IN 102 - Cours 3

ENSTA ParisTech

1 Types de base de C : approfondissement

1.1 Les entiers

Les entiers **int** existent en plusieurs tailles : **short**, **long**, **long long**.

long int i = −1234567 ;

La longueur d'un entier (nombre de bits) détermine le nombre de valeurs qu'il permet de représenter.

Par défaut les entiers sont signés. On peut explicitement déclarer une variable entière comme non signée par le modificateur **unsigned**.

unsigned long int i = 1234567 ;

Le signe d'un entier signé réquisitionne 1 bit (celui de poids fort), d'où 1 bit de moins pour représenter la valeur absolue de l'entier.

La combinaison signe/longueur d'un entier détermine la plage de valeurs qu'il permet de représenter.

L'arithmétique entière traitant des entiers de longueur finie, elle est modulo cette longueur.

Une arithmétique modulo implique des problèmes de débordement lors des opérations arithmétiques et des conversions signé ↔ non signé.

Une arithmétique entière implique la perte de propriétés mathématiques comme l'associativité de * et de /.

1.2 Les flottants

Les flottants sont toujours signés.

Les flottants étant de taille finie, on a un problème de représentation des valeurs minimales et maximales. De plus, comme entre 2 rationnels existe une infinité de rationnels, on a en plus un problème de précision (≪ *taille du trou* ≫ entre 2 flottants).

Les flottants existent en 2 tailles et donc en 2 précisions.

Des approximations (arrondis) surviennent lors des calculs.

Des flottants sensés représenter des valeurs identiques peuvent apparaître différents.

Des flottants sensés représenter des valeurs différentes peuvent apparaître identiques.

Ne jamais tester l'égalité de flottants avec l'opérateur =. Effectuer un test ≪ *à ϵ près* ≫ (i.e. tester si $\mid x - y \mid < \epsilon$ pour un ϵ choisi).

2 Les chaînes de caractères

Ce sont en fait des ≪ tableaux ≫ de caractères (suite contiguë de caractères terminée par le caractère '\0').

Les caractères sont représentés par le type prédéfini **char** (entiers sur 8 bits).

char c = 'V' ;

On assigne par convention à chaque ≪ *lettre* ≫ une valeur entière (son ≪ *code ASCII* ≫).

Les caractères ≪ *spéciaux* ≫ peuvent être représenté par des séquences dites d'échappement ('\n', '\\', '\t' ...)

3 Constructions de boucles

Certains processus (algorithmes) nécessitent de répéter (itérer) un traitement T.

La répétition peut avoir plusieurs formes :
- Nombre de fois connu à l'avance ?
- Sous une condition C devant être testée avant le 1^{er} traitement T ?
- Sous une condition C devant être testée après le 1^{er} traitement T ?

C propose 3 constructions d'itération (de boucle).

3.1 Faire n fois : la boucle `for`

for (*instruction*$_1$ **;** *expression* **;** *instruction*$_2$) **{** *instruction*$_3(s)$ **}**

```
#include <stdio.h>

int main ()
{
  int i ;
  for (i = 0; i < 10; i++) {
    printf ("%d\n", i) ;
  }
  return (0) ;
}
```

Sémantique :

1 Initialisation : exécuter *instruction*$_1$.

2 Si *expression* s'évalue en `true`

2.1 Exécuter *instruction*$_3(s)$.

2.2 Exécuter *instruction*$_2$ (≪post-traitement ≫).

2.3 Retourner en 2.

3.2 Tant que . . . faire : la boucle `while`

Vu dans le résumé du cours numéro 1.

3.3 Faire . . . tant que : la boucle `do-while`

do { *instruction*(s) **} while (***expression***) ;**

```
#include <stdio.h>

int main ()
{
  char c ;
  do {
    printf ("Entrez un chiffe: ") ;     // Lecture...
    scanf ("%c", &c) ;
  } while ((c < '0') ||  (c > '9')) ; // Jusqu'a succes.

  return (0) ;
}
```

Sémantique :

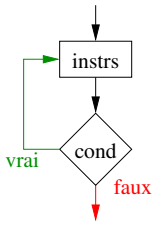
1 Exécuter *instruction*(s).

2 Si *expression* s'évalue en `true`

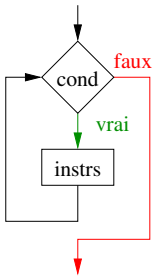
2.1 Retourner en 1.

3.4 Résumé des boucles

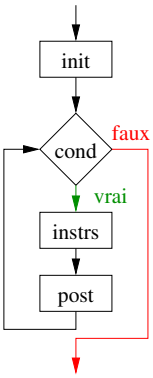
do { instrs } while (cond) ;



while (cond) { instrs }



for (init; cond; post) { instrs }



4 Conditionnelle généralisée

La conditionnelle généralisée permet de tester une valeur contre plusieurs cas mutuellement exclusifs sans cascader des **if** imbriqués.

```
#include <stdio.h>

int main ()
{
  int x ;

  printf ("Entrez un nombre: ") ;
  scanf ("%d", &d) ;              // Lecture...
  switch (x) {
    case 0: printf ("x: 0\n") ; break ;
    case 1: printf ("x: 1\n") ; break ;
    case 4: printf ("x: 4\n") ; break ;
    default: printf ("Autre cas\n") ; break ;
  }
  return (0) ;
}
```

Chaque **case** doit être suivi d'une constante (entière) puis de :

Chaque fin de cas doit être terminé par **break ;**

Le traitement exécuté est celui du cas dont la constante est égale à la valeur testée.

5 Quelques «outils»

5.1 Constantes, macros et préprocesseur

Plutôt que des constantes littérales, (42) il est préférable d'utiliser des macros.

#define SIZE (42)

Les macros sont remplacées textuellement avant compilation par le préprocesseur C (`cpp`).

```
#define SIZE (   42 )
    ...
int i = SIZE *    SIZE ;
if (j < SIZE - 1) ...
```

⇒

```
int i = (   42 ) *    (   42 ) ;
if (j < (   42 ) - 1) ...
```

Les macros peuvent être paramétrées : remplacement textuel en substituant le paramètre par le texte argument lors de l(utilisation.

```
#define TWICE(x)((x)* (x))
#define MUL(a,b)((a)* (b))

int v = Bla TWICE(ah bon ?) ;
int w = MUL(((15 )) , autre truc) ;
```

⇒

```
int v = Bla ((ah bon ?) * (ah bon ?)) ;
int w = Bli ((((15 )))* ( autre truc)) ;
```

5.2 Alias de type

Il est possible de nommer un type déjà existant par un alias.

Le nouveau nom peut être utilisé comme n'importe quelle expression de type.

typedef *expression-de-type nouveau-nom* **;**

```
typedef unsigned long long int ulli_t ;

ulli_t myvar ;
ulli_t myfun (int i, ulli_t j) { ... }
```

5.3 Contrainte de type

Permet de changer le type selon lequel une donnée est ≪vue≫.

Utilisation d'un cast : (*nom-type*) *expression*.

```
float i = ... ;
j = 15 * ((unsigned long) (i + 1)) + j ;
```

Attention aux pertes d'information / corruptions.