Plan

- Introduction, rappels, généralités
- Approche impérative
- Approche fonctionnelle
- Approche objet
- Approche générique
- Compléments

Points abordés

- Définition de fonction
- Passage d'arguments
- Retour de valeur
- Signature, type d'une fonction
- Pointeur de fonction
- Fonction anonyme
- Coroutine
- Exception

Programmation fonctionnelle

- Approche “mathématique”, non basée sur l'architecture des ordinateurs
- Décomposition des traitements en une suite de fonctions : résultat = fonction(données)
- Composition de fonctions, lambda calcul
- Récursion plutôt que boucles
- Souvent associée aux listes (Ancêtre LISP = LIST Processor)

Intérêt de l'approche fonctionnelle

- Pas de modification de l'état (pas d'effet de bord)
  - Parallélisme

```
f() + g()
```

- Transparence référentielle
  - Les appels à une fonction avec le(s) même(s) argument(s) retournent les mêmes résultats

```
f( 3 ) + f( 3 ) == 2 * f( 3 )
```

Exemple fonctionnel en C

```c
/*
 * Pré-condition : tab contient au moins
 *     un entier égal à 0
 */
int longueur_tableau( int tab[] ) {
   if( tab[0] == 0) {
      return 0;
   }
   return 1 + longueur_tableau( tab + 1 );
}
```

Pas d'affectation

Démo

GitLab

cours/fonctionnel/Recursion.cpp

Fonction

- Un ***nom***
- Une zone mémoire correspondant au code (repérée par une adresse)
- Zéro ou plus ***paramètres formels*** (nom, type)
- Un type qualifiant la valeur de retour
  - `void` pour indiquer aucune valeur de retour
  - Une fonction qui ne retourne rien n'est utile que si elle a un ***effet de bord***

Définition de fonction nommée

- Syntaxe

```
double carre( double d ) {
  return d * d;
}
```

- C++ : notation post-fixée possible pour le type de retour

```
auto carre( double d ) -> double {
  return d * d;
}
```

- En C++, on utilise parfois le terme de **fonction libre** (en opposition à **fonction membre** (méthode, voir le prochain cours)

Déclaration de fonction

- Syntaxe (`extern` est optionnel)

```
extern double carre( double d );
```

- C++

```
auto carre( double d ) -> double;
```

- Une fonction doit être déclarée avant d'être utilisée
- Une définition est une déclaration
- Une fonction doit être définie une et une seule fois

Passage d'arguments (1)

- Passage ***par valeur***

```
void f1( int i ) {
   int k = i;
   i = 3;
}
```

```
void g1() {
   f1( 5 );
   int j = 8;
   f1( j );
   f1( j + 2 );
}
```

- Le ***paramètre effectif*** est une copie de l'***argument*** (pas de modification de ce dernier)

Passage d'arguments (2)

- Passage ***par adresse***

```
void f2( int * i ) {
  int k = *i;
  *i = 3;
}
```

```
void g2() {
  // f2( &5 );
  int j = 8;
  // f2( j );
  f2( &j );
  // f2( &( j + 2 ));
}
```

- Terminologie abusive (passage par valeur d'un pointeur)

C++ : Passage d'arguments (3)

- Passage ***par référence sur valeur gauche***

```
void f3( int & i ) {
  int k{ i };
  i = 3;
}
```

```
void g3() {
  // f3( 5 );
  int j{ 8 };
  f3( j );
  // f3( j + 2 );
}
```

- Le paramètre est un autre nom de l'argument (modification)

C++ : Passage d'arguments (4)

- Passage ***par référence sur valeur gauche constante***

```
void f4( const int & i ) {
  int k{ i };
  // i = 3;
}
```

```
void g4() {
  f4( 5 );
  int j{ 8 };
  f4( j );
  f4( j + 2 );
}
```

- Syntaxe et sémantique du passage par valeur, performance du passage par adresse

Démo

GitLab

cours/fonctionnel/Arguments.cpp

Retour de valeur (1)

- Retour ***par valeur***

```
double carre( double d ) {
  double res = d * d;
  return res;
}
```

- La valeur retournée est une copie du résultat de l'évaluation de l'expression utilisée dans l'instruction `return`

Retour de valeur (2)

- Retour ***par adresse***

```
int * getTab() {
  static int tab[10];
  return tab;
}
// Exemple d'utilisation
// getTab()[3] = 8;
```

- Terminologie abusive (retour de la valeur d'un pointeur)
- ATTENTION : la donnée pointée doit continuer à exister après la fin de la fonction

C++ : Retour de valeur (3)

- Retour ***par référence sur valeur gauche***

```
int & getVal( unsigned i ) {
  static int tab[10];
  return tab[i];
}
// Exemple d'utilisation
// getVal( 3 ) = 8;
```

- ATTENTION : la donnée retournée doit continuer à exister après la fin de la fonction
- Permet des fonctions pouvant être utilisée comme ***valeur gauche***

Retours de valeur multiples

- Python

```
def order( a, b ):
  if a < b:
    return a, b
  return b, a

_, max = order( 23, 12 )
```

- C++17

```
std::tuple< int, int > order( int a, int b ) {
  if( a < b ) return std::make_tuple( a, b );
  return { b, a };
}

auto [ std::ignore, max ] = order( 23, 12 );
```

Signature d'une fonction

- La signature de la fonction

```
long partie_entiere( double d );
```

est

```
partie_entiere( double )
```

- Les noms des paramètres sont ignorés, ainsi que le type de retour
- Il est interdit d'avoir deux fonctions ayant la même signature (sauf, en C++, si elles sont dans des espaces de noms différents)

C++ : Surcharge de fonctions

- Possibilité de définir plusieurs fonctions avec le même nom, mais ayant un nombre et/ou des types de paramètres formels différents

```
int    max(    int a,    int b );
double max( double a, double b );
```

- Le compilateur choisira la bonne version selon les arguments donnés

- Les fonctions de même nom doivent avoir des signatures différentes

C++ : Valeur par défaut

- Possibilité de donner une valeur par défaut aux paramètres d'une fonction ; cette valeur par défaut est utilisée si l'argument correspondant n'est pas donné lors de l'appel

```
int str_to_int( std::string s, int base = 10 );

int dix  = str_to_int( "10"     );
int deux = str_to_int( "10", 2 );
```

- Si un paramètre a une valeur par défaut, tous les paramètres suivants doivent aussi en avoir une

Type d'une fonction (1)

- Le type de la fonction

```
long partie_entiere( double d );
```

est

```
long( double )
```

- Ce type ne peut pas être utilisé pour définir une variable (une variable ne peut pas *contenir* une fonction, mais peut contenir un lien vers une fonction).
- C++ : comme il existe plusieurs moyens d'avoir un lien vers une fonction, la solution générale passe par l'utilisation d'un *wrapper* de la bibliothèque standard :

```
std::function< long( double ) >
```

C++ : Type d'une fonction (2)

- Ce type peut être utilisé pour définir des variables :

```
std::function< long( double ) > foo;
// Affectation
foo = partie_entiere;
// Appel
auto l = foo( 2.5 );
```

ou des paramètres ou type de retour :

```
// Déclaration
void bar( std::function< long( double ) > f );
// Appel
bar( partie_entiere );
```

Pointeur de fonction

- Il est possible de manipuler des pointeurs de fonctions

```
long ( *pf )( double ) = &partie_entiere;
```

- Appel de la fonction pointée

```
long l = ( *pf )( 2.0 );
```

- Conversion implicite entre fonction et pointeur

```
pf = partie_entiere;
l = pf( 3.0 );
```

- C++ : un pointeur sur fonction est compatible avec `std::fonction`

```
std::function< long( double ) > foo{ pf };
```

Fonction comme argument en C

- Utilisation des pointeurs sur fonction
- Exemple dans la bibliothèque standard

```
void qsort(
  void * base, size_t nel, size_t width,
  int ( *compar )( const void *, const void * )
);
```

Fonction anonyme

- Il n'est pas possible en C++ de définir une fonction nommée à l'intérieur d'une autre fonction (avec accès au contexte de cette dernière).
- Il est parfois utile de définir un comportement fonctionnel sans vouloir passer par la définition d'une fonction nommée.
- Ces deux besoins peuvent être satisfaits en utilisant des ***fonctions lambda***, concept issu des langages fonctionnels et maintenant présent dans la majorité des langages de programmation.

Définition d'une fonction *lambda*

- Exemple d'expression définissant une ***fonction lambda*** qui indique si un entier est pair :

```
[]( int n ) -> bool { return n % 2 == 0; }
```

- les crochets introduisent la fonction lambda, on y précise si besoin le mode de capture du contexte ;
- les parenthèses contiennent la liste des paramètres ;
- le type de retour en notation post-fixée est optionnel si le compilateur peut déduire cette information du code ;
- le bloc contient les instructions de la fonction lambda.

Utilisation d'une fonction *lambda*

- Une fonction lambda peut être stockée dans une variable ; le type de cette variable étant compliqué et sans intérêt, on utilise un typage par inférence :

```
auto f1 = ( int n ) -> bool { return n % 2 == 0; };
```

ou le type général `std::fonction`

```
std::fonction< int( bool ) > f2{ f1 };
```

- Une expression de type fonction lambda peut aussi être directement passée comme argument :

```
int tab[] = { 1, 2, 3, 4 };
std::for_each(
    tab, tab + 4, []( int & n ){ n++; } );
```

Capture du contexte

- Une fonction lambda peut utiliser des variables visibles dans son contexte : on parle de ***capture de variable*** ; cette capture peut être faite par valeur ou par référence :

```
int i{ 1 }, j{ 2 };

// i par valeur, j par référence
auto sum = [i, &j]() { return i + j; };

// Affiche 3
std::cout << sum() << "\n";

++i; ++j;
// Affiche 4
std::cout << sum() << "\n";
```

Démo

GitLab

cours/fonctionnel/Lambda.cpp

Coroutine : concepts (1)

- La sortie d'une fonction (*subroutine*) provoque la destruction de toutes ses variables locales.
- Une coroutine peut être suspendue, puis reprendre son exécution avec son état mémorisé au moment de la suspension. Deux modèles :
  - coroutines asymétriques : la suspension de la coroutine se traduit pas un retour à la fonction appelante,
  - coroutines symétriques : lors de sa suspension, une coroutine choisit celle qui sera activée.
- Ces deux modèles sont équivalents fonctionnellement ; dans les deux cas, une valeur peut être transférée.

Coroutine : concepts (2)

- Les coroutines sont associées au multitâche coopératif (*fiber*) par opposition au multitâche préemptif (*threads*) :
  - il n'est pas nécessaire de mettre en place des mécanismes de synchronisation.
- Dans beaucoup de langages, les coroutines asymétriques sont associées à l'asynchronisme (*async/await*).
- Les générateurs sont des coroutines asymétriques particulières dont le rôle est de fournir des valeurs à la demande (*yield*).

Coroutine : difficultés

- Une coroutine doit conserver son état local en dehors de la pile standard d'exécution :
  - *stackless* : seule la coroutine peut provoquer sa suspension ➔ seul son état local doit être sauvegardé.
  - *stackful* : toute fonction appelée par la coroutine peut provoquer la suspension ➔ une pile complète est nécessaire.
- L'état sauvegardé :
  - est unique pour chaque coroutine symétrique ;
  - est unique à chaque fonction appelante pour les coroutines asymétriques : cet état local doit être retourné à la fonction appelante lors de la suspension, qui le redonnera à la coroutine lors de la reprise ;

Coroutines en C++

- Les coroutines en C++20 sont sans pile (l'état est stocké dans un bloc alloué dynamiquement) et sans lien avec l'asynchronisme (mais ces deux fonctionnalités peuvent être combinées).
- Le mécanisme est très générique, avec de nombreuses possibilités d'adaptation :
  - utilisation complexe sans une bibliothèque masquant cette complexité.
- Trois mots-clefs sont utilisés pour la mise en œuvre des coroutines : `co_await`, `co_yield`, `co_return` (le premier est un opérateur).

Démo

GitLab

cours/fonctionnel/Coroutine.cpp

Démo

GitLab

cours/fonctionnel/Fibonacci.cpp

Gestion des erreurs détectées à l'exécution

- Problème du signalement d'une erreur inattendue
  - le code qui détecte l'erreur ne sait pas comment la traiter
- Solutions *anciennes* (cf TP1 Programmation système)
  - une variable globale contient le code d'erreur
    ⇨ oubli de vérification de la variable, écrasement de sa valeur, problème général des globales
  - les fonctions retournent un code d'erreur
    ⇨ le code logique de la fonction est noyé dans le code de test des erreurs

Exceptions

- Une exception transfert le contrôle d'exécution du point de détection de l'erreur vers le code (situé plus haut dans la pile d'appels) déclaré comme pouvant traiter l'erreur
- Une exception est un type dont les valeurs peuvent porter des informations
  - Il est en général possible de définir des nouveaux types d'exceptions
  - Une exception est signalée par émission d'une valeur de ce type
  - Le code en charge de traiter l'exception indique sur quel(s) type(s) il doit être déclenché

Exceptions en C++

- Syntaxe :

```
try {
  /* Une fonction appelée effectuera
     potentiellement
     throw TypeException{};
  */
}
catch( TypeException e ) {
  // ...
}
```

- Pas de restrictions sur la valeur émise par un `throw` (type primitif, classe, allocation automatique ou dynamique…)
- Destruction de toutes les variables locales lors du dépilage des contextes (cf. destructeur d'une classe)
- Attention aux objets alloués dynamiquement (utiliser `std::unique_ptr`)