Brève introduction au C++

Simon Rohou

Mars 2019

V1.00
Supports de cours disponibles sur `www.simon-rohou.fr/cours/c++/`

Le module 4.8 Middleware

Introduction

Bases

Les classes

Section 1

Le module 4.8 Middleware

Le module 4.8 Middleware

Middleware : définition

De l'architecture logicielle en robotique :

Un **Middleware** (intergiciel) est un logiciel tiers qui créé un réseau d'échanges d'informations entre différentes applications informatiques.

Il permet de diviser la partie logicielle du robot en une suite de processus pouvant être démarrés simultanément ou séquentiellement au lancement d'une mission.

Le module 4.8 Middleware

Utilité d'un middleware

L'intérêt d'un middleware ? C'est pouvoir :

- **séparer** les applications
  et donc clarifier la répartition du travail au sein d'une équipe
- **paralléliser** facilement les processus
  et tirer profit des n cœurs du processeur utilisé
- **distribuer** les applications sur différentes machines
  car elles sont toutes liées à un serveur
- **rejouer** des missions
  car il est facile de logger tous les changements de variables
- **réutiliser** les applications déjà implémentées
  qui répondent à nos problématiques

Le module 4.8 Middleware

Middlewares existants

Quelques middlewares courants :

- **ROS** : Robot Operating System
- **MOOS** : Mission Oriented Operating Suite
- **YARP** : Yet Another Robot Platform
- **MIRA** : Middleware for Robotic Applications
- **LCM** : Lightweight Communications and Marshalling
- **RTMaps** : Real Time, Multisensor applications
- ...

Le module 4.8 Middleware

Planning

1. **Lundi 11 mars :** introduction au C++
   Intervenants : Simon Rohou
2. **Lundi 18 mars :** middleware ROS (partie 1)
   Intervenants : Thomas Le Mézo, Simon Rohou
3. **Lundi 25 mars :** middleware ROS (partie 2)
   Intervenants : Thomas Le Mézo, Simon Rohou
4. **Lundi 1er avril :** middleware ROS (partie 3)
   Intervenants : Thomas Le Mézo, Simon Rohou
5. **Lundi 8 avril :** projet sur plate-formes DART
   Intervenants : Thomas Le Mézo, Simon Rohou, Benoit Zerr
6. **Lundi 29 avril :** projet sur plate-formes DART
   Intervenants : Thomas Le Mézo, Simon Rohou, Benoit Zerr
7. **Lundi 6 mai :** projet sur plate-formes DART
   Intervenants : Thomas Le Mézo, Simon Rohou, Benoit Zerr
8. **Lundi 13 mai :** évaluation

ENSTA Bretagne

Section 2

Introduction

Introduction

Le langage C++

- début en 1983, aujourd'hui normalisé (C++11)
- langage de programmation **compilé**
- amélioration du langage C
- programmation sous de multiples paradigmes
  - programmation procédurale
  - programmation orientée objet (POO)
  - programmation générique
- **bonnes performances**
- très utilisé, notamment en robotique

ENSTA Bretagne

Introduction

Programmer en C++

1. Éditer le programme avec votre éditeur de texte favori
   - dans ce cours, nous vous proposons *Sublime Text 3*
2. Compiler le programme
   - en utilisant g++
   - ou toute surcouche de compilation telle que `cmake`
3. Exécuter le programme

Section 3

Bases

Bases

Hello World

Programme principal : `main.cpp`

```
1   #include <cstdlib>
2   #include <iostream>
3
4   using namespace std;
5
6   int main()
7   {
8     cout << "Hello World" << endl;
9
10    return EXIT_SUCCESS;
11  }
```

Bases

Hello World

Programme principal : `main.cpp`

```
1   #include <cstdlib> // bibliothèque générique standard
2   #include <iostream> // bibliothèque d'entrées/sorties
3
4   // On se place dans l'espace de nommage std
5   using namespace std;
6
7   // Programme principal
8   int main()
9   {
10    // Affichage à l'écran :
11    cout << "Hello World" << endl;
12
13    // Retourne une valeur de succès d'exécution :
14    return EXIT_SUCCESS;
15  }
```

Bases

Hello World

Programme principal : `main.cpp`

```
1   #include <cstdlib> // bibliothèque générique standard
2   #include <iostream> // bibliothèque d'entrées/sorties
3
4   // On se place dans l'espace de nommage std
5   //using namespace std;
6
7   // Programme principal
8   int main()
9   {
10    // Affichage à l'écran :
11    std::cout << "Hello World" << std::endl;
12
13    // Retourne une valeur de succès d'exécution :
14    return EXIT_SUCCESS;
15  }
```

Bases

Hello World : compilation

Programme principal : `main.cpp`

Compilation avec g++ :

```
g++ main.cpp -o output
```

Résultat :

```
Hello World
```

Bases

Types élémentaires

En C++, toutes les variables sont typées.

- `int` : entiers (voir aussi `short int`, `long`)
- `float` : les *flottants*, des nombres à virgule flottante pour représenter des décimaux
- `double` : flottants à double précision
- `char` : représentation de caractères `'a'`, `'M'`, `'$'`...
- `bool` : booléens, `true` ou `false`

Ainsi que d'autres types particuliers, tels que `size t`.

Bases

Déclaration de variables

Le type d'une variable se définit à sa déclaration. Par exemple :

```
1 float heading;
2 int robots_nb;
3 float robot_speed = 10.; // variable initialisée
4 const float robot_length = 4.; // constante symbolique
```

Notes :

- toute variable doit être définie avant utilisation
- une déclaration peut se faire à tout moment
  - mais on privilégie les déclarations en en-tête de blocs
- une variable reste valable dans le bloc où elle a été déclarée
  - autrement, on parle de variables globales
- une variable non initialisée peut prendre n'importe quelle valeur

Bases

Chaînes de caractères

Les chaînes de caractères n'apparaissent pas nativement en C++. Il faut utiliser la classe `string`. Exemple :

```
1   #include <string>
2
3   string name; // une chaîne de caractères vide
4   string color = "#3D93C1"; // une couleur en convention HTML
5   string method = "euler";
6
7   // Une variable string est un objet :
8   int size = method.size(); // size == 5
9   char c = method[1]; // c == 'u'
10
11  // Concaténations
12  string b = "robotique";
13  string c = "mobile";
14  string a = b + " " + c; // a == "robotique mobile"
```

Bases

Structures de données : classe vector

Il existe de nombreuses structures de données, telles que `stack`, `list`, `map`, largement documentées. On s'intéressera à la classe `vector` qui s'utilise comme un tableau.

```
1   #include <vector>
2
3   vector<float> v_headings; // un tableau de décimaux
4   // Un tableau de tableaux d'entiers :
5   vector<vector<int> > v_v_numbers;
6
7   v_headings.push_back(2.5);
8   v_headings.push_back(0.3);
9
10  float theta0 = v_headings[0]; // theta0 == 2.5
11  double theta1 = v_headings[1]; // cast automatique en double
12
13  int size = v_headings.size(); // size == 2
14  bool is_empty = v_v_numbers.empty(); // is_empty == true
```

Bases

Structures de données : classe vector

Pour parcourir une structure de données, on utilise des `iterator`.
Pour la classe `vector`, on peut simplement itérer les valeurs :

```
1   vector<float> v_sin;
2
3   // Ajout de valeurs dans le tableau :
4   for(int i = 0 ; i < 100 ; i++)
5   {
6     v_sin.push_back(sin(0.1 * i));
7   } // la variable i n'existe plus
8
9   // Mise à jour de ces valeurs
10  for(int i = 0 ; i < v_sin.size() ; i++)
11    v_sin[i] += M_PI / 2.;
12
13  // Affichage
14  for(int i = 0 ; i < v_sin.size() ; i++)
15    cout << v_sin[i] << endl;
```

Bases

Fonctions

Le programme suivant affiche la valeur correspondant à $\sqrt{x^2 + y^2}$.

```
1   #include <cstdlib>
2   #include <iostream>
3   #include <math.h> // nécessaire pour le calcul de sqrt, pow...
4
5   float dist(float x, float y)
6   {
7     return sqrt(pow(x,2) + pow(y,2));
8   }
9
10  int main()
11  {
12    float pos_x = 2., pos_y = 6.;
13    std::cout << dist(pos_x, pos_y) << std::endl;
14    return EXIT_SUCCESS;
15  }
```

Bases

Fonctions

Le programme suivant affiche la valeur correspondant à $\sqrt{x^2 + y^2}$.

```
1   #include <cstdlib>
2   #include <iostream>
3   #include <math.h>
4
5   float dist(float x, float y); // prototype de la fonction
6
7   int main()
8   {
9     float pos_x = 2., pos_y = 6.;
10    std::cout << dist(pos_x, pos_y) << std::endl;
11    return EXIT_SUCCESS;
12  }
13
14  float dist(float x, float y) // définition de la fonction
15  {
16    return sqrt(pow(x,2) + pow(y,2));
17  }
```

Bases

Fonctions procédures

Une *procédure* ne retourne pas de valeurs.
On la déclare de type `void` :

```
1  void write_sqrd_value(float x)
2  {
3    cout << "value: " << pow(x, 2) << endl;
4  }
```

Lorsque plusieurs paramètres sont à retourner, on fait **un passage par référence**.

```
1  void polar(float x, float y, float &rho, float &theta)
2  {
3    rho = sqrt(pow(x,2)+pow(y,2));
4    theta = atan2(y, x);
5
6    // lecture de x, y
7    // écriture de rho, theta
8  }
```

Bases

Valeurs, références et pointeurs

- **Passage par valeur** :

```
1   void write_sqrd_value(float x);
2   // x est en lecture seule
```

- **Passage par référence** :

```
1   void polar(float x, float y, float &rho, float &theta);
2   // x et y en lecture, rho et theta en écriture
```

- **Passage par pointeur** :

```
1   void display_name(string *name);
2   // valeur accessible via l'adresse de la variable name
```

Bases

Les références la modification de valeurs

Pour **modifier la valeur** d'un paramètre dans une fonction, il faut passer ce paramètre par référence.

Une référence sur une variable est un synonyme de cette variable, c'est-à-dire une autre manière de désigner le même emplacement de la mémoire.

On utilise le symbole & pour déclarer une référence.

Bases

Les références pour l'efficacité

Un passage de paramètre par valeur recopie la variable dans le corps de la fonction. Lorsque des objets lourds sont passés par valeur (en lecture), leur copie peut prendre du temps.

On privilégie alors un **passage par référence** pour indiquer une référence directe sur la variable à lire. Mais pour empêcher sa modification, on ajoute le mot-clé `const`.

```
1   // Fonctions équivalentes :
2
3   void display_name(string name)
4   {
5     cout << name << endl;
6   }
7
8   void display_name(const string& name)
9   { // plus rapide à l'exécution
10    cout << name << endl;
11  }
```

Bases

Les pointeurs

Les pointeurs sont très utilisés en C/C++.

- un pointeur est un type de variable renseignant une adresse

```
1   int a = 2;
2   int *ptr_a = NULL; // init. : aucune variable pointée
3   ptr_a = &a; // le pointeur pointe sur a
```

- il permet d'accéder aux valeurs d'une variable en ne conservant que son adresse

```
1   cout << *ptr_a << endl; // affiche la valeur de a : 2
2   ptr_a = 3; // opération interdite (changement d'adresse)
3   *ptr_a = 3; // modifie la valeur de a : a == 3
```

Bases

Les pointeurs : exemple

```
1   void display1(const string *name) {
2     *name = "autre"; // operation interdite (ne compile pas)
3     cout << *name << endl;
4   }
5
6   void display2(string *name) {
7     *name = "autre"; // operation permise
8     cout << *name << endl;
9   }
10
11  int main()
12  {
13    string n = "test";
14    // Passage par pointeur de l'adresse de la variable 'n'
15    display1(&n);      // affiche "test"
16    display2(&n);      // affiche "autre"
17    cout << n << endl; // affiche "autre"
18  }
```

Section 4

Les classes

Les classes

Exemple (Robot.h)

Dans un fichier `Robot.h` :

```
1  #ifndef __ROBOT_H__ // pour éviter les inclusions cycliques
2  #define __ROBOT_H__
3
4
5
6
7
8
9
10
11
12
13
14
15
16 #endif
```

Les classes

Exemple (Robot.h)

Dans un fichier `Robot.h` :

```
1   #ifndef __ROBOT_H__ // pour éviter les inclusions cycliques
2   #define __ROBOT_H__
3
4   class Robot
5   {
6     public:
7
8
9
10
11
12    protected:
13
14  };
15
16  #endif
```

Les classes

Exemple (Robot.h)

Dans un fichier `Robot.h` :

```
1   #ifndef __ROBOT_H__ // pour éviter les inclusions cycliques
2   #define __ROBOT_H__
3
4   class Robot
5   {
6     public:
7
8
9
10
11
12    protected:
13      float m_x = 0., m_y = 0.; // variables de classe
14  };
15
16  #endif
```

Les classes

Exemple (Robot.h)

Dans un fichier `Robot.h` :

```
1   #ifndef __ROBOT_H__ // pour éviter les inclusions cycliques
2   #define __ROBOT_H__
3
4   class Robot
5   {
6     public:
7       Robot(); // constructeur par défaut
8       Robot(float x, float y); // constructeur
9       float pos_x() const; // accesseur (lecture de m_x)
10      void set_pos_x(float x = 0.); // mutateur (écriture de m_x)
11
12    protected:
13      float m_x = 0., m_y = 0.; // variables de classe
14  };
15
16  #endif
```

Les classes

Exemple (Robot.cpp)

Dans un fichier `Robot.cpp` :

```
1   #include "Robot.h"
2
3   Robot::Robot() : m_x(0.), m_y(0.)
4   {
5     // valeurs par défaut
6   }
7
8   Robot::Robot(float x, float y) : m_x(x), m_y(y)
9   {
10    // valeurs configurées
11  }
12
13  Robot::~Robot()
14  {
15    // destructeur
16  }
17
18  // ...
```

Les classes

Exemple (Robot.cpp)

Dans un fichier `Robot.cpp` :

```
1   // ...
2
3
4
5
6   // Accesseur : lecture de la variable de classe m_x
7   // La méthode est 'const' -> elle ne modifie pas l'objet
8   float Robot::pos_x() const
9   {
10    return m_x;
11  }
12
13  // Mutateur : modification de la variable m_x
14  // L'utilisation du 'const' n'est pas possible ici
15  void Robot::set_pos_x(float x)
16  {
17    m_x = x;
18  }
```

Les classes

Héritage

Dans un fichier `UnderwaterRobot.h` :

```
1   class UnderwaterRobot : Robot // héritage de Robot
2   {
3     public:
4       UnderwaterRobot(); // constructeur
5       ~UnderwaterRobot();
6       // ...
7
8     protected:
9       float m_z = 0.; // ajout d'une variable de classe
10  };
```

Les classes

Déclaration

On parle d'instanciation d'objets :

```
1   #include "Robot.h"
2
3   int main()
4   {
5     Robot r1;
6     Robot r2(3., 5.);
7
8     cout << r1.pos_x() << endl; // affiche 0.
9     r2.set_pos_x(6.);
10    cout << r2.pos_x() << endl; // affiche 6.
11
12    return EXIT_SUCCESS;
13  }
```