Plan

Gestion des erreurs et des exceptions

Gestion des erreurs par retour de valeur

- Technique populaire pour C et C++
- C'est la responsabilité de l'appelant de différentier un résultat valide et une erreur
- Pas de sémantique commune : chaque programmeur peut implémenter de sa manière la logique de différentiation entre un code d'erreur ou un résultat valide
- Facile d'ignorer les erreurs

Gestion des erreurs et des exceptions

Les codes d'erreurs sont souvent ignorés

- Une erreur commune est de ne pas valider les valeurs de retour
- Peut produire des vulnérabilités

Valeur de retour ignorée peut produire un buffer overflow

```
char buf[10], cp_buf[10];
fgets(buf, 10, stdin);
strcpy(cp_buf, buf);
```

Version correcte

```
char buf[10], cp_buf[10];
char * ret = fgets(buf, 10, stdin);
if (ret != buf) {
report_error(errno);
return;
}
strcpy(cp_buf, buf);
```

Gestion des erreurs et des exceptions

Encodage d'un code d'erreur

Une fonction ne possède qu'une seule valeur de retour. Plusieurs techniques sont utilisées pour encoder un code d'erreur dans une valeur de retour :

- Utiliser une valeur < 0 pour indiquer une erreur lorsque une valeur de retour valide est un entier positif
- Utiliser la valeur 0 pour indiquer une erreur en utilisant une variable globale pour indiquer le code d'erreur
- Inclure une valeur valide pour indiquer l'existence d'une erreur (Par exemple 0 indique à la fois une valeur correcte ou bien l'existence d'une erreur)

Gestion des erreurs et des exceptions

Identifier le problème dans ce code

```
1  #define MAXVALUE 1023
2  #define ERR 0
3  int myStrlen(char *s) {
4     int c=0,i;
5     if (s == NULL) return ERR;
6     for (i = 0; i < MAXVALUE; i++) {
7        if (s[i] == '\0') return c;
8        c++;
9     }
10    return ERR;
11 }
```

Gestion des erreurs et des exceptions

Implémentation de la logique de gestion d'erreurs

➥ Identifier le problème

```
1  #define MAXVALUE 1023
2  #define ERR_NULL -1
3  #define ERR_TOOBIG -2
4
5  /*** Creates a copy of a C-string ***/
6  char * createCopy(char *s) {
7     int c=0,i; char *s1;
8     sl = malloc(sizeof(char)*MAXVALUE);
9     if (s == NULL || sl == NULL)
10       return (char *) ERR_NULL;
11    for (i = 0; i < MAXVALUE; i++) {
12       sl[i] = s[i];
13       if (s[i] == '\0') return sl;
14    }
15    free(sl);
16    return (char *) ERR_TOOBIG;
17 }
```

Gestion des erreurs et des exceptions

Implémentation de la logique de gestion d'erreurs

```
1  char * createCopy(char *s) {
2     int c=0,i,err_val=0;
3     char * sl=NULL;
4     sl = malloc(sizeof(char)*MAXVALUE);
5     if (s == NULL || sl == NULL) {
6        err_val = ERR_NULL;
7        goto ERR_HANDLER;
8     }
9     for (i = 0; i < MAXVALUE; i++) {
10       sl[i] = s[i];
11       if (s[i] == '\0') return sl;
12    }
13    err_val = ERR_TOOBIG;
14 ERR_HANDLER:
15    if (sl != NULL) free(sl);
16    return err_val;
17 }
```


- Ne pas mélanger la gestion d'erreurs avec la logique fonctionnelle
  - Permet d'avoir un code plus lisible et plus facile à maintenir
  - Éviter de changer la logique fonctionnelle à la production d'une erreur
  - Rendre les activités de recouvrement plus consistantes : libération de l'espace mémoire alloué, libération de verrous, etc.

Gestion des erreurs et des exceptions

Les assertions

- Une assertion est un macro qui permet d'arrêter le programme dans le cas où la condition spécifiée n'est pas vérifiée

```
void assert (int condition);
```

- Les assertions sont utiles pour le débogage
  - Affichage détaillé de l'emplacement de la violation de la condition
- L'option `-DNDEBUG` du compilateur GCC permet d'ignorer toutes les assertions

Gestion des erreurs et des exceptions

Exemple

```
1  #include <iostream>
2  #include <cassert>
3  using namespace std ;
4  int main() {
5     int c,a=1,b=0;
6     assert (b!=0);
7     c = a/b ;
8     cout << "c : " << c << endl ;
9     return 0;
10 }
```

Gestion des erreurs et des exceptions

Exemple

```
1  #include <iostream>
2  #include <cassert>
3  using namespace std ;
4  int main() {
5     int c,a=1,b=0;
6     assert (b!=0);
7     c = a/b ;
8     cout << "c : " << c << endl ;
9     return 0;
10 }
```

Sortie avec `g++ main.cpp`

```
a.out: main.cpp:6: int main(): Assertion `b !=0' failed.
Abandon (core dumped)
```

Gestion des erreurs et des exceptions

Exemple

```
1  #include <iostream>
2  #include <cassert>
3  using namespace std ;
4  int main() {
5     int c,a=1,b=0;
6     assert (b!=0);
7     c = a/b ;
8     cout << "c : " << c << endl ;
9     return 0;
10 }
```


Sortie avec `g++ main.cpp`

```
a.out: main.cpp:6: int main(): Assertion `b !=0' failed.
Abandon (core dumped)
```

Sortie avec `g++ main.cpp -DNDEBUG`

```
Exception en point flottant (core dumped)
```

Gestion des erreurs et des exceptions

Les exceptions : principe

- Mécanisme permettant de prévoir une erreur à un endroit et de la gérer à un autre endroit
  - Lorsque qu'une erreur a été détectée à un endroit, on la signale en «lançant» un objet contenant toutes les informations que l'on souhaite donner sur l'erreur
  - À l'endroit où l'on souhaite gérer l'erreur, on peut «attraper» l'objet «lancé»
  - Si un objet «lancé» n'est pas attrapé du tout, cela provoque l'arrêt du programme : toute erreur non gérée provoque l'arrêt
- Les exceptions permettent le découplage total entre la détection d'une anomalie et son traitement

Gestion des erreurs et des exceptions

Les exceptions

- On cherche à remplir 3 tâches élémentaires
  1. Signaler une erreur
  2. Marquer les endroits réceptifs aux erreurs
  3. Leur associer (à chaque endroit réceptif aux erreurs) un moyen de gérer leurs erreurs
- On a donc 3 mots clés dédiés à la gestion des exceptions
  1. `throw` indique l'erreur («lance» l'exception)
  2. `try` indique un bloc réceptif aux erreurs
  3. `catch` gère les erreurs associées (les «attrape»)
- Le compilateur C++ ne vérifie pas que les exceptions sont gérées dans le code (JAVA le fait)

Les exceptions : générer une exception

- `throw` permet de générer une exception : signaler l'erreur au reste du programme
- Syntaxe : `throw expression`

Exemples

```
1  throw 21 ; // Exception de type entier
2  throw string("quelle erreur !") ; // Exception de typee string
3
4  struct Erreur {
5  int code ;
6  string message ;
7  } ;
8  ...
9  Erreur faute ;
10 ...
11 faute.code = 12 ;
12 faute.message = "Division par 0" ;
13 throw faute ;
```

Les exceptions : le bloc `try`

- `try` permet de définir un ensemble d'instructions à surveiller
- Si une exception est déclenchée durant l'exécution du bloc try, il sera alors suspendu et l'exécution se poursuivra dans un bloc catch

Exemples

```
1  try {
2  ...
3  if (x == 0.0) throw string("valeur nulle") ;
4  ...
5  }
6
7  try {
8  ...
9  y = f(x) ; // f peut lancer une exception
10 ...
11 }
```

Les exceptions : attraper une exception

- `catch` introduit un bloc dédié à la gestion d'une ou plusieurs exceptions
- Tout bloc `try` doit toujours être suivi d'un bloc catch
  - Si une exception est lancée mais n'est pas interceptée par le catch correspondant, le programme s'arrête
- Syntaxe

```
catch (type nom) {
   ...
}
```

  - `catch(...)` intercepte n'importe quel type d'exception
  - Pour les exceptions de type complexe, on préfère passer par référence

```
catch (Erreur const& e)
```

Exemples

```
try {
   ...
   if (x == 0.0) throw string("valeur nulle") ;
   ...
   if (j >= 3) throw j ;
}
// Attraper les exceptions de type string
catch(string const& erreur) {
   cerr << "Erreur : " << erreur << endl ;
}
// Attraper les exceptions de type int
catch(int erreur) {
   cerr << "Avertissement : je n'aurais pas du avoir la valeur "<< erreur << endl ;
}
```

Gestion des erreurs et des exceptions