

COURS N° 1 : C++ UN MEILLEUR C

- ✗ renforcer le contrôle de type :
 - + définitions, prototypes de fonctions
 - + surcharge
 - + références
 - + conversions de void *
- ✗ préférer le compilateur au préprocesseur
- ✗ considérer les entrées/sorties opérations abstraites
- ✗ préférer new/delete à free et malloc
- ✗ petites différences
- ✗ notes aux programmeurs C

DÉFINITION ET PROTOTYPE DE FONCTIONS (1)

- ✗ quand le compilateur C++ rencontre un appel de fonction
 - + il ne l'accepte que si la fonction a déjà été déclarée ou définie
 - + puis il compare
 - ✗ le type des arguments effectifs à celui des paramètres formels
 - ✗ en cas de différences des promotions numériques et des conversions standard sont implicitement mises en place

DÉFINITION ET PROTOTYPE DE FONCTIONS (2)

- ✗ Comme en C

 - + fichier .h

 - + portée du prototype

- ✗ Différences

 - + fonctions sans arguments

 - ✗ en C : `int f(void)`

 - ✗ en C++ : `int f()`

 - + fonctions sans valeurs de retour

 - ✗ en C : `f(int, int)` ou `void f(int, int)`

 - ✗ en C++ : `void f(int, int)`

 - + arguments par défaut

 - ✗ nouveauté très utilisée

COURS N° 1 : C++ UN MEILLEUR C

- ✗ renforcer le contrôle de type :
 - + définitions, prototypes de fonctions
 - surcharge
 - ✗ surcharge des fonctions
 - ✗ polymorphisme ad hoc
 - ✗ recherche du code
 - ✗ surcharge et éditions de liens
 - + références
 - + conversions de void *
- ✗ préférer le compilateur au préprocesseur
- ✗ considérer les entrées/sorties opérations abstraites
- ✗ préférer new/delete à free et malloc

SURCHARGE DES FONCTIONS (ET DES OPÉRATEURS)

- ✗ permet de réaliser le "polymorphisme ad hoc"
- ✗ mécanisme :
 - + un même nom désigne différentes fonctions dans une même portée
 - + le compilateur choisit la fonction à appliquer en fonction du type des arguments
- ✗ simplifie la tâche du programmeur qui n'a plus à mémoriser et à taper des noms barbares

LE POLYMORPHISME (PARENTHÈSE)

- ✗ concept très important en programmation et particulièrement en POO :
- ✗ du grec, signifie plusieurs (poly) formes (morphos)
- ✗ définition :
un même nom représente plusieurs objets selon le contexte
- ✗ 3 types de polymorphisme
 - + le polymorphisme ad hoc
 - + le polymorphisme d'héritage (Cf. cour 3)
 - + le polymorphisme paramétrique
- ✗ polymorphisme ad hoc ou surcharge (overloading) : définir plusieurs fonctions de même nom, chacune correspondant à des types de paramètres précis

RECHERCHE D'UNE FONCTION SURCHARGÉE(1)

fonction à un argument : meilleure correspondance possible

1/ correspondance exacte

2/ promotions numériques

3/ conversions standard (dégradantes)

4/ conversions définies par l'utilisateur

arrêt :

- ✗ au premier niveau où \exists correspondance unique
- ✗ si à un même niveau plusieurs correspondances : erreur à la compilation (ambiguïté)
- ✗ si aucune fonction ne convient : erreur de compilation (fonction inconnue)

RECHERCHE D'UNE FONCTION SURCHARGÉE(2)

fonctions à plusieurs arguments

1/ le compilateur sélectionne pour chaque argument la (ou les) meilleure(s) fonction(s)

2/ il considère l'intersection des ensembles des meilleures fonctions pour chaque argument

Si cette intersection contient une seule fonction, c'est la fonction choisie

sinon erreur de compilation

Remarque :

fonction avec un ou plusieurs arguments par défaut

traitée comme plusieurs fonctions différentes avec un nombre croissant d'arguments

SURCHARGE ET ÉDITION DES LIENS

- ✗ Le compilateur modifie les noms externes des fonctions
- ✗ Nouveau nom : nom interne + nombre et type des arguments
- ✗ Problème des fonctions C : fournir un prototype précédé de extern "C"

COURS N° 1 : C++ UN MEILLEUR C

- ✗ renforcer le contrôle de type :
 - + définitions, prototypes de fonctions
 - + surcharge
 - références
 - ✗ passage des paramètres
 - ✗ retour d'un résultat
 - ✗ plus généralement
 - + conversions de void *
- ✗ préférer le compilateur au préprocesseur
- ✗ considérer les entrées/sorties opérations abstraites
- ✗ préférer new/delete à free et malloc
- ✗ petites différences
- ✗ notes aux programmeurs C

RÉFÉRENCES(1)

- ✗ une référence est un alias pour un objet
 - + comme un pointeur
elle est représentée par l'adresse d'un objet
 - + contrairement à un pointeur
on accède directement à l'objet référencé sans
recourir à l'opérateur d'indirection
- ✗ utilisation :
 - pour éviter des manipulations sur les
pointeurs
- ✗ en particulier :
 - + passage des paramètres d'une fonction
 - + type de retour pour une fonction

PASSAGE DES PARAMÈTRES D'UNE FONCTION (1)

- ✗ en C :
 - + par défaut : passage par valeur
 - + argument modifié ou "gros objet" : passage par adresse, i.e. passage par valeur d'un pointeur sur l'objet
- ✗ en C++ :
 - + par défaut : idem, passage par valeur
 - + argument modifié ou "gros objet" : passage par références
 - ✗ le compilateur prend en charge la transmission des arguments par adresse
 - ✗ l'utilisateur manipule une variable normal
- ✗ philosophie C++ : les ptr indiquent une indirection, pas des manipulations pour faire remonter des modifications vers le code appelant

PASSAGE DES PARAMÈTRES D'UNE FONCTION (2)

➔ passage par adresse (pointeur)

✗ avantage

- + l'utilisateur sait que le paramètre effectif peut être modifié car il passe un ptr à l'appel
- + contrôle des effets de bords indésirés

✗ inconvénients

- + lourdeur d'écriture, sources d'erreurs
- + l'utilisateur doit savoir qu'il faut passer un pointeur

✗ utilisation en C

- + paramètre modifié par une fonction
- + gros objet passé en paramètre pour éviter une copie

PASSAGE DES PARAMÈTRES D'UNE FONCTION

(3)

➔ passage par référence

✗ avantage

- + simplifie l'écriture : on manipule (à l'appel et dans la définition) le nom des variables, pas un ptr
- + choix au moment de la définition, à l'appel rien à faire de spécial

✗ inconvénients

- + risque d'effets de bord indésirés si l'utilisateur oublie que le paramètre peut être modifié
 - + essayer d'éviter les fonctions qui modifient leurs paramètres
- #### ✗ utilisation en C++ : remplace le passage de paramètre par adresse du C

RETOUR DU RÉSULTAT D'UNE FONCTION

- ➔ retour du résultat d'une fonction
 - ✗ la valeur retournée par une fonction est passée par valeur
 - ✗ pour les gros objets, plus efficace de retourner une référence sur l'objet
 - 2 difficultés
 - + retourner une référence sur un objet local (bogue de la "référence pendante", message d'avertissement)
(remarque : même pb si on retourne un ptr sur un objet local bogue du "ptr fou")
 - + retourner une lvalue (surtout utilisé pour la surcharge d'opérateurs)

RÉFÉRENCES (BILAN PROVISOIRE)

- ✗ notion de référence : plus générale que ce qui est présenté ici
- ✗ réalisation d'une référence
pointeur constant qui est dérérérencé (par le compilateur) à chaque utilisation
- ✗ une référence doit être initialisée
 - + au moment de sa déclaration
 - + par une lvalue (objet dont on peut prendre l'adresse)
- ✗ une référence ne peut pas être modifiée
(seul l'objet référencé peut l'être s'il n'est pas constant)
- ✗ pour les types de base : lvalue obligatoire
- ✗ pour les références à des constantes : subtil (cf doc)
utile pour la définition d'arguments par défaut

COURS N ° 1 : C++ UN MEILLEUR C

- ✗ renforcer le contrôle de type :
 - + définitions, prototypes de fonctions
 - + surcharge
 - + références
 - conversions de void *
- ✗ préférer le compilateur au préprocesseur
- ✗ considérer les entrées/sorties opérations abstraites
- ✗ préférer new/delete à free et malloc
- ✗ petites différences
- ✗ notes aux programmeurs C

CONVERSION DE VOID *

- ✗ En C++,
 - + les pointeurs de type void * ne sont pas convertis implicitement en un pointeur d'un autre type,
 - + mais un pointeur de type quelconque est converti implicitement en void *
- ✗ En C, conversions implicites dans les deux sens

COURS N° 1 : C++ UN MEILLEUR C

- ✗ renforcer le contrôle de type :
 - préférer le compilateur au préprocesseur
 - + const préféré à #define
 - + fonctions en ligne préférées aux macros
- ✗ considérer les entrées/sorties opérations abstraites
- ✗ préférer new/delete à free et malloc
- ✗ petites différences
- ✗ notes aux programmeurs C

NOMMER DES CONSTANTES (1)

- ✗ pour nommer des constantes
 - + en C : `#define`
 - + en C++ : le spécificateur de type `const`
- ✗ avantages :
 - + limiter la portée des constantes (à un fichier, à une classe)
 - + ne pas encombrer l'espace global des noms
 - + accepter des expressions
- ✗ similitudes C++/C ANSI :
 - + un objet déclaré constant ne peut être modifié
 - + le compilateur signale les tentatives de modifications
 - + un objet constant doit être initialisé

FONCTIONS INLINE (EN LIGNE, DÉVELOPPABLES) (1)

- ✗ quand une fonction est déclarée inline, le compilateur essaiera d'insérer (de développer) le code (machine) de la fonction aux endroits où la fonction est appelée
 - + économie de temps (pas d'appel de fonction)
 - + taille du code croît avec le nombre d'appels
- ✗ syntaxe : placer le mot clé inline devant sa définition
- ✗ la définition d'une fonction en ligne doit apparaître avant tout appel de la fonction dans le même fichier
 - interdit la compilation séparée de la fonction en ligne
 - oblige à placer la définition des fct inline dans les fichiers d'en-tête
- ✗ le compilateur peut ne pas pouvoir développer une fct en ligne (boucle...)

FONCTIONS INLINE (2)

✗ avantages :

- + éviter le surcoût en temps d'un appel de fonction (Coplien : temps divisé par 4)
- + préférable aux macro :
 - ✗ compilateur/ préprocesseur
 - ✗ contrôle de type comme pour les fct
 - ✗ pas les effets de bords des macros
 - ✗ permet le déverminage (pas tjrs)

✗ inconvénients

- + ne permet pas la compilation séparée (entorse à la séparation client/programmeur)
- + augmentation du code si la fct est longue

✗ utilisation : ne pas abusez

régle des 80/20 : constructeurs, fct très courtes

C++ UN MEILLEUR C

- × meilleur typage :
- × préférez le compilateur au préprocesseur
entrées/sorties opérations abstraites
- × préférez new/delete à free et malloc
- × petites différences
- × Notes aux programmeurs C

ENTRÉES/SORTIES EN C++(1)

- ✗ la bibliothèque `stdio.h` de C peut être utilisée, mais c'est dommage
- ✗ `iostream.h` permet de disposer d'E/S
 - + faciles à utiliser (plus de format)
 - + module objet plus petit (C++ n'inclut que le code nécessaire)
 - + extensible aux classes utilisateurs
 - + conforme au modèle de programmation objet

ÉCRITURE SUR LA SORTIE STANDARD (2)

- ✗ `cout` objet prédéfini (un flot) de C++ : associé à sortie standard (`stdout`)
- ✗ on envoie à `cout` un message `<<`

```
int n = 4;  
cout << "voilà un entier :" << n ;
```
- ✗ l'opérateur `<<` (de décalage) est surchargé pour `cout` et les types usuels
- ✗ il provoque l'écriture sur `cout` de l'argument (ici la chaîne) et retourne `cout`
- ✗ par le mécanisme de surcharge, `cout` reconnaît le type de l'argument de l'opérateur de décalage et lui applique le code voulu ;
- ✗ cela dispense le programmeur de l'écriture du format

ÉCRITURE SUR LA SORTIE STANDARD (3)

on utilise l'opérateur << pour envoyer sur `cout`

- ✗ type de base quelconque (`char`, `int`, `float`)
- ✗ chaîne de caractères (`char *`) : tous les caractères de la chaîne sont écrits dans `cout`
- ✗ pointeur (autre que des `char *`) : écriture de l'adresse correspondante
- ✗ pour avoir l'adresse d'une chaîne : la "caster" `void *`
(pourquoi ?)
- ✗ possibilité de surcharger << pour afficher des objets des classes utilisateurs

LECTURE SUR L'ENTRÉE STANDARD (1)

- ✗ `cin` objet prédéfini (un flot) de C++ : associé à entrée standard (`stdin`)
- ✗ on envoie à `cin` un message :

```
int n, p ;  
cin >> n >> p;
```
- ✗ l'opérateur `>>` (de décalage) est surchargé pour `cin` et les types usuels
- ✗ les paramètres sont passés par référence plus besoin d'appeler avec le `&`
- ✗ lecture des caractères dans le flot d'entrée `stdin` jusqu'à rencontrer un séparateur et les convertit en une valeur (ici entière) rangée dans `n` ; enfin `cin` est retourné

LECTURE SUR L'ENTRÉE STANDARD (2)

- ✗ exploration du flot jusqu'à la rencontre d'un séparateur (espace, tabulation ...)
- ✗ caractère invalide pour le type à lire provoque l'arrêt de l'exploration mais ce caractère est repris lors de la prochaine lecture
- ✗ lecture d'un caractère sur `cin` commence par sauter les séparateurs

NEW ET DELETE

✗ syntaxe

new type

- + retourne un pointeur sur la zone allouée pour le type ou null
- + fait appel au constructeur du type (cf. cours 4)

new type [n]

- + retourne un pointeur sur n cellules contiguës
- delete ad
- + libère delete [] ad
 - + pour les tableaux d'objets (Cf. cours 5)

PETITES DIFFÉRENCES

- ✗ les constantes caractère ne sont pas des entiers
- ✗ commentaire en fin de ligne : //
- ✗ déclarations et initialisations peuvent être placées où l'on veut (mais avant d'être utilisées)

NOTES AUX PROGRAMMEURS EN C (BJARNE STROUSTRUP)

- ✗ écrire du C++ avec un style C : perdre bénéfices de C++
- ✗ utilisez `const` et `inline` plutôt que `#define`
- ✗ ne déclarez pas de variable avant d'en avoir besoin
- ✗ n'utilisez pas `malloc/free` :
`new/delete` fait le même travail et mieux
- ✗ préférez `iostream.h` à `stdio.h`
- ✗ évitez `void*`, arithmétique de pointeur, les tableaux C, les conversions de types explicites
- ✗ penser un programme C++
 - + comme un ensemble de concepts interagissant
 - + non comme un ensemble de structures de données manipulés par des fonctions

QUESTIONS
