Formation C++

life and death pointers, references const

(c) Guy Dalberto 2007, 15 octobre 2007

Alcos

Plan Général

Alcos

- Mémoire, Pile, durée de vie
- références (homonymes)
- pointeurs
- const
- #define

Références

Alcos

- C++ in action, Bartosz Milewski, http://www.relisoft.com
  sous-titre : ***Industrial Strength Programming Techniques***
- C++ Coding Standards, Sutter && Alexandrescu
- http://www.freshsources.com, Chuck Allison
- The C++ Programming Language 3th Ed, Stroustrup
- Thinking in C++, Bruce Eckel, http://www.mindview.net

Mémoire, Pile, durée de vie

Alcos

Variables, déclaration, localisation

Alcos

- Pour pouvoir écrire des expressions telles que : **x = y + 2 ;**
  Les variables x et y doivent avoir été déclarées
- La déclaration spécifie le type

```
int  x ;
Y   y ;
```

- La déclaration spécifie aussi la **localisation** :
  - variables automatique (valides à l'intérieur d'un bloc de code)

```
{ …  ; int   x ; …  }
```

  - arguments d'une fonction

```
void f(int x) { … }
```

  - variables globales ou statiques

```
int  g_x ;
int fct() { static int s_x ; ... }
extern int  g_x ;
```

Zones mémoire : Code, pile, tas

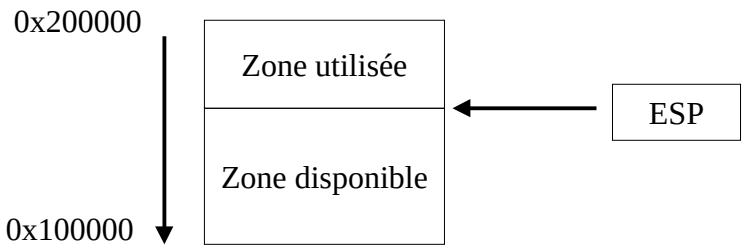
Alcos

- chaque processus dispose d'un espace mémoire alloué par le système d 'exploitation. Cet espace mémoire est partagé en zones
- le code exécutable (protégé en général en écriture)
- les données dont la durée de vie est celle du processus
  - les constantes
  - les variables globales (éventuellement statiques)
- les données dont la vie est automatiquement contrôlée par le point d 'exécution courant, contenues dans la pile (Stack)
  - utilisée pour : arguments de fonction, variables locales et temporaires
  - chaque Thread d 'un processus dispose d 'une pile
- les données dont la vie est spécifiquement contrôlée par les décisions du programme.
  - elles sont allouées dynamiquement (Heap)
  - malloc, free, new et delete

Pile

Alcos

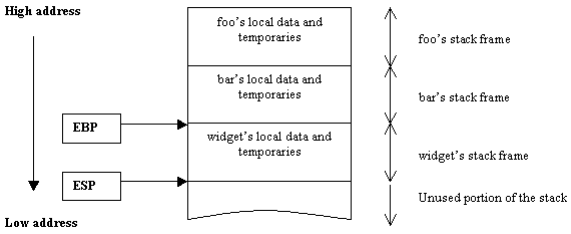
- utilisée en mode LIFO (Last In First Out)
- Un registre spécial (ESP) contient l 'adresse de la limite entre la zone utilisée et la zone disponible
- La zone utilisée augmente lors des appels de fonctions et diminue lors des retours. La pile représentée grandit vers le bas.
- Des instructions spéciales manipulent la pile (push, call, ret)



Stack Frame et Frame Pointer

Alcos

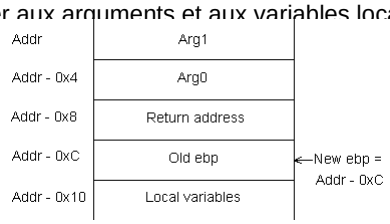
- Etat de la pile lorsque la fonction foo appelle la fonction bar qui appelle widget.
- Les UCs Intel 32 bits fournissent aussi un registre EBP, qui est souvent utilisé pour accéder :
  ```
  aux arguments (indexation positive)
  aux variables locales (indexation négative)
  à l'EBP de la fonction appelante
  ```
- L 'ensemble de ces données s 'appelle un Stack Frame



Appel de fonction, détail

Alcos

- Quand une fonction est appelée, l'appelant :
  - met en pile les arguments dans un ordre qui dépend des conventions d 'appels (en général du dernier au premier).
  - exécute l 'instruction **call** qui met en pile le registre EIP (adresse de l 'instruction suivante) et met dans le EIP l 'adresse de la fonction appelée
- L'appelé :
  - sauvegarde le EBP de l 'appelant dans la pile
  - copie le ESP courant dans le EBP
  - décrémente le ESP de la taille nécessaire pour les variables locales
  - utilise le EBP pour accéder aux arguments et aux variables locales



Optimisation, mémoire et registres

Alcos

- Le compilateur a beaucoup de latitude quand à l'implémentation
- Il peut stocker les variables temporaires ou locales dans des registres pour en accélérer l 'accès
- Il peut les supprimer complètement (optimisation)
- Certaines conventions d 'appels prévoient de passer certains arguments par registres plutôt que par la pile.

Références = Homonymes

Alcos

Référence : Un autre nom

Alcos

- Une Référence est un autre nom permettant de désigner un objet, Stroustrup §5.5
- notation :X& rx // rx référence une variable de type X
- L'utilisation principale des références est de spécifier les arguments et les valeurs rendues des fonctions en général et des opérateurs surchargés en particulier

Référence dans un bloc

Alcos

- Il est possible d 'utiliser une référence à l 'intérieur d 'un bloc :

```
int  getValue_4() {
    int i = 1 ;
    int& k = i ; // k est un alias de i
    i = 3 ;      // => k == 3
    k = 2 ;      // => i == 2
    i++ ;
    k++ ;
    return k ;   // (k == 4) && (i == 4)
}
```

Référence dans un bloc, code généré

Alcos

```
int  add2toV(int v) {
    int i = v ;
    int& k = i ;
    i++ ;
00401F00  mov         eax,dword ptr
 [esp+4]
    k++ ;
00401F04  add         eax,2
    return k ;  // v + 2
}
00401F07  ret
```

- **k** et **i** sont des noms qui n'existent que dans le code source

Références, initialisation

Alcos

- Une référence est **toujours** initialisée.
- Une variable de bloc doit spécifier l'initialisation

```
int& k = i ;
```

- Un argument de fonction sera initialisé dynamiquement lors de l'appel

```
void  f(X& x) {
    ...
}
void f() {
   X  x ;
   f( x ) ;  // écrit la référence
             // dans la pile
}
```

Références, opérations

Alcos

- En dépit des apparences, **aucun** opérateur n'agit sur une référence *Stroustrup § 5.5*
- Toutes les opérations ont lieu sur l'objet référencé.

```
void g() {
  int ii = 0 ;
  int& rr = ii ;
  rr++  ;  // ii is incremented to 1
  int* pp = &rr ; // pp points to ii
  rr = 4 ; // modifie ii
}
```

- Il n'est pas possible de réaffecter une référence

Références valides et invalides

Alcos

- Un référence est valide si elle désigne un objet vivant

```
// compiler warning
// returning address of local variable
static int& getInvalidReference() {
   int i = 5 ;
   return i ;
}
// NO warning
static int& getInvalidReference() {
   int i = 5 ;
   int& ri = i ;
   return ri ;
}
```

Pointeurs

A Pointer is a mutable reference

Bartosz Milewski

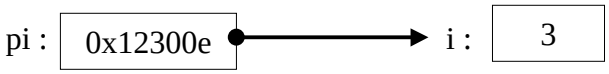
Alcos

Pointeur : déclaration et initialisation

Alcos

- L'opérateur **&** (**AdressOf)** permet d 'obtenir l 'adresse d 'un objet
- Un pointeur est une variable qui peut contenir l 'adresse d 'un objet

```
int i ;
i = 3 ;          // 3 => i
int* pi = &i ;   // adresse de i => pi
```



Pointeur : déréférencement

Alcos

- L'opération principale effectuée avec un pointeur est le **déréférencement**, qui est l'opération consistant à obtenir la référence de l'objet pointé avec l 'opérateur *

```
int i = 3 ; int j ;
```

- avec un pointeur

```
int* pi = &i ;
*pi = 5 ;        // 5 => i
j = *pi ;        // copie i => j
```

- avec une référence

```
int& ri = i ;
ri = 5 ;         // 5 => i
j = ri ;         // copie i => j
```

Pointeur : accès à une Structure

Alcos

- Un pointeur de type X* permet d accéder aux éléments d 'une structure X

```
struct X {
 int  i ;
 long j ;
} ;
X  x1 ;
X* px = &x1 ;
```

- A l 'aide de l 'opérateur -> structure dereference operator

```
px->i = 5 ;
px->j = 7 ;
```

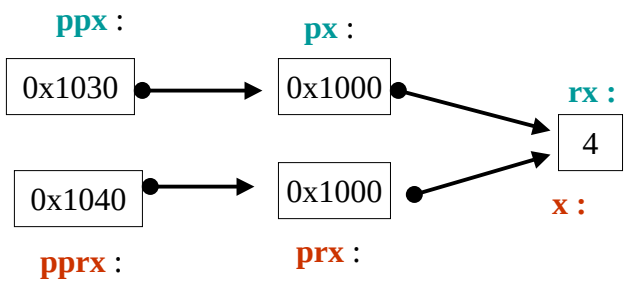
- Qui n 'est qu'un raccourci d 'écriture pour :

```
(*px).i = 5 ;
(*px).j = 7 ;
```

Pointeur sur un pointeur

Alcos

```
int  x   = 4 ;
int* px = &x ;
int** ppx = &px ;
int& rx = x ;
int* prx = &rx ;
int** pprx = &prx ;
```



Pointeur, quel usage ?

Alcos

Don't use pointers unless there is no other way (Milewski)

- Chaque fois qu'une fonction utilise un pointeur fixe sur un et un seul objet, elle devrait utiliser une référence.
- Autrement dit, il vaut mieux écrire :

```
int getI(X& rx) { return rx.i ; }
void setI(X& rx, int i) {rx.i= i ; }
```

que

```
int  getI(X* px) { return px->i ; }
void  setI(X* px, int i) {px->i = i ; }
```

- Il est très simple de transformer un pointeur en référence :

```
X* px = new X ;
setI( *px, 4 ) ;
X& x = *px ;
int i = getI(x) ;
```

Pointeur valide et invalide

Alcos

- Un pointeur est valide s 'il contient l 'adresse d 'un objet vivant du type adéquat
- Un pointeur est invalide dans tous les autres cas
- Le déréférencement (utilisation du contenu) d 'un pointeur invalide provoque un comportement indéterminé
- Dans le meilleur des cas on aura immédiatement une exception « Access Violation »
- Dans le pire des cas, on va écraser une variable qui ne servira que 2 minutes plus tard
- C 'est la manière la plus simple de créer des bugs pseudo aléatoires

Pointeur Nul

Alcos

- Aucun objet ne peut légalement avoir pour adresse 0
- L'utilisation d 'un pointeur nul arrête **immédiatement** le programme fautif

```
int* pi = 0 ;
int j = *pi ;
Access violation reading location 0x00000000
*pi = 2 ;
Access violation writing location 0x00000000
```

- Ou si vous préférez la constante NULL

```
const   int   NULL = 0 ;
int* pi = NULL ;
```

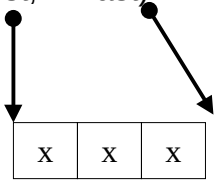
- Une erreur due à un pointeur nul est **beaucoup moins difficile à identifier** que celle produite par un pointeur non nul pointant sur un objet invalide
- Conclusion, mettre 0 dans tous les pointeurs invalides

Pointeur, rôle

Alcos

- Un pointeur n 'est pas lié en permanence à un objet particulier. Il peut être modifié par le programmeur.
- Cela permet d 'utiliser des pointeurs pour accéder à des tableaux ou à des objets dynamiquement alloués
- Autrement dit : un pointeur permet d 'accéder à des **ensembles variables** d'objets

. long total(X* first, X* last)



const

Alcos

const est votre ami

Alcos

- Le mot-clé **const** modifie un type, c'est à dire qu'il :
  - restreint la manière dont un objet peut être utilisé
  - ne spécifie pas la manière dont l'objet est alloué
- exemples de signature :

```
int    strcpy(char* dest, const char* src) ;
```

- qui protègera les programmeurs dislexiques

```
void doSomething(const char* msg) {
  char   buf[256] ;
  strcpy(msg, buf) ;       // oops !
  ..
}
```

'strcpy' : cannot convert parameter 1 from 'const char *' to 'char *'

#define

Alcos

- de nombreuses déclarations **#define** devraient être remplacées par des **enum** ou des **const**

```
enum { Red=0xFF0000, Green=0x00FF00, Blue=0x0000FF } ;
```

- en C++, le code suivant est légal :

```
const   int  xxxSize = 20 ;
char    buf  [xxxSize]  ;
```

- en C ce code est illégal et produit une erreur :

  expected constant expression

Références et Pointeurs const

Alcos

- référence sur un objet constant :

  const X& x ;

- pointeur sur un objet constant :

  const X* px ;

- pointeur constant sur un objet variable :

  X* const px ;

- pointeur constant sur un objet constant :

  const X* const px ;

const en C

Alcos

- Le mot-clé const est utilisé de manière légèrement différente en C++ et en C.
- En language C, le compilateur attribue toujours de la mémoire.
  - **const X** est simplement un **X** non modifiable
  - il ya toujours une zone mémoire allouée, associée à un symbole de type **external linkage**
  - si un fichier header contenant une définition est inclus par plusieurs unités de compilation, le linker signale une erreur double définition
    ```
    const int bufSize = 100 ;
    ```
  - on peut déclarer l'existence d'une variable const, le linker résoudra :
    ```
    extern const int bufSize;
    ```
  - il n'est pas possible d'écrire :
    ```
    const int bufSize = 100 ;
    int buf[bufSize] ;
    ```

const en C+

Alcos

- le compilateur évalue immédiatement la constante
- il est possible d'écrire :
  ```
  const int bufSize = 100 ; int buf[bufSize] ;
  ```
- le compilateur n'attribue de la mémoire que si c'est nécessaire, autrement dit :
  - si cela devient compliqué : const struct X [ ] = { {… }, {…} }
  - si quelqu'un prélève l'adresse d'une constante
    ```
    void f(const int& ri) ; void g(const int* pi) ;
    const int a = 2 ; const int b = 2 ;
    f(a) ; g(&b) ;
    ```
- par défaut la constante est de type **internal linkage**
  - si un fichier header contenant une définition est inclus par plusieurs unités de compilation, de 0 à N zones mémoires seront allouées
  - on peut passser en **external linkage** en rajoutant le mot clé extern
    ```
    extern const int bufSize = 100;
    ```

const se répand comme un virus

Alcos

- const a tendance à se répandre, car la plupart des paramètres sont constants et la plupart des fonctions membres ne modifient pas les objets
- mais malheureusement toutes les bibliothèques ne sont pas const-correct
- dans ce cas, on peut supprimer la constness en écrivant :

```
void f(const X& cx) {
 X& mx = const_cast<X&>(cx) ; // parce que ….
 ….
}
```

Signatures

Alcos

Passage par valeur et par référence

Alcos

- Il y a toujours une copie sur la pile ou dans un registre
  - Le passage par valeur copie le paramètre, donc au minimum la taille d'un int, et au maximum .....
  - Le passage par référence copie l'adresse du paramètre
  - Le passage d'un pointeur est un passage par valeur. Le paramètre copié est une adresse. Le coût est donc exactement le même que le passage par référence.
- Syntaxe et taille des paramètres sur la pile (UC 32 bits)

```
void fct(int, X*, X&) => 12
void fct(int, float, float*, X*, X&) => 20
```

- Passage d 'une structure par valeur

```
void fct(int, float, float*, X*, X) => 16 + sizeof(X)
```

Pointeurs et Références, comparaison

Alcos

- 2 manières de dire la même chose au compilateur :

```
int dateCmp(const Date* const lhs , const Date* const rhs) {
        if ( lhs->annee < rhs->annee )
        ...
int dateCmp(const Date& lhs, const Date& rhs) const {
        if ( lhs.annee < rhs.annee )
        ...
```

- La différence est dans l'appel :

```
if ( 0 < dateCmp( &dateA, &dateB ) )
      ..
```

devient

```
if ( 0 < dateCmp( dateA, dateB ) )
      ..
```