

Modélisation Transactionnelle des Systèmes sur Puces en SystemC

Ensimag 3A — filière SLE

Grenoble-INP

(Ré)visions de C++ : épisode 1

Matthieu Moy
(transparents originaux de Jérôme Cornet)

Matthieu.Moy@imag.fr

2013-2014

Planning approximatif des séances

- ➊ Introduction : les systèmes sur puce
- ➋ Introduction : modélisation au niveau transactionnel (TLM)
- ➌ **Introduction au C++**
- ➍ Présentation de SystemC, éléments de base
- ➎ Communications haut-niveau en SystemC
- ➏ Modélisation TLM en SystemC
- ➐ TP1 : Première plateforme SystemC/TLM
- ➑ Utilisations des plateformes TLM
- ➒ TP2 (1/2) : Utilisation de modules existants (affichage)
- ➓ TP2 (2/2) : Utilisation de modules existants (affichage)
- ➔ Notions Avancé en SystemC/TLM
- ➖ TP3 (1/3) : Intégration du logiciel embarqué
- ➗ TP3 (2/3) : Intégration du logiciel embarqué
- ➙ TP3 (3/3) : Intégration du logiciel embarqué
- ➚ Intervenant extérieur : ?
- ➛ Perspectives et conclusion

Sommaire

1 C++ : Présentation

2 Bases

3 Classes et objets

4 Généricité

5 Surcharge d'opérateurs

6 Le mot de la fin ...

Présentation

- Langage « objet »
- Création : 1985, Standardisation ISO : 1998, 2003, 2011
- Points forts
 - ▶ Vitesse d'exécution, accès aux couches de bas niveau
 - ▶ Compatible avec le langage C (presque)
 - ▶ Bibliothèque standard (STL = Standard Template Library)
- Points faibles
 - ▶ Gestion de la mémoire manuelle (pas de GC)
 - ▶ Syntaxe
 - ▶ Complexité

C Vs C++

- Techniquement :
 - ▶ C++ est (presque) un sur-ensemble de C
 - ▶ “Il suffit d'apprendre ce qu'il y a en plus dans C++”
- En pratique :
 - ▶ C et C++ sont des langages **différents**.
 - ▶ Les bonnes pratiques de C sont considérées comme mauvaises en C++ !
 - ▶ Les experts C++ recommandent de ne pas se baser sur la connaissance du C pour apprendre C++.
- Dans ce cours :
 - ▶ On fait quand même ce qu'il ne faudrait pas ;-).

Sommaire

1 C++ : Présentation

2 Bases

3 Classes et objets

4 Généricité

5 Surcharge d'opérateurs

6 Le mot de la fin ...

Hello, World !

```
$ cat hello.cpp
#include <iostream>
int main() {
    std::cout << "Hello, world!"
               << std::endl;
    return 0;
}
$ g++ hello.cpp -o hello
$ ./hello
Hello, world!
$
```

Espaces de noms (1/2)

- Encapsulation d'éléments de code dans un espace global
 - ▶ noms de variables
 - ▶ procédures
 - ▶ types
 - ▶ constantes
- On en aura besoin pour la suite...
- Exemple

```
namespace A
{
    typedef uint8 myint;

    void my_function();
}
```


Espaces de noms (2/2)

- Utilisation

```
...  
{  
    A::myint i = 42;  
  
    A::my_function();  
}
```

- Raccourci

```
...  
{  
    using namespace A;  
  
    my_function();  
}
```

Entrées/sorties (1/2)

- Entête `iostream`, espace de nom `std`
- Exemple de sorties écran

```
#include <iostream> // pas de .h !
```

```
using namespace std;
```

```
int main(int argc, char **argv)
```

```
{
```

```
    double d = 4.5;
```

```
    int     i = 3;
```

```
// endl : aller a la ligne, vider le buffer
```

```
cout << "Bonjour" << endl;
```

```
cout << "d : " << d << " i : " << i << endl;
```

```
return 0;
```

```
}
```

Entrées/sorties (2/2)

- Saisie clavier : même principe

```
#include <iostream> // pas de .h !

using namespace std;

int main(int argc, char **argv)
{
    int choix = -1;

    cout << "Dites 33 : " << endl;
    cin >> choix;

    if (choix == 33)
        cout << "Tout va bien" << endl;
    // ...
}
```

Types utilitaires

- Classe string

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

using namespace std;

int main(int argc, char **argv)
{
    string s("bravo"), s2(" jerome!");

    string s3 = s1 + s2; // concatenation

    cout << s3 << endl;

    if (s2 == string(" jerome!")) // comparaison
        cout << "Tout va bien" << endl;
    // ...
}
```

Passages de paramètres (1/3)

- Passage par **valeur** (comme en C)

- ▶ Valeur copiée dans la pile
- ▶ Lecture seule

```
void afficher_nombre(int i)
{
    cout << "nombre : " << i << endl;
}
```

- Passage par **pointeur** (comme en C)

- ▶ **Pointeur sur la valeur** recopié dans la pile
- ▶ Lecture/écriture

```
void incrementer_nombre(int *i)
{
    (*i) = (*i) + 1;
}
```

Passages de paramètres (2/3)

● Passage par **référence**

- ▶ Référence : sorte de « pointeur »
- ▶ « Pointe » toujours sur quelque chose
- ▶ Utilisation possible sur des variables ou des paramètres
- ▶ Exemple sur un paramètre :

```
void incrementer_nombre(int & i)
{
    // i est utilise comme un parametre normal
    i = i + 1;
    // meme effet qu'avec pointeur
}
```

- ▶ Référence **non constante** : accès en lecture/écriture
- ▶ Référence **constante** : accès en lecture seule

Passages de paramètres (3/3)

- Passage par **référence** (suite)

- ▶ Exemple de référence **constante** :

```
void afficher_string(const string & s)
{
    cout << s << endl;
}
```

- ▶ Références : à utiliser de préférence pour le passage en paramètre d'objets
- ▶ Évitent la recopie de tout l'objet sur la pile
- Un bon style de codage en C++ : « On passe des `const` & sauf si on a besoin d'autre chose explicitement ».

Sommaire

1 C++ : Présentation

2 Bases

3 Classes et objets

4 Généricité

5 Surcharge d'opérateurs

6 Le mot de la fin ...

Sommaire de cette section

3

Classes et objets

- Notions de base
- Héritage

Classes et objets

- Classe : regroupement de variables et de code agissant sur ces variables
 - ▶ Variables : **attributs**
 - ▶ Code : **méthodes**
- Objet : instance particulière d'une classe
 - ▶ Valeurs des attributs propres à l'instance
 - ▶ Méthodes partagées par tous les objets
- Comme en Java (ou presque)...

Classes (1/2)

- Séparation **entête** (.h) / **source** (.cpp)
- Définition d'une classe : entête

```
class Camion
{
    public:
        Camion(int positiondorigine);
        void rouler();

    private:
        int position;

}; // attention au ; a la fin!
```

- Modificateurs d'accès : **public**, **private**, **protected**...
- Mot clé particulier **struct** : classe où tout est public

Classes (2/2)

- Définition d'une classe : source

```
#include "Camion.h"

// constructeur
Camion::Camion(int positiondorigine)
{
    position = positiondorigine;
}

// methode rouler
void Camion::rouler()
{
    // ca roule
}
```

- Constructeur exécuté à chaque création d'objet

Création d'objets (1/2)

- Allocation sur la pile

- ▶ À préférer...
- ▶ Mémoire libérée automatiquement en fin de vie de l'objet
- ▶ Syntaxe analogue à la déclaration de variables simples
- ▶ Exemple :

```
void mon_code()  
{  
    Camion c(3); // position d'origine : 3  
  
    c.rouler();  
} // objet c détruit a cet endroit
```

Création d'objets (2/2)

- Allocation dynamique

- ▶ Dans certains cas : tableaux, etc.
- ▶ Penser à libérer la mémoire (pas comme en Java)
- ▶ Exemple :

```
void mon_code_2()  
{  
    Camion *c = new Camion(3);  
  
    c->rouler();  
  
    // destruction du camion  
    delete c;  
}
```

Constructeur par défaut

- Constructeur par défaut = constructeur sans argument
- Appelé par défaut à chaque création d'objet

```
class A {  
public:  
    A() {  
        cout << "Building a A" << endl;  
    }  
};  
  
int main () {  
    A a, b;  
}
```

Quelques éléments en plus

- Symétrique du constructeur : le destructeur

```
// constructeur
```

```
SmartPointer::SmartPointer()  
{  
    objet = new TypeObjet();  
}
```

```
// destructeur
```


```
SmartPointer::~SmartPointer()  
{  
    // liberation de la memoire allouee pour cases  
    // a la destruction de l'objet  
    delete objet;  
}
```


Création de tableaux d'objets

- Opérateurs `new ... []/delete []`

```
TableauDynamique::TableauDynamique(int taille)
{
    objet = new int[taille];
    //          ^^^^^^^^-- Tableau
}
```

```
TableauDynamique::~~TableauDynamique()
{
    // liberation de la memoire allouee pour cases
    // a la destruction de l'objet
    delete [] objet;
    //      ^^-- tableau
}
```

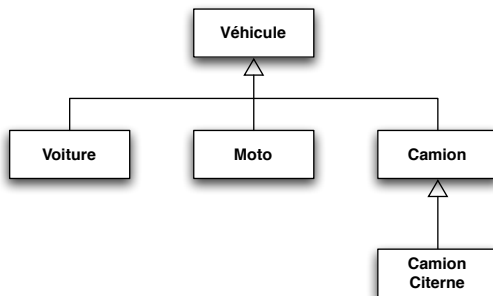
-  Association `new []/delete []` de la responsabilité du programmeur.

Sommaire de cette section

- 3 Classes et objets
 - Notions de base
 - Héritage

Héritage : présentation

- Organisation des classes en une hiérarchie
- Récupération par les classes **filles** des attributs et méthodes des classes **mères**



- Possibilité d'héritage multiple en C++

Héritage : syntaxe

- Exemple : classe mère Vehicule, classe fille Voiture

```
class Vehicule
{
    public:
        Vehicule(const string & immatricul);
    private:
        string immatriculation;
};
```

```
class Voiture : public Vehicule
{
    public:
        Voiture(const string & immatricul,
                int nombredeportes);
    private:
        int nbportes;
};
```

Public, private, protected

```
class toto {  
public:  
    int x;  
    int y;  
private:  
    char z;  
protected:  
    string foo;  
};
```

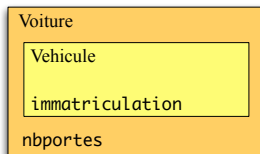
- `public` : visible partout
- `private` : visible uniquement dans cette classe
- `protected` : visible dans cette classe et ses classes filles.

Chaînage de constructeurs (1/2)

cf. code/heritage

- Exemple (suite) : constructeur de Voiture

```
Voiture::Voiture(const string & immatricul,  
                int nombreportes)  
    // chaînage sur classe de base  
    : Vehicule(immatricul)  
{  
    // suite des initialisations  
    nbportes = nombreportes;  
}
```



- Si pas de chaînage explicite, appel du constructeur par défaut (sans argument)
- Chaînage explicite obligatoire s'il n'y a pas de constructeur par défaut dans la classe de base.
- Remarque : sur les destructeurs, pas de paramètres
⇒ chaînage automatique

Chaînage de constructeurs (2/2)

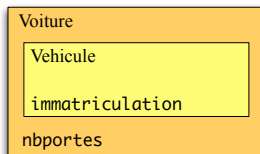
cf. code/heritage

● Exemple (suite) : constructeur de Voiture

```
Voiture::Voiture(const string & immatricul,  
                int nombredeportes)  
    // chaînage sur classe de base ...  
    : Vehicule(immatricul),  
      // ... et sur un champ  
      nbportes(nombredeportes)  
{  
    // rien  
}
```

● Intérêts :

- ▶ On peut initialiser un champ `const`
- ▶ Un seul appel de constructeur
(au lieu d'un appel de constructeur
+ une affectation)



Sommaire

1 C++ : Présentation

2 Bases

3 Classes et objets

4 Généricité

5 Surcharge d'opérateurs

6 Le mot de la fin ...

Sommaire de cette section

4

Généricité

- Définition de patron (template)
- STL : Standard Template Library

Généricité (1/2)

cf. `code/template/`

- Notion déjà vue en VHDL, Ada et Java
- Deux types de généricité :
 - ▶ Sur les **constantes**
 - ▶ Sur les **types**
- Exemple d'utilisation (**instanciation**) :

```
void my_code()  
{  
    TableauInt<10> t;           // sur les constantes  
    TableauDynamique<int> t2;   // sur les types  
  
    TableauDynamique<Camion> *t3; // sur les types  
    t3 = new TableauDynamique<Camion>(20);  
    ...  
}
```

Généricité (2/2)

cf. `code/template/`

- Exemple de généricité sur les constantes :

```
template<int nbcases>
class TableauInt
{
    ...
    int cases[nbcases];
};
```

- Exemple de généricité sur les types :

```
template<typename T>      // on peut trouver
class TableauDynamique    // aussi template<class T>
{
    ...
    T *cases;
};
```

Sommaire de cette section

- 4 **Généricité**
 - Définition de patron (template)
 - STL : Standard Template Library

STL : Bibliothèque de conteneurs génériques

- `std::vector<type>` : tableau redimensionnable
- `std::list<type>` : liste doublement chaînée
- `std::map<key, type, ...>` : tableau associatif
- ...
- cf. <http://www.cplusplus.com/reference/stl/>
- Bibliothèque alternative (non-standard, partiellement obsolète avec C++11) : <http://www.boost.org/>

Sommaire

1 C++ : Présentation

2 Bases

3 Classes et objets

4 Généricité

5 Surcharge d'opérateurs

6 Le mot de la fin ...

Surcharge d'opérateurs

- Surcharge d'opérateur : sémantique d'appel de fonction, avec la syntaxe des opérateurs usuels
- On peut presque tout surcharger :
 - ▶ Opérateurs arithmético-logiques : +, -, =, !, &&...
 - ▶ Accesseurs : [], (), ...
 - ▶ Gestion de la mémoire : new, delete, ->, *, ...
- Non surchargeables : ::, sizeof, *, ? :
- Ne vous étonnez pas si `x = x + 1;` fait 12 appels de fonctions à l'exécution ;-).

Sommaire

1 C++ : Présentation

2 Bases

3 Classes et objets

4 Généricité

5 Surcharge d'opérateurs

6 Le mot de la fin ...

Pratiques courantes en C à éviter en C++

- `type var[]` → `vector<type> var;`
- `char *` → `string`
- `malloc/free` → `new/delete`, ou “smart pointers”.
- `void *` → `templates`
- `pointeurs` → `smart pointers` (cf. STL, boost, ...)
- ...

Un exemple

- Smart pointer avec Comptage de Référence.
- Cf. feuilles distribuées, ou `code/smartpointer`

La nouvelle norme : C++11

(supportée par GCC 4.8.2 et clang 3.3 avec `-std=c++11`)

- Lambda fonctions
- Inférence de type (`auto`)
- Boucle for simplifiée pour itérer sur des conteneurs (comme Java)
- Support natif des threads, opérations atomiques
- Move constructors
- Initialisations unifiées
- `template1<template2<int>>` accepté
- ...