (norme C89, C99) → #include <stdbool.h>
`signed char`	`char (byte)`	1 octet	de –128 à 127
`unsigned char`	`unsigned char`	1 octet	de 0 à 255
`signed short int`	`short`	2 octets	de -32 768 à 32 767
`unsigned short int`	`unsigned short`	2 octets	de 0 à 65 535
`signed int`	`int`	2 ou 4 octets	de -32 768 à 32 767 ou -2 147 483 648 à 2 147 483 647
`unsigned int`	`unsigned int`	2 ou 4 octets	de 0 à 65 535 ou 0 à 4 294 967 296
`signed long int`	`long`	4 ou 8 octets	de -2 147 483 648 à 2 147 483 647 ou -9223372036854775808 à 9223372036854775807
`unsigned long int`	`unsigned long`	4 ou 8 octets	de 0 à 4 294 967 296 ou $2^{64}-1$
`long long int`	`long long`	8 octets au minimum	de -9223372036854775808 à 9223372036854775807
`unsigned long long int`	`unsigned long long`	8 octets au minimum	de 0 à 18446744073709551615

Type primitif des variables

Type primitif, taille et valeurs maximales et minimales en C (suite)

`float`	`float`	4 octets	IEEE 754 single-precision binary floating-point format. ±3.402823e+38, ±1.175494e-38
`double`	`double`	8 octets	IEEE 754 double-precision binary floating-point format. ±1.797693e+308, ±2.225074e-308
`long double`	`long double`	12 octets	Dépend de l'implémentation (80 bits, 128 bits utilisé sous AIX d'IBM)

Type primitif des variables

- Le fichier <limit.h> contient les macros définissant les valeurs minimales et maximales pour les types entiers

- Le fichier <float.h> contient les macros définissant les valeurs minimales et maximales pour les types float et double

- Le type chaîne de caractères (String) n'existe pas en C

→ une chaîne de caractères est un tableau de caractères terminé par \0

Types primitifs : conversion (cast)

Promotion (cast implicite)

-char → short → int → long → float → double

Cast explicite (dangereux car pas de vérification en C)

-short → char

-long → short

-double → float

-float → int

Types primitifs : taille, affichage

-sizeof est l’opérateur permettant d'avoir la taille en mémoire d'un type primitif

-printf("format" , ...) permet d'afficher dans la console la valeur d'une variable.

→ format est une chaîne de caractères qui peut contenir

-du texte

-des séquences d’échappement

- doit contenir des spécificateurs de format : %i ou %f ou %e ou %c

Opérateurs

- Opérateurs arithmétiques :

\+ - * / % ++ --

- Opérateurs relationnels :

== >= <= < < != < >

- Opérateurs logiques :

&& || !

- Operateurs bit a bit (bitwise operators) :

& | ^ ~ >> <<

- Operateurs d'affectations (assignment operators) :

= += -= *= /= %= >>= <<= &= ^= |=

Opérateurs

- Autres

- « Sizeof » : Taille du type de la variable
- « & » : Adresse mémoire d'une variables
- « * » : Valeur a l'adresse pointé
- « → » : Déreference sur une structure
- « . » : Référence sur une structure
- « ?: » : Opérateur ternaire
- « [ ] » : Indexeur
- « ( ) » : appel de fonction
- « , » : opérateur séquentiel
- « (int) » ou « (long) », « (float) » : Opérateur de coercition

Opérateurs : Priorité

Cf :http://en.wikipedia.org/wiki/Operators_in_C_and_C++

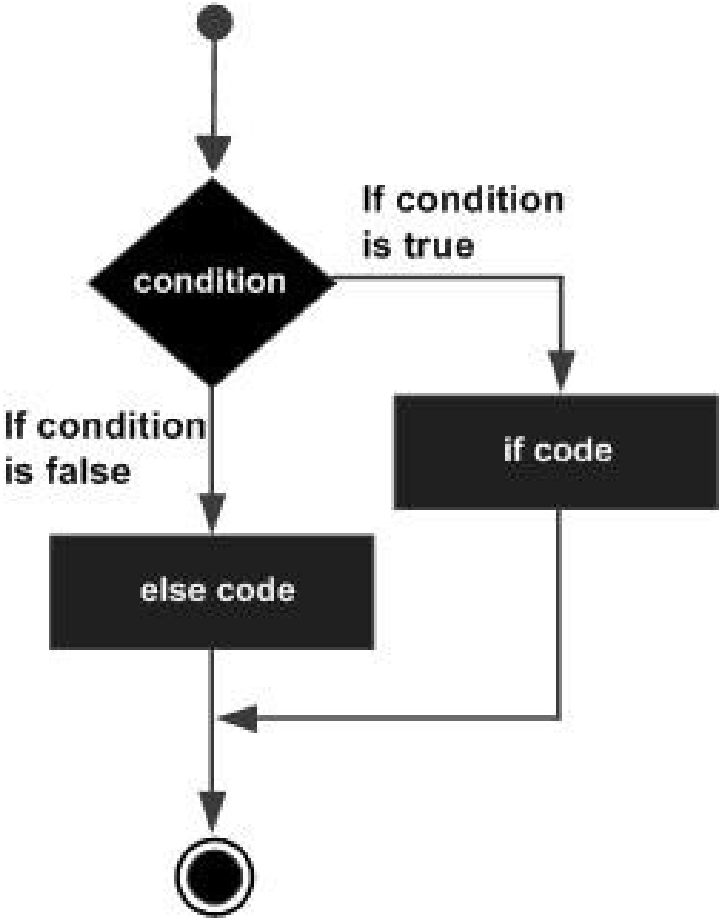
Cf : PDF « Operators priority »

Instructions

Structure conditionnelle

```
if (expression_boolean) {
    //Code a executer
} else {
    //Code alternatif a executer
}

if (expression_boolean) {
    //Code a executer
} else if (expression_boolean) {
    //Code a executer
} else {
    //Code a executer
}
```

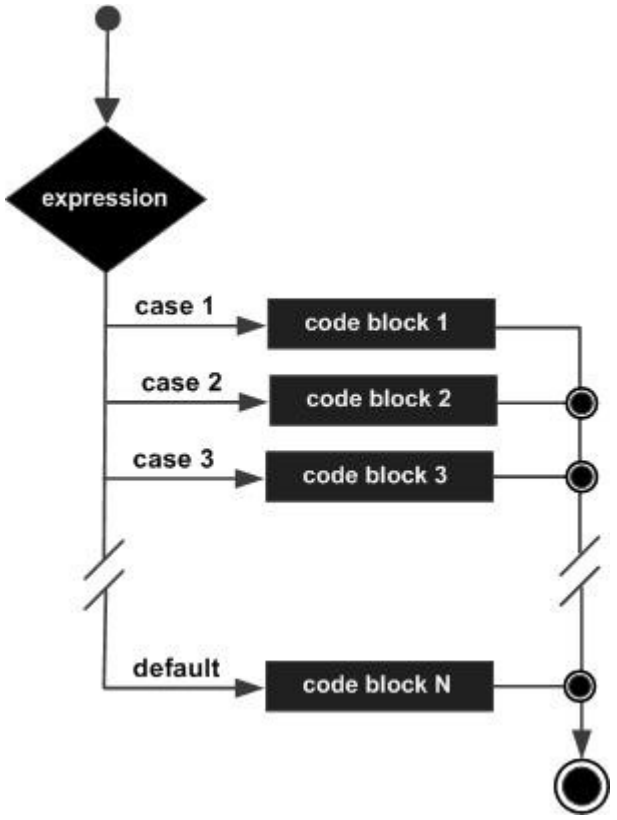


Attention à la portée des variables

Instructions

Structure de branchement (branches controls)

```
switch (a) {
    case 1 : {
        //Code a executer
        break;
    }
    case 2 : {
        //Code a executer
        break;
    }
    case 3 : {
        //Code a executer
        break;
    }
    default : {
        //Code a executer
    }
}
```



Attention au break

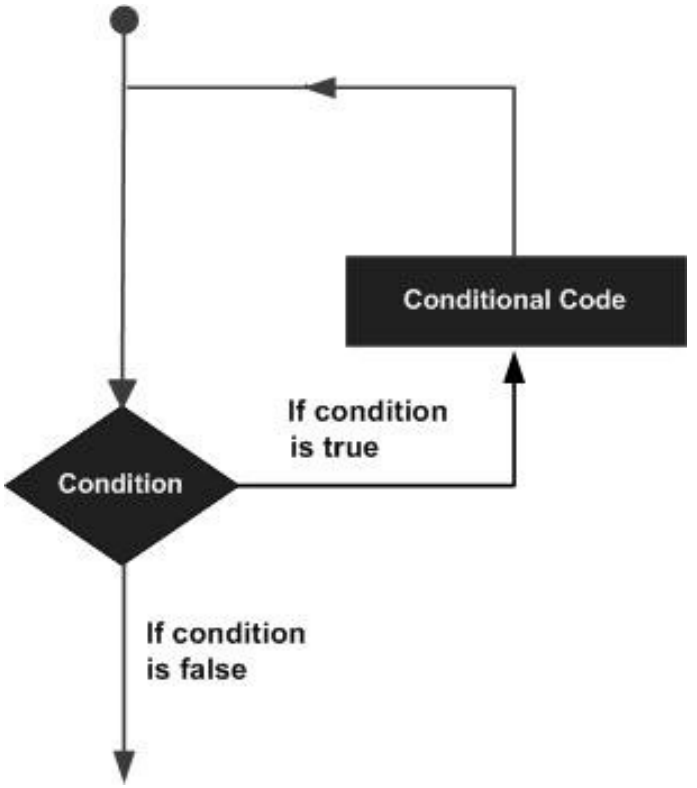
Instructions

Boucles

```java
//Boucle deterministe
for (int i = 0; i < 100; i++) {
    //Code a executer
}

//Boucles non deterministe
while (expression_boolean) {
    //Code a executer
}

do {
    //Code a executer
} while (expression_boolean);
```



Instructions

Boucles

```
//Boucle infinie
while (true) {
    //Code a executer
}

//Boucle for infinie
for (;;) {
    //Code a executer
}
```

Instructions

Sortie de boucle (for, switch, while, do) : break

```
// break
while (true) {
    //Code a executer
    if (expression_boolean) {
        break;
    }
}

while (expression_boolean) {
    //Code a executer
    if (expression_boolean) {
        break;
    }
}
```

Instructions

Redémarrage d'une itération (for, while, do) : continue

```
while (expression_boolean) {
    //Code a executer
    if (expression_boolean) {
        continue;
    }
    //Code a executer
}
```

Instructions

Saut inconditionnel : goto vers un label

```
encore:
    printf("running....");
    //Code...
    //saut inconditionnel
    goto sortie;
    // un label

    printf("fin....");

sortie:

    //Code ...
    //saut conditionnel
    if (expression_boolean) goto encore;
```

A proscrire dans la mesure du possible
Et c'est très souvent possible dans la majorité des ca

Interdit de sauter dans
une b

Instructions

Autres :

const : déclaration d'une constante

enum : déclaration d'une énumération

extern : autorise l'utilisation d'une variable déclarée dans un autre fichier *.c

register : invite de compilateur à utiliser un registre CPU pour conserver cette variable

static : rend une variable/fonction visible dans tout le programme. Si appliqué sur une variable local → préserve la variable entre les appels successifs

Instructions

Autres (suite):

volatile : indique au compilateur que le contenu d'une variable peut changer de façon imprévisible (contexte multithreading)

struct : défini la structure d'un enregistrement

typedef : déclare un nouveau type

union : groupe des variables dans le même espace de mémoire

Les tableaux

Tableau à 1 Dimension

```c
//Tableau monodimensionel statique de 10 int
int tabA[10];
//Dimensionnement et initialisation
int tabB[] = {10, 15, 48, 32, 100};

/*
 * Dimensionnement dynamique d'un tableau de float
 *(C99 et C11 uniquement !! )
 */
int n = rand() % 100;
float tabC[n];
```

Les tableaux

Tableaux multi dimension

```
//Tableau de 10x20 (matrice 10x20);
float tab2DA[10][20];

//Dimensionnement et initialisation
//Matrice 3x3

short tab2DB[][3] = {
    {1, 2, 3},
    {10, 20, 30},
    {8, 88, 888}
};

int n = rand() % 100;
int m = rand() % 100;
//Dimensionnement dynamique
int tab2DC[n][m];
```

Les tableaux

Un tableau (comme une variable) doit être initialisé, sinon il contient des valeurs aléatoires !

> **In keeping with C's free-wheeling, "I assume you know what you're doing" policy, the compiler does not complain if you try to write to elements of an array that do not exist.**

Les tableaux

Organisation d'un tableau en mémoire

```
short tab2DB[][] = {
    {10, 20, 30,40},
    {8, 88, 888,8888}
};
```

10	20	30	40
8	88	888	8888

10	20	30	40	8	88	888	8888

Chaine de caractères

→ la syntaxe de la déclaration va dépendre de l'usage de la chaine de caractères

-Déclaration avec un pointeur → texte pour affichage (la chaine de caracteres n'est pas mutable)

```
/*Declaration avec un pointeur
 *la chaine est considére comme une constante (non modifiable !)
 *exemple : impossible de remplacer le 'o' par un espace
 */
char* msgA = "Hello";
printf("msgA : %s\n", msgA);
```

Chaine de caractères

- Déclaration avec un tableau

```
/*Déclaration d'un tableau de 7 char et initialisation
 *Attention aux indices :0-6
 * 2 Possibilites pour l'init. Notez le 0 en dernier caractere
 * de la seconde possibilité
 */
char msgB[] = "Bonjour";
char msgb[] = {'B', 'o', 'n', 'j', 'o', 'u', 'r', 0};
//Affiche Bonjour
printf("  msgB : %s\n", msgB);
```

Chaine de caractères

- Le dernier caractères d'une chaine (en mémoire) doit être '\0' : c'est le marqueur de fin de chaine (NULL terminated string par opposition aux PASCAL string/ length prefixed)

Beaucoup de prudence lors de la manipulation de chaine → dépassement de pile

Chaine de caractères

- Déclaration avec un pointeur en dynamique

```
#define MAX_STRING_SIZE 100
```

char* et malloc → un avant goût des pointeurs
Noter l'utilisation de la directive #define

```
    //Allocation mémoire
    char* msgD = malloc(MAX_STRING_SIZE);
    strncpy(msgD, "ca marche enfin", MAX_STRING_SIZE);
    msgD[0] = 'C';
    //Affiche Ca marche enfin
    printf("msgD : %s\n", msgD);
```

Les pointeurs

Définition :

- Un pointeur est une variable contenant l'adresse d'un espace mémoire
- L'espace mémoire pointé est typé
- Il est lui même une variable de type int/long
- Il dispose d'une arithmétique restreinte et particulière

Les pointeurs

Déclaration

```
short *ptr0;//Pointeur sur un espace memore contenant une valeur de type short

int *ptr1;//Pointeur sur un espace memore contenant une valeur de type int

float *ptr2;//Pointeur sur un espace memore contenant une valeur de type float

char *ptr4;//Pointeur sur un espace memore contenant une chaine de caracteres
```

Les pointeurs

Initialisation

```
ptr0 = (short *) 0x8000;//pointeur sur l'adresse 0x8000...
ptr1 =(int *) 0x8004;


ptr0 = (short *) malloc(sizeof(short));// initialisation et reservation de la memoire
ptr1 = malloc(sizeof(int));// correct, mais warning du compilateur

ptr4 = (char*) malloc(200);// Init et reservation d'un espace memoire
                           // pour une chaine de 200 caracteres \0 compris !
```

Les pointeurs

Initialisation et affectation

```
int *ptr5 = (int *) 0x8004;

float *ptr6 = (float *) 0x8010;

float *ptr7 = (float *) malloc(sizeof(float));

char ptr8 = (char *) malloc(1024 * sizeof(char));

short *ptr9 = malloc(sizeof(*ptr9));
```

Les pointeurs

Pointeur et opérateur de référencement (&)

```
int age = 25;
int *ptrAge = &age;

printf("  adresse de age %p,   valeur de age : %d\n", &age, age);
printf("  valeur de ptrAge %p,  valeur pointee par ptrAge : %d\n", ptrAge, *ptrAge);

age++;
printf("  adresse de age %p,   valeur de age : %d\n", &age, age);
printf("  valeur de ptrAge %p,  valeur pointee par ptrAge : %d\n", ptrAge, *ptrAge);

*ptrAge = *ptrAge+1;
printf("  adresse de age %p,   valeur de age : %d\n", &age, age);
printf("  valeur de ptrAge %p,  valeur pointee par ptrAge : %d\n", ptrAge, *ptrAge);
```

Les pointeurs

Arithmétique des pointeurs

Seul des opérateurs + et - (ainsi que ++ et --) ont un sens !

```
int age = 25;
int *ptrAge = &age;

printf("  adresse de age %p,   valeur de age : %d\n", &age, age);
printf("  valeur de ptrAge %p,  valeur pointee par ptrAge : %d\n", ptrAge, *ptrAge);

printf("\n");

ptrAge++;
printf("  valeur de ptrAge %p,  valeur pointee par ptrAge : %d\n", ptrAge, *ptrAge);
```

Les pointeurs

Pointeur et tableau

Attention on parle ici de tableau ET pointeur et non de tableau de pointeurs !!!

Les pointeurs

Utilisation d'un tableau avec les pointeurs

```
float tab[] = {10,20,30,40,80};
float *ptrTabA=tab;// Pas d'utilisation de l'operateur &

float *ptrTabB;
ptrTabB = tab;// Pas d'utilisation de l'operateur &

printf("  adresse de la la valeur a la position 0 : %p\n", ptrTabA+0);
printf("  valeur de tab a l'index +0 : %f\n", *(ptrTabA+0));

printf("  adresse de la la valeur a la position 3 : %p\n", ptrTabA+3);
printf("  valeur de tab a l'index +3 : %f\n", *(ptrTabA+3));
```

Noter qu'on n'utilise pas l'opérateur de référencement pour les tableaux

Les pointeurs

Création d'un « tableau » avec un pointeur

```
int i = 0;
float *ptrTab = (float *) malloc(4 * sizeof (float)); // reserve la memoire
*(ptrTab + 0) = 3.14f;
*(ptrTab + 1) = 1.61f;
*(ptrTab + 2) = 2.71f;
*(ptrTab + 3) = sqrt(2);


for (i = 0; i < 4; i++) {
    printf("  adresse %p, valeur a l'index +%d : %f\n", ptrTab + i, i, *(ptrTab + i));
}
```

Les pointeurs

Différence entre un tableau et un tableau (via les pointeurs)

Il existe des différences fondamentales :

- L'espace mémoire des tableaux est alloué sur la pile (stack)
- Cet espace est limité et non libérable par le développeur
- Il est plus rapide d'accès
- Cette espace stock aussi toutes les variables locales, les adresses de retours des appels de fonction et les arguments.
- L'espace mémoire utilisé par les pointeurs est alloué sur le tas (heap)
- Il est limité par la mémoire géré par le système
- Il doit être géré avec prudence par de développeur → fuite mémoire, plantage OS

Les pointeurs

→ utilisation de tableau de grande taille, ou des appels a des fonctions récursives de trop grande profondeurs peut bloquer le programme

→ Il est possible de changer (dans certaines limites et selon le système) la taille de la stack

- → commande ulimit sous linux
- → option de compilation -Wl,--stack=8388608
- → setrlimit en C

Les pointeurs

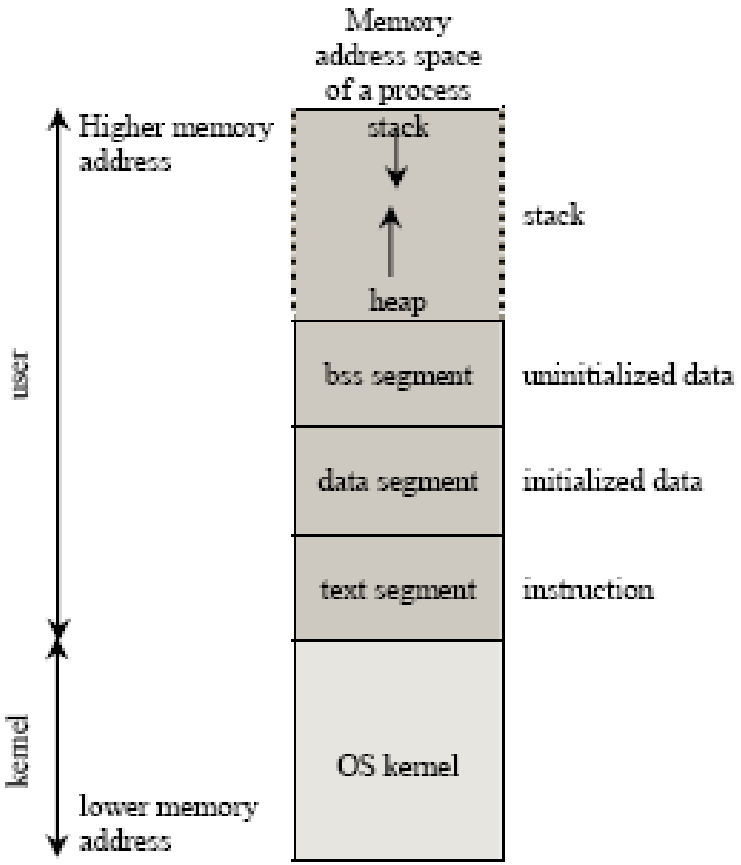


Schéma d'un programme en mémoire

Source : http://www.tenouk.com/Bufferoverflowc/Bufferoverflow1c.html
http://www.tenouk.com/ModuleW.html

Les pointeurs

Stack: (tableau via les [])

- Stored in computer RAM just like the heap.
- Variables created on the stack will go out of scope and automatically deallocate.
- Much faster to allocate in comparison to variables on the heap.
- Implemented with an actual stack data structure.
- Stores local data, return addresses, used for parameter passing
- Can have a stack overflow when too much of the stack is used. (mostly from infinite (or too much) recursion, very large allocations)
- Data created on the stack can be used without pointers.
- You would use the stack if you know exactly how much data you need to allocate before compile time and it is not too big.
- Usually has a maximum size already determined when your program starts

Les pointeurs

Heap: (tableau via les pointeurs)

- Stored in computer RAM just like the stack.
- Variables on the heap must be destroyed manually and never fall out of scope. The data is freed with delete, delete[] or free
- Slower to allocate in comparison to variables on the stack.
- Used on demand to allocate a block of data for use by the program.
- Can have fragmentation when there are a lot of allocations and deallocations
- Can have allocation failures if too big of a buffer is requested to be allocated.
- You would use the heap if you don't know exactly how much data you will need at runtime or if you need to allocate a lot of data.
- Responsible for memory leaks

Les pointeurs

Le pointeur NULL

- Valeur prise par un pointeur non initialisé ;
- Valeur de retour de certains fonction (malloc, calloc…) en cas d'erreur d'allocation

```c
int *ptr=NULL;
float *tab;

if(ptr==NULL) {
    printf("Pointeur nul");
}
//Tentative allocation memoire
tab=(float*) malloc(10*sizeof(*tab));
//Verification
if(tab==NULL) {
    printf("Echec allocation memoire");
    exit(-1);
}
```

Les pointeurs

Pointeur générique

- pointeur dont le type n'est pas défini

```
void *ptr = malloc(1024);
printf("  adresse ptr : %p\n", ptr );
ptr++;
printf("  adresse ptr : %p\n", ptr );
```

Les pointeurs

Le transtypage de pointeur (cast)

```
void *ptr = malloc(1024);
char *ptr1 = (char *) ptr;
int *ptr2 = (int *) ptr;
float *ptr3 = (float *) ptr;

strncpy(ptr, "Hello world", 1024); //Pour eviter de n'avoir que des zero...

printf("  adresse ptr1 : %p valeur %c \n", ptr1, *ptr1);
ptr1++;
printf("  adresse ptr1 : %p valeur %c \n", ptr1, *ptr1);

printf("  adresse ptr2 : %p valeur %d \n", ptr2, *ptr2);
ptr2++;
printf("  adresse ptr2 : %p valeur %d \n", ptr2, *ptr2);

printf("  adresse ptr3 : %p valeur %f \n", ptr3, *ptr3);
ptr3++;
printf("  adresse ptr3 : %p valeur %f \n", ptr3, *ptr3);
```

Les pointeurs

Allocation et Désallocation de la mémoire

-demande une allocation mémoire au système

malloc(size_t N) : réserve N octets de mémoire

calloc(size_t N, size_t type) : réserve N*type de mémoire et initialise le bloc a zéro

-demande d'une réallocation

realloc(ptr, size_t N) : tente d'allouer un bloc mémoire de taille N

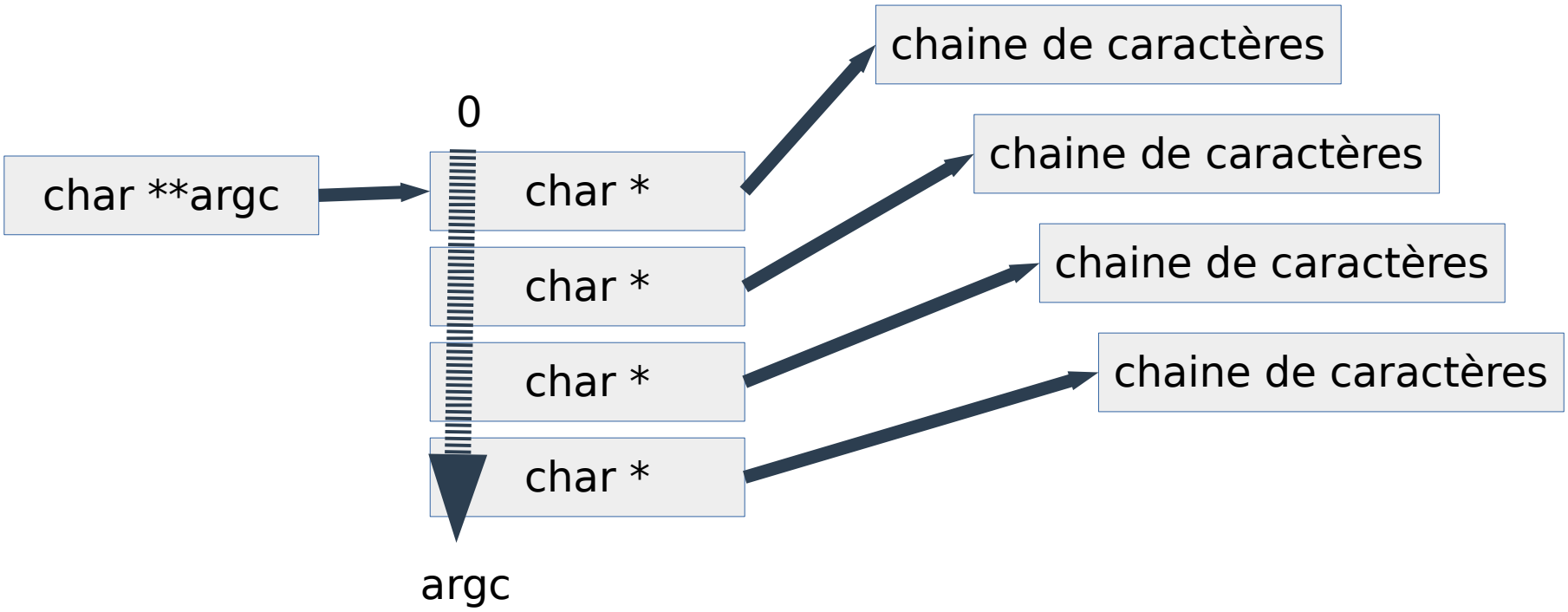
-Libère la mémoire

free(ptr) ;

malloc, calloc et realloc renvoi une adresse memoire ou NULL en cas d'echec

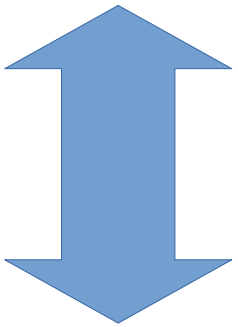
Les pointeurs

Pointeur de pointeur : exemple avec **argv



Les pointeurs

```
int main(int argc, char** argv)
```



Deux écritures équivalentes...

```
int main(int argc, char *argv[])
```

Les pointeurs

Pointeur de pointeur : exemple de code

```c
int main(int argc, char** argv) {

    int i;
    printf("  argc : %d \n", argc);
    for (i = 0; i < argc; i++) {

        printf("  argv : %p\n", argv);
        printf("  adresse de arg[%i] : %p\n", i, argv[i]);
        printf("  chaine : %s\n", argv[i]);
    }
    return (EXIT_SUCCESS);
}
```

Les entrées

Entrée de valeurs : la fonction scanf

- Cette fonction accepte en paramètre
  - Une chaine de formatage
  - une adresse mémoire
- Il existe aussi sscanf et fscanf…
- Utilisation possible de fgets (a utiliser de préférence)

Les structures et énumérations

Définition

Une structure est un assemblage de variables qui peuvent avoir différents types.

Exemple : information sur une personne :

→ nom (chaine de caractères)
→ age (entier)
→ poids (réel)
→ taille (réel)
→ numéros de téléphone (tableau de chaînes)

Les structures et énumérations

Définition d'une structure

```
struct personne {
    char nom[30];
    char prenom[30];
    unsigned int age;
    float poids;
    float taille;
    char *telephone[15];
};
```

Attention au point virgule a la fin de la déclaration

Les structures et énumérations

Définition d'un nouveau type de variable : typedef

```c
struct personne {
    char nom[30];
    char prenom[30];
    unsigned int age;
    float poids;
    float taille;
    char *telephone[15];
};

typedef struct personne Individu;
```

```c
void test() {

    Individu p2;
    p2.age = 10;
    strncpy(p2.nom, "Dupont", 30);
}
```

Les structures et énumérations

Définition d'un nouveau type de variable : typedef

```c
#include <string.h>

typedef struct {
    char nom[30];
    char prenom[30];
    unsigned int age;
    float poids;
    float taille;
    char *telephone[15];
} Individu;
```

Syntaxe commune

Les structures et énumérations

```c
#include <string.h>

typedef struct  {
    char *batiment;
    char *num_rue;
    char *nom_rue;
    int code_postal;
    char *ville;
} Adresse;

typedef struct {
    char nom[30];
    char prenom[30];
    unsigned int age;
    float poids;
    float taille;
    char *telephone[15];
    Adresse adresse;
} Individu;
```

Une structure peu contenir une autre structure

Les structures et énumérations

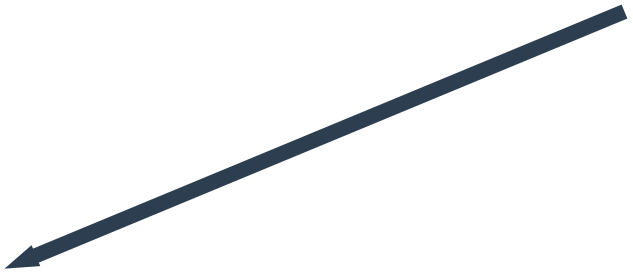
Pointeur sur une structure

```
Individu *p3;

p3 = (Individu *) malloc(sizeof (Individu));

p3->age = 10;
p3->poids = 65;
strncpy(p3->nom, "Durant", 30);
strncpy((*p3).prenom, "Jean", 30);
p3->telephone[0] = (char *) malloc(11 * sizeof (char));
strncpy(p3->telephone[0], "0487124574", 10);
p3->telephone[4] = (char *) malloc(11 * sizeof (char));
strncpy(p3->telephone[4], "0911111111", 10);
```

Variante peu utilisée



Les structures et énumérations

tableau de structure

```
Individu tab[5];

tab[1].age = 10;
tab[1].poids = 72.3;
strncpy(tab[1].nom, "Durant", 30);
tab[1].telephone[0] = (char*) malloc(11 * sizeof (char));
strncpy(tab[1].telephone[0], "0762149888", 11);
tab[1].telephone[1] = (char*) malloc(11 * sizeof (char));
strncpy(tab[1].telephone[1], "0411111111", 11);
```

Les structures et énumérations

Déclaration anticipée des structures

```c
typedef struct {
    char *nom;
    char prenom[121];
    int age;
    char genre;
    Personne enfant[10];
} Personne;
```

Ne compile pas →

```
main.c:27:5: error: unknown type name 'Personne'
    Personne enfant[10];
    ^
```

Les structures et énumérations

Déclaration anticipée :

```c
typedef struct personne Personne;

struct personne{
    char *nom;
    char prenom[121];
    int age;
    char genre;
    Personne enfant[10];
} ;
```

Les structures et énumérations

Enumérations

```
typedef enum {
    none,
    secret,
    top_secret,
    black_ops
} secret_levels;
```

```
secret_levels l;
l=top_secret;
printf("  level : %d\n" ,l);
```

```
level : 2
```

Les structures et énumérations

```
typedef enum {
    coeur = 10,
    carreau = 20,
    trefle ,
    pique
} Couleur;
```

```
Couleur c1 = trefle;
printf("  Couleur c1 : %d\n", c1);
```

```
Couleur c1 : 21
```

Fonctions

Déclaration

```
return_type function_name( parameter list ) {
        //code
}
```

Exemple 1 :

```
int max(int valA, int valB) {
    int m = valA > valB ? valA : valB ;
    return m;
}
```

Exemple 2 :

```
void sort(float tab[], size_t size) {

}
```

Fonctions

Appel d'une fonction

```
int maxi =  max(16, p.age);

printf("max = %d\n", max(10,p.age));
```

En C, les fonctions doivent être déclarées avant les appels

Fonctions

Prototype (ou signature) : A placer dans un fichier .h qui porte le même nom que le fichier .c

```c
#ifndef MESSTRUCTURES_H
#define MESSTRUCTURES_H
#endif  /* MESSTRUCTURES_H */

typedef struct {
    char *nom;
    char *prenom;
    int age;
    int taille;
} Personne;
```

```c
#ifndef MESFONCTIONS_H
#define MESFONCTIONS_H
#endif  /* MESFONCTIONS_H */

#include "messtructures.h"


int max(int, int);

void sort(float[], size_t size);

Personne *create_personne(char *);

void change_personne_1(Personne );

void change_personne_2(Personne*);
```

Préprocesseur et directives

Définition

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

-l'incorporation de fichiers source (#include),

-la définition de constantes symboliques et de macros (#define),

-la compilation conditionnelle (#if, #ifdef,...).

Préprocesseur et directives

Directive #include

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

#include <nom-de-fichier>

→ recherche le fichier mentionné dans un ou plusieurs répertoires systèmes définis par l'implémentation (par exemple, /usr/include/) ;

#include "nom-de-fichier"

→ recherche le fichier dans le répertoire courant (celui où se trouve le fichier source). On peut spécifier d'autres répertoires à l'aide de l'option -I du compilateur.

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

Préprocesseur et directives

Directive #define

La directive #define permet de définir :

- des constantes symboliques,
- des macros avec paramètres.

La directive #undefine permet de supprimer

Préprocesseur et directives

Définition de constantes symboliques

#define name value

→ demande au préprocesseur de substituer toute occurrence de nom par la chaîne de caractères value dans la suite du fichier source. Son utilité principale est de donner un nom parlant à une constante, qui pourra être aisément modifiée. Par exemple :

```
#define MAX_SIZE 10
#define DEBUG
#define  SAVE_NAME "/temp/data.txt"
```

Préprocesseur et directives

Définition de macros

#define NAME(params) definition

→ params est une liste d'identificateurs séparés par des virgules.

Par exemple, avec la directive

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

ATTENTION : un appel a CARRE(2+5) → 2+5*2+5

Préprocesseur et directives

Compilation conditionnel #if, #ifdef

La compilation conditionnelle a pour but d'incorporer ou d'exclure des parties du code source dans le texte qui sera généré par le préprocesseur. Elle permet d'adapter le programme au matériel ou à l'environnement sur lequel il s'exécute, ou d'introduire dans le programme des instructions de débogage.

Les directives de compilation conditionnelle se répartissent en deux catégories, suivant le type de condition invoquée :

- la valeur d'une expression
- l'existence ou l'inexistence de symboles.

Préprocesseur et directives

Condition liée à la valeur d'une expression

```
#if SIZE < 10
  ...
#elif SIZE >= 40
  ...
#else
  ...
#endif
```

Préprocesseur et directives

Condition liée à l'existence d'un symbole

```
  ...
#ifdef DEBUG
  ...
#else
  ...
#endif
```

Cette dernière directive peut être remplacée par l'option de compilation -Dsymbole, qui permet de définir un symbole. On peut remplacer
#define DEBUG
en compilant le programme par
gcc -DDEBUG main.c

Préprocesseur et directives

Extension du compilateur #pragma

Permet de passer des options au compilateur

Message d'erreur

#warning msg → emet un avertissement avec le texte msg

#error msg → emet un message et arrête la compilation

Préprocesseur et directives

Déclarations automatiques

Le langage C impose que le compilateur définisse un certain nombre de constantes. Sans énumérer toutes celles spécifiques à chaque compilateur, on peut néanmoins compter sur :

__FILE__ (char *) : une chaîne de caractères représentant le nom de fichier dans lequel on se trouve. Pratique pour diagnostiquer les erreurs.

__LINE__ (int) : le numéro de la ligne en cours dans le fichier.

__DATE__ (char *) : la date en cours (incluant le jour, le mois et l'année).

__TIME__ (char *) : l'heure courante (HH:MM:SS).

__STDC__ (int) : cette constante est en général définie si le compilateur suit les règles du C ANSI (sans les spécificités du compilateur). Cela permet d'encadrer des portions de code non portables et fournir une implémentation moins optimisée, mais ayant plus de chance de compiler sur d'autres systèmes.

__func__ : le nom de la fonction en cours dans le fichier (gcc)

Règles de programmation

- Règle des 3 : factoriser son code dès qu'il est redondant plus de 3 fois
- Règle des 80/20 : corrections de 20 % des bugs les plus importants éliminent 80% des crashs
- Respect des conventions d’écriture et de nommage
- Obligation de structurer le code (fonction, classes, organisation et répartition sur plusieurs fichiers)
- Obligation de commenter le code
- Obligation de documenter le code

Règles de programmation

- Obligation de gérer les versions

- Obligation d’écrire des tests unitaires et d’utiliser les assertions (en mode debug)

-Utilisation d’outils collaboratif (messagerie, suivi des tache)

Commentaires

Plusieurs syntaxes pour marquer un commentaire

→ /** */

→ /* ! */

→ // !

→ ///

Restez homogène lors de l'annotation de votre code

Documentation

- Décrit l'utilisation d'une fonction/méthode/classe dans le code

- Deux grands types d'annotations :

-Format javadoc (exemple pour pour documenter un fichier)

```
/**
 * @file pour documenter un fichier.
 * @author pour donner le nom de l'auteur.
 * @version pour donner le numéro de version.
 * @brief pour donner une description courte.
 * @details pour donner une description longue (pas obligatoire).
 * @date 01/01/2001
 */
```

Documentation

Pour documenter une fonction :

```
/**
 * @param pour documenter un paramètre de fonction/méthode.
 * @warning pour attirer l'attention.
 * @return pour documenter les valeurs de retour d'une méthode/fonction.
 * @see pour renvoyer le lecteur vers quelque chose (une fonction, une classe,
 * @since pour faire une note de version (ex : Disponible depuis ...).
 * @deprecated pour spécifier qu'une fonction/méthode/variable... n'est plus ut
 *
 */
```

Documentation

Autre type d'annotion aux format javadoc

```
/**
 * @struct pour documenter une structure C.
 * @union pour documenter une union C.
 * @enum pour documenter un type énuméré.
 * @fn pour documenter une fonction.
 * @var pour documenter une variable / un typedef / un énuméré.
 * @def pour documenter un #define.
 * @typedef pour documenter la définition d'un type.
 */
```

Cf https://www.oracle.com/technetwork/articles/java/index-137868.html

Documentation

-Format doxygen

Cf : http://www.doxygen.nl/manual/commands.html

Documentation

Annotation « aide mémoire »

```
/**
 * @todo pour indiquer un code "à faire".
 * @fixme pour indiquer un code défectueux, "à réparer".
 */
```

Selon l'IDE, il existe des annotations particulières (exemple netbeans) :

```
//todo : a finir pour demain
//fixme : ca marche pas les lundi
//XXX : a finir avant le commit
//PENDING : en attende de decision du chef de projet
//<<<<<<< Help !!!
```

Documentation

Générateur de documentation :

- javadoc
- Doxygen
- oCalmDoc
- Sphinx ...