Introduction à C++

et outils de développement

Jean-Christophe Fabre

LISAH
Laboratoire d'étude des Interactions
Sol-Agrosystème-Hydrosystème

Plan

Plan

Faisons connaissance...

C++ est un langage procédural, orienté objet et générique :

- Défini par Bjarne Stroustrup dans les années 1980, sur la base du langage C
- Normalisé ISO en 1998, mise à jour la plus récente en 2011
- Disponible sur quasiment tous les systèmes et architectures
- 4ème langage le plus utilisé au monde, 1er des langages objets "performants"

Quelques caractéristiques générales :

- Langage compilé, générant des programmes natifs
- Sensible à la casse
- Intègre les fonctionnalités et la syntaxe du C

Commentaires

Les commentaires sont des parties du code source qui seront ignorées.
Deux syntaxes sont utilisables :

- tout ce qui se trouve entre /* et */
- tout ce qui suit // jusqu'à la fin de la ligne

Exemple

```
/* ceci est un commentaire sur plusieurs lignes. ceci est un
  commentaire sur plusieurs lignes.
  ceci est un commentaire sur  plusieurs lignes.
  ceci est un commentaire sur plusieurs
  lignes. */

// ceci est un commentaire sur une ligne
int x; // ceci est un commentaire sur une ligne
       // ceci est un commentaire sur une ligne
```

Eléments de structure

Un code source C++ est composé de blocs et d'expressions:

- Une expression est terminée par un point-virgule ;
- Un bloc contient des expressions et des sous-blocs
- Un bloc est délimité entre { et }

Exemple

```
{                       // debut de bloc
  int a = 0;

  {                     // un sous-bloc
    a = a + 1;          // la variable a est toujours disponible
  }
}                       // fin de bloc
```

Types de données simples

Nom	Type	Plage de valeurs
`bool`	un booléen	[true, false]
`char`	un entier	$[-128, 127]$
`short`	un entier	$[-2^{15}, 2^{15} - 1]$
`int`	un entier	$[-2^{31}, 2^{31} - 1]$
`long`	un entier	$[-2^{63}, 2^{63} - 1]$
`unsigned char`	un entier sans signe	$[0, 255]$
`unsigned short`	un entier sans signe	$[0, 2^{16} - 1]$
`unsigned int`	un entier sans signe	$[0, 2^{32} - 1]$
`unsigned long`	un entier sans signe	$[0, 2^{64} - 1]$
`float`	un réel	$[-3.4 \times 10^{38}, 3.4 \times 10^{38}]$
`double`	un réel	$[-1.7 \times 10^{308}, 1.7 \times 10^{308}]$
`long double`	un réel	$[-3.4 \times 10^{4932}, 3.4 \times 10^{4932}]$

Variables et constantes

Les variables et constantes sont :

- Obligatoirement typées
- Accessibles dans leur bloc de déclaration
- Identifiées avec des caractères alphanumériques

$$A - Z, a - z, 0 - 9, -, \_$$

Exemple

```
{
  int x;  // declaration d'une variable entiere x
  const double pi = 3.14;  // declaration d'une constante reelle pi
  bool b = true;  // declaration d'une variable booleenne b initialisee

  x = 5;
  {                 // un sous-bloc
    int aa = 18;    // declaration d'une variable entiere aa initialisee
    x = aa;
  }
  aa = aa + 1;      // erreur! (aa non utilisable)
}
```

Opérateurs d'affectation et booléens

L'opérateur d'affectation est le = (ne pas confondre avec l'opérateur d'égalité)

Opérateurs booléens :

- ! (NON), & (ET), | (OU), && (ET fainéant), || (OU fainéant)
- Les opérateurs fainéants sont à privilégier!
- Le résultat d'une opération booléenne est de type booléen

Exemple

```
{
  bool a = (6 > 2);  // a vaut true

  bool x = false && (a || true);  // x vaut false
}
```

Opérateurs d'arithmétiques et de comparaison

Opérateurs :

- Arithmétiques : +, -, *, /, %
- De comparaison : ==, !=, <, >, <=, >=

Types des résultats d'opérations :

- Le type du résultat d'une opération arithmétique dépend des types des opérandes
- Le type du résultat d'une opération de comparaison est un booléen

Exemple

```
{
  int x = 5;
  int y = 3;

  int r = x % 3;       // r vaut 2 (reste de la division Euclidienne)

  bool b = (6 < 2);  // b vaut false
}
```

Pointeurs et variables pointées

Un pointeur permet de stocker une adresse mémoire pour une variable du type du pointeur.

Ils sont utilisés notamment pour :

- allocation dynamique de zones mémoire
- tableaux "old-school"
- chaînes de caractères "old-school"

Exemple

```
{
  int* pa; // pointeur sur une adresse de variable de type entier
  int x = 5;

  pa = &x;  // pa pointe maintenant vers la variable x
  *pa = 10; // affectation de la valeur 10 dans variable pointee par pa
  pa = 2;  // interdit
}
```

Condition if

Permet d'exécuter un bloc uniquement si une condition est satisfaite

- Le test de condition `if` doit retourner un booléen
- Possibilité de proposer un bloc alternatif `else` en cas de non satisfaction de la condition `if`

Exemple

```
{
  double N, D, Result;

  if (D != 0)  // le bloc n'est execute que si D est different de 0
  {
    Result = N / D;
  }
  else  // sinon le bloc suivant est execute
  {
    std::cout << "Erreur!" << std::endl; // affichage d'une erreur
  }
}
```

Condition switch

Permet de traiter des conditions avec plusieurs valeurs possibles

- Un `switch` teste une variable ou une expression
- Chaque valeur possible doit être exprimée au travers d'un `case`
- Un cas `default` est possible si non satisfaction d'aucune autre condition
- Non recommandé avec des variables de type "texte"

Exemple

```
{
  switch (valeur)
  {
    case 1 :  autrevaleur = 5; break;
    case 2 : autrevaleur = 1; break;
    case 3 : autrevaleur = 7; break;
    default : autrevaleur = 0;
  }
}
```

Boucle for

Une boucle `for` permet de répéter un bloc un nombre de fois défini à l'avance

- La valeur initiale, la condition d'arrêt et l'incrément de boucle sont donnés en une seule fois
- Une boucle `for` peut ne pas être exécutée si la condition d'arrêt est directement satisfaite

Exemple

```
for (int i = 0; i < 10; i++)
{

  // ...

}
```

Boucle while

Une boucle `while` ou `do ... while` permet de répéter un bloc sous condition(s)

- La condition de boucle est définie avant ou après le bloc à exécuter
- Dans le cas d'un `do ... while`, le bloc est obligatoirement exécuté au moins une fois

Exemple while

```
int i = 0;
while (i < 100)
{
  // ...
  i++;
}
```

Exemple do ... while

```
int i = 0;
do
{
  //...
  i++;
} while (i < 100);
```

Fonctions

Une fonction est un bloc d'instructions d'un programme, nommé, que l'on peut appeler depuis d'autres parties du programme

- Une fonction renvoie une valeur de retour
- Le type `void` peut être utilisé pour une fonction qui ne renvoie rien
- Une fonction peut comporter des paramètres

Exemples de fonctions

```
void NeFaitRien()
{
}

double RetournePi()
{
  return 3.1415927;
}
```

Exemple d'utilisation d'une fonction

```
double CalculeAireCercle()
{
  // calcule l'aire d'un cercle de rayon 3
  double AireCercle = RetournePi() * 3.0 * 3.0;
  return AireCercle;
}
```

Fonctions : variables locales, variables globales

Comme dans tout bloc, une fonction peut déclarer des variables accessibles uniquement dans la fonction : les variables locales.
Les variables globales sont également accessibles, mais il est fortement déconseillé de les utiliser.

Exemple d'utilisation d'une fonction

```
{
  double globalA = 2.5;

  double Fonction1()
  {
    double A = 1.2;
    return A;
  }

  double Fonction2()
  {
    double A = 2.7;
    return A + globalA; // fortement deconseille
  }
}
```

Fonctions : Passage de paramètres par valeur

Le passage de paramètres par valeur copie la valeur des paramètres passés à la fonction comme variable locale de la fonction

- Coûts de mémoire et de performance dûs à la copie
- Pas de risque de modification de la valeur de l'appelant

Exemple

```
unsigned double CalculeAireCercle(unsigned double R)
{
  // calcule l'aire d'un cercle de rayon R
  unsigned double AireCercle = RetournePi() * R * R;
  return AireCercle;
}
```

Fonctions : Passage de paramètres par référence

Le passage de paramètres par référence permet d'utiliser directement la variable passée par l'appelant

- Peut être utilisé comme valeur de retour
- Possibilité de modification de la valeur de l'appelant, pouvant être neutralisée par l'utilisation du mot-clé `const`
- Mode à privilégier pour le passage des paramètres

Exemple

```
void CalculeAireCercle(const unsigned double& R, unsigned double& A)
{
  // calcule l'aire d'un cercle de rayon R
  A = RetournePi() * R * R;
}
```

Fonctions : Passage de paramètres par pointeur

Le passage de paramètre par pointeur est en fait un passage par valeur d'un pointeur sur la variable (et non pas de la variable elle-même)

- Peut être utilisé comme valeur de retour
- Possibilité de modification de la valeur de l'appelant, pouvant être neutralisée par l'utilisation du mot-clé `const`

Exemple

```
void CalculeAireCercle(const unsigned double& R, unsigned double* A)
{
  // calcule l'aire d'un cercle de rayon R
  *A = RetournePi() * R * R;
}
```

Structure générale d'un programme

Un programme est constitué d'une fonction principale appelée `main()`, elle-même pouvant faire appel à d'autres fonctions

Exemple

```
double Moyenne(double Nombre1, double Nombre2)
{
  return ((Nombre1+Nombre2)/2);
}

int main(int argc,char **argv)
{
  double Y = Moyenne(5.3,7.9);
}
```

Espaces de nommage

Les espaces de nommage permettent de regrouper des variables et fonctions, marquant l'origine du code utilisé

Exemple LibA

```
namespace LibA {
unsigned double AuCarre(double Nombre)
{
  return (Nombre*Nombre);
}
}
```

Exemple LibB

```
namespace LibB {
unsigned double AuCarre(double Nombre)
{
  return (pow(Nombre,2));
}
}
```

Exemple intégrant LibA et LibB

```
int main(int argc,char **argv)
{
  double Y = LibA::AuCarre(5);  // utilisation de AuCarre de LibA
  double Z = LibB::AuCarre(5);  // utilisation de AuCarre de LibB
}
```

Plan

Programmation orientée objet

La programmation orientée objet a pour objectifs de simplifier et structurer le développement logiciel

Elle s'articule notamment autour de briques logicielles, et fait appel aux concepts suivants:

- Classes et objets
- Attributs et méthodes
- Encapsulation
- Héritage et surcharge
- Polymorphisme

Classe

Une classe est une définition d'une entité informatique caractérisée par

- un identifiant
- un ou plusieurs attributs donnant les propriétés
- une ou plusieurs méthodes décrivant le fonctionnement
- un ou plusieurs constructeurs
- un destructeur

Exemple

```
class Voiture
{
  unsigned int Annee;  // attribut
  unsigned int CodeCouleur;  // attribut

  Voiture();  // constructeur
  ~Voiture();  // destructeur

  void Avancer();  // methode
  void Reculer();  // methode
  void Tourner(unsigned int Direction);  // methode
};
```

Constructeur et destructeur

Un constructeur est appelé lorqu'un objet est créé afin d'initialiser l'objet, notamment via des listes d'initialisation.

Un destructeur est appelé lorqu'un objet est détruit, afin de réaliser des "opérations de nettoyage" si nécessaire.

Exemple de déclaration

```
class Voiture
{
  unsigned int Annee;  // attribut
  unsigned int CodeCouleur;  // attribut

  Voiture();  // constructeur
  // autre constructeur
  Voiture(unsigned int A,
          unsigned int CC);
  ~Voiture();  // destructeur
};
```

Exemple de définition

```
Voiture::Voiture()
:  Annee(1900), CodeCouleur(0)
{
}

Voiture::Voiture(unsigned int A,
                 unsigned int CC)
: Annee(A), CodeCouleur(CC)
{
}

Voiture::~Voiture()
{
}
```

Encapsulation et visibilité

Une classe encapsule des attributs et des méthodes, et peut leur donner des caractéristiques de visibilité

Exemple

```
class Voiture
{
  private:
    unsigned int Annee;  // attribut
    unsigned int CodeCouleur;  // attribut

  public:
    Voiture(unsigned int A, unsigned int CC);  // autre constructeur
    ~Voiture();  // destructeur

    unsigned int getAnnee() { return Annee; }; // retourne l'annee
    unsigned int getCodeCouleur() { return CodeCouleur; }; // retourne la couleur
};
```

Par bonne pratique, un attribut est préférablement privé (private), accessible via des méthodes publiques (public)

Objet

Une classe permet d'instancier un ou plusieurs objets indépendants, avec des valeurs attributaires différentes

- Lors de l'instanciation d'un objet, un des constructeurs est obligatoirement appelé
- Lors de la destruction d'un objet, le destructeur est automatiquement appelé
- L'accès aux méthodes se fait via `NomObjet.NomMethode(Paramètres)`

Exemple

```
{
  Voiture Simca1000 = Voiture();
  Voiture Peugeot308;

  Peugeot308 = Voiture(2008,5);
  Simca1000 = Voiture(1976,17);

  unsigned int A = Simca1000.getAnnee();
}
```

New et delete

Un objet peut être créé via un pointeur sur une variable du type de sa classe

- Un objet pointé est instancié via l'opérateur `new`
- Un objet pointé est détruit via l'opérateur `delete`
- L'accès à ses méthodes se fait via l'appel à `NomObjet->NomMethode(Paramètres)`

Exemple

```
{
  Voiture* Simca1000;
  Simca1000 = new Voiture(1976,17);

  unsigned int A = Simca1000->getAnnee();
  delete Simca1000;
}
```

Par bonne pratique, à un `new` doit correspondre un `delete` (afin d'éviter les "fuites mémoire")

Cycle de vie des objets et persistance

Un objet a une vie durant laquelle ses attributs et méthodes sont disponibles (persistent)

- Objet classique : dans le bloc de déclaration de l'objet
- Objet pointé : entre le `new` et le `delete` de l'objet

Exemple

```
{
  Voiture* Simca1000;
  unsigned int A;

  {
    Voiture Peugeot308;

    Simca1000 = new Voiture(1976,17);
    A = Peugeot308.getAnnee(); // OK
  }
  A = Simca1000->getAnnee(); // OK
  A = Peugeot308.getAnnee(); // erreur!

  delete Simca1000;

  A = Simca1000->getAnnee(); // erreur!
}
```

Héritage

L'héritage permet de créer des classes à partir d'autres classes pour les spécialiser

- Meilleure structuration et consolidation du code
- Utilisation de la visibilité protected pour un accès aux attributs par les sous classes directes

Exemple super-classe

```
class Vehicule
{
  protected:
    unsigned int Annee;
    unsigned int CodeCouleur;

  public:
    void Avancer();
    void Reculer();
};
```

Exemple sous-classe

```
class Voiture : public Vehicule
{
  public:
    void Tourner(unsigned int D);
    void ChangerVitesse(unsigned int R);
};
```

Méthode virtuelle

Une méthode virtuelle définit un comportement par défaut pour une classe, et qui peut être spécialisé dans une sous classe

Exemple super-classe

```
class Vehicule
{
  protected:
    bool EnAvant;

  public:
    virtual void Reculer();
};

Vehicule::Reculer()
{
  EnAvant = false;
}
```

Exemple sous-classe

```
class Voiture : public Vehicule
{
  private:
    int Vitesse;
    int Rapport;

  public:
    void Reculer();
    ChangerVitesse(int R)
      { Rapport = R; };
};

Voiture::Reculer()
{
  EnAvant = false;
  ChangerVitesse(-1);  // ou R = -1
}
```

Classes abstraites

Une classe abstraite comporte au moins une méthode virtuelle pure

- Elle ne peut être instanciée directement
- Elle peut être instanciée via une sous-classe qui doit définir toutes les méthodes virtuelles pures de la classe abstraite

Exemple classe abstraite

```
class Vehicule
{
  protected:
    unsigned int Annee;

  public:
    virtual void Reculer() = 0;
    virtual void Avancer() = 0;
    unsigned int getAnnee()
      { return Annee; };
};
```

Exemple sous-classe

```
class Voiture : public Vehicule
{
  private:
    bool EnAvant;

  public:
    void Reculer()
      { EnAvant = false; };
    void Avancer();
      { EnAvant = true; };
};
```

Patrons de classe

Un patron de classe (ou template) définit une classe prenant un ou plusieurs types en paramètre(s)

Exemple

```
template<class T>
class Point
{
  private:
    T x, y;

  public:
    T getX() { return x; };
    T getY() { return y; };
    void setXY(T ValueX, T ValueY) { x = ValueX; y = ValueY; };
};

{
  Point<double> PointDouble;
  Point<unsigned int> PointUInt;

  PointDouble.setXY(1.5,-2.7);
  PointUInt.setXY(3,7);
}
```

Autres notions avancées

Pour aller plus loin:

- Polymorphisme
- Surcharge d'opérateurs
- Classes friends
- Méthodes inline
- Héritage multiple
- Code constant
- Code statique
- Traitement des exceptions
- ...

Plan

La Standard Template Library C++ (STL)

La Standard Template Library (STL) est une bibliothèque C++, normalisée par l'ISO (ISO/CEI 14882)

- Gestionnaires de flux de données
- Containers : chaînes de caractères, vecteurs, listes, tableaux associatifs, ...
- Itérateurs pour le parcours de containers
- Algorithmes

Elle utilise massivement toutes les fonctionnalités du C++, notamment les patrons de classe (templates)

Elle est intégralement inclue dans l'espace de nommage `std`

Elle est disponible pour la majorité des systèmes et outils de compilation C++

Flux d'entrées-sorties

La STL définit un ensemble de classes de gestion de flux de données (classes std::ios et ses sous-classes)

- Affichage à l'écran, saisie au clavier
- Lecture-écriture dans des fichiers
- Emission-réception sur un support réseau
- ...

Exemple

```
#include <fstream>
#include <string>
{
  std::string Chaine = "Hello";
  std::ofstream Fichier("fichier.txt");  // ouverture du flux vers fichier.txt

  Fichier << Chaine;  // ecriture de "Hello" dans le fichier
  Fichier << "␣World!" << std::endl; // ecriture de "World!" + saut de ligne
  Fichier.close();  // fermeture du flux vers fichier.txt
}
```

Flux standards

La STL définit un ensemble de flux standards sous la forme d'objets disponibles via <iostream>

- std::cout : flux de sortie standard (affiché à l'écran)
- std::cerr : flux de sortie d'erreurs (affiché à l'écran)
- std::cin : flux d'entrée standard (saisi au clavier)

Exemple

```
#include <iostream>
#include <string>

{
  std::string Saisie;

  std::cin >> Saisie;  // saisie d'une chaine de caracteres
  std::cout << "Valeur saisie : " << Saisie; // affichage de la valeur saisie

  const double Pi = 3.1415927;
  std::cout << "Valeur de PI : " << Pi;
}
```

Les conteneurs (ou containers)

Un conteneur est une structure qui peut contenir un ensemble de données : variables classiques (int, double, ..) ou objets
Deux grandes catégories de conteneurs :

- séquentiels : std::string, std::vector, std::list, std::deque, ...
- associatifs : std::map, std::set, std::multimap, ...

Des itérateurs permettent de les parcourir sans en connaître la structure.

Conteneur std::string (chaîne de caractère)

Les chaînes de caractères peuvent être stockées dans des objets de classe std::string, définie dans <string>

- Stockage du contenu de la chaîne (sans les double-quotes)
- Information sur la chaîne : taille, contenu, est-elle vide?, ...
- Manipulation de la chaîne : concaténation, extraction de sous-chaînes,insertion, remplacement, ...

Exemple

```
#include <string>

{
  std::string Chaine;
  unsigned int a;

  Chaine = "Hello";
  Chaine = Chaine + "␣World!"; // Chaine vaut "Hello World!"
  a = Chaine.size();  // a vaut 12;
  a = Chaine.find("World");  // a vaut 7;
  Chaine.clear();  // Chaine vaut ""
}
```

Conteneur std::vector (vecteur)

Le conteneur std::vector propose une structure séquentielle, de taille dynamique, avec un accès indexé aux éléments contenus

- méthode at(i) ou opérateur [i] pour accéder à l'élément d'indice i. Attention : 1er élément = indice 0
- méthode push_back(E) pour ajouter l'élément E en fin de vecteur
- méthode size() pour obtenir la taille du vecteur

Exemple

```
#include <vector>

{
  std::vector<int> VecteurEntiers;  // creation d'un vecteur d'entiers
  VecteurEntiers.push_back(5);   // ajout de la valeur 5
  VecteurEntiers.push_back(-3);  // ajout de la valeur -3

  std::cout << "Taille du vecteur : " << VecteurEntiers.size() << std::endl;
  std::cout << "Element d'indice 1 : " << VecteurEntiers[1] << std::endl;
}
```

Conteneur std::list (liste)

Le conteneur std::list propose une structure séquentielle, de taille dynamique, avec un accès séquentiel aux éléments contenus

- accès aux éléments via des itérateurs uniquement
- méthode push_back(E) pour ajouter l'élément E en fin de liste
- méthode size() pour obtenir la taille de la liste

Exemple

```
#include <list>
{
  // creation d'une liste de pointeurs sur des voitures
  std::list<Voiture*> ListeVoiture;

  // ajout d'une voiture de 1995 de couleur 5
  ListeVoiture.push_back(new Voiture(1995,5));
  // ajout d'une voiture de 2008 de couleur 1
  ListeVoiture.push_back(new Voiture(2008,1));

  // affichage de l'annee de la voiture contenue en tete de liste
  std::cout << ListeVoiture.front()->getAnnee() << std::end;
}
```

Conteneur std::map (tableau associatif)

Le conteneur std::map propose une structure associative, de type clé-valeur et de taille dynamique, avec un accès indexé par des clés aux éléments contenus

- opérateur [key] pour accéder à l'élément indexé par key
- méthode size() pour obtenir la taille du tableau associatif

Exemple

```
#include <map>
{
  // creation d'un tableau associatif ayant une chaine de caracteres
  // comme cle, et un reel double comme valeur
  std::map<std::string,double> Proprietes;

  Proprietes["nmanning"] = 0.2;
  Proprietes["slope"] = 0.0001;

  std::cout << "Le coeff de manning vaut : " << Proprietes["nmanning"] << std::end;
  std::cout << "La pente vaut : " << Proprietes.at("slope") << "m/m" <<  std::end;
}
```

Plan

Code source

Le code source est écrit sous la forme de fichiers textes:

- Fichiers ".hpp" pour la déclaration
- Fichiers ".cpp" pour la définition

Un fichier "B.cpp" peut utiliser ce qui est défini dans un fichier "A.cpp" via l'inclusion de son fichier de déclaration "A.hpp"

A.hpp

```
class A
{
  private:
    int Attr;
  public:
    void setAttr(int a);
};
```

A.cpp

```
#include <A.hpp>

void A::setAttr(int a)
{
  Attr = a;
}
```

B.cpp

```
#include <A.hpp>

{
  A AObject;

  AObject.setAttr(5);
}
```

Compilateurs GCC

GCC : GNU C Compiler devenu GNU Compiler Collection

- C, C++, ObjectiveC
- Java
- Fortran, Pascal, Ada
- VHDL
- D

GCC permet de créer des programmes exécutables à partir d'un code source dans un des langages supportés

GCC est disponible sur toutes les variantes d'Unix, VMS, Windows, et sur les architectures AMD64, ARM, DEC Alpha, M68k, MIPS, PowerPC, SPARC, x86, Hitachi H8

Make et CMake

Make est un outil permettant d'automatiser une compilation de fichiers sources

- Compilation uniquement des fichiers modifiés depuis la dernière compilation
- Utilisation d'un makefile décrivant l'ensemble des fichiers source constituant le programme

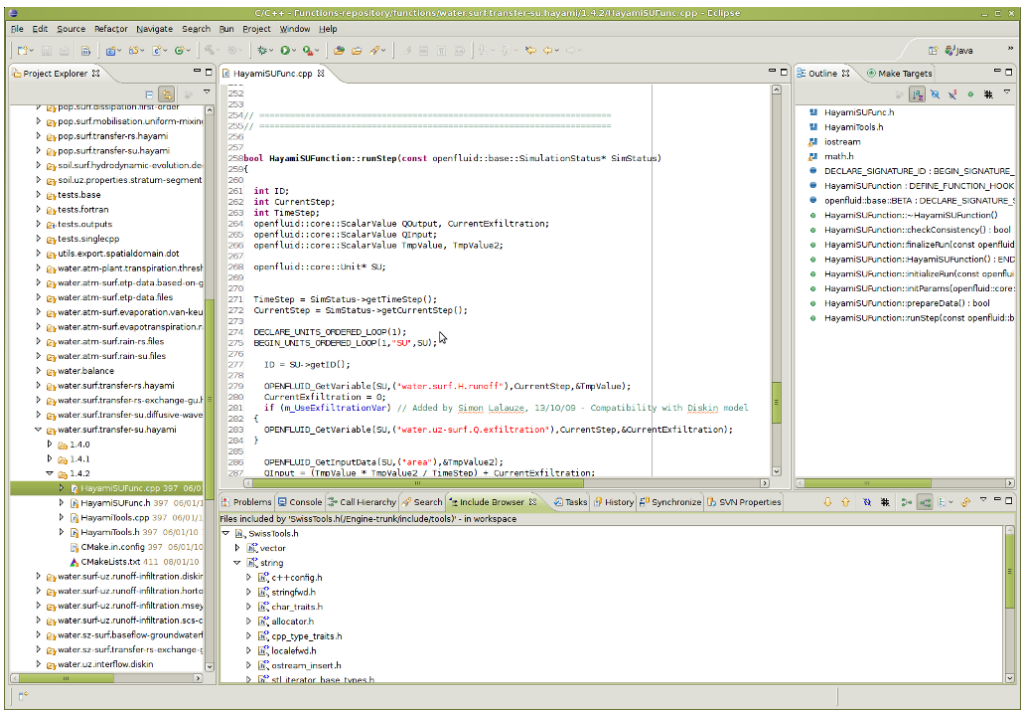
CMake est un outil de génération automatique de makefile, en fonction de l'environnement de développement courant

- Adaptation au système et à l'architecture de la machine
- Vérification de disponibilités des outils et librairies nécessaires

Eclipse

Eclipse est un environnement de développement intégré libre, extensible, universel et polyvalent

- Multi-langage : Java, C, C++, Fortran, PHP, HTML, XML, LaTeX 2_ε, SQL, UML, ...
- Multi-outils : Make, CMake, Subversion, CVS, ...



Bonnes pratiques de développement

De bonnes pratiques de développement, c'est :

- Réguler l'écriture du code source et l'avancée des développements
- Faciliter le partage ou la reprise de développements
- Améliorer globalement la qualité des développements

Ces bonnes pratiques s'appliquent :

- Au nommage des fichiers, variables, classes, ...
- A la forme du code source : indentation, aération, structure, format de commentaires, ...
- Aux choix techniques qui guident le développement

Il est indispensable de définir et suivre de bonnes pratiques de développement dans le cadre de travaux collectifs ou à visée collective

Références & Ressources I


Jean-Christophe Fabre.
Guide du développeur LISAH.
fabrejc@supagro.inra.fr.


Deitel Harvey M. and Deitel Paul J.
C++ - Comment programmer.
Reynald Goulet, 2000.


Björn Karlsson.
Beyond the C++ Standard Library.
Addison-Wesley Professional, 2005.


Gauthier Quesnel.
Introduction au langage C++.
see http://www.vle-project.org/.

Références & Ressources II


C++FAQ Lite.
http://www.parashift.com/c++-faq-lite/.


Developpez.com - C++.
http://cpp.developpez.com/.


Eclipse IDE.
http://www.eclipse.org/.


GCC, the GNU Compiler Collection.
http://gcc.gnu.org/.


Site web OpenFLUID Community.
http://www.openfluid-project.org/community/.


Standard Template Library Programmer's Guide.
http://www.sgi.com/tech/stl/.

Références & Ressources III


The C++ Resources Network.
http://www.cplusplus.com/.