Université de Reims Champagne-Ardenne
UFR Sciences Exactes et Naturelles
Licence Informatique
INFO0402: Programmation Orientée Objet

TD N°1

Exercice 1: Modèle de données

1. A quoi correspond le modèle de données LP64 ?
2. Si je compile un programme écrit sur un système avec un modèle de données LP64, et que je le recompile sur un système avec un modèle de données ILP32, quels sont les changements que je peux observer et quels problèmes cela peut-il poser ?
3. pourquoi est-il utile d'avoir des types entiers à longueur fixe ?

Solution:

1. int=4/long int=8/*=8 (système 64 bits) Linux, Mac OS X.
2. sizeof(long int) et sizeof(*) changent. Donc,
   - la taille de toute structure qui utilise ces types va changer.
   - si un long int utilise une valeur > max(int), le calcul ne sera plus correct.
   - un pointeur 32 bits ne permet d'accéder au plus 4GB de mémoire (en général, pas plus de 2GB). Donc, si plus de 2GB de mémoire sont alloués, le programme échouera.
3. permet de construire plus facilement des structures de données avec des tailles fixes (fichiers, paquets, ...).

Exercice 2: Conversions

1. Qu'est ce qu'une promotion numérique ?
2. Une promotion numérique est-elle toujours sans perte de précision ?
3. Que se passe-t-il si l'on convertit une entier non signé vers entier non signé plus petit ?
4. Soit le code suivant :

```
unsigned char c1 = 100, c2 = 3, c3 = 4;
unsigned char r1 = c1*c2;
unsigned char r2 = r1 / c3;
unsigned char r = c1 * c2 / c3;
```

   Donner la valeur de `r1`, `r2` et `r`.
5. Existe-t-il des entiers qui peuvent être convertit en flottant exactement ?
6. Existe-t-il des flottants qui peuvent être convertit en entier exactement ?
7. Quelle est la différence entre la troncature d'un flottant en entier, et les fonctions d'arrondi `floorf` et `ceilf` ?

Solution:

1. conversion d'un type permettant de stocker le type original "sans perte de précision".
2. oui (entier vers entier plus grand, float vers double).
3. modulo 2^p où p est le nb de bits du type vers lequel on convertit.
4. $r1 = (100 \times 3) \mod 256 = 300 \mod 256 = 44$, $r2 = 44/4 = 11$, $r = (100 \times 3/4) \mod 256 = (300/4) \mod 256 = 75 \mod 256 = 75$. Pour le dernier calcul, remarquer que les entiers 8 bits sont convertis en 32 bits, s'effectuent en 32 bits, puis sont reconvertis en 8 bits.
5. oui, pour un nombre entier dont le nb de bits significatif d'un entier est inférieur à celui de la mantisse.
6. oui, les mêmes qu'à la question précédente.
7. conversion flottant $\rightarrow$ entier : $3,2 \rightarrow 3$, $-3,2 \rightarrow -3$ = vers 0
   `floorf` : $3,2 \rightarrow 3$, $-3,2 \rightarrow -4$ = vers le bas
   `ceilf` : $3,2 \rightarrow 4$, $-3,2 \rightarrow -3$ = vers le haut.

Exercice 3: Comprendre `lvalue` et `rvalue`

Rappel : dans ce cours, on utilise les sens de `lvalue` et `rvalue` donnés à partir du C_{11}^{++}, et qui ne recouvrent pas le sens qu'on leurs donne habituellement.

1. Donner le sens de `lvalue` et `rvalue`.
2. Quelle est la différence entre une variable sans modificateur et avec les modificateurs `&`, `&&` et `*` ?
3. Pourquoi, lorsque je déclare une variable de type référence sur une `rvalue`, alors cette variable est une `lvalue` ?

```
int a = 5;
int &b = a;
int c = b;
int d = (a+4)/2;
int &e = 5;
int &f = a/2;
int &&g = a;
int &&h = b;
int &&i = a/4;
int &&j = 8;
```

code 1

```
int    fun1(), &fun2(), &&fun3();
int A = fun1();
int &B = fun2(a);
int &&C = fun3(5);
int D = fun2(a);
int E = fun3(7);
int &F = fun3(a);
int &G = fun1(a);
int &&H = fun2(a);
int &&I = fun1();
```

code 2

4. Dans le code 1 ci-dessus, pour chaque ligne, identifier si dans les parties gauche et droite des définition, ce sont des `lvalue`s ou des `rvalue`s, puis en déduire si c'est une expression C_{11}^{++}valide.
5. Dans le code 2 ci-dessus, pour chaque ligne, identifier si dans les parties gauche et droite des définitions, ce sont des `lvalue`s ou des `rvalue`s, puis en déduire si c'est une expression C_{11}^{++}valide.
6. Quel est le sens du qualificateur `const` lorsqu'il est utilisé dans la définition d'une variable de type `int`, `int&`, `int&&` ?
7. Que deviennent les affectations de la question 3 si l'on qualifie `const` toutes les variables ?

Solution:

1. une `lvalue` est une expression identifiable (= on peut récupérer son adresse). Une `rvalue` est une expression déplaçable.

   Par défaut, une `lvalue` n'est pas déplaçable, et une `rvalue` n'est pas identifiable.

   Voir l'annexe sur la typologie des valeurs et le déplacement de la leçon 2.
2. sans modificateur = variable lvalue, `&` = référence à une lvalue, `&&` = référence à une rvalue, `*` = pointeur (produit un **type différent**).
3. elle est nommée. c'est donc une `lvalue`.
4. voir le tableau (I=identifiable, D=déplaçable, N+=non)

code	gauche	droite	valide	explication
`int a = 5;`	lvalue (I/ND)	rvalue (NI/D)	oui	rvalue copiée dans lvalue
`int &b = a;`	lvalue (I/ND)	lvalue (I/ND)	oui	ref lvalue depuis lvalue
`int c = b;`	lvalue (I/ND)	lvalue (I/ND)	oui	ref lvalue copiée dans lvalue
`int d = (a+4)/2;`	lvalue (I/ND)	rvalue (NI/D)	oui	rvalue copiée dans lvalue
`int &e = 5;`	lvalue (I/ND)	rvalue (NI/D)	non	impossible rvalue ⇒ ref lvalue
`int &f = a/2;`	lvalue (I/ND)	rvalue (NI/D)	non	impossible rvalue ⇒ ref lvalue
`int &&g = a;`	lvalue (I/ND)	lvalue (I/ND)	non	impossible lvalue ⇒ ref rvalue
`int &&h = b;`	lvalue (I/ND)	lvalue (I/ND)	non	impossible lvalue ⇒ ref rvalue
`int &&i = a/4;`	lvalue (I/ND)	rvalue (NI/D)	oui	ref rvalue depuis rvalue
`int &&j = 8;`	lvalue (I/ND)	rvalue (NI/D)	oui	ref rvalue depuis rvalue

5. voir le tableau

code	gauche	droite	valide	explication
`int A = fun1();`	lvalue (I/ND)	rvalue (NI/D)	oui	rvalue copiée dans lvalue
`int &B = fun2(a);`	lvalue (I/ND)	lvalue (I/ND)	oui	ref lvalue depuis lvalue
`int &&C = fun3(5);`	lvalue (I/ND)	xvalue (I/D)	oui	ref rvalue depuis rvalue
`int D = fun2(a);`	lvalue (I/ND)	lvalue (I/ND)	oui	ref lvalue copiée dans lvalue
`int E = fun3(7);`	lvalue (I/ND)	xvalue (I/D)	oui	ref rvalue copiée dans lvalue
`int &F = fun3(a);`	lvalue (I/ND)	xvalue (I/D)	non	impossible ref rvalue ⇒ ref lvalue
`int &G = fun1(a);`	lvalue (I/ND)	rvalue (NI/D)	non	impossible rvalue ⇒ ref lvalue
`int &&H = fun2(a);`	lvalue (I/ND)	lvalue (I/ND)	non	impossible ref lvalue ⇒ ref rvalue
`int &&I = fun1();`	lvalue (I/ND)	rvalue (NI/D)	oui	rvalue vers ref rvalue

Une `xvalue` est une expression qui est à la fois identifiable (on a sa référence), et déplaçable (elle est en fin de vie).

6. qualification `const` :

   `const int` : entier constant

   `const int&` : réference vers un entier constant

   `const int&&` : réference vers un entier temporaire constant (en pratique, inutile)

7. avec le qualificateur `const`

code	gauche	droite	valide	explication
`const int a = 5;`	lvalue (I/ND)	rvalue (NI/D)	oui	rvalue copiée dans clvalue
`const int &b = a;`	lvalue (I/ND)	lvalue (I/ND)	oui	cref lvalue sur lvalue (RO)
`const int c = b;`	lvalue (I/ND)	lvalue (I/ND)	oui	cref lvalue copiée dans clvalue
`const int d = (a+4)/2;`	lvalue (I/ND)	rvalue (NI/D)	oui	rvalue copiée dans clvalue
`const int &e = 5;`	lvalue (I/ND)	rvalue (NI/D)	oui	rvalue ⇒ cref lvalue
`const int &f = a/2;`	lvalue (I/ND)	rvalue (NI/D)	oui	rvalue ⇒ cref lvalue
`const int &&g = a;`	lvalue (I/ND)	lvalue (I/ND)	non	impossible lvalue ⇒ cref rvalue
`const int &&h = b;`	lvalue (I/ND)	lvalue (I/ND)	non	impossible lvalue ⇒ cref rvalue
`const int &&i = a/4;`	lvalue (I/ND)	rvalue (NI/D)	oui	cref rvalue depuis rvalue
`const int &&j = 8;`	lvalue (I/ND)	rvalue (NI/D)	oui	cref rvalue depuis rvalue

rvalue ⇒ cref lvalue : possible car la cref lvalue fait référence à une variable temporaire sans la modifier. Elle peut prolonger sa durée de vie si nécessaire sur la portée de la lvalue. Destruction de la lvalue ⇒ destruction de la rvalue. Rappel : une rvalue nommée est une lvalue.

lvalue ⇒ cref rvalue : pas de sens, car un objet temporaire ne peut décider de la destruction d'un objet non temporaire.

Exercice 4: Unité de traduction, durée de stockage et liens

1. Rappeler ce qu'est une unité de traduction.
2. Donner un exemple d'un code contenant des objets ayant des durées de stockage automatique, statique et dynamique. On essayera de donner des plusieurs exemples pour chaque durée de stockage.
3. Donner un exemple de code modulaire contenant des objets sans lien, un lien interne, un lien externe.
4. Soit le code suivant :

```
void fun() {
    char *a = "tralala", b[] = "tralala";
    a[0] = 'T';
    b[0] = 'T';
}
```

   Pourquoi ce code provoque-t-il une erreur de segmentation ?

Solution:

1. cf section 4.1 du chapitre 1.
2. construire un exemple avec :

   durée de stockage automatique : variable locale, variable dans un sous-bloc, paramètre d'une fonction

   durée de stockage statique : variable globale, variable statique

   durée de stockage dynamique : dynamique : allocation dynamique

3. construire un exemple avec :

   objet sans lien : variable locale dans une fonction.

   objet avec lien interne : appel d'une fonction ou d'une variable globale définie dans l'unité de traduction.

   objet avec lien externe : appel d'une fonction ou d'une variable globale définie dans une autre unité de traduction.

4. `a` pointe vers une zone de mémoire constante (zone `text` du programme), et sa modification provoque l'erreur.

   `b` est un tableau dans le stack (comme une variable locale) initialisé avec la chaine "tralala". Donc, pas de problème.

Exercice 5: Surcharge de fonction 1

1. Est-il possible en C++ de faire en sorte que la fonction `max` fonctionne à la fois pour des entiers et des flottants sans utiliser ni les macros du préprocesseurs ni les templates.
2. Pourquoi l'appel `max(3,4.5f)` échoue-t-il alors ?
3. Pourquoi a-t-on intérêt à définir cette fonction inline ?
4. On veut écrire une seule fonction `Rand` qui permet les appels suivants :
   - `Rand()` retourne un nombre aléatoire entre 0 et 1.
   - `Rand(6)` retourne un nombre aléatoire entre 0 et 6.
   - `Rand(2,10)` retourne un nombre aléatoire entre 2 et 10.

   Expliquer comment écrire une telle fonction.
5. Pourquoi serait-il préférable que cette fonction soit `inline` ?

Solution:

1. en utilisant les surcharges de fonction, int max(int,int) et float max(float,float).
2. le compilateur n'est pas en mesure de déterminer quelle fonction appeler.
3. car c'est une fonction courte, et qu'un appel de fonction est un gaspillage de ressources dans ce cas.
4. `int Rand(int u=0,int v=0)`

   si $u = v$, alors la loi est sur $[0, 1]$, si $u > v$ alors la loi est sur $[0, u]$, si $u < v$ alors la loi est sur $[u, v]$.

   Afin que cela fonctionne sur tous les entiers, remplacer 0 par `std::numeric_limits<int>::min()`.
5. Petite fonction possiblement très fréquemment appelée.

Exercice 6: Surcharge de fonction sur un type qualifié et modifié

La table et les règles qui permettent de répondre à ces questions sont à la page 29. Pour les combinaisons de surcharges suivantes, on se pose les deux questions :

a ces surcharges sont-elles possibles ?
b quels sont les types captés par les différentes surcharges, et quelles sont ceux qui ne le sont pas ?

1. `void fun(int)` et `void fun(const int)`
2. `void fun(int)` et `void fun(int&)`
3. `void fun(int)` et `void fun(int&&)`
4. `void fun(const int)` et `void fun(int&)`
5. `void fun(int&)` et `void fun(const int&)`
6. `void fun(int&)` et `void fun(int&&)`
7. `void fun(int&)` et `void fun(const int&&)`
8. `void fun(const int&)` et `void fun(int&&)`
9. `void fun(const int&)` et `void fun(const int&&)`
10. `void fun(int&&)` et `void fun(const int&&)`

Solution:

	surcharges	solution
1.	`void fun(int)` et `void fun(const int)`	non : règle 1 (conflit même catégorie, ici value)
2.	`void fun(int)` et `void fun(int&)`	non : règle 2 (pas de surcharge value par lvalue)
3.	`void fun(int)` et `void fun(int&&)`	non : règle 2 (pas de surcharge value par rvalue)
4.	`void fun(const int)` et `void fun(int&)`	non : règle 2 (pas de surcharge value par lvalue)
5.	`void fun(int&)` et `void fun(const int&)`	non : règle 1 (conflit même catégorie, ici lvalue)
6.	`void fun(int&)` et `void fun(int&&)`	oui, `fun(int&)` capte lvalue, `fun(int&&)` capte rvalue
7.	`void fun(int&)` et `void fun(const int&&)`	oui, `fun(int&)` capte lvalue, `fun(const int&&)` capte rvalue + const rvalue
8.	`void fun(const int&)` et `void fun(int&&)`	oui, `fun(const int&)` capte lvalue + const lvalue, `fun(int&&)` capte rvalue
9.	`void fun(const int&)` et `void fun(const int&&)`	oui, `fun(const int&)` capte lvalue + const lvalue, `fun(int&&)` capte rvalue + const rvalue
10.	`void fun(int&&)` et `void fun(const int&&)`	non : règle 1 (conflit même catégorie, ici rvalue)

Exercice 7: Macro du précompilateur

1. soit le macro suivante : `#define MYMACRO(a,b) if (a) fun(b)`. Pourquoi cette macro peut-elle poser des problèmes et comment le corriger ?
2. soit le macro suivante : `#define abs(x) ((x)>=0 ? (x) : -(x))`. Pourquoi cette macro peut-elle poser des problèmes et comment le corriger ?
3. soit le macro suivante :
```
#define MYMACRO(a,b) \
    instruction1; \
    instruction2; \
    /*...*/ \
    instructionN;
```
.
Pourquoi cette macro peut-elle poser des problèmes et comment le corriger ?

Solution:

1. si on écrit :
```
if (x) MYMACRO(3,4)
else x++;
```
alors le code compris par le compilateur est :
```
if (x) {
    if (a) fun(b) else x++;
}
```
2. `MYMACRO(fun(x))`
alors le code compilé est :
`((fun(x))>=0 ? (fun(x)) : -(fun(x)))`
Donc, `fun(x)` est appelée deux fois.
Utiliser une fonction inline qui fait la même chose.
3. La solution n'est pas simple :
   - Avec `while (x) MYMACRO(a,b);`, seule la première instruction est exécutée par le `while`.
   - si ajoute des accolades autour des instructions, alors l'écriture :
     `if (whatever) MYMACRO(foo, bar); else baz;`
     provoque une erreur de compilateur (en raison du ; après le bloc).
   - une solution serait de placer les instructions dans le bloc :
```
#define MYMACRO(a,b) \
    do { \
      instruction1; \
      instruction2; \
      /*...*/ \
      instructionN; \
    } while(false)      // pas de ; ici
```

alors l'écriture `if (whatever) MYMACRO(foo, bar); else baz;` provoque le résultat attendu.

Conclusion :

- dans la plupart des cas, on peut trouver un appel de la macro qui a un comportement non souhaitable.
- éviter d'utiliser des macros sauf lorsque cela est vraiment nécessaire, et ne peuvent pas être réalisée en utilisant des fonctions inline ou des templates C++.
- le seul cas où l'on ne peut vraiment pas s'en passer est pour la réalisation de code dans la compilation conditionnelle.