* Antoine Jouglet
* TP/TD : 4h par semaine, un coup l’un un coup l’autre
* Examens
  + 20% TP + 40% M + 40% F
  + Poly annoté autorisé au final
  + Médian 25 avril
* Compilateur : Penser à Qt creator

Compte LO21

* + lo21p035
  + wejo8DUP
* TD/TP
  + Thibaut Arribe
  + thibaut.arribe@utc.fr
* API
  + Application Programming Interface
  + Permet d’interagir avec un code déjà programmé
  + Mis à disposition plutôt par des services en ligne
    - Exemples : Google, Youtube, FaceBook, …
* SDK
  + Software Development Kit
  + Permet de développer pour un contexte unique
  + Utilisé notamment par des plates formes mobiles
    - Exemples : iOS, Android
* Framework
  + Pour faciliter la vie du programmeur
  + Plus générique qu’un SDK
  + Librairies
* Objet
  + Ce n’est forcément quelque chose qui se touche
  + C’est un truc
  + Nous avons la faculté de percevoir des objets grâce à leurs attributs. Ils nous permettent de définir différents objets.

Attributs

* + Ce ne sont pas des objets, ils servent à les caractériser
  + Se retrouve dans un certain nombre d’attributs
  + Objet au croisement de différents attributs
  + Ensemble d’objets qui partagent des attributs communs : classe
* Classe
  + Abstraction
* Modélisation orientée objet est toujours contextuelle : on choisit des attributs pertinents p/r l’utilisation qu’on va faire de cette modélisation.
* Approche objet
  + Utilisation d’objets.
  + Ils interagissent entre eux pour réaliser une fonction => système
  + Système décomposé sur ce qu’il est et fait
* Attributs
  + Chaque objet est responsable de ses attributs
* Méthode
  + Fonction qui s’exécute sur un objet particulier
* Avantages approche OO (espèce d’arc-en-ciel)
  + Stabilité
* Analyse et conception orientées objet
  + Découpage naturel et intuitif contre le complexité d’un problème.
  + Découpage horizontal
    - Exemple : (polygones (quadrilatères (parallélogrammes (losanges rectangles))triangles pentagones))
* Outils
  + UML
    - Langage graphique
    - On l’utilise comme on veut
    - Degré de détails pertinent à ce qu’on veut en faire
  + Design Pattern
* Préprocesseur
  + Plus de .h.
  + iostream : entrée sortie
  + string : bibliothèque pour générer les caractères
  + namespace
    - portée globale/portée locale d’une fonction
    - portée globale divisée en plusieurs sous espaces
      * tous les éléments des bibliothèques standard sont déclarés dans std.
* cout
  + objet qui appartient à la classe ostream, qui représente la sortie standard
  + Pendant de printf
  + cout<<“Bonjour ! \n“ ; = cout.operator<<(“Bonjour ! \n“) ;
* cin
  + Pendant de scanf.
* Définition et déclaration d’une variable
  + Définir une variable : réserver un espace mémoire.
  + Déclarer une variable : dire que quelque part il y aura une variable (mot clé : extern)
  + Initialiser les variables, c’est bien sinon on n’est pas sûr de sa valeur
  + Une définition est aussi une déclaration
  + Identificateur de fonction ou de variable doit être déclaré avant utilisation
* Données
  + Portée
    - Où est par qui la donnée est-elle visible ?
    - Visible qu’à partir de la ligne où elle a été déclarée

|  |
| --- |
| int i  for (i=0 ; i<10 ; i++)  cout << i << “ \n ” ;  int i = 5 ;  for (int i=0 ; i<10 ; i++)  {  cout << i << “\n” ; \\ affiche i, le compteur de la boucle, ce n’est pas le même i que celui qui avait été déclaré avant la boucle for.  cout << ::i << “\n” ; \\ affiche 5  } |

* + Durée de vie
    - Quand est-ce que la donnée existe ? = quand est-ce que la données a un espace mémoire qui lui a été réservé ?
  + Segments mémoire
    - Segment statique
      * Variables globales et statiques
      * Durée de vie = celle du programme
      * static
      * Quand on a besoin qu’une variable soit toujours là
    - Pile
      * stack
      * variables automatiques
      * vie dans le bloc à partir de leur déclaration
    - Tas
      * Heap

Taille des objets

* + Sous forme de multiple de la taille de char.
  + Taille de char = 1 (une fois moins 8 bits, mais ça peut être 16, 32, …)
* Initialisation et affection
  + Initialisation : int i=4 ;
  + Affectation : i=5 ;
* Constantes
  + const
  + Super important
* Structures
  + struct
  + Forme simple de class
  + struct id  
    {  
     type1 id1 ;  
     …  
     typen idn ;  
    }

Objet

* + Région contiguë de mémoire
  + Au plus un identificateur : pas forcément besoin de connaître son nom
* Adressage indirect
  + Pointeurs
    - A une valeur entière
    - Numéro d’un emplacement mémoire
      * Peut être valide ou non
      * Peut correspondre ou non à une adresse allouée
    - type\* : pointeur de type
    - pointeur nul : 0
    - Conversion implicite en bool :
      * Nul => False
      * Non nul => True
    - Pointeurs const
      * const type\* pt
      * Il pointe sur une zone qu’on ne peut pas modifier par l’intermédiaire du pointeur
      * En revanche on peut la modifier par l’intermédiaire d’autre chose
    - Pointeurs constants (≠ pointeurs const)
      * Le pointeur lui-même ne peut pas changer de valeur
      * type\* const pt
  + Références
    - Doit être initialisée
    - Ne pas être réassignée, constante par nature
    - Jamais nulle
    - miaou& = variable ; référence à une variable de type miaou qui pointe sur **la zone de variable**
    - S’initialise avec une valeur de type int
    - cout << ref << ; affiche le contenu de la zone mémoire
    - Référence const
      * const miaou& ref=x ;
      * Equivaut à miaou const& ref=x
      * On ne peut pas modifier la zone référencée par ref si on a une référence const
* lvalue
  + A l’origine, expression qu’on peut être à gauche de l’opérateur d’affection
  + Exp = … ; Exp est une lvalue
  + Maintenant, elle désigne un emplacement mémoire
* Allocation dynamique en C++
  + new, delete, new[], delete[]
  + new et new[] permettent d’allouer de la mémoire
    - int\* mem = new int ;
    - int\* mem2 = new int[10] ; tableau dynamique de taille 10
  + delete et delete[] permettent de libérer de la mémoire
    - delete mem
    - delete[10] mem2 ;
* Namespaces
  + Utiliser un namespace :
    - Pour une portée : using namespace mon namespace
    - Pour un appel simple : ::monnamespace::
  + Déclarer :
    - Dans le header : namespace monnamespace{ … }
    - Dans le fichier source : void monnamespace ∷mafonction{ … }
  + Alias : namespace monnamspace2 = monnamespace ;
  + Possible de déclarer des namespaces dans un autre
* Transmission : le passage par valeur
  + Arguments toujours transmis par une valeur
  + Paramètre de fonction : variable locale initialisée avec le résultat de l’expression passée en argument, i.e. une valeur
  + De même pour le retour
    - return
  + Unique mode de transmission possible en C/C++
* Valeurs par défaut
  + Toujours en derniers dans la liste
  + Dans le header
    - type funcname (type1 var1, type2 var2=constante) ;
      * var2 a la valeur constante par défaut
    - char\* funcname (char\* dest, const char\* str1) ;
* Fonction inline
* Déclaration d'une classe en C++ avec ses méthodes
  + Dans un header classe.cpp
    - struct nom  
      {  
       int heure; // attribut  
       int minute; // attribut  
       void afficher(); // méthode  
      }
  + Dans un fichier source pour les méthodes de la classe classe.cpp
    - #include "classe.h"  
      void **horaire::**afficher() // cette fonction est une méthode de la classe horaire  
      {  
       std::cout << this->heure << "H" << this -> minute;  
      }



* + Dans le fichier source principal main.cpp
    - #include "classe.h"  
      int main()  
      {  
      horaire t;  
      t.heure = 14;  
      t.minute = 29;  
      t.afficher(); // c'est comme ça qu'on applique la méthode à l'objet t  
      }
  + this est un mot clé qui correspond à l'objet sur lequel on appliquera la méthode. Il s'agit d'une adresse, c'est un pointeur de type horaire\* dans l'exemple ci-dessus.
  + **Argument implicite** : objet sur lequel s'applique la méthode. Une méthode a toujours au moins un paramètre, c'est cet argument implicite.
  + Possibilité de recevoir d'autres arguments
    - Exemple : dans la déclaration de horaire :  
      void set-horaire (int h, int m)  
      {  
       heure=h;  
       minute=m;  
      }
      * Cette méthode sera une fonction inline
      * Il est fréquent de définir de petites fonctions directement dans la déclaration de la classe
  + Impossible de donner une valeur par défaut aux attributs d'une classe dans la déclaration de classe
  + Normalement, déclaration de classes à un niveau global ou dans un namespace
    - Néanmoins, possible de déclarer une classe dans une autre
    - Possible de déclarer une classe dans une fonction
  + Méthodes const
    - Dans le header : void afficher() const;
    - Dans le fichier source : void horaire::afficher() const { … };
    - Impossible de modifier les attributs de l'objet au sein d'une méthode const
    - A partir d'une référence const on ne peut utiliser qu'une méthode const
    - Const horaire& t;  
      t.set-horaire(12, 54); // erreur, car set-horaire est une méthode non const
* Séparation pour l'objet
  + Partie utile pour les autres objets
    - Ce que les autres objets doivent savoir pour collaborer
  + Partie nécessaire à son fonctionnement propre
    - Ces données peuvent être cachées
* Limiteurs d'accès
  + public:
    - utilisateurs de la classe ont accès aux membres publics
  + private:
    - les méthodes de la classe propriétaire ont accès aux membre privés
  + protected:
    - Cf. héritage
  + On place ces limiteurs d'accès (avec les deux points) avant de déclarer les membres
* Class ou struct ?
  + Presque pareils
  + Membres privés par défaut pour class
  + Membres publiques par défaut pour struct
* Accesseurs
  + Souvent méthodes inline
  + Pour respecter le principe d'encapsulation
* Motivation
  + Protection de l'utilisateur
  + Transparence pour l'utilisateur en cas de modification de la structure de données
  + Couplage faible
* Exception : l'amitié
  + friend:
  + I'll be there for you
  + Fonction amie d'une classe
    - friend: puis le prototype de la fonction
  + Méthode amie d'une autre classe
    - friend: puis le prototype de la méthode avec résolution de portée
  + Classe amie (toutes ses méthodes sont amies) d'une autre classe
    - friend: puis le nom de la classe amie
  + Exemple : dans la déclaration de la classe qui a des amis
    - class A  
      {  
       friend int f(); // fonction amie de A  
       friend B::f(); // méthode de B amie de A  
       friend class C; // classe C amie de A  
      }

Constructeur

* + Méthode
  + Appelée à chaque création d'objet
  + Surchargeable
  + Constructeur par défaut généré si aucun proposé dans la déclaration de classe
    - Il ne fait rien
    - Si la classe a un attribut de type d'une autre classe, le constructeur par défaut appelle quand même le constructeur de cette classe attribut
  + S'il existe un constructeur, obligation de l'utiliser à la création d'un objet
* Destructeur
  + Un seul par objet
  + Implémentation :
    - Dans le header, dans la déclaration de la classe, il porte le nom de la classe avec une ~ devant
  + Pas de type de retour
  + Peut être inline
* Constructeur de recopie
  + Il y en a toujours un généré par défaut
  + Ex :   
    horaire h1(10, 51);  
    horaire h2=h1; // utilisation du constructeur de recopie !!!
  + On peut le surcharger
* Attributs statiques
  + Un seul exemplaire pour tous les objets de la classe
  + static dans la déclaration de classe dans le .h
    - classe priorite {  
       int p;  
       static int min;  
      }
  + Il doit être initialisé explicitement à l'extérieur de la déclaration de classe dans une unité de compilation (obligatoire)
    - int priorite::min=0;
* Méthodes statiques
  + Elles n'ont pas besoin d'objet pour être utilisées
  + Jamais const
  + Pas de this défini
  + static dans la déclaration de classe dans le .h
  + Utilisation dans un .cpp : nom de la classe::methode
* Conditions exceptionnelles
  + Risquent de compromettre la poursuite de son éxécution
* Traitement standard en C
  + On renvoie une valeur d'erreur
  + Avantage : séparation détection du traitement
  + Inconvénient : il faut toujours examiner les valeurs retours
* Séparer détection et traitement
  + Nécessaire pour la réutilisabilité des composants
    - Ce sont les composants qui détectent les incidents exceptionnels
    - L'utilisateur choisie la conduite à adopter en cas d'incident
* Avantages exception
  + Se propage depuis l'appelé vers l'appelant
  + Exception doit être traitées : non ignorable
* Gestion des erreurs : exceptions
  + Levée d'exception
    - Rupture
    - Instruction throw
      * throw "erreur";
      * Tout ce qui est après va être ignoré pour se brancher à l'endroit où on va traiter l'erreur
  + Après le déclanchement
    - Séquence d'instructions interrompue
    - Variables automatiques détruites
  + Capture
    - Exception déclenchée dans un bloc try doit être capturée pour être traitée par un gestionnaire d'exception
    - catch

Choix du gestionnaire

* + Ils sont essayés dans l'ordre
  + Dès qu'un gestionnaire correspond, il est exécuté
* Pourquoi ?
  + Concevoir, c'est long
  + Il y a des problèmes récurrents
    - Souvent on y apporte les mêmes solutions
* Qu'est-ce ?
  + Solution de conception à un problème récurrent
  + Modèle de solution à adapter en fonction du problème
* Avantages
  + Réutilisation des bonnes solutions d'architecture et conception plus rapide et plus facile