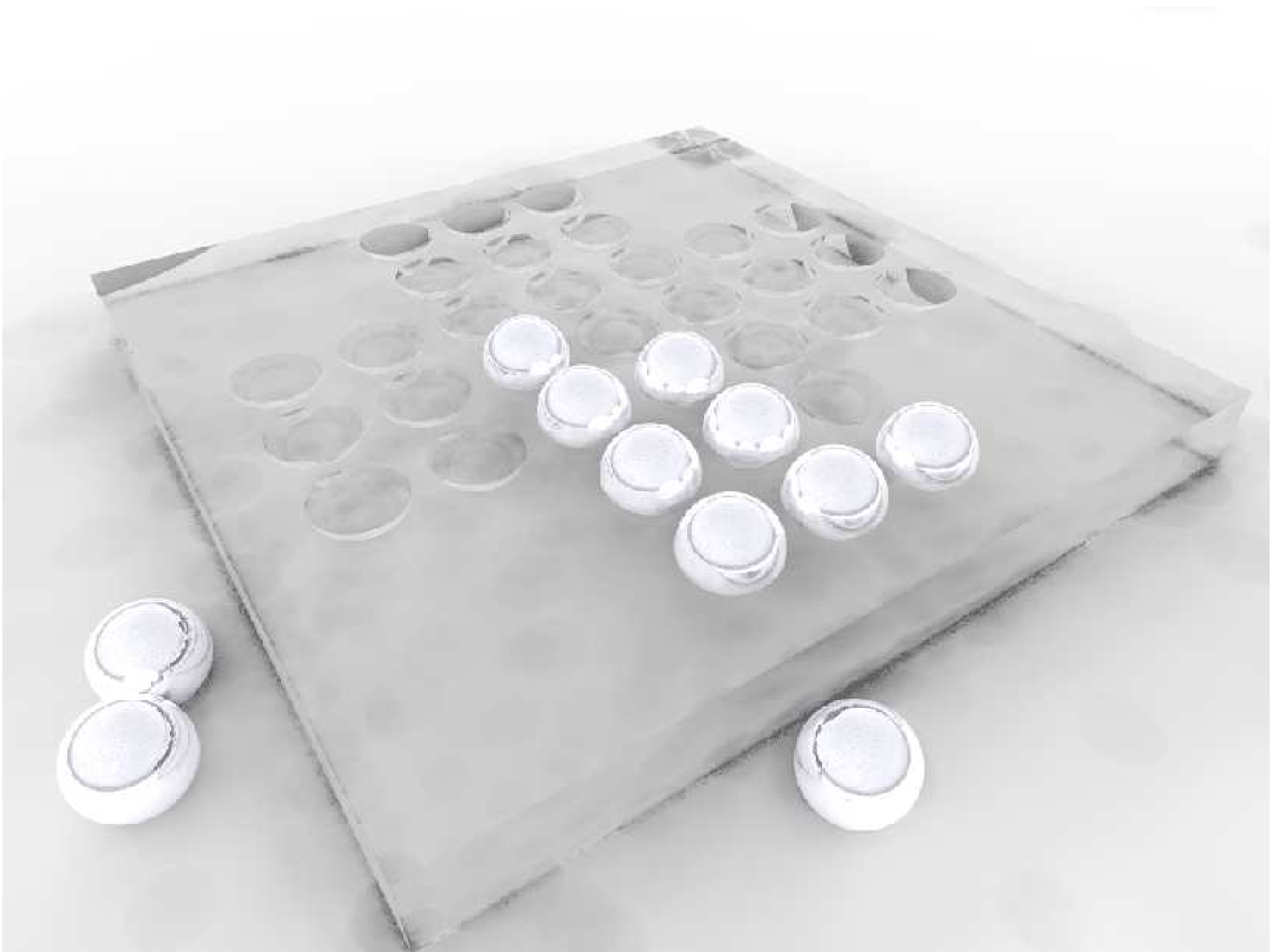
Conception de systèmes matériels et logiciels

Master Ingénierie des Systèmes Industriels Complexes

Fabrice Derepas



Ce document est publié sous la Creative Commons Paternité 2.0 France.
http://creativecommons.org/licenses/by/2.0/fr/legalcode

Image de couverture réalisée à l'aide du logiciel povray http://www.povray.org.

Plan

1 C++ et la modélisation **9**

1.1 Introduction . . . 9
1.2 Un objectif : les besoins de la gestion de projets . . . 10
1.2.1 Ressources . . . 10
1.2.2 Temps . . . 11
1.2.3 Contenu . . . 11
1.3 Qu'est-ce que C++ ? . . . 11
1.3.1 Les entiers en C++ . . . 11
1.3.2 Définir des classes d'objets . . . 12
1.3.3 Une classe est une portée . . . 12
1.3.4 Constructeur et destructeur . . . 13
1.3.5 Tableaux . . . 14
1.3.6 Types génériques . . . 15
1.3.7 Héritage . . . 16
1.3.8 Méthodes virtuelles . . . 17
1.3.9 Définitions modulaires . . . 18
1.3.10 Des données bien ordonnées . . . 19
1.3.11 Autres primitives d'entrées/sorties . . . 20
1.3.12 Structures de contrôle . . . 21
1.3.13 Le préprocesseur . . . 23
1.4 Mise en oeuvre dans SystemC . . . 23
1.4.1 Qu'est-ce que SystemC . . . 23
1.4.2 Pourquoi SystemC ? . . . 24
1.4.3 Exemples d'abstractions utilisées . . . 24
1.4.4 Additionneur sur 8 bits . . . 25
1.4.5 Structure générale des modèles SystemC . . . 25
1.4.6 Exemple de code SystemC . . . 26
1.4.7 Tester le module écrit . . . 27
1.5 Conclusion . . . 29
1.6 Exercices . . . 30

2 Le système Jasip **35**

2.1 Compléments C++ . . . 35
2.1.1 Précision sur le passage de paramètres . . . 35
2.1.2 Pointeurs . . . 35
2.1.3 Références . . . 36
2.1.4 Différence entre références et pointeurs . . . 36
2.1.5 Déréférencement de pointeurs . . . 37
2.1.6 Pointeurs de fonction . . . 38
2.1.7 Pointeurs de méthodes . . . 39
2.1.8 Structure de contrôle pour choix multiples . . . 39
2.1.9 Types énumérés . . . 40
2.1.10 Notion d'opérateurs . . . 41
2.2 Sémantique d'exécution SystemC . . . 47
2.2.1 Phases de simulation d'un modèle SystemC . . . 47

2.2.2 Temps et horloge . . . 48
2.2.3 Instances d'exécutions . . . 49
2.2.4 Communication par files de messages . . . 50
2.2.5 Passage d'arguments à la simulation . . . 52
2.3 Le microprocesseur Jasip . . . 53
2.3.1 Les éléments en jeux . . . 53
2.3.2 Description du cache . . . 54
2.3.3 Description de la RAM . . . 54
2.3.4 Le besoin de modélisation . . . 54
2.4 Exercices sur C++ . . . 54
2.5 Exercices sur le système Jasip . . . 57
2.6 Exercices SystemC . . . 58

3 Test et intégration **59**
3.1 Exemple d'utilisation de la STL . . . 59
3.1.1 Chaînes de caractères . . . 59
3.1.2 Ecrire dans une chaîne de caractères . . . 60
3.1.3 Listes . . . 60
3.1.4 Associations . . . 61
3.2 Architecture de la STL . . . 61
3.2.1 Containers . . . 61
3.2.2 Presque des containers . . . 62
3.2.3 Itérateurs . . . 62
3.2.4 Algorithmes . . . 63
3.2.5 Flux d'entrée/sortie . . . 63
3.3 Style de codage modulaire . . . 64
3.3.1 Interfaces . . . 64
3.3.2 Héritage multiple . . . 65
3.3.3 Classes de base virtuelles . . . 66
3.3.4 Application au codage modulaire . . . 67
3.4 Test des modèles SystemC . . . 67
3.4.1 Test boite noire . . . 68
3.4.2 Test boite blanche . . . 68
3.5 Exercices C++ . . . 68
3.6 Exercices SystemC . . . 69

4 Réseau sur puce **71**
4.1 Complément C++ . . . 71
4.1.1 Retour sur la STL : compléxité des opérations . . . 71
4.1.2 Exceptions . . . 71
4.1.3 Coût des méthodes virtuelles . . . 72
4.1.4 Copie en ligne . . . 72
4.2 Structures internes de SystemC . . . 73
4.2.1 `SC_MODULE` . . . 73
4.2.2 `SC_CTOR` . . . 73
4.2.3 `SC_METHOD` . . . 73
4.3 Canaux hiérarchiques . . . 74
4.3.1 Définition de l'interface . . . 74
4.3.2 Implémentation du canal . . . 74
4.3.3 Utilisation des ports . . . 75
4.4 Ajout de périphériques . . . 76
4.4.1 Opérations à effectuer . . . 76
4.4.2 Mise en mémoire . . . 76
4.4.3 Exemples . . . 77
4.4.4 Zones mémoires . . . 77
4.4.5 Une interface de plus haut niveau . . . 77
4.5 Réseaux sur puce . . . 78

4.6 Exercices SystemC . . . 78

5 Exemple de co-design **83**
5.1 Complément C++ : relecture de code . . . 83
5.1.1 Qu'est-ce que la relecture de code ? . . . 83
5.1.2 Quand faire des relectures de code ? . . . 83
5.1.3 Règles de codage . . . 83
5.1.4 Mémoire . . . 84
5.1.5 Macros . . . 84
5.1.6 Logique . . . 85
5.1.7 Séquence d'appel . . . 85
5.1.8 Outils complémentaires . . . 86
5.2 Cas d'étude pour le co-design . . . 86
5.2.1 Equation des ondes . . . 86
5.2.2 Calcul approché . . . 86
5.3 Exemples de co-design . . . 87
5.3.1 Validation du principe . . . 87
5.3.2 Implémentation Logicielle . . . 87
5.3.3 Ajout de nouvelles commandes . . . 88
5.3.4 Adjonction d'un co-processeur . . . 88
5.3.5 Matériel dédié . . . 89
5.4 Exercice . . . 89

6 Dans le monde réel **91**
6.1 Approches à la Jasip . . . 91
6.1.1 J2ME . . . 91
6.1.2 Android . . . 92
6.1.3 Comparaisons . . . 93
6.2 Unisim . . . 93
6.2.1 Rôle des plate-formes virtuelles . . . 93
6.2.2 Un exemple de réalisation . . . 94
6.2.3 L'interface TLM . . . 94
6.2.4 La gestion mémoire . . . 95
6.3 Exercice . . . 97

7 Fiche Pratique 1 : Compilation C++ **99**
7.1 Compilation . . . 99
7.1.1 Qu'est-ce que la compilation ? . . . 99
7.1.2 Compilation d'un fichier *.cc . . . 99
7.1.3 Génération de l'exécutable ou édition de liens . . . 99
7.1.4 Symboles dans l'exécutable . . . 100
7.2 Automatisation de la compilation . . . 100
7.2.1 Exemple simple . . . 100
7.2.2 Utilisation de variables . . . 101
7.2.3 Règles génériques . . . 102
7.2.4 Dépendance avec les fichiers d'entête . . . 102
7.2.5 Résumé . . . 102

8 Fiche Pratique 2 : compilation Java **105**
8.1 Compilation . . . 105
8.1.1 Qu'est-ce que la compilation Java ? . . . 105
8.1.2 Compilation d'un ensemble de fichiers *.java . . . 105
8.1.3 Visualisation du bytecode . . . 105
8.2 Automatisation de la compilation . . . 105
8.2.1 Makefiles . . . 105
8.2.2 ant . . . 106

Introduction

Ambition du cours

Voici les deux buts principaux de ce cours :

- Appréhender la conception de systèmes complexes du point de vue de l'architecte. Ce cours met en avant le besoin de recourir à différents niveaux d'abstraction permettant de réaliser, dans une logique d'assemblage, des systèmes complexes possédant un nombre de fonctionnalités élevé.
- Acquérir des compétences au niveau chef de projets dans un langage généraliste comme C++. Ce cours n'a pas pour but de former des experts en C++, mais de faire sentir comment un langage généraliste orienté objet peut permettre la mise en oeuvre de systèmes complexes, quelles sont les forces et les faiblesses de projets basés sur des développements en C++.

Pour ce qui concerne la conception de systèmes matériels, le cours se concentre sur des modèles de haut niveau écrits en SystemC, les notions élémentaires de conception de circuits intégrés sont mentionnées. Un simulateur de microprocesseur, un réseau sur puce sont réalisés. Ces architectures sont utilisées pour aborder la problématique de la répartition des tâches matérielles et logicielles.

Pour ce qui concerne la conception de systèmes logiciels c'est le langage C++ qui a été choisi. La Standard Template Library est présentée permettant de dégager les paradigmes importants utilisés en architecture logicielle.

Plan du cours

Aucune hypothèse n'est faite sur les prérequis nécessaires pour suivre le cours. C'est pourquoi l'ensemble des notions C++ nécessaires à un usage raisonnable de SystemC est décrit. Il est cependant clair qu'une pratique préalable d'un langage de programmation permet de tirer un meilleur parti des différents chapitres et que de bonnes compétences en C++, Java ou C# constituent des atouts importants pour suivre le cours avec facilité.

Le chapitre 1 décrit les notions de C++ relatives aux aspects orientés objet. Des exemples simples de SystemC sont exposés.

Le chapitre 2 apporte un complément sur des notions fondamentales en C++ qui n'ont pas été abordées au chapitre précédent. La sémantique des modèles SystemC est décrite avec précision. Le but des exercices SystemC est de simuler la partie mémoire d'un microprocesseur simpliste Jasip.

Le chapitre 3 décrit les principales classes ainsi que l'architecture générale de la STL, la librairie normalisée de C++. Différentes méthodologies de test sont présentées. Les exercices visent à tester les modules créés dans le chapitre précédent.

Le chapitre 4 aborde les constructions internes utilisées dans SystemC. Le but des exercices SystemC est de simuler un microprocesseur multicœur en se basant sur les architectures de réseau sur puce dont la modélisation en SystemC va nous permettre de trouver le bon dimensionnement.

Le chapitre 5 présente l'usage des principaux motifs de conception (design patterns) souvent utilisés en C++. Les aspects méthodologiques en terme de flexibilité, de réutilisabilité de SystemC sont abordés. Différentes façons de partitionner les tâches entre matériel et logiciel sont présentées. Les exercices visent à évaluer l'opportunité de recourir à des modules spéficiques effectuant le calcul de la propagation d'une onde.

Mise en œuvre

Le cours se base sur de nombreux exemples permettant d'illustrer sur des cas concrêts les concepts introduits. Chaque chapitre est suivi d'exercices, qui forment une partie importante de l'apprentissage. La difficulté de chaque exercice est indiquée avec un nombre d'étoiles :

- * : la solution de l'exercice est presque immédiate. Il est donc important de savoir le résoudre.
- ** : la solution de l'exercice peu prendre un peu de temps. Il est bon de faire l'exercice ou de comprendre la solution.
- *** : l'exercice n'est pas évident ou bien concerne les personnes désireuses d'approfondir un sujet donné.

Les exercices sont la plus part du temps corrigés en ligne à l'url :

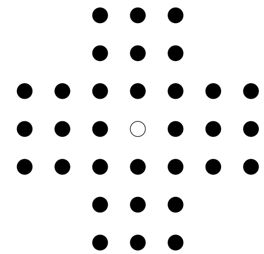
`http://www.enseignement.polytechnique.fr/profs/informatique/Fabrice.Derepas/`

Chapitre 1

C++ et la modélisation

1.1 Introduction

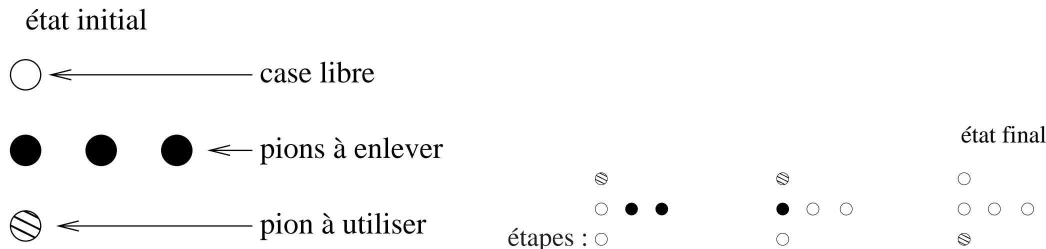
Certains problèmes sont difficiles à résoudre entièrement. Il est alors utile de les décomposer en problèmes plus petits, puis d'assembler les solutions obtenues pour résoudre le problème d'origine.



Prenons l'exemple du jeu de solitaire. Le jeu se déroule sur un plateau contenant des positions qui peuvent être occupées ou non. Dans l'état initial toutes les positions sont occupées sauf le centre, comme illustré ci-contre. Le problème est de trouver la suite de déplacements permettant de finir avec un seul pion au milieu du plateau (c'est à dire à l'emplacement initialement vide). Les pions sont enlevés et déplacés en utilisant la règle suivante : un pion peut sauter par dessus l'un de ses quatre voisins immédiats (en haut, en bas, à droite ou à gauche) s'il arrive dans un emplacement vide. Le pion par dessus lequel le saut s'est effectué est alors retiré du plateau. Ci-dessous figure un exemple de déplacement :

On peut bien entendu essayer de résoudre le problème dans sa globalité au hasard. Cela peut prendre beaucoup de temps. Une autre idée est de diviser le problème en problèmes plus simples : nous allons apprendre à enlever des formes de base, puis nous essayerons, un peu comme dans un puzzle de résoudre le problème complet à l'aide de ces problèmes plus simples. Les formes dont nous allons nous servir pour résoudre le problème sont données par la pratique.

La première forme simple à enlever est une rangée de trois pions (en noir sur le schéma) à l'aide d'un pion hachuré en utilisant la place vacante :

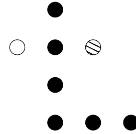


Il n'y a pas le choix : il faut se servir d'abord du pion hachuré, puis du pion de droite pour retirer le pion du milieu. Enfin le pion hachuré revient à sa place. Le bilan est que l'on a fait disparaître les pions noirs à l'aide du pion hachuré, en disposant de la case vacante.

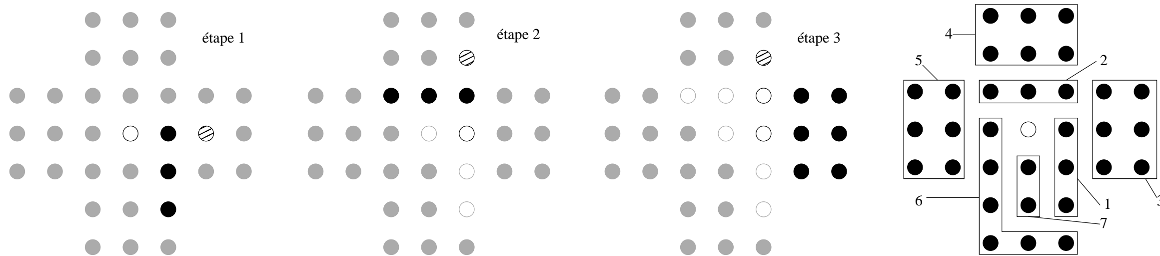
Ainsi un nouveau mouvement a été mis en évidence : quand le motif des trois pions avec la case libre et le pion à utiliser est rencontré alors les trois pions peuvent être enlevés directement.

De manière naturelle au regard du découpage du plateau, la forme suivante est un bloc de 6 pions à retirer, en voici deux exemples :

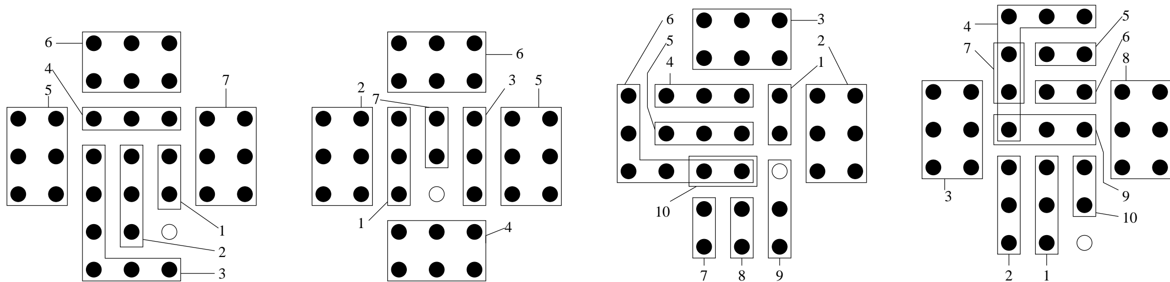
Enfin la troisième forme de base est le « L » :



Voici la solution du problème initial où les numéros représentent l'ordre dans lequel enlever les formes :



Cette méthode permet également de trouver des solutions pour d'autres problèmes, par exemple avec un état initial différent :



Nous manipulons ainsi des notions plus abstraites qui sont les formes avec lesquelles nous pavons le plateau. Bien entendu une telle méthode possède des inconvénients : un raisonnement par blocs peut nous priver de certaines solutions.

Ce type de démarche est fréquent en informatique, elle s'applique par exemple à du matériel, où l'on cherche à s'abstraire du comportement des transistors, pour dégager des fonctionnalités, puis à nouveau en utilisant un niveau d'abstraction supplémentaire, des fonctionnalités sont regroupées en blocs communiquants entre eux.

De même, de nos jours la programmation en assembleur est peu fréquente, des concepts de plus haut niveau qui tendent à faire abstraction de la machine sont utilisés : ainsi le langage C fait abstraction des instructions du microprocesseur sous jacent, le langage Java tend à faire oublier au programmeur la manipulation de la mémoire.

1.2 Un objectif : les besoins de la gestion de projets

Même si la programmation peut être un loisir, l'objectif principal est la réalisation de projets. Il est donc fondamental de prendre en compte les besoins nécessaires à la bonne marche des projets.

La gestion de projets vise à la réalisation, dans un temps imparti, d'objectifs fixés, en utilisant des ressources données. Voici quelques rappels sur les besoins nécessaires à une bonne gestion de projets. La suite de ce chapitre présente comment un langage comme C++ peut répondre à ces besoins.

1.2.1 Ressources

Les personnes participant au projet sont une ressource essentielle. Chacun a ses compétences propres et tout le monde n'est pas interchangeable. Il faut souvent pouvoir au sein du même projet faire cohabiter du code écrit par des personnes ayant des compétences spécifiques, ou pouvoir réutiliser du code existant ayant déjà été écrit avant même le début du projet. Ainsi il est important que tout le monde ne travaille pas sur le même fichier, qu'il existe des espaces de nommage bien déterminés pour que les noms des variables ou des fonctions ne rentrent pas en conflit.

Le matériel, une autre partie importante des ressources est également à prendre en compte. Ce dernier pouvant changer rapidement (taille du disque dur, nouveau périphérique) il faut pouvoir abstraire si possible de tout comportement spécifique à un matériel. Ainsi le matériel est généralement manipulé via des classes d'objets comportant une partie publique : la liste des actions à effectuer sur le matériel, et une partie privée qui est spécifique au matériel. Lors de la prise en charge d'un nouveau matériel, il suffit alors théoriquement de changer la partie privée, mais la manipulation se fera de la même manière.

1.2.2 Temps

Le temps est un facteur fréquent d'échec de projets informatiques. Une bonne gestion de projets va définir des tâches et des jalons permettant de valider le bon avancement de ces tâches. Ce découpage, qui est une étape nécessaire, doit être rendu possible par un langage permettant une approche modulaire. La notion de classe d'objets permet d'avoir cette modularité.

L'approche modulaire implique une définition d'interfaces permettant aux différents modules de communiquer. Nous verrons qu'il est possible lors de l'écriture d'un programme de séparer la définition des classes objets de leur fonctionnement. Il suffit alors de fournir le fichier de définition aux personnes souhaitant utiliser la classe d'objets.

1.2.3 Contenu

Une partie importante du projet est la définition du contenu du projet. Le contenu peut varier facilement, le plus souvent par des ajouts de nouvelles fonctionnalités, qui prises indépendamment les unes des autres sont mineures mais dont la somme est importante.

C++ permet facilement de manipuler des objets ayant des niveaux d'abstraction différents. Cela permet d'avoir dès les premières étapes du développement des prototypes permettant de cerner au plus tôt le périmètre du projet.

1.3 Qu'est-ce que C++ ?

Nous avons précédemment évoqué les notions de classe, d'espace de nommage, d'actions à effectuer sur un objet. Nous allons ici développer ces notions. Une série d'exemples propose une approche intuitive au langage de programmation C++. C++ est une norme ISO [iso03] tout comme C [iso99]. Bjarne Stroustrup, à l'initiative de C++, a écrit un ouvrage de référence sur C++ : The C++ Programming Language [Str97].

1.3.1 Les entiers en C++

Les nombres entiers sont fréquemment manipulés. Voici les principaux types d'entiers en C++ :

- `int` type entier signé sur 32 bits (sur une machine 32 bits) allant de $-2147483648 = -2^{31}$ à $2147483647 = 2^{31} - 1$.
- `unsigned int` entier signé sur 32 bits (sur une machine 32 bits) allant de 0 à $4294967295 = 2^{32} - 1$.
- `short` type entier signé sur 16 bits allant de $-32768 = -2^{15}$ à $32767 = 2^{15} - 1$.
- `unsigned short` entier signé sur 16 bits allant de 0 à $65535 = 2^{16} - 1$.
- `char` type entier signé sur 8 bits allant de $-128 = -2^{7}$ à $127 = 2^{7} - 1$.
- `unsigned char` entier signé sur 8 bits allant de 0 à $255 = 2^{8} - 1$.
- `long long int` type entier signé sur 64 bits (équivalent à `int` sur une machine 64 bits).
- `unsigned long long int` type entier non signé sur 64 bits.

A la page 20 figure un listing permettant d'afficher les bornes sur les entiers en fonction de la machine sur laquelle on se trouve.
Voici quelques instructions où les valeurs des variables sont données en commentaire :

```
1     unsigned char c1 = 255;    // définition de c1 comme un unsigned char valant 255
2     c1=c1+1;                   // incrémenter c1 de 1, c1 vaut maintenant 0
3     char c2 = 127;             // définition de c2 comme un char valant 127
4     c2=c2+1;                   // c2 vaut maintenant -128
5     short s1 = 32767;
6     s1=s1+1;                   // s1 vaut maintenant -32768
7     unsigned short s2 = 65535;
8     s2=s2+1;                   // s2 vaut maintenant 0
9     int i=2147483647;
10    i=i+1;                     // i vaut maintenant -2147483648
11    unsigned int j=4294967295;
12    j=j+1;                     // j vaut maintenant 0
13    unsigned long long int lli = 0;
```

```
14    lli=lli-1;                // lli vaut maintenant 18446744073709551615=2^64-1
```

Toutes les instructions se terminent par le caractère `;`. Les commentaires sont soit compris entre `//` et la fin de la ligne ou entre `/*` et `*/`. Voici un exemple :

```
1  // ceci est un commentaire sur une ligne
2  /* ceci est un commentaire
3     sur plusieurs lignes */
```

Outre les opérations arithmétiques il existe des opérations dites bit à bit sur les entiers. Citons en particulier :

– le « ou » bit à bit, noté `i|j` pour deux entiers `i` et `j`, le n-ième bit de `i|j` est le résultat de l'opération logique ou entre le n-ième bit de `i` et le n-ième bit de `j`.
– le « et » bit à bit, noté `i&j` pour deux entiers `i` et `j`, le n-ième bit de `i&j` est le résultat de l'opération logique et entre le n-ième bit de `i` et le n-ième bit de `j`.

Voici un exemple pour 12 et 6, on constate que $12|6 = 14$ et $12\&6 = 4$:

12	1	1	0	0		1	1	0	0
6	0	1	1	0		0	1	1	0
12|6	1	1	1	0	12&6	0	1	0	0

1.3.2 Définir des classes d'objets

En C++ on peut définir des classes d'objets. Par exemple une voiture est une classe d'objet. Les éléments appartenant à la classe sont nommés instances. Par exemple ma voiture est une instance de la classe d'objets voiture. Plus généralement une classe représente un ensemble d'objets ayant un comportement similaire. Voici la syntaxe C++ pour définir ces concepts :

```
1  class Voiture { };    // définition de la classe Voiture.
2  Voiture maVoiture;    // création d'une instance particulière nommée maVoiture.
```

La première ligne déclare une classe nommée `Voiture`. La syntaxe est la suivante :

`class` <*nom de la classe* >{ <*description* >} `;`

Dans cet exemple, la description de la classe est vide. La deuxième ligne déclare une instance de la classe `Voiture` nommée `maVoiture`. Comme les types primitifs (`int` , `short`, `char`, ...), la syntaxe est la suivante :

<*nom de la classe* ><*nom de l'instance* > `;`

Les classes que nous pouvons définir forment, comme les entiers que nous avons vus, des types sur lesquels nous allons pouvoir réaliser des actions.

1.3.3 Une classe est une portée

Les langages informatiques permettent d'appréhender un monde complexe où des objets de différentes échelles vont pouvoir être exprimés : une voiture, le régulateur de vitesse de la voiture, le microprocesseur dans le régulateur de vitesse, etc... . Seule une approche modulaire permet d'exprimer une telle différence d'échelle. C'est pourquoi une classe peut être considérée comme une unité logique d'expression ou la portée logique de certaines notions.

Ainsi on peut définir d'autres concepts qui peuvent interagir entre eux : par exemple la voiture contient 4 roues et un régulateur de vitesse. Le régulateur de vitesse contient lui même un microprocesseur :

```
1  class Microprocesseur { };
2  class Pneu { };
3  class Regulateur {
4          Microprocesseur proc;  // déclaration d'une variable au sein de la classe
5  };
6  class Voiture {
7          Pneu pneuAvantDroit;
```

```
8           Pneu pneuAvantGauche;
9           Pneu pneuArriereDroit;
10          Pneu pneuArriereGauche;
11          Regulateur regulateurDeVitesse;
12   };
```

Une variable déclarée au sein d'une classe est nommée champ de la classe, ainsi la classe `Regulateur` comporte un champ `proc` qui est une instance de `Microprocesseur`. Ainsi toute instance de `Regulateur` comporte une instance de `Micropro cesseur` qui lui est propre. La classe étant une portée logique il faut définir ce qui est visible de l'extérieur (c'est à dire ce qui est public) et ce qui est « caché » à l'intérieur (c'est à dire ce qui est privé). Nous allons définir une classe d'objets nommée `Voiture`, où sur les instances de cette classe il sera possible d'effectuer les actions suivantes : ouvrir ou fermer une porte. Nous allons modéliser l'état de la porte par un booléen, c'est le type C++ `bool` (à `true` pour ouvert et à `false` pour fermé) :

```
1    class Voiture {              // définition de la classe Voiture
2    public:
3           void ouvrePorte() {       // définition de la méthode ouvrePorte.
4                   etatPorte=true;   // affecte la valeur true à la variable etatPorte
5           }
6           void fermePorte() {       // définition de la méthode fermePorte.
7                   etatPorte=false;  // affecte la valeur false à la variable etatPorte
8           }
9    private:
10          bool etatPorte;           // définition d'une variable booléenne, etatPorte.
11   };
12   int main() {                // définition d'une fonction nommée main
13          Voiture maVoiture;        // création d'une variable nommée maVoiture
14          maVoiture.ouvrePorte();   // appel de la méthode ouvrePorte
15                                    // sur l'instance maVoiture
16          return 0;                 // retourne la valeur zéro, puisque l'exécution
17                                    // s'est bien déroulée.
18   }
```

Les déclarations situées après l'attribut `public:` sont publiques, c'est à dire accessibles depuis l'extérieur de la classe (la fonction `main` appelle ainsi la méthode `ouvrePorte` qui est publique). Les déclarations situées après `private:` sont privées, ainsi la fonction `main` ne pourrait effectuer l'instruction `maVoiture.etatPorte=true`, seules le méthodes de la classe `Voiture` peuvent manipuler les champs ou les méthodes privées. Les actions effectuées au sein d'une classe sont nommées méthodes. Par exemple `ouvrePorte` est une méthode de la classe `Voiture`. Ici `void` est le type renvoyé par la méthode `ouvrePorte`, en l'occurrence rien du tout, et les parenthèses `()` contiennent la liste des arguments de la fonction ou de la méthode, en l'occurrence aucun argument. La définition de la méthode suit alors entre les accolades `{ }`.

Hors d'une classe les actions sont regroupées en fonctions. Dans l'exemple précédent, on trouve une fonction nommée `main`, ne prenant aucun argument et renvoyant un entier `int`. Par convention la fonction nommée `main` est la fonction appelée au début du programme. L'instruction `return 0` retourne la valeur zéro et signifie par convention que l'exécution du programme s'est bien déroulée.

Voici la syntaxe de la déclaration d'une fonction ou d'une méthode :

<type de retour><nom de la fonction>`(` *<liste d'arguments >*`)`
`{` *<corps de la fonction >*`}`

Une liste d'arguments est soit vide soit une liste d'éléments de la forme *<type ><nom de l'argument >*séparés par des virgules. Dans la section suivante figure un exemple pour le constructeur `Voiture` à la ligne 8.

1.3.4 Constructeur et destructeur

Dans l'exemple précédent à la section 1.3.3 un point n'est pas précisé : en effet dans la fonction `main` après la création de la variable `maVoiture` à la ligne 13 et avant l'appel à la méthode `ouvrePorte` à la ligne 14, que vaut la variable `etatPorte` de l'instance `maVoiture` ? Sa valeur n'est pas définie, cela ne pose pas de problème dans cette exemple précis car elle n'a pas été lue. Il est cependant souhaitable que tous les champs d'une instance soient initialisés correctement dès sa création. Dans ce but, on rajoute une méthode un peu particulière dénommée constructeur, automatiquement appelée à la création de l'objet.

Nous allons définir une classe comportant deux constructeurs : un constructeur sans argument où l'état de la porte est initialisé à `false`, et un autre constructeur permettant de choisir la valeur de l'état initial de la porte.

```
1   class Voiture {
2   public:
3           // constructeur par défaut (sans arguments), la porte est fermée
4           Voiture () {
5                  etatPorte=false;
6           }
7           // constructeur où l'état initial est précisé
8           Voiture (bool etatInitial) {
9                   etatPorte=etatInitial;
10          }
11  private :
12          bool etatInitial;
13  }
14  int main() {               // définition d'une méthode nommée main
15          Voiture maVoiture (true);   // création de variable et appel de constructeur
16                                      // ou l'état initial est précisé.
17          Voiture monAutreVoiture;    // appel implicite au constructeur par défaut.
18          return 0;
19  }
```

Symétriquement au constructeur un destructeur nommé `~Voiture` est appelé quand la variable est détruite. Dans l'exemple ci-dessous, l'état de la porte est fixé à `true` quand l'instance de la classe est détruite :

```
1   class Voiture {
2   public:
3           // constructeur par défaut, la porte est fermée
4           Voiture () {
5                  etatPorte=false;
6           }
7           // destructeur : ouvre les portes avant de détruire l'objet.
8           ~Voiture () {
9                   etatPorte=true;
10          }
11  private :
12          bool etatInitial;
13  }
14  int main() {
15          Voiture maVoiture; // appel au constructeur par défaut.
16          return 0;
17  }  // en fermant l'accolade toutes les variables créées dans la fonction sont
18     // détruites, les destructeurs sont alors appelés.
```

Ici l'action d'affecter `true` au champ `etatPorte` au moment de la destruction peut sembler dérisoire. En effet à quoi cela sert-il puisque l'objet va être supprimé de la mémoire ? Si le code C++ pilote des équipements externes, affecter certaines valeurs à des variables peut conditionner effectivement le déplacement d'une porte. Le plus souvent le code que l'on trouve dans le destructeur consiste à libérer la mémoire qui a été allouée tout au long de la vie de l'objet. Le chapitre 2 détaille les aspects liés à l'allocation dynamique de mémoire.

Le fait de disposer de deux fonctions ayant le même nom mais des arguemnts différents (comme pour les constructeurs dans l'exemple précédent) est nommé surcharge.

1.3.5 Tableaux

Lorsque l'on manipule une série de données de même type, il est souvent utile de les indexer par un entier, on peut alors parler du 3eme objet, de l'objet suivant, ou du précédent. Les structures de données les plus simples permettant de faire cela sont les tableaux.

Nous allons définir un tableau comportant quatre instances de la classe `Voiture` :

```
1   // définit la classe Voiture
```

```
2   class Voiture {
3   public :
4           Voiture() { valeur = -1;}   // constructeur par défaut.
5           Voiture(int prix) { valeur = prix; }
6   private :
7           int valeur;
8   };
9   // définit un entier initialisé à la valeur 4
10  int maxNombreDeVoitures = 4;
11  // définit un tableau à maxNombreDeVoitures éléments de type Voiture
12  Voiture monTableau [maxNombreDeVoitures];
```

Dans cet exemple les indices du tableau vont de 0 à 3 (c'est à dire `maxNombreDeVoitures`-1). On remarque que le constructeur attend un argument `i` qui est de type `int`. Voici par exemple une initialisation du tableau dans la fonction `main` :

```
13  int main() {
14          monTableau[0]=Voiture(2700); // initialise le 1er élément du tableau
15          monTableau[1]=Voiture(2334);
16          monTableau[2]=Voiture(4750);
17          monTableau[3]=Voiture(4406); // initialise le 4eme élément du tableau
18          return 0;
19  }
```

Dans l'exemple ci-dessus on aurait également pu initialiser le contenu du tableau au moment de sa déclaration de la manière suivante :

```
11  Voiture monTableau [] = {
12          Voiture(2700),
13          Voiture(2334),
14          Voiture(4750),
15          Voiture(4406)
16  };
17  int main() {
18          return 0;
19  }
```

On remarque alors qu'il n'y pas besoin de préciser la taille du tableau entre crochets, puisqu'elle est donnée par le nombre d'éléments. Il est à noter que si `main` est la première fonction exécutée, les constructeurs des variables globales du programme (ici les constructeurs des éléments du tableau) vont être exécutés avant la fonction `main`.

1.3.6 Types génériques

Nous avons abordé dans les contraintes de la gestion de projets la possibilité de réutiliser du code. Par exemple supposons l'existence une classe représentant une voiture nommée `Voiture1` comporte 4 pneus. Puis un peu plus tard apparaît le besoin de disposer d'une classe `Voiture2` comportant 4 pneus d'une marque différente. On peut bien entendu réécrire la classe `Voiture2` *ex nihilo*. Mais plus tard le besoin pourrait se faire sentir pour encore d'autres marques de pneus : des Michelins, des Goodyears, des Bridgestones, des Pirellis, où les mêmes opérations sont effectuées sur la voiture quelque soit la marque de pneus. Pour ne pas réécrire à chaque fois une nouvelle classe, nous allons déclarer un type générique qui pourra être n'importe quel type de pneu :

```
1   template <class Pneu> class Voiture {
2   public:
3           Voiture () { }
4           void retirePneu(int i) { mesPneus[i].retire();}
5           void mettrePneu(int i) { mesPneus[i].mettre();}
6   private:
7           Pneu mesPneus[4];  // conserver les 4 pneus dans un tableau
8   };
```

On utilise ici une classe `Pneu` qui n'a pas encore été définie. On dit que `Voiture` est une classe template ou patron. On s'attend juste à pouvoir appeler sur les instances ayant le type `Pneu` les méthodes `retire` et `mettre`.

Si l'on veut alors créer des instances de `Voiture`, il faut préciser le type de pneus. Considérons alors un type `Michelin` et un type `Bridgestone` :

```
9   class Michelin {
10  public :
11          Michelin() { etatPneu=false;}
12          void mettre() { etatPneu=true;}
13          void retire() { etatPneu=false;}
14  private :
15          bool etatPneu;
16  };
17  class Bridgestone {
18  public :
19          Bridgestone() { etat=0;}
20          void mettre() { etat=1;}
21          void retire() { etat=0;}
22  private :
23          int etat;
24  };
25  int main() {
26          // créer une voiture avec des pneus Michelin :
27          Voiture<Michelin> maVoiture();
28          maVoiture.mettrePneu(0);
29          maVoiture.mettrePneu(1);
30          maVoiture.mettrePneu(2);
31          maVoiture.mettrePneu(3);
32          // créer une voiture avec des pneus Bridgestone :
33          Voiture<Brigestone> autreVoiture();
34          return 0;
35  }
```

Tout d'abord deux classes `Michelin` et `Bridgestone` sont définies. Elles ont des mécanismes différents (l'une conserve l'état dans un booléen, l'autre dans un entier), mais implémentent toutes deux les méthodes `mettre` et `retire`. Ces méthodes étant définies, on peut dans la suite du programme parler des types `Voiture<Michelin>` ou `Voiture<Brigestone>`. Ces deux types sont nommés instance de template.

L'exercice 1.6 propose une approche intensive de l'usage des templates permettant de mieux comprendre le mécanisme de typage de C++.

1.3.7 Héritage

Il existe des types particuliers de voitures : par exemple la voiture de sport, ou la voiture de luxe. A chaque fois les caractéristiques de base de la voiture sont présentes, mais avec des possibilités supplémentaires. Il serait fastidieux de réécrire ce qui a été déjà fait dans la classe modélisant une voiture, par ailleurs en terme de maintenance le travail serait plus important. Nous souhaitons simplement étendre le concept de voiture en ajoutant des données supplémentaires. Ce mécanisme se nomme héritage.

Voici comment à partir d'une classe `Voiture` définir une classe `VoitureDeSport` reprenant les méthodes de `Voiture` et comportant un champ de type `SuperMoteur` :

```
1   // définir une voiture
2   class Voiture {
3   public:
4           Voiture () { etatVoiture = false; }
5           void demarre() { etatVoiture=true; }
6           void arrete() { etatVoiture=false; }
7   private:
8           bool etatVoiture;
```

```
 9  };
10  class SuperMoteur { };
11  class VoitureDeSport : public Voiture {
12  public :
13          VoitureDeSport () { arrete(); }
14  private :
15          SuperMoteur leMoteur;
16  };
```

On a tout d'abord définit une classe `Voiture` et une classe `SuperMoteur`. Ensuite la syntaxe `class VoitureDeSport : public Voiture` crée une définition de classe se basant sur `Voiture`. C'est-à-dire que toute instance de `VoitureDeSport` pourra également être considéré comme une instance de `Voiture`. Tous les champs et toutes les méthodes publiques de `Voiture`, la classe parent, sont disponibles dans la classe fille `VoitureDeSport`.

Dans la classe `VoitureDeSport` on ne peut pas manipuler le champ `etatVoiture` qui est déclaré avec l'attribut `private`. Pour rendre le champ `etatVoiture` disponible dans les classes héritées sans pour autant qu'il soit publique on peut le déclarer `protected`. Voici un exemple équivalent au précédent :

```
 1  // définir une voiture
 2  class Voiture {
 3  public:
 4          Voiture () { etatVoiture = false; }
 5          void demarre() { etatVoiture=true; }
 6          void arrete() { etatVoiture=false; }
 7  protected:
 8          bool etatVoiture;
 9  };
10  class SuperMoteur { };
11  class VoitureDeSport : public Voiture {
12  public :
13          VoitureDeSport () { etatVoiture=false; }
14  private :
15          SuperMoteur leMoteur;
16  };
```

1.3.8 Méthodes virtuelles

Reprenons l'exemple précédent, en ajoutant une méthode `demarre` différente dans la voiture de sport :

```
11  class VoitureDeSport : public Voiture {
12  public :
13          VoitureDeSport () { etatVoiture = false; }
14          void demarre() { etatVoiture=true; leMoteur.demarre(); }
15  private :
16          SuperMoteur leMoteur;
17  };
```

Supposons que l'on dispose d'une fonction, nommée `testVoiture`, permettant de tester le comportement d'une voiture :

```
18  void testVoiture ( Voiture v) {
19          // effectue les tests:
20          v.demarre();
21          v.arrete();
22  }
```

Une instance de `VoitureDeSport` pouvant également être considérée comme une instance de `Voiture` on peut appeler la fonction `testVoiture` sur une instance de `VoitureDeSport`. Lorsque l'on appelle `testVoiture` avec un objet de type `VoitureDeSport` on souhaite que la méthode `demarre` appelée soit celle définie dans `VoitureDeSport`. Ce n'est pas le cas dans avec le code écrit jusqu'ici. Pour que cela soit le cas, il faut déclarer la méthode `demarre` avec l'attribut `virtual` dans la classe `Voiture` :

```
// définir une voiture
class Voiture {
public:
        virtual void demarre() { ... }
        // ...
};
class VoitureDeSport : public Voiture {
public :
        void demarre() { ... }
        // ...
};
```

Si on omet l'attribut virtual dans la classe `Voiture` alors dans la fonction `testVoiture` c'est la méthode `demarre` correspondant à la classe `Voiture` et non à `VoitureDeSport` qui est appelée.

1.3.9 Définitions modulaires

Généralement plusieurs personnes travaillent simultanément sur un projet. Ainsi quand une personne p_1 écrit une classe voiture, pour qu'une personne p_2 s'en serve, il n'y pas nécessairement besoin de connaître exactement le contenu de chaque méthode mais seulement d'avoir une idée du squelette de la classe créée.

Imaginons que p_1 souhaite écrire une classe représentant une voiture, et que outre le constructeur la seule méthode sur cette classe renvoie dans un entier le prix de la voiture. Alors p_1 peut écrire le code relatif à cette voiture dans deux fichiers : un fichier d'entête (header file) se terminant par `.h`, et un fichier contenant le corps des méthodes, se terminant généralement par `.cc`, `.cxx` ou `.cpp`. Voici le contenu de `voiture.h`, on remarque que le corps des méthodes n'est pas présent, en revanche les champs de la classe sont présents :

```
// fichier voiture.h
class Voiture {
public:
        Voiture(int prix);
        int donnePrix() const;
private :
        int prixDeLaVoiture;
};
```

On remarque que le fichier `voiture.h` donne la définition de la structure de la classe : elle comporte un constructeur attendant un entier, une méthode, et un champ de type `int`. On remarque le qualificatif `const` après la méthode `donnePrix` : à cet endroit (après les arguments d'une méthode) cela signifie que l'appel méthode `donnePrix` ne modifie pas l'instance sur laquelle la méthode est appelée. La vérification qu'une méthode puisse bien être qualifiée par `const` est effectuée au moment de la compilation, permettant ainsi d'éviter des erreurs à l'exécution. Le corps du constructeur et de la méthode sont décrits dans le fichier `voiture.cc` :

```
// fichier voiture.cc
#include "voiture.h"  // inclure le fichier voiture.h

/**
 * Donne le corps du constructeur de la classe Voiture.
 */
Voiture::Voiture(int prix) {
        prixDeLaVoiture=prix;
}

/**
 * Donne le corps de la méthode donnePrix.
 */
int Voiture::donnePrix() const {
        return prixDeLaVoiture;
}
```

On remarque l'usage du symbole `::` qui signifie que l'on va déclarer un symbole se situant dans la classe `Voiture`. De manière générale cet opérateur permet d'accéder à un espace de nommage, ici celui définit par la classe `Voiture`.

Ainsi pour utiliser une librairie il suffit de donner le fichier d'entête, le fichier de définition n'étant pas nécessaire. Voici un usage de la classe voiture dans le fichier `main.cc` :

```
1  // fichier main.cc
2  #include "voiture.h"
3  int main() {
4          Voiture maVoiture(123);
5          return 0;
6  }
```

Le fichier `main.cc` constitue une unité de compilation, même si les corps des méthodes de la classe `Voiture` ne sont pas connues. On peut donc le compiler séparément du reste du code. Le fichier `voiture.cc` constitue également une unité de compilation.

Nous avons vu comment partager des définitions de classes. Il peut également être utile de partager des instanciations de classes ou des variables sur des types entiers. Ainsi si dans une unité de compilation on dispose d'une variable globale `i` :

```
1  // fichier unite1.cc
2  int i=4;
```

pour référencer cette variable depuis une autre unité de compilation il suffit de rajouter une déclaration sans valeur initiale, précédée de l'attribut `extern` :

```
1  // fichier unite2.cc
2  extern int i;
3  int multiplicationParI(int j) {
4          return i*j;
5  }
```

Si le mot clé `extern` ne figure pas dans `unite2.cc`, la compilation se fera sans problèmes mais c'est à l'édition de liens que le compilateur va rencontrer un symbole dupliqué. Lire la fiche pratique 1 sur la compilation pour plus de détails sur ce qu'est l'édition de liens.

1.3.10 Des données bien ordonnées

Tout comme il n'est pas pratique de mettre tous ses fichiers dans le même répertoire, il est bon de ne pas mettre toutes ses fonctions et ses classes dans le même espace de nom. Ainsi par exemple pour écrire une librairie nommée communication dont le but est la communication entre un client et un serveur, il est logique de mettre les fichiers C++ dans un répertoire nommé `communication`. Prenons alors l'exemple déclarant la classe `Client` :

```
1  class Client {
2          // .. definition
3  };
```

Il est possible que le nom de la classe `Client` définit dans la librairie puisse entrer en conflit avec une autre classe qui serait déjà nommée `Client` ailleurs dans le programme. Ainsi il est préférable (mais en aucun cas nécessaire) de déclarer la classe client dans un nouvel espace de nommage (`namespace` en C++).

```
1  namespace Communication {
2          class Client {
3                  // .. definition
4          };
5  }
```

Dans le programme on peut alors créer une instance de la classe client comme suit :

```
6  Communication::Client monClient ();
```

ou si l'on veut se débarrasser du préfixe `Communication::` de manière similaire :

```
6   using namespace Communication; // importe tous les noms situés dans Communication
7   Client nomClient();
```

Un espace de nommage où de nombreuses librairies sont présentes existe déjà : `std`. Par exemple le programme suivant affiche bonjour à l'écran :

```
1   #include <iostream>  // demande d'inclure les définitions pour l'affichage
2   using namespace std; // utilise les noms de l'espace de nommage std
3   int main() {
4           cout << "Bonjour!" << endl;
5           return 0;
6   }
```

Plus généralement l'espace de nommage `std` contient l'ensemble des bibliothèques du standard C++.
Voici le programme équivalent sans l'usage de la directive `using` :

```
1   #include <iostream>  // demande d'inclure les définitions pour l'affichage
2   int main() {
3           std::cout << "Bonjour!" << std::endl;
4           return 0;
5   }
```

On peut également obtenir les bornes sur les entiers présentés à la section 1.3.1 qui dépendent de la machine à l'aide du programme suivant :

```
1   #include <limits>
2   #include <iostream>
3   using namespace std;
4   int main() {
5     cout << "plus grand entier " << numeric_limits<int>::max() << endl;
6     cout << "plus petit entier " << numeric_limits<int>::min() << endl;
7     return 0;
8   }
```

Sur une machine 32 bits ce programme affiche :

```
plus grand entier 2147483647
plus petit entier -2147483648
```

1.3.11 Autres primitives d'entrées/sorties

Nous venons de voir comment afficher des données à l'écran. Voici plus de détails sur les entrées/sorties : dans des fichiers, dans des chaînes de caractères, ou pour des classes.

Voici comment écrire la chaîne de caractères `Bonjour` dans un fichier nommé `montexte` :

```
1   #include <fstream>
2   #include <iostream>
3   using namespace std;
4   int main() {
5           ofstream f ("montexte");  // ofstream = output file stream
6           if (f.is_open()) {
7                   // le fichier montexte a bien été ouvert en écriture
8                   f << "Bonjour" << endl;
9                   f.close();
10                  return 0;
11          }
12          // le fichier montexte n'a pu être ouvert en écriture
13          cerr << "impossible d'ouvrir le fichier 'montexte'" << endl;
14          return 1;
15  }
```

On créé une instance de `ofstream` nommée `f`. Si la création du fichier `montexte` est possible alors la méthode `is_open` renvoie `true`, on peut alors écrire dans le fichier comme s'il s'agissait de la sortie standard. Une fois l'écriture terminée on peut refermer le fichier à l'aide de la méthode `close`. La variable `cerr` est le flux d'erreur standard dans lequel un processus peut écrire les messages d'erreur le concernant.

On peut définir l'opérateur << sur des classes pour pouvoir directement les imprimer dans un fichier ou sur `cout`. Prenons l'exemple d'une classe représentant un point par un couple d'entiers :

```
0   class Point {
1   public :
2           Point(int _x, int _y) {
3                   x=_x; y=_y;
4           }
5           int getX() const { return x;}
6           int getY() const { return y;}
7   private:
8           int x,y;
9   };
```

Ajoutons alors la définition de l'opérateur << :

```
10  ostream & operator<< ( ostream & os, const Point & p ) {
11          os << "(" << p.getX() << "," << p.getY() << ")" << endl;
12          return os;
13  }
```

Les opérateurs sont définis plus en détails dans le chapitre suivant. Le symbole `&` constitue un passage par référence et est détaillé dans le chapitre suivant. L'attribut `const` signifie ici (avant le type d'un argument) que l'argument `p` ne va pas être modifié par l'appel à l'opérateur `operator<<`. En particulier, cela impose que seules des méthodes de `Point` qualifiées par `const` peuvent être appelées dans l'opérateur `operator<<`. Alors pour afficher les coordonnées d'un point on peut se servir du code ci-dessous :

```
14  #include <iostream>
15  using namespace std;
16  int main() {
17          Point p (12,13);
18          cout << p << endl;  // appel à l'opérateur << définit précédemment
19          return 0;
20  }
```

1.3.12 Structures de contrôle

On nomme structure de contrôle l'ensemble des opérations qui permettent de choisir si des instructions doivent être exécutées ou non.

Test

La structure la plus simple est le test. Par exemple voici une fonction qui indique si son argument est égal à 4 :

```
1   #include <iostream>  // demande d'inclure la définition
2                        // des primitives d'entrée sortie
3   using namespace std; // utilise l'espace de nommage std
4
5   void afficheSiEgalA4 (int i ) {
6           if (i==4) {
7                   cout << "i vaut 4" << endl;
8           } else {
9                   cout << "i ne vaut pas 4" << endl;
10          }
11  }
```

La syntaxe des tests est la suivante :

`if (` *<condition>* `)` *<bloc exécuté si condition vraie>*[`else` *<bloc exécuté si condition fausse>*]

La partie avec le `else` est optionnelle.

Boucles for

Une autre structure fréquente sont les boucles. Voici une façon d'initialiser tous les éléments d'un tableau à 0 :

```
const int tailleMax = 30000; // nombre d'éléments dans le tableau
int monTableau[tailleMax];   // création d'un tableau à tailleMax éléments
int main() {
        // pour i variant de 0 à tailleMax-1 de un en un effectuer monTableau[i]=0
        for (int i=0; i<tailleMax; i=i+1) {
                monTableau[i]=0;
        }
        return 0;
}
```

La structure des boucles `for` est la suivante :

`for (` *<initialisation>* `;` *<condition de fin >* `;` *<action à faire à la fin de chaque boucle>* `)`

Le corps de la boucle est ensuite déclaré dans des accolades.

Boucles while

Une autre syntaxe consiste à utiliser les boucles `while` :

```
const int tailleMax = 30000;
int monTableau[tailleMax];
int main() {
        int i=0;
        while (i<tailleMax) {
                monTableau[i]=0;
                i=i+1;
        }
        return 0;
}
```

La structure des boucles `while` est la suivante :

`while (` *<condition pour effectuer la boucle >* `)` { *<corps de la boucle >*}

Le corps de la boucle est ensuite déclaré dans ses accolades.

Boucles do while

Il existe également des boucles `do ... while`, où la condition de sortie de la boucle n'est évaluée qu'à la fin :

```
const int tailleMax = 30000;
int monTableau[tailleMax];
int main() {
        int i=0;
        do {
                monTableau[i]=0;
                i=i+1;
        } while (i<=tailleMax);
        return 0;
}
```

La structure des boucles `do ... while` est la suivante :

`do` { *<corps de la boucle >*} `while (` *<condition pour effectuer la boucle >* `);`

1.3.13 Le préprocesseur

Il se peut que l'on ait à écrire des bouts de codes répétitifs. On souhaiterais donner un nom à ces séquences de caractères et invoquer le nom pour ne pas avoir à retaper les mêmes séquences à chaque fois. Le préprocesseur fait ce travail. Voici un exemple indiquant d'une fonction provoquant une erreur quand son argument est négatif :

```
#include <iostream>
#define ERREUR std::cerr << "Il y a une erreur" << std::endl; exit(1)
int doubleValeurSiPositif(int i) {
        if (i<0) {
                ERREUR;
        }
        return 2*i;
}
```

A chaque fois que l'on écrit `ERREUR` la séquence `std::cerr ... exit(1)` est écrite à la place puis donnée au compilateur. Voici la syntaxe de la directive `#define` :

`#define` *<nom à définir><valeur>*

Le nom définit à l'aide de la commande define est nommé macro. Le remplacement des macros par leur valeur est purement syntaxique, il s'agit simplement de caractères remplacés les uns par les autres, avant la phase de compilation, en particulier il n'y a pas de notion de type.

On souhaite passer un argument donnant plus d'information sur l'erreur. On peut encore le faire à l'aide des macros, en lui donnant un argument :

```
#include <iostream>
#define ERREUR(x) std::cerr << "Il y a une erreur : "  << x << std::endl; exit(1)
int doubleValeurSiPositif(int i) {
        if (i<0) {
                ERREUR("i est négatif");
        }
        return 2*i;
}
```

En déclarant `ERREUR(x)` partout où l'identifiant `x` apparaît la valeur de l'argument sera substituée. Il existe également des variables particulières qui donnent le nom du fichier en cours `__FILE__` et le numéro de la ligne en cours `__LINE__`. Ainsi pour facilement retrouver l'endroit dans le code où l'erreur s'est produite on peut rajouter :

```
#include <iostream>
#define ERREUR(x) std::cerr << __FILE__ << ":" << __LINE__ << ": erreur : " \
                            << x << std::endl; exit(1)
int doubleValeurSiPositif(int i) {
        if (i<0) {
                ERREUR("i est négatif");
        }
        return 2*i;
}
```

Noter la présence du caractère \ permettant d'écrire la macro sur plusieurs lignes. A l'exécution en cas d'erreur on obtiendra un affichage de la forme :

```
toto.cc:6: erreur : i est négatif
```

Nous verrons au paragraphe 4.1.2 le mécanisme d'exception permettant de gérer les cas d'erreur de manière plus sémantique.

1.4 Mise en oeuvre dans SystemC

1.4.1 Qu'est-ce que SystemC

SystemC est un ensemble de classes C++ permettant de décrire du matériel ou du logiciel : des microprocesseurs avec un programme fonctionnant dessus, un bus avec des périphériques communiquant.

Le site web permettant d'obtenir la documentation et les logiciels est `http://www.systemc.org`. On y trouve en particulier le manuel de référence du langage [osc05], rédigé par l'Open SystemC Initiative (OSCI). Le noyau dur des membres de l'Open SystemC Initiative (OSCI) sont Arm, Cadence, CoWare, Forte, Mentor Graphics, STMicroelectronics, Synopsys, Philips.

1.4.2 Pourquoi SystemC ?

Au début de l'électronique il y avait de nombreux composants différents sur une carte : des transistors, des condensateurs, des résistances. Les composants n'ont cessés d'être plus compacts et plus intégrés. Actuellement, la tendance est même d'avoir plusieurs cœurs de microprocesseurs connectés entre eux dans un seul composant (c'est le type d'architecture proposé en exercice au chapitre 4). On trouve de plus des circuits permettant d'offrir de nouvelles fonctionnalités pour une consommation réduite : encodeur/décodeur MPEG4 pour la vidéo par exemple. Ainsi pour faire un nouveau téléphone portable supportant la vidéo il faut prendre un microprocesseur qui possède déjà cette fonctionnalité.

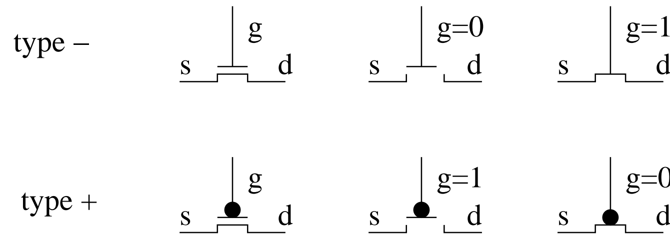
La conception de matériel est de plus en plus dans une logique d'assemblage, où un composant est constitué d'un microprocesseur et de modules (comme l'encodeur/décodeur MPEG4) relié par un bus ou un réseau sur puce. Cette conception spécifique est amortie par les volumes importants qui vont être produits.

Cette phase d'assemblage prend du temps. Il peut y avoir des difficultés à assembler les différents composants. Ces difficultés doivent être repérées le plus tôt possible. C'est là où les modèles écrits en SystemC interviennent. On réalise tout d'abord un modèle SystemC relativement haut niveau des composants à intégrer. Ce modèle ne comportant pas tous les détails au niveau des transistors est relativement rapide à simuler. On peut alors dès ce stade avancé de la conception étudier les performances du système, ou observer si le comportement est correct.

1.4.3 Exemples d'abstractions utilisées

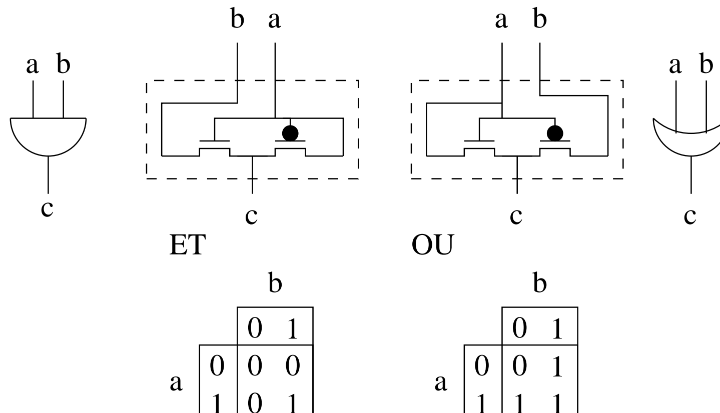
Le microprocesseur est uniquement constitué de signaux électriques. On considère au dessus d'un certain voltage que la valeur du signal représente le nombre 1, et en dessous elle représente le nombre 0.

La brique de base est le transistor, que l'on peut voir ici comme un petit interrupteur. Voici les deux types de transistors - et + :



Dans le transistor de type - le courant passe entre s et d quand g vaut 1. C'est l'inverse dans le transistor +. Un microprocesseur Pentium 4 contient environ 55 millions de transistors de ce type. La taille d'un tel transistor de type MOSFET (Metal Oxide Semiconductor Field Effect Transistor) est d'environ 150 nano mètre. Le temps de basculement d'un transistor est fonction de sa taille : plus le transistor est petit plus le temps de basculement est petit. Ainsi plus on souhaite avoir une fréquence d'horloge élevée plus il faut des transistors petits. Pour plus de détails sur ce sujet se reporter à l'ouvrage Basic VLSI Design [PE94].

On abstrait rapidement les transistors en utilisant des portes logiques ET (c vaut 1 si et seulement si a et b valent 1) et OU (c vaut 1 si et seulement si a ou b vaut 1) :



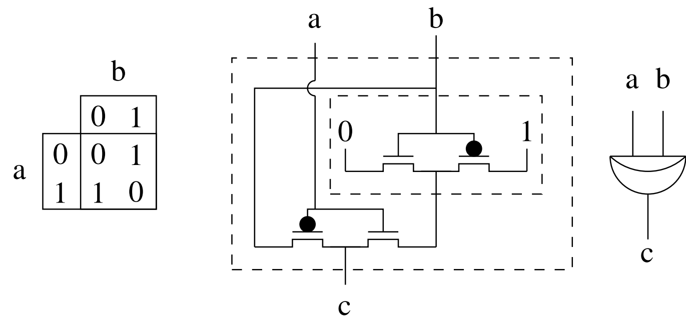
ET :

a \ b	0	1
0	0	0
1	0	1

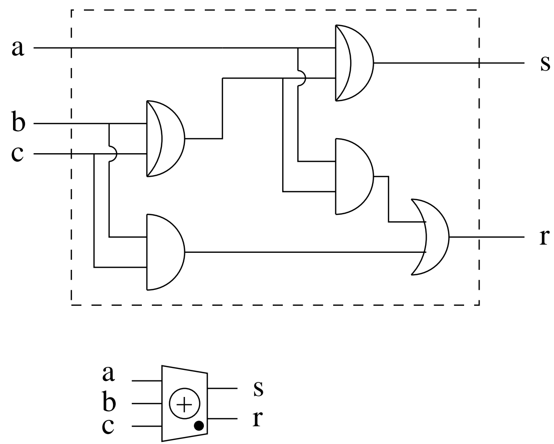
OU :

a \ b	0	1
0	0	1
1	1	1

Voici également XOR : le OU exclusif (c vaut 1 si et seulement si a ou b vaut 1, mais pas les deux à la fois) :



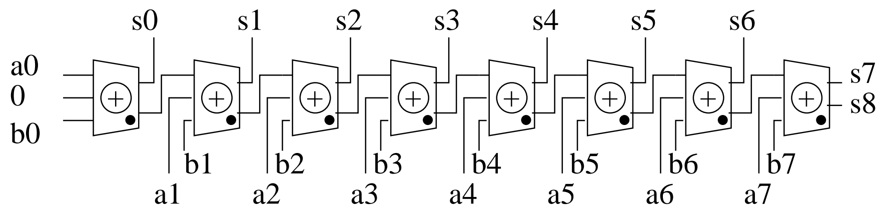
L'addition de trois chiffres binaires *a*, *b* et *c* est dès lors possible. Le résultat varie de 0 à 3, soit comme valeur possible en écriture binaire : 0,1, 10 (pour le nombre décimal 2) et 11 (pour le nombre décimal 3). Le chiffre des unités est noté *s* comme somme et l'autre chiffre est noté *r* comme retenue. On peut alors calculer *s* et *r* à l'aide du circuit suivant qui est un assemblage des portes précédentes :



Ce circuit est appelé additionneur complet (full adder en anglais). On le note avec le plus entouré d'un rond et un point noir pour indiquer l'emplacement de la sortie donnant la retenue.

1.4.4 Additionneur sur 8 bits

En se servant de l'additionneur complet on peut alors écrire un circuit additionnant deux nombres *a* et *b* respectivement d'écriture binaire sur 8 bits *a*7, *a*6, *a*5, *a*4, *a*3, *a*2, *a*1, *a*0 et *b*7, *b*6, *b*5, *b*4, *b*3, *b*2, *b*1, *b*0 :

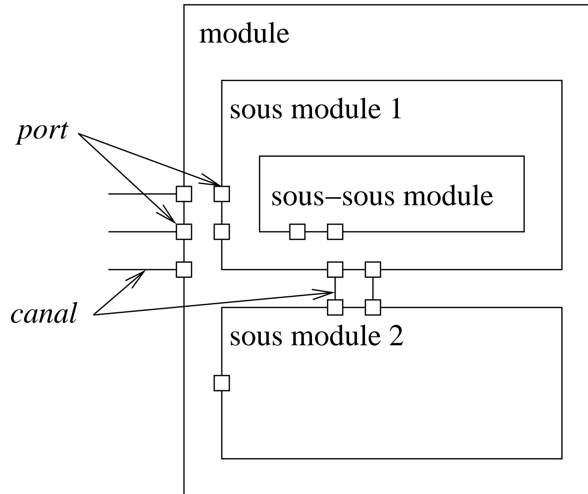


Le résultat *s*8, *s*7, *s*6, *s*5, *s*4, *s*3, *s*2, *s*1, *s*0 est lui sur 9 bits.

Comme précédemment mentionné les transistors ont un petit temps de basculement, que l'on souhaite minimiser pour pouvoir augmenter la fréquence de l'horloge. Ici on a une chaîne de huit additionneurs complets mis en série. Il faut donc attendre que les huit additionneurs se stabilisent, chacun attendant le précédent. Ainsi plus les nombres seront grand plus l'addition prendra du temps. On peut se débrouiller pour avoir un nombre d'additionneurs complets en série logarithmique en la taille des nombres additionnés. La réalisation d'un tel additionneur est proposée dans l'exercice 1.6.

1.4.5 Structure générale des modèles SystemC

En SystemC la notion fondamentale est celle de module. Elle est relativement similaire aux boites représentées dans les exemples précédentes. En pratique un module va être une classe C++ héritant d'une classe nommée `sc_module` et pouvant faire pratiquement n'importe quoi.



Un module va ensuite comporter des ports sur lesquels vous pouvoir venir se greffer des canaux. Dans l'exemple ci-dessus on a un module de haut niveau contenant deux sous modules et trois ports d'entrée.

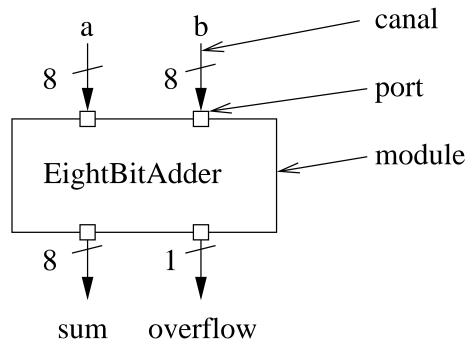
On distingue principalement trois types de ports d'entrée :

– `sc_in<T>` ports en entrée dans un module, véhiculant des données de type `T`,
– `sc_out<T>` ports en sortie dans un module, véhiculant des données de type `T`,
– `sc_inout<T>` port en entrée et sortie dans un module , véhiculant de données de type `T`.

Il existe de nombreux autres types de ports qui seront détaillés dans les chapitres ultérieurs.

1.4.6 Exemple de code SystemC

Nous allons écrire le code correspondant à un additionneur 8 bits comme présenté précédemment. Les modèles SystemC se situent généralement un niveau au dessus de la modélisation avec les additionneurs complets : l'additionneur sur 8 bits peut être modélisé par un module ayant deux entrées et deux sorties : une sortie sur 8 bits donnant le résultat et un bit supplémentaire nommé overflow indiquant s'il y a eu dépassement de la capacité :



Voici le code C++ correspondant dans le fichier `eightbitadder.h` :

```
1   #include "systemc.h"   // inclut les définitions SystemC
2   typedef unsigned char u8; // définit 'u8' comme un type char non signé
3
4   SC_MODULE(EightBitAdder) {  // définit un module nommé EightBitAdder
5     // liste des ports
6     sc_in<u8> a,b;      // spécifie que le module possède deux ports d'entrée
7     sc_out<u8> sum;     // le module possède un port de sortie sur 8 bits
8     sc_out<bool> overFlow;  // le module possède un port de sortie sur 1 bit
9
10    // définition des actions à effectuer
11    void prcEightBitAdder() {
12      int intSum = a;      // calcul de la somme sur un entier
13      intSum=intSum+b;
```

```
14        sum = intSum & 255;          // calcul de la somme sur 8 bits
15        overFlow = ((intSum & 256) != 0);  // calcul de l'overflow
16      }
17
18      // constructeur du module
19      SC_CTOR (EightBitAdder) {
20        // spécifie que la méthode prcEightBitAdder
21        // est sensible aux signaux a et b
22        SC_METHOD (prcEightBitAdder);
23        sensitive << a << b;
24      }
25    };
```

Le programme commence par importer les définitions nécessaires à l'usage des classes SystemC à l'aide de la directive `#include "systemc.h"`.

Le type `unsigned char` est ensuite nommé `u8` à l'aide de la commande `typedef`. Nous pouvons donner de nouveaux noms à des types. Par exemple ci dessous au lieu de réécrire à chaque fois `Toto::MaClasse` nous l'abrégeons en `Tata` :

```
1   namespace Toto {
2           class MaClasse {} ;
3   }
4   typedef Toto::MaClasse Tata;
5   Tata maVariable;
```

Voici la syntaxe de `typedef` :

`typedef` *<nom de type><nom supplémentaire pour le même type>* ;

La syntaxe `SC_MODULE(EightBitAdder)` déclare un module SystemC, c'est une boite sur laquelle nous pourrons faire des entrées et des sorties. En pratique cette notation est équivalente à créer une classe qui hérite de `sc_module` dont tous les champs ou toutes les méthodes sont publiques :

```
class EightBitAdder : sc_module {
public:
        // définition du module
        ...
};
```

Ceci est rendu possible grâce aux macros C++. Ainsi en définissant la macro `SC_MODULE` avec l'argument `x` par :

```
#define SC_MODULE(x) class x : sc_module
```

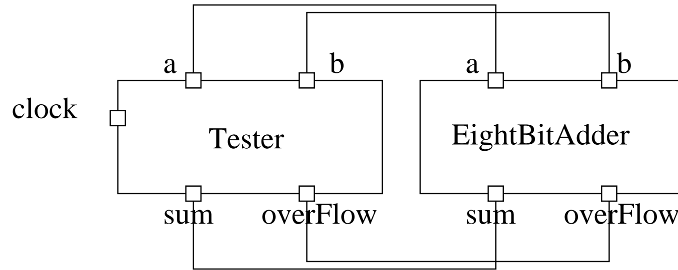
La valeur exacte de la macro `SC_MODULE` est donnée à la section 4.2.1. Dans le corps du module on remarque 4 champs : des champs de la forme `sc_in` définissant les ports d'entrée dans le module, des champs de la forme `sc_out` définissant les ports de sortie du module. Il s'agit de classes template qui sont, dans l'exemple, instanciées avec le type de données en entrée ou en sortie.

Les modules SystemC étant des classes, on trouve bien entendu le constructeur. C'est ce qui est fait à la ligne 19 par la déclaration `SC_CTOR (EightBitAdder)`. `SC_CTOR` est en fait une macro (tout comme `SC_MODULE`) permettant de déclarer un constructeur prenant en argument un chaîne de caractères.

Dans le corps du constructeur est ensuite spécifié qu'une méthode SystemC est présente, par la macro `SC_METHOD (prcEightBitAdder)`. La déclaration `sensitive << a << b` indique au moteur de simulation SystemC qu'il faudra exécuter la dernière méthode enregistrée (en l'occurrence `prcEightBitAdder`) si une nouvelle donnée arrive sur les ports `a` ou `b`. Sur un tel évènement la méthode `prcEightBitAdder` est exécutée jusqu'au bout avant que l'ordonnanceur SystemC ne choisisse la prochaine méthode à appeler.

1.4.7 Tester le module écrit

Maintenant que nous avons un module `EightBitAdder` nous souhaiterions tester son bon fonctionnement. Ci-dessous figure le dispositif de test utilisé :



Nous utilisons un module nommé `Tester` qui envoie les données à additionner, puis récupère le résultat et vérifie le bon fonctionnement. Voici le code implémentant ce mécanisme :

```
1  #include "eightbitadder.h"
2
3  SC_MODULE(Tester) {   // définition du module Tester
4    // ports
5    sc_out<u8> a,b;
6    sc_in<u8> sum;
7    sc_in<bool> overFlow;
8    sc_in_clk clock;
9    // conserve les valeurs fournies à l'additionneur
10   int aValue, bValue;
11   bool firstTime;
12   void sendInput() {
13     if (firstTime) { // envoyer zéro la première fois
14       firstTime=false;
15       aValue=0;bValue=0;
16       a=0;b=0;
17       return;
18     }
19     aValue = rand() & 255;  // génère une valeur aléatoire pour a
20     bValue = rand() & 255;  // génère une valeur aléatoire pour b
21     a.write(aValue);  // écrit la valeur aléatoire sur le port a
22     b.write(bValue);  // écrit la valeur aléatoire sur le port b
23     cout << "Sending " << aValue << " and " << bValue << endl;
24   }
25   void checkOutput() {
26     int sumValueRead = sum.read();
27     int overFlowValueRead = overFlow.read();
28     if (aValue+bValue!= (sumValueRead + (overFlowValueRead?256:0))) {
29       cout << "Error : sent " << aValue << " and " << bValue
30            << " but received " << (sumValueRead + (overFlowValueRead?256:0)) << endl;
31     }
32   }
33   SC_CTOR(Tester) {
34     srand(0);               // initialisation du générateur aléatoire
35     firstTime=true;
36     SC_METHOD(sendInput);
37     sensitive << clock;
38     SC_METHOD(checkOutput);
39     sensitive << sum << overFlow;
40   }
41 };
```

On retrouve sur le modèle de `Tester` le pendant les ports sur `EightBitAdder` et le port pour l'horloge `clock`. L'horloge déclenche la méthode `sendInput`. On remarque l'usage de la fonction `rand` pour la génération de nombres aléatoires. C'est ici raisonnable car on ne cherche pas à disposer de nombre aléatoires d'une bonne qualité, se reporter à [PFTV92] ou [Knu81] pour disposer de nombre aléatoire d'une bonne qualité. Nous pouvons alors lancer la simulation en rajoutant le code suivant :

```
42  int sc_main(int argc, char** argv) {
43    // les 4 signaux permettant de relier les modules tester et eightBitAdder
44    sc_signal<u8> testerAOut, testerBOut;
45    sc_signal<u8> testerSumIn;
46    sc_signal<bool> testerOverIn;
47    // créer une horloge avec une période de 1ns  donnant au tester les impulsions
48    // pour envoyer de nouveaux échantillons
49    sc_clock clock("clockForTester",sc_time(1,SC_NS));
50    // instancier les modules tester et eightBitAdder
51    Tester tester ("tester");
52    EightBitAdder eightBitAdder ("eightBitAdder");
53    // lier les ports aux signaux.
54    tester.a(testerAOut);
55    eightBitAdder.a(testerAOut);
56    tester.b(testerBOut);
57    eightBitAdder.b(testerBOut);
58    tester.sum(testerSumIn);
59    eightBitAdder.sum(testerSumIn);
60    tester.overFlow(testerOverIn);
61    eightBitAdder.overFlow(testerOverIn);
62    tester.clock(clock);
63    // lancer la simulation pour 100 nano secondes
64    sc_start(sc_time(100,SC_NS));
65    // tout s'est bien passé renvoyer zéro.
66    return 0;
67  }
```

Dans la fonction `sc_main`, qui est l'équivalent de la fonction `main` dans un programme C++, figure les quatre signaux permettant de relier les modules, ainsi que la définition de l'horloge. Les arguments de la fonction `sc_main` sont détaillés ultérieurement à la section 2.2.5.

L'instruction `sc_start` lance la simulation. Nous n'avons pas jusqu'à présent parlé de la sémantique d'exécution de SystemC, mais c'est ce qui en fait sont principal intérêt : les modèles sont exécutables et on peut voir leur comportements. Nous ne détaillons que sommairement la sémantique d'exécution de SystemC dans ce chapitre.

Voici comment compiler le modèle SystemC :

```
g++ -o eightbitadder.exe -I$SC_HOME/include eightbitadderwrapper.cc \
    -L$SC_HOME/lib-linux -lsystemc
```

On suppose que `$SC_HOME` est une variable d'environnement de l'interpréteur donnant le répertoire dans lequel SystemC est installé. L'instruction de compilation signifie que le fichier généré doit être `eightbitadder.exe`, que le répertoire `$SC_HOME/include` va être utilisé pour lire des fichiers d'entête, que le répertoire `$SC_HOME/lib-linux` va être utilisé pour lire les librairies, et que la librairie `SystemC` va être liée à l'exécutable. Voici le résultat de l'exécution :

```
             SystemC 2.1_oct_12_04.beta --- Jun 21 2005 16:28:31
        Copyright (c) 1996-2004 by all Contributors
                    ALL RIGHTS RESERVED
Sending 103 and 198
Sending 105 and 115
Sending 81 and 255
...
```

Aucune erreur n'est affichée, les tests se sont bien déroulés.

1.5 Conclusion

Nous avons sommairement décrit les contraintes nécessaires à la bonne réalisation d'un projet. Nous avons détaillé les constructions de base de C++ permettant de répondre à certains besoins imposés par la gestion de projets.

Enfin nous avons décrit le positionnement du langage SystemC et décrit comment le langage C++ permet facilement de faire un nouveau dialecte pour décrire, entre autre, du matériel.

1.6 Exercices

Ex 1.1 * Hello World !

Ecrire un programme affichant « `Hello World !` », et l'exécuter. Se reporter à la fiche pratique `Compilation C++` à la page 99 pour les détails permettant d'exécuter le programme.

Ex 1.2 * Inclusion de .h

On inclut les fichiers à l'aide de la directive `#include "` *nom de fichier* `"`. On considère les deux fichiers suivants d'une ligne seulement :

```
**** f1.h : #include "f2.h"
**** f2.h : #include "f1.h"
```

Q1 Essayer de compiler le fichier main.cc suivant :

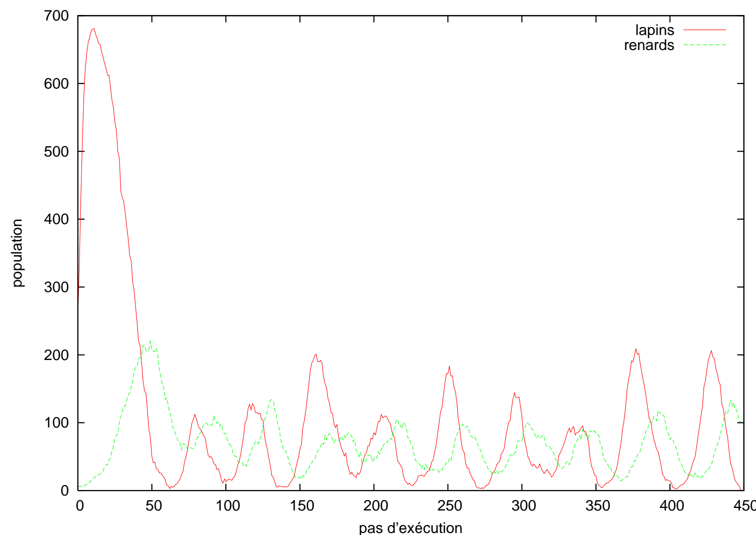
```
#include "f1.h"
```

Q2 Pour remédier à l'erreur précédente mettre en place un mécanisme permettant de n'inclure qu'une fois le fichier `f1.h` à l'aide des directives suivantes :

- `#define TOTO` définit le symbole `TOTO`
- `#ifndef TOTO`
  `// code C++`
  `#endif` où le code C++ n'est exécuté que si le symbole `TOTO` n'est pas définit.

Ex 1.3 ** Simulation d'écosystème

On s'intéresse à modéliser un écosystème contenant des lapins et des renards. Ci-dessous figure un exemple de courbe obtenue donnant les variations de populations. Après une phase initiale transitoire, on observe un rythme régulier : la population de lapins croit, cela permet alors aux renards de se développer, mais ils consomment alors trop de lapins, la population de lapins décroît alors, il n'y a donc plus assez de lapins et la population de renards décroît alors à son tour.



On se donne une classe générique nommée `Animal` définie ci-dessous :

```
1  #include <string>
2
3  using namespace std;
4
5  class Animal {
6  public :
7      Animal() { x=0; y=0; energie=0; energieRepro=100;}
8      Animal(int x, int y) { fixePosition(x,y); }
```

```
9        void fixePosition(int xx,int yy) { x=xx; y=yy; }
10       void energieAjoute(int e) { energie=energie+e; }
11       int donneEnergie() { return energie; }
12       int lireX() { return x;}
13       int lireY() { return y;}
14       void initEnergieRepro(int i) { energieRepro=i; energie=0;}
15       int donneEnergieRepro() { return energieRepro; }
16       void tue() { energie=-100;}
17       virtual int valeurEnergetique() { return 0;}
18       virtual string donneLettre() { return "A";}
19   private :
20       int x;
21       int y;
22       int energie;
23       int energieRepro;
24   };
```

Q1 Dans un fichier nommé `lapin.h` écrire une classe nommée `Lapin` héritant de `Animal` pour laquelle la méthode `valeurEnergetique` renvoie 5 et `donneLettre` renvoie `"L"`. Faire de même dans un fichier `renard.h` avec une classe `Renard` pour laquelle les valeurs renvoyées sont 10 et `"R"`. Le code source de la class `Animal` est disponible à l'url suivant :

`http://www.derepas.com/csml/ex_eco_download.tar.gz`

Q2 Dans l'archive précédente se trouve définie la classe `Monde` dans les fichiers `monde.h` et `monde.cc`. Ecrire un fichier nommé `simule.cc` contenant une procédure nommée `main` et effectuant les actions suivantes :
- Créer une instance de la classe `Monde` nommée `m`, sur un carré de 30×30 comportant 150 lapins et 5 renards à l'aide de l'instruction `Monde m (30,150,5)`.
- Tant que `m.finished()` est faux effectuer un appel à la méthode `m.cycleSuivant()`. Afficher le résultat à l'aide de la commande `cout << m << endl`

La solution de l'exercice est disponible à l'url suivant :

`http://www.derepas.com/csml/ex_eco_solution.tar.gz`

Ex 1.4 *** Addition en temps logarithmique

Le but de cet exercice est de prouver la correction de l'additionneur en temps logarithmique évoqué à la section 1.4.4.

Q1 Considérons deux nombres A et B ayant une écriture binaire de taille 2^n. On note $S = A + B$ et $S' = A + B + 1$. On découpe A et B en $A = A_0 + 2^n A_1$ et $B = B_0 + 2^n B_1$, avec $A_0, A_1, B_0, B_1 < 2^n$. On note $S_0 = A_0 + B_0$, $S'_0 = A_0 + B_0 + 1$, $S_1 = A_1 + B_1$ et $S'_1 = A_1 + B_1 + 1$. Calculer S et S' à l'aide de l'écriture de S_0, S'_0, S_1 et S'_1.

Q2 A l'aide de transistors plus et moins, créer un bloc à trois entrées sur un bit : a, b, c et une sortie s, qui vaut b si a vaut zéro et c si a vaut 1.

Q3 Utiliser le bloc de la question 2 pour implémenter sur silicium l'algorithme 1 réalisant un additionneur sur 8 bits en temps logarithmique.

Q4 Implémeter l'additionneur en temps logarithmique en SystemC.

Ex 1.5 *** Méta programmation

Le but de l'exercice est de prouver que 13 est premier en utilisant le mécanisme de typage des templates de C++. L'exercice se base sur un programme écrit par Dan Piponi (`http://www.sigfpe.com`).

Précisons tout d'abord une construction qui n'a pas été présentée dans le cours : pour définir une classe dont tous les champs sont publics, on peut utiliser le mot clé `struct` en lieu et place de `class`.

Pour définir les entiers nous allons utiliser les axiomes de Péano : nous disposons d'un ensemble de valeurs nommé *Naturel*. Une de ces valeurs est nommée O (la lettre o en majuscule), elle représente le nombre zéro. Nous disposons d'une opération nommée successeur notée $S : Naturel \rightarrow Naturel$. Ainsi le nombre 1 sera représenté par $S(O)$, 2 par $S(S(O))$ et ainsi de suite.

On se fixe les définitions suivantes :

```
1   template<class T> struct Naturel { typedef T valeur; };
2   // définition de zéro
3   struct zero : public Naturel<zero> { };
4   // définition d'un successeur et d'un prédécesseur d'un entier
```

```
5   template<class C> struct S
6             : public Naturel<S<C> > { typedef C predecesseur; };
```

Q1 En utilisant un `typedef` définir les types un, deux, trois, quatre, cinq, six, sept huit, neuf et dix représentants les entiers de valeur correspondante.

Nous pouvons définir l'addition par les axiomes suivants :

- pour tout nombre C, $C + O = C$,
- si un nombre D a un prédécesseur D' alors $C + D = S(C + D')$.

Nous pouvons écrire ces deux axiomes en utilisant le système de typage de C++ :

```
7   template<class C,class D> struct plus
8             : public S<plus<C,typename D::predecesseur> > { };
9   template<class C> struct plus<C,zero>
10            : public C { };
```

On remarque l'usage du qualificatif `typename` pour préciser que `D::predecesseur` définit bien un type.

Q2 Voici la définition de la soustraction :

- pour tout entier C, $C - 0 = C$,
- si un nombre D a un prédécesseur D' alors $C - D$ est le prédécesseur de $C - D'$.

Ecrire les types correspondant à cette définition.

Q3 Voici la définition de la multiplication :

- pour tout entier C, $C \times 0 = 0$
- si un nombre D a un prédécesseur D' alors $C \times D = C + C \times D'$.

Ecrire les types correspondant à cette définition.

Q4 Voici les deux axiomes définissant l'opération « plus grand ou égal à » :

- si un nombre C a un prédécesseur C' et si un nombre D a un prédécesseur D' alors $C \geq D$ est équivalent à $C' \geq D'$.
- pour tout entier C différent de zero $C \geq 0$ est équivalent au type `un` c'est à dire `S<zero>`, (nous coderons la valeur de vérité booléenne par `un`).
- pour tout entier C différent de zero $0 \geq C$ est équivalent au type `zero`.
- $0 \geq 0$ est équivalent au type `un`.

Ecrire les types nommés `ge` correspondant à cette définition.

Q5 Dans cette question nous allons mettre en place un test logique pour savoir si un élément est divisible par un autre. L'expression booléenne équivalente à D divise C sera représentée par le type `Divise<C,D>`. Nous introduisons un troisième argument par défaut égal à deux, que l'on va remplacer par la valeur booléenne de $C \geq D$:

```
11  template<class C,class D,class E = S<S<zero> > > struct Divisible { };
12
13  template<class C,class D> struct Divisible<C,D,S<S<zero> > >
14            : public Divisible<C,D,typename ge<C,D>::valeur> { };
```

Ainsi le type `Divise<C,D>`, est équivalent au type `Divise<C,D,deux>` qui hérite de `Divisible<C, D, typename ge<C,D>::valeur>` .

En se servant du fait que si $C < D$ alors D divise C si et seulement si C vaut zéro, écrire les valeurs de `Divisible<C, D, zero>`.

En se servant du fait que si $C \geq D$ alors D divise C si et seulement si D divises $C - D$, écrire les valeurs de `Divisible<C, D, un>`.

Q6 Ecrire un type nommé `Premier` vérifiant si un entier naturel est premier en utilisant un algorithme simple : il est premier si et seulement si il ne peut être divisé par nombre supérieur ou égal à 2 plus petit que lui. Tout comme nous avons réussi à faire un test à la question précédente, il faut ici faire une boucle.

Pour vérifier que 13 est premier il suffit alors de définir la valeur à partir de l'écriture décimale :

```
15  template<class C,class D> struct Decimal
16            : public plus<typename fois<dix,C>::valeur,D> { };
```

Et d'exécuter les instructions suivantes :

```
17  #include <string>
18  #include <iostream>
19  using namespace std;
```

```
20  template<class C> string output(C);
21  template<> string output(zero) { return "Non"; }
22  template<> string output(un) { return "Oui"; }
23
24  int main() {
25          // Est-ce que 13 est premier ?
26          cout << output(Premier<Decimal<un,trois>::valeur>::valeur()) << endl;
27          return 0;
28  }
```

Chapitre 2

Le système Jasip

Ce chapitre donne un complément sur le langage C++ et SystemC. Nous nous servons alors de SystemC pour simuler un microprocesseur simpliste exécutant du byte code java : Jasip.

2.1 Compléments C++

Cette section ajoute quelques compléments sur le langage C++ qui n'étaient pas présents dans le premier cours.

2.1.1 Précision sur le passage de paramètres

Voici le code suivant où une fonction nommée `test` attend en paramètre une instance de classe :

```
1  class Voiture { };
2  void test(Voiture v) {
3          // faire des tests
4  }
5  int main() {
6          Voiture voiture();
7          test(voiture);
8          return 0;
9  }
```

Voici les actions effectuées lors de l'exécution de la fonction main du programme :
- initialiser la variable `voiture`.
- créer une nouvelle variable `v` et recopier le contenu de `voiture` dedans.
- appeler la fonction `test` avec `v` comme argument.

C'est un peu long, de plus il y a une recopie du contenu de l'instance `voiture` ce qui peut prendre du temps. Deux solutions alternatives existent :
- passer une référence à la variable `voiture` plutôt que de recopier sa valeur,
- passer l'adresse mémoire où la variable `voiture` est stockée.

Ces approches sont détaillées dans les deux sections suivantes.

2.1.2 Pointeurs

Pour chaque variable dans le programme on peut obtenir l'adresse (*i.e.*, sa position dans la mémoire) de celle-ci en ajoutant le symbole `&` devant. Ainsi le programme suivant affiche l'adresse de la variable entière `i` :

```
1  #include <iostream>
2  int main() {
3          int i=0;
4          std::cout << &i << std::endl;
5          return 0;
6  }
```

Si ce programme est dans le fichier `t.cc` on obtient l'exécution suivante :

```
>g++ -o t.exe t.cc
>./t.exe
0xbffffa54
```

On sait alors que la valeur de `i` est stockée à partir de la case mémoire numéro `bffffa54` (en hexadécimal). Si une variable `t` à un type `Toto` alors le type de `&t` l'adresse de `t` est noté `Toto *`. Ainsi un programme équivalent au précédent est :

```
1   #include <iostream>
2   int main() {
3           int i=0;
4           int * j = &i;
5           std::cout << j << std::endl;
6           return 0;
7   }
```

Ainsi pour éviter la recopie dans l'appel de fonction présenté dans la section précédente, il suffit d'écrire :

```
1   class Voiture { };
2   void test(Voiture * v) {
3       // faire des tests
4   }
5   int main() {
6           Voiture voiture();
7           test(&voiture);
8           return 0;
9   }
```

2.1.3 Références

Une autre façon existe pour éviter la recopie : le passage par référence. La syntaxe est simple il suffit dans le prototype de la fonction ou de la méthode de rajouter le symbole `&` après le nom de type. Voici le résultat sur l'exemple :

```
1   class Voiture { };
2   void test(Voiture & v) {
3       // faire des tests
4   }
5   int main() {
6           Voiture voiture();
7           test(voiture);
8           return 0;
9   }
```

2.1.4 Différence entre références et pointeurs

Les références et les pointeurs sont en fait la même chose : une adresse mémoire associée à un type décrivant l'objet pointé. La différence est qu'une référence doit obligatoirement référencer un objet, ce qui n'est pas le cas d'un pointeur qui peut être n'importe quelle adresse mémoire, y compris une adresse ne correspondant en fait à rien. Une telle adresse généralement utilisée est un pointeur vers l'adresse mémoire 0 de type `void *`, et noté `NULL`, définit dans le fichier `stdlib.h` :

```
1   #include <stdlib.h>
2   int main() {
3           int * intPointer = NULL;   // on peut initialiser le pointeur sans avoir
4                                      // la variable
5           int i=0;
6           int & intReference = i;    // on doit avoir la variable pour initialiser
7                                      // la référence.
8           intPointer=&i;
9           return 0;
10  }
```

La question de l'initialisation est encore plus sensible dans le cas de champs d'une classe. Voici par exemple une classe possédant un champ qui est un pointeur :

```
1  class MaClasse {
2  public :
3          MaClasse(int * i) { pointeur=i; }
4          int * pointeur;
5  };
```

En revanche le code suivant aurait été impossible à écrire avec une référence, en effet la référence doit être initialisée avant même l'exécution du corps du constructeur. Ainsi pour les références on utilise la syntaxe suivante :

```
1  class MaClasse {
2  public :
3          MaClasse(int & i) : maReference(i)  {
4                  // corps du constructeur
5          }
6          int & maReference;
7  };
```

Ce type d'initialisation (hors du constructeur) est également utile pour les classes ne disposant pas d'un constructeur par défaut (c'est à dire d'un constructeur ne prenant aucun argument). Ainsi le code suivant ne compile pas :

```
1  // définition d'une classe A ne possédant pas de constructeur par défaut
2  class A {
3  public :
4    A(int i) {}
5  };
6  // définition d'une classe B possédant un champ de type A
7  class B {
8  public:
9    B(int i) { a=A(i);}    // ERREUR : pas de constructeur par défaut pour A !
10   A a;
11 };
```

Le programme ne peux compiler car `a` doit être initialisé avant l'entrée dans le constructeur de `B`. Voici comment écrire le constructeur de `B` :

```
1  class A {
2  public :
3    A(int i) {}
4  };
5  class B {
6  public:
7    B(int i) : a(i) { }
8    A a;
9  };
```

2.1.5 Déréférencement de pointeurs

Ainsi nous avons des pointeurs qui sont des adresses mémoire vers des objets d'un certain type. Pour récupérer l'objet pointé il existe un opérateur `*`, c'est en quelque sorte l'inverse de l'opérateur `&` :

```
1  #include <iostream>
2  #include <stdlib.h>
3  class Voiture {
4  public :
5          int valeur;
6  };
7  void afficheValeur(const Voiture * v) {
```

```
8           if (v==NULL) return;
9           Voiture voiture= *v;
10          std::cout << voiture.valeur << std::endl;
11  }
```

La fonction précédente fonctionne correctement mais créé à la ligne 9 une nouvelle instance de voiture dans laquelle est recopiée le contenu de la valeur pointée par `v`. On perd le bénéfice du passage par pointeur. Il faut soit utiliser des références, soit utiliser l'opérateur `->` permettant d'accéder à une méthode ou un champ d'une classe pointée :

```
1   #include <iostream>
2   #include <stdlib.h>
3   class Voiture {
4   public :
5           int valeur;
6   };
7   void afficheValeur(const Voiture * v) {
8           if (v==NULL) return;       // si v ne pointe vers rien, quitter.
9           std::cout << v->valeur << std::endl;
10  }
```

Avec des références, il n'y a en revanche pas besoin de tester la validité :

```
1   #include <iostream>
2   class Voiture {
3   public :
4           int valeur;
5   };
6   void afficheValeur(const Voiture & v) {
7           std::cout << v.valeur << std::endl;
8   }
```

2.1.6 Pointeurs de fonction

Comme on définit des pointeurs vers des variables, on peut également définir des pointeurs vers des fonctions.
Par exemple considérons le code suivant :

```
1   #include <string>
2   // la fonction f renvoie la taille de la chaîne de caractères
3   int f(std::string s) {
4           return s.size();
5   }
```

Cela définit le symbole `f` comme un pointeur de fonction. Nous allons définir le type `PointeurIntString` qui pointe vers une fonction renvoyant un `int` en prenant en argument un objet de type `std::string` :

```
6   typedef int (*PointeurIntString)(std::string);
```

Voici la syntaxe pour déclarer un type pointeur de fonction :

`typedef` *<type de retour>*`(*` *<nouveau nom>*`) (`*<liste des types d'arguments>*`);`

On peut alors mettre la valeur du symbole `f` dans une variable `i` et appeler la fonction :

```
7   #include <iostream>
8   int main() {
9     PointeurIntString i = f;
10    std::cout << i("mon texte") << std::endl;
11    return 0;
12  }
```

Les pointeurs de fonctions sont utiles quand le choix de la fonction à appeler varie. Par exemple pour écrire le code d'une calculatrice l'opération va dépendre du bouton sur lequel l'utilisateur appuie. Si les pointeurs de fonctions sont dans un tableau il suffit alors d'appeler la fonction dont l'indice correspond au bouton.

Voici un exemple de tableau de pointeurs de fonctions :

```
1   int plus (int x, int y) { return x+y;}
2   int minus (int x, int y) { return x-y;}
3   int times (int x, int y) { return x*y;}
4   int div (int x, int y) { return x/y;}
5
6   int (*fun_array[]) (int,int) = { plus,minus,times,div};
7
8   int donneResultat(int operation, int operande1, int operande2) {
9           return *(fun_array[operation])(operande1,operande2);
10  }
```

Voici la syntaxe pour déclarer directement un tableau de pointeurs de fonction :

<type de retour>`(*` *<nom du tableau>*`[]) (` *<liste des types d'arguments>*`);`

2.1.7 Pointeurs de méthodes

La section précédente abordait le principe des pointeurs de fonctions. On trouve l'équivalent pour les méthodes des classes.

```
1   class A {
2   public :
3           int methode1() { return 14;}
4           int methode2() { return 300;}
5   };
6
7   typedef int (A::*MethodeDansA)();
8
9   MethodeDansA  methodTab [] = { &A::methode1, & A::methode2 };
```

Dans cet exemple on définit une classe `A` comportant deux méthodes ayant la même signature : elles ne prennent pas d'argument et renvoient un entier. On définit alors à l'aide de la commande `typedef` le type `MethodeDansA` comme étant un pointeur vers une méthode de `A` ne prenant pas d'argument et renvoyant un entier. Voici la syntaxe pour déclarer à l'aide de `typedef` un type pointeur de méthode :

`typedef` *<type de retour>*`(`*<nom de classe>*`::*`*<nom du nouveau type>*`)(`*<liste des types d'arguments>*`);`

On peut alors définir un tableau `methodTab` conservant des pointeurs vers `methode1` et `methode2`.

2.1.8 Structure de contrôle pour choix multiples

Nous avons présenté différentes structures de contrôle : `if`, `for`, `while` ou `do while`. Il existe en outre le `switch` qui permet de manière efficace d'effectuer l'équivalent d'une suite de commandes `if`. Ainsi le code suivant permet d'effectuer différentes actions en fonction de la valeur du caractère `c` :

```
1   void lireTouche(char c) {
2           switch (c) {
3           case 'q' :
4                   // l'utilisateur a tapé 'q' nous allons quitter l'application
5                   exit(0);
6           case 'l' :
7                   // effectuer la lecture d'un fichier
8                   lireFichier();
9                   break;        // quitter le switch
10          case 'a':
11          case 'b':
```

```
12                  cerr << "Action impossible" << endl;
13                  break;
14          default:
15                  // la touche n'est pas reconnue
16                  cerr << "La touche " << c << " n'est pas reconnue." << endl;
17          }
18  }
```

Quelque soit le nombre de choix de la commande switch, la bonne option est déterminée en temps quasi constant. Avec des commandes `if` à la suite on aurait en moyenne un temps linéaire si les choix sont équiprobables. L'exercice 2.4 montre l'efficacité de la commande `switch`.

2.1.9 Types énumérés

Un type C++ que nous n'avons pas abordé jusqu'à présent sont les énumérés. Ainsi pour une voiture si on dispose de quatre couleurs possibles blanc, rouge, jaune et noir il est possible de coder la couleur dans un champ en utilisant une convention : par exemple 0 signifie blanc, 1 rouge, 2 jaune et 3 noir. On obtient alors la classe suivante :

```
1   class Voiture {
2   public:
3           int couleur;
4   };
5   int main() {
6           Voiture v;
7           v.couleur=2;
8           return 0;
9   }
```

Une telle convention peut ne pas être extrêmement pratique, il faut se souvenir du code ou bien faire des macros en utilisant `#define`. Il existe un type énuméré permettant de manipuler la liste de couleurs souhaitées :

```
1   enum CouleurVoiture {
2           CV_BLANC,
3           CV_ROUGE,
4           CV_JAUNE,
5           CV_NOIR
6   };
7   class Voiture {
8   public:
9           CouleurVoiture couleur;
10  };
11  int main() {
12          Voiture v;
13          v.couleur=CV_JAUNE;
14          return 0;
15  }
```

La syntaxe pour déclarer un type énuméré est :

`enum` <*nom du type*>{ <*liste des symboles*>} `;`

La liste des symboles étant une séquence d'identifiants séparés par des virgules. Les identifiants constituant l'énumération sont des symboles qui font partie de l'espace de nommage dans lequel l'énuméré est déclaré. C'est pourquoi les identifiants de l'énuméré commencent tous par le préfixe `CV_` (comme couleur voiture) pour éviter les conflits de nom avec une autre définition de blanc ou noir. C'est pourquoi il peut être judicieux de déclarer l'énuméré dans la classe (on peut alors omettre le préfixe `CV_`) :

```
1   class Voiture {
2   public:
3           enum CouleurVoiture {
4                   BLANC,
```

```
5                   ROUGE,
6                   JAUNE,
7                   NOIR
8           };
9           CouleurVoiture couleur;
10  };
11  int main() {
12          Voiture v;
13          v.couleur=Voiture::JAUNE;
14          return 0;
15  }
```

En pratique les énumérés sont implémentés par des type entiers. Les valeurs commencent à zéros puis vont de un en un dans l'ordre de déclaration. Dans l'exemple précédent en pratique BLANC vaut zéro, ROUGE vaut 1, JAUNE vaut 2 et NOIR 3. On peut si on le souhaite forcer les valeurs :

```
1   enum CouleurVoiture {
2           BLANC=4,  // BLANC vaut 4
3           ROUGE,    // ROUGE vaut 5
4           JAUNE,    // JAUNE vaut 6
5           NOIR=50   // NOIR vaut 50
6   };
```

A noter que si nous avions imposé NOIR=5 alors ROUGE et NOIR auraient eu la même valeur, ce qui peut poser de graves problèmes.

2.1.10 Notion d'opérateurs

Les opérateurs sont les opérations classiques que l'on effectue en général sur des entiers, mais déclinées sur des classes. Pour cette raison les opérateurs peuvent être définis pour une maîtrise totale de leur action.

Ainsi par exemple pour deux entiers `i` et `j` on peut écrire l'expression `i+j`. Qu'en est-il pour deux classes ? Il faut définir l'opérateur + sur cette classe. Ainsi on peut définir l'addition sur une classe comportant des vecteurs de taille 4 :

```
1   class Vector {
2   public :
3       // construction du vecteur à l'aide de 4 entiers
4       Vector (int i,int j,int k,int l) {
5           valeur[0]=i;
6           valeur[1]=j;
7           valeur[2]=k;
8           valeur[3]=l;
9       }
10      // définition de l'opérateur + entre
11      // l'instance de la classe et l'argument v
12      Vector operator+(const Vector & v) const {
13          return Vector(valeur[0]+v.valeur[0],
14                        valeur[1]+v.valeur[1],
15                        valeur[2]+v.valeur[2],
16                        valeur[3]+v.valeur[3]);
17      }
18  private:
19      int valeur[4];
20  };
21
22  int main() {
23    Vector v1 (1,2,3,4);
24    Vector v2 (5,6,7,8);
25    Vector v3 = v1+v2;    // appel à l'opérateur +
```

```
26                          // v3 est égal à [ 6 8 10 12 ]
27    return 0;
28  }
```

Pour définir un opérateur la syntaxe est semblable à celle d'une methode :

<type de retour>`operator`*<description>*`(`*<liste d'arguments>*`)` { *<corps >*`)` }

Le nombre d'arguments est fixé par l'opérateur. Ici c'est 1 argument pour l'opérateur + définit dans une classe ou deux arguments si l'opérateur est définit hors de la classe.

Voici comment définir l'opérateur + hors de la classe `Vector`. Comme nous souhaitons accéder aux champs privés de la classe `Vector` depuis l'opérateur + qui va être déclaré de manière externe à la classe il nous faut tout d'abord déclarer le prototype de l'opérateur + avec l'attribut `friend` dans la classe `Vector`. On peut alors librement définir l'opérateur + hors de la classe, avec cette fois deux arguments, qui sont les deux vecteurs à ajouter.

```
1   class Vector {
2       // déclarer l'opérateur + comme \og ami\fg~pour qu'il puisse
3       // accéder aux champs privés des instances de Vector.
4       friend Vector operator+(const Vector &, const Vector &);
5   public :
6       // construction du vecteur à l'aide de 4 entiers
7       Vector (int i,int j,int k,int l) {
8           valeur[0]=i;
9           valeur[1]=j;
10          valeur[2]=k;
11          valeur[3]=l;
12      }
13  private:
14      int valeur[4];
15  };
16
17  // définition de l'opérateur + entre deux vecteurs
18  Vector operator+(const Vector & v1, const Vector & v2) {
19      return Vector(v1.valeur[0]+v2.valeur[0],
20                    v1.valeur[1]+v2.valeur[1],
21                    v1.valeur[2]+v2.valeur[2],
22                    v1.valeur[3]+v2.valeur[3]);
23  }
24  int main() {
25    Vector v1 (1,2,3,4);
26    Vector v2 (5,6,7,8);
27    Vector v3 = v1+v2;    // appel à l'opérateur +
28                          // v3 est égal à [ 6 8 10 12 ]
29    return 0;
30  }
```

Beaucoup d'autres opérations sont des opérateurs. Voici la liste complète des opérateurs en C++ :

```
+    -      *    /      %    ^         &    |    ~    !    =    <    >
+=   -=     *=   /=     %=   ^=        &=   |=   <<   >>   >>=  <<=  ==
!=   <=     >=   &&     ||   ++        --   ,    ->*  ->   ()   []
new  delete      new[]       delete[]
```

Ci-dessous figure une liste d'exemples d'opérateurs qu'il est souvent utile de redéfinir. Les opérateurs `*` unaire ainsi que de pré et de post incrémentation et décrémentation `++` et `--` sont détaillés au chapitre suivant.

opérateurs algébriques usuels

Nous venons de détailler l'usage de `operator+` l'opérateur d'addition. Il existe de même les équivalents pour les autres opérations usuelles possédant deux arguments : `operator-` binaire pour la soustraction, `operator*` pour la multiplication,

`operator/` pour la division, `operator&&` pour le et logique, `operator||` pour le ou logique, `operator|` pour le ou bit à bit, `operator&` pour le et bit à bit, `operator^` pour le ou exclusif.

Pour ces opérateurs nous trouvons également la variante d'assignation. Ainsi pour une variable `i` de type `int` l'expression `i+=4;` signifie `i=i+4;`, mais `+=` est un opérateur spécifique. On trouve ainsi les opérateurs suivants : `operator+=` pour l'addition, `operator-=` pour la soustraction, `operator*=` pour la multiplication, `operator/=` pour la division, `operator|=` pour le ou bit à bit, `operator&=` pour le et bit à bit, `operator^=` pour le ou exclusif.

Il est à noter que `operator*` et `operator&` donnés ici en exemple sont des opérateurs binaires (c'est à dire supportant deux arguments). Il existe les mêmes opérateurs unaires, détaillés un peu plus loin et ayant un sens différent : le `operator*` unaire est l'opérateur de déréférençage, et `operator&` unaire est l'opérateur permettant d'accéder à l'adresse d'un élément.

operator<<

L'opérateur `<<` de décalage binaire est souvent utilisé pour l'affichage. Initialement ce symbole est le décalage binaire sur la gauche (c'est à dire de rajouter un zéro à droite dans l'écriture binaire). Ainsi pour une variable `x` de type `int`, `x << 1` est équivalent à `x*2` et `x << n` revient à ajouter `n` zéros à droite en écriture binaire , c'est à dire à multiplier par 2^n.

Pour l'affichage le flux est redirigé sur un objet de type `ostream`. Voici un exemple sur un objet de type `Vector`. L'exécution de l'exemple affiche `[1 2 3 4 ]` :

```
1   #include <iostream>
2   using namespace std;
3   class Vector {
4       friend ostream & operator<< (ostream &, const Vector &);
5   public :
6       Vector (int i,int j,int k,int l) {
7           valeur[0]=i;
8           valeur[1]=j;
9           valeur[2]=k;
10          valeur[3]=l;
11      }
12  private:
13      int valeur[4];
14  };
15
16  ostream & operator<< (ostream & os, const Vector & v) {
17      os << "[";
18      for (int i=0;i<4;++i) {
19          os << v.valeur[i] << " ";
20      }
21      os << "]";
22      return os;
23  }
24
25  int main() {
26      Vector v1 (1,2,3,4);
27      cout << v1 << endl;
28      return 0;
29  }
```

operator new

On trouve deux types de variables : des variables globales accessibles depuis n'importe où dans le programme ainsi que des variables locales à chaque fonction. Parfois on peut avoir besoin de créer de nouvelles variables globales, ou au moins partagées entre plusieurs fonctions. Pour cela on peut réserver de la mémoire à l'aide de l'opérateur `new`.

Ainsi, étant donné une classe `A` l'instruction ci-dessous :

```
1   A * a = new A();
```

va réserver une zone mémoire contenant une instance de `A` initialisée avec le constructeur par défaut. Cette adresse peut alors être manipulée par n'importe quelle fonction du programme. La réservation continue jusqu'à ce que l'opérateur `delete` soit appelé quelque part dans le programme :

```
2  delete a;
```

On appelle généralement fuite mémoire un oubli répété de libération de mémoire à l'aide de la commande `delete`. Un programme qui fonctionne pendant 10 minutes et a une fuite mémoire ne pose pas de problèmes. Cependant, si ce programme tourne plusieurs heures il peut finir par consommer l'ensemble de la mémoire de la machine.

L'instruction `new` est en fait l'appel à un opérateur et peut donc être surchargé. Cela peut être utile pour forcer les endroits dans la mémoire où sont alloués les objets, ou pour simplement tracer les allocations réalisées. Voici l'exemple d'une classe `V` où les instances allouées via `new` le seront dans un tableau nommé `tableauDeV`.

```
1  class V {
2  public:
3    void * operator new (unsigned int);
4    int a;
5  };
6  #define NULL (void*)0
7  V tableauDeV [10];
8  int indexInArray=0;
9  void * V::operator new (unsigned int size) {
10   if (indexInArray>=10) return NULL;
11   return &(tableauDeV[indexInArray++]);
12 }
```

Au bout de 10 allocations new renverra `NULL`.

operator=

L'opérateur d'assignation est utilisé quand une variable est affectée à une autre. Considérons une classe `Vector` représentant un vecteur de taille variable. La taille du vecteur est donnée en argument au constructeur.

```
1  class Vector {
2  public :
3      Vector (int i) {
4          // allocation d'un tableau de taille i
5          valeur = new int[taille=i]; // appel à l'opérateur new[] sur les entiers
6      }
7      ~Vector() {
8           // effacer l'allocation faite dans le constructeur
9           delete [] valeur;          // appel à l'opérateur delete[]
10     }
11     void setValue(int n, int i) {
12         if (i<0 || i>=taille) return;
13         valeur[i]=n;
14     }
15     Vector & operator=(const Vector & v) {
16         if (taille!=v.taille) {
17             delete [] valeur;
18             valeur = new int[taille=v.taille];
19         }
20         for (int i=0;i<taille;++i) {
21             valeur[i]=v.valeur[i];
22         }
23     }
24 private:
25     int * valeur;
26     int taille;
```

```
27  };
28  int main() {
29      Vector v1 (10);
30      for (int i=0;i<10;++i) v1.setValue(i,i);
31      Vector v2 (10);
32      v2=v1;
33      return 0;
34  }
```

Il est à noter que si nous avions écrit directement `Vector v2 = v1` c'est le constructeur par recopie (détaillé ci-dessous) qui est appelé et non l'opérateur d'affectation.

Aparté sur le constructeur par recopie

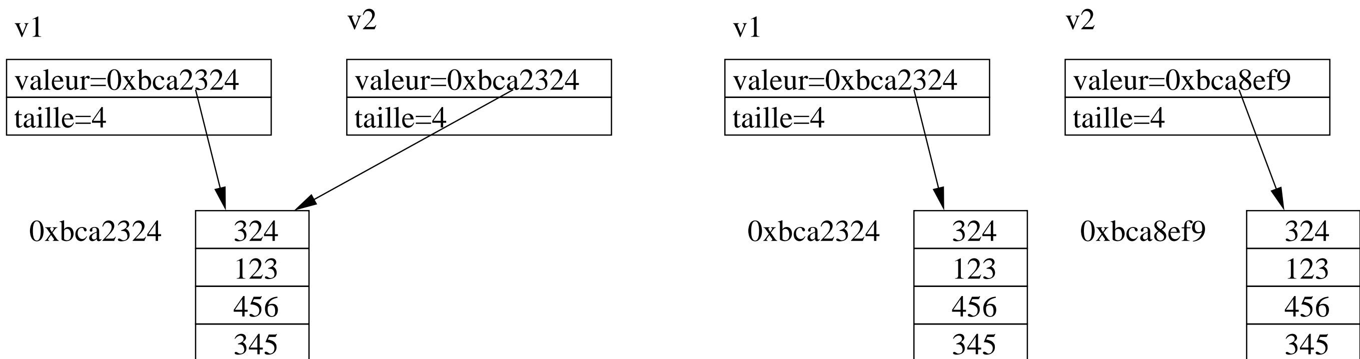
Notons que l'exemple précédent n'est pas pratique. En effet pour une expression `Vector v2 = v1` ce n'est pas le constructeur par défaut qui est appelé pour `v2` (il n'y en a d'ailleurs pas), suivit d'une affectation par l'opérateur = mais le constructeur par recopie. Le constructeur par recopie est un constructeur créé par défaut permettant de recopier une instance de d'une classe dans une autre. La signature de ce constructeur est pour la classe `Vector` :

```
Vector(const Vector &)
```

Ainsi dans ce cas précis pour définir le constructeur par recopie il faut écrire :

```
Vector (const Vector & v) {
    if (taille!=v.taille) {
        delete [] valeur;
        valeur = new int[taille=v.taille];
    }
    for (int i=0;i<taille;++i) {
        valeur[i]=v.valeur[i];
    }
}
```

Voici la différence entre le constructeur par recopie par défaut et le nouveau constructeur par recopie où le tableau est réalloué :



`operator==` et `operator !=`

Voici la définition de l'opérateur == sur la classe vecteur précédemment décrite :

```
bool operator==(const Vector & v) {
    if (taille!=v.taille) {
        return false;
    }
    for (int i=0;i<taille;++i) {
        if (valeur[i]!=v.valeur[i])
```

```
            return false;
      }
  }
```

Tout comme l'opérateur `operator==` on trouve son inverse `operator !=`. Il est *a priori* saint de définir `operator !=` comme la négation de `operator==` même si rien ne le contraint. Voici l'exemple sur la classe `Vector` :

```
bool operator!=(const Vector & v) {
  return !(*this==v);  // appel à l'opérateur == précédemment définit.
}
```

La variable `this` est un pointeur vers l'instance sur laquelle la méthode a été appelée. Elle peut être utilisée dans toute méthode non statique d'une classe.

operator[]

Pour accéder aux valeurs données d'une instance `v` de la classe `Vector`, il est pratique d'utiliser la syntaxe `v[i]`. Il faut pour cela redéfinir l'opérateur `[]` :

```
int operator[](int i) {
    if (i<0 || i>=taille) return 0;
    return valeur[i];
}
```

Le cas où `i` n'est pas dans le bon intervalle pourrait également être traité par le mécanisme des exceptions (cf. section 4.1.2) plutôt que de simplement retourner zéro.

operator()

L'opérateur `()` permet d'utiliser les instances de classes avec une syntaxe d'appel de fonction. Cela peut être une alternative à l'usage des pointeurs de fonction.

Imaginons que nous disposons d'une fonction nommée `trieList` qui trie une liste à laquelle on souhaite passer en argument la manière de comparer deux éléments. On pourrait utiliser un pointeur vers la fonction effectuant la comparaison.

Nous allons créer une classe nommée `Compare` dans laquelle nous allons définir l'opérateur `()`. Le type de comparaison sera donné au constructeur de l'instance par un type énuméré `CmpType`.

```
1   enum CmpType {
2     CMP_PLUS_GRAND,
3     CMP_PLUS_GRAND_OU_EGAL,
4     CMP_PLUS_PETIT,
5     CMP_PLUS_PETIT_QUE
6   };
7
8   class Compare {
9   public:
10    Compare(CmpType t) {type=t;}
11    bool operator()(int i1,int i2) {
12      switch (type) {
13      case CMP_PLUS_GRAND:
14        return i1>i2;
15      case CMP_PLUS_GRAND_OU_EGAL:
16        return i1>=i2;
17      case CMP_PLUS_PETIT:
18        return i1<i2;
19      case CMP_PLUS_PETIT_QUE:
20        return i1<=i2;
21      }
22      return true;
23    }
```

```
24  private:
25    CmpType type;
26  };
27
28  void trieList(Compare &);
29
30  int main() {
31    Compare cmp (CMP_PLUS_PETIT_QUE);
32    bool b1 = cmp(1,3);  // b1 vaut true
33    bool b2 = cmp(4,2);  // b2 vaut false
34    trieList(cmp);  // trier au regard de la fonction définie par cmp
35    return 0;
36  }
```

operator->

On peut redéfinir l'opérateur `->` ce qui peut permettre par exemple d'appeler des méthodes d'un autre objet. Ainsi par exemple ici nous définissons une classe `A` et nous appelons sur une instance de `t` de `T` une méthode de `A` via l'opérateur `->` :

```
1   #include <iostream>
2   class A {               // définition d'une classe A
3   public:
4     A() { }
5     int getValue() { return 314; }
6   };
7   class T {               // définition de T
8   public:
9     T(A *_a) { a=_a;}
10    A * operator -> () {  // redéfinition de ->
11      return a;
12    }
13    A * a;
14  };
15  int main() {
16    A * a = new A();
17    T t (a);
18    std::cout << t->getValue() << std::endl;
19    return 0;
20  }
```

L'exécution du programme ci-dessus affiche 314.

2.2 Sémantique d'exécution SystemC

Outre la description de modèles abordée dans le cours précédent, SystemC propose une sémantique pour l'exécution de ses modèles. L'exécution est basée sur un modèle événementiel qui se déroule en plusieurs phases.

2.2.1 Phases de simulation d'un modèle SystemC

Elaboration

Il y a tout d'abord la phase d'élaboration. Elle commence avec l'invocation de la fonction `sc_main`, et elle finit avec l'invocation de la fonction `sc_start`. C'est la phase dans laquelle le modèle est construit : les modules sont instanciés, les ports sont attachés aux signaux.

Simulation

La simulation commence avec l'invocation de la fonction `sc_start`. Si aucun paramètre n'est fourni la simulation finit avec l'invocation à `sc_stop`. Si un temps est donné en argument à `sc_start` la simulation s'arrête quand ce temps est écoulé ou sur un appel à `sc_stop`.

Voici les étapes répétées de la simulation :

1. Initialise la liste des modules à exécuter.
2. Dans la liste des modules à exécuter en prendre un et reprendre son exécution. L'ordre dans lequel les modules s'exécutent n'est pas spécifié. L'exécution d'un module peut déclencher des évènements qui eux-mêmes peuvent réveiller d'autres modules.
3. répéter l'étape 2 tant qu'il y a des modules à exécuter.
4. mise à jour : exécuter les appels à `update()`.
5. répéter l'étape 2 suite aux appels à `update()`.
6. si il n'y a plus d'évènement possible alors la simulation prend fin.
7. sinon avancer l'horloge à la nouvelle date.
8. déterminer les modules à exécuter suite au changement d'horloge et aller dans l'étape 2.

L'étape 2 est nommée delta-cycle, et les étapes 2 à 8 forment un cycle, où l'horloge est effectivement incrémentée. Par rapport au temps dans la simulation, la durée d'un delta cycle est nulle. Il peut y avoir un nombre arbitraire de delta-cycles dans un cycle d'horloge.

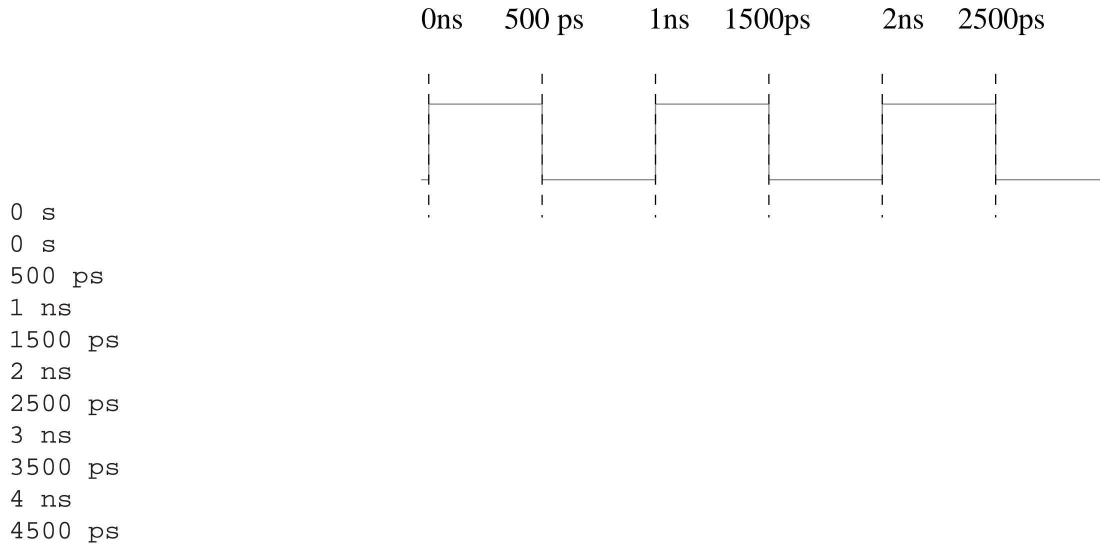
Les phases de mise à jour (update) sont utiles par exemple pour les canaux fifos où les valeurs ne sont écrites qu'au moment des phases de mise à jour.

2.2.2 Temps et horloge

Voici un module `GiveTime` affichant la valeur de l'horloge :

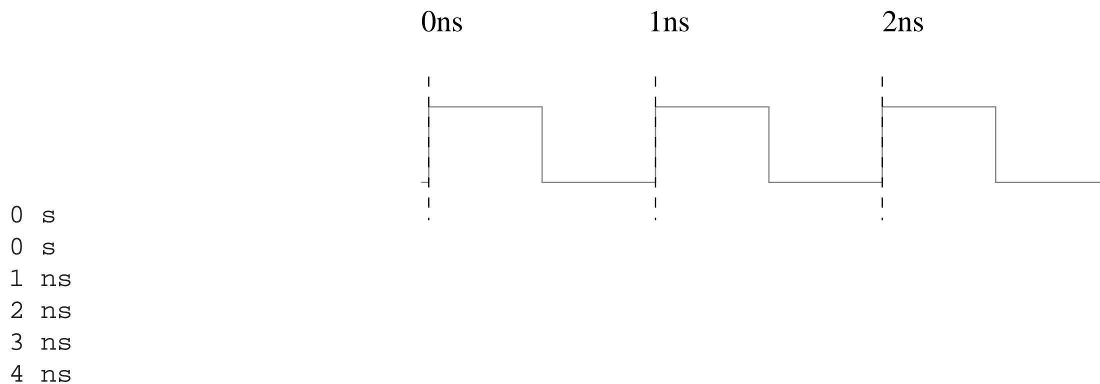
```
1   #include "systemc.h"   // inclut les définitions SystemC
2
3   SC_MODULE(GiveTime) {  // définit un module nommé GiveTime
4     // un seul port d'entrée : l'horloge.
5     sc_in_clk clock;
6     void showClock() {
7       cout << sc_time_stamp() << endl;
8     }
9     SC_CTOR(GiveTime) {
10      SC_METHOD(showClock);
11      sensitive << clock;
12    }
13  };
14  int sc_main(int argc, char** argv) {
15    sc_clock clock("clock",sc_time(1,SC_NS)); // créé un objet représentant l'horloge
16                                              // avec une période de 1ns.
17    GiveTime giveTime("giveTime");   // créé l'instance du module
18    giveTime.clock(clock);           // lie l'horloge clock au port giveTime.clock
19    sc_start(sc_time(5,SC_NS));      // lance la simulation jusqu'à 5 ns
20    return 0;
21  }
```

L'horloge du système est déclarée à la ligne 15. Les horloges sont des instances de `sc_clock`. Elles sont construites avec un nom et une période. Le port associé dans le module `GiveTime` est de type `sc_in_clk`. Voici le résultat de l'exécution du module :



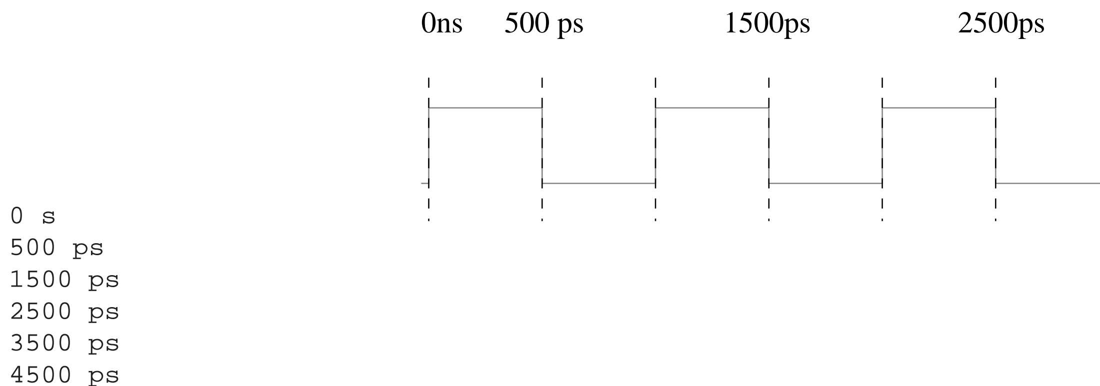

```
0 s
0 s
500 ps
1 ns
1500 ps
2 ns
2500 ps
3 ns
3500 ps
4 ns
4500 ps
```

La période de l'horloge étant de 1 ns, la valeur de celle-ci change toutes les 500ps. La valeur zéro répétée correspond à la phase d'initialisation. En effet la directive `sensitive << clock` spécifie que la méthode `showClock` est appelée à chaque changement de valeur de l'horloge. Si l'on ne souhaite afficher que les moments où le front d'horloge est montant il faut écrire `sensitive << clock.pos()`. On obtient alors l'affichage :




```
0 s
0 s
1 ns
2 ns
3 ns
4 ns
```

Pour être sensible au front descendant de l'horloge il faut écrire `sensitive << clock.neg()`. On obtient alors l'affichage :




```
0 s
500 ps
1500 ps
2500 ps
3500 ps
4500 ps
```

2.2.3 Instances d'exécutions

Les `SC_METHOD` rencontrées dans le chapitre précédent doivent être entièrement exécutées et forment donc des delta cycles. Pour écrire des procédures sur plusieurs delta cycles (ce qui permet d'écrire un code plus lisible), il existe une autre déclaration : `SC_THREAD`.

Les `SC_THREAD` sont semblables aux `SC_METHOD` par les caractéristiques suivantes :

– il s'agit de méthodes d'un `SC_MODULE` ayant la signature `void ()`.

– les méthodes sont sensibles à une série d'évènements.

Les SC_THREAD diffèrent des SC_METHOD par les caractéristiques suivantes :

– initialement chaque SC_THREAD est lancé
– l'exécution d'un SC_THREAD peut être suspendue par le SC_THREAD lui même lors qu'un signal est attendu.

Le fait que les SC_THREAD puissent être interrompus permet de faciliter l'écriture dans les fifos où les lectures ou écritures peuvent être bloquantes :

```
#include "systemc.h"
SC_MODULE(MonModule) {
  // ports en sortie
  sc_fifo_out<int> outFifo;
  // ports en entrée
  sc_fifo_in<int> inFifo;

  void prcMonModule() {
    while (true) {
      outFifo.write(inFifo.read());
    }
  }
  SC_CTOR(MonModule) {
    SC_THREAD(prcMonModule);
    sensitive << inFifo.data_written();
  }
};
```

Voici le même code en utilisant une SC_METHOD :

```
#include "systemc.h"
SC_MODULE(MonModule) {
  // ports en sortie
  sc_fifo_out<int> outFifo;
  // ports en entrée
  sc_fifo_in<int> inFifo;

  void prcMonModule() {
    if (inFifo.num_available()<1) { return;}
    if (outFifo.num_free()<1) { return;}
    outFifo.write(inFifo.read());
  }
  SC_CTOR(MonModule) {
    SC_METHOD(prcMonModule);
    sensitive << inFifo.data_written();
  }
};
```

2.2.4 Communication par files de messages

SystemC propose d'autres classes que les signaux pour communiquer entre les modules. Une structure fréquente est nommée fifo et représente une file de messages de type fifo (First In First Out). Voici la définition d'une file de message de type fifo :

> Une file de messages de type fifo de capacité n peut contenir n messages. Deux types d'opérations sont possibles sur les fifos : l'écriture et la lecture. L'écriture ajoute un message dans la file, la lecture donne et supprime le message le plus ancien. Pour chaque opération, on distingue des variantes bloquantes et non bloquantes : écriture bloquante ou non bloquante si la file est pleine, lecture bloquante ou non bloquante si la file est vide.

Le type représentant la fifo est `sc_fifo`. Les ports associés sont `sc_fifo_in` et `sc_fifo_out`.

Prenons un exemple avec une classe `Producer` qui écrit dans une fifo et une classe `Consumer` qui lit cette même fifo.

```
1  #include "systemc.h"
2  #include <iostream>
```

```
3   using namespace std;
4   SC_MODULE(Producer) {
5     sc_fifo_out<int> fifoOut;    // un seul port en entrée : la fifo
6     int counter;                 // compteur à incrémenter à chaque envoi
7     void prcProducer() {
8       while(true) {
9         fifoOut.write(counter);
10        cout << "Sending " << counter++ << endl;
11        if (counter>=100) {
12          sc_stop();
13        }
14      }
15    }
16    SC_CTOR(Producer) {
17      counter=0;
18      SC_THREAD(prcProducer);
19    }
20  };
```

Le producteur envoie les nombres de 1 à 100 puis arrête la simulation. Le consommateur lui affiche les nombres lus :

```
21  SC_MODULE(Consumer) {
22    sc_fifo_in<int> fifoIn;
23    void prcConsumer() {
24      while (true) {
25        int counter = fifoIn.read();
26        cout << "Receiving " << counter << endl;
27      }
28    }
29    SC_CTOR(Consumer) {
30      SC_THREAD(prcConsumer);
31      sensitive << fifoIn.data_written();
32    }
33  };
```

Il reste maintenant à relier les deux modules à l'aide d'une fifo, que nous choisissons de taille 5 :

```
34  int sc_main(int argc, char ** argv) {
35    sc_fifo<int> myFifo (5);  // création d'une fifo de taille 5
36    Producer p ("producer");  // instanciation du producteur
37    p.fifoOut(myFifo);        // lier la fifo au port du producteur
38    Consumer c ("consumer");  // instanciation du consommateur
39    c.fifoIn(myFifo);         // lier la fifo au port du consommateur
40    sc_start();               // lancer la simulation
41    return 0;
42  }
```

Voici le résultat tronqué de l'exécution :

```
Sending 0
Sending 1
Sending 2
Sending 3
Sending 4
Receiving 0
Receiving 1
Receiving 2
Receiving 3
Receiving 4
```

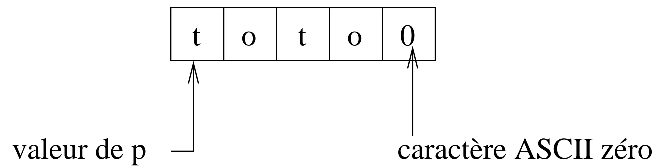
```
Sending 5
Sending 6
...
Sending 98
Sending 99
SystemC: simulation stopped by user.
```

On remarque ici la politique d'ordonnancement du moteur de simulation SystemC : si un SC_THREAD est lancé, son exécution continue jusqu'à ce qu'un blocage intervienne. En effet le producteur rempli complètement la fifo, le consommateur pourrait commencer de lire, mais l'ordonnanceur ne lui donne pas la main.

2.2.5 Passage d'arguments à la simulation

Nous avons vu au chapitre précédent que le prototype de la fonction sc_main est :

```
int sc_main(int argc, char * argv[]);
```



Les variables argc (argument count) et argv (argument value) représentent respectivement le nombre d'arguments en ligne de commande et leur valeurs. En effet char * représente un pointer vers des caractères. Ce type est souvent utilisé dans le langage C pour désigner une chaîne de caractères se terminant par zéro. Ainsi en C ou en C++ l'instruction char * p = "toto" ; fait pointer p vers une zone mémoire de 5 caractères comme représentée à la figure ci-contre. Si le modèle SystemC compilé est nommé toto.exe, l'appel sur la ligne de commande

```
toto.exe -monparametre 314
```

va appeler appeler main avec argc valant 3 et argv valant :

```
{ "toto.exe", "-monparametre", "314"}
```

Voici comment récupérer la valeur après l'option -monparametre :

```
1   #include <string>
2   #include <iostream>
3   #include <stdlib.h> // prototype de la fonction atoi
4   using namespace std;
5   int monParametre;   // variable à affecter
6   int main(int argc, char * argv[] ) {
7     int i=1;
8     while (i<argc) {  // faire parcourir à i tous les indexes possibles
9       if (string(argv[i])=="-monparametre") {
10        // l'option -monparametre a été passée par l'utilisateur
11        ++i;
12        if (i>=argc) {
13          // la ligne de commande s'arrete après -monparametre
14          cerr << "erreur une valeur est attendue après -monparametre" << endl;
15          return 1;
16        } else {
17          // nous utilisons la fonction atoi (Ascii To Integer) pour obtenir la valeur.
18          monParametre=atoi(argv[i]);
19        }
20      }
21      ++i;
22    }
23    cout << "monParametre vaut " << monParametre << endl;
24    return 0;
25  }
```

2.3 Le microprocesseur Jasip

Dans cette section, nous décrivons schématiquement les fonctionnalités requises par un microprocesseur, et nous proposons une architecture extrêmement simplifiée permettant de les implémenter.

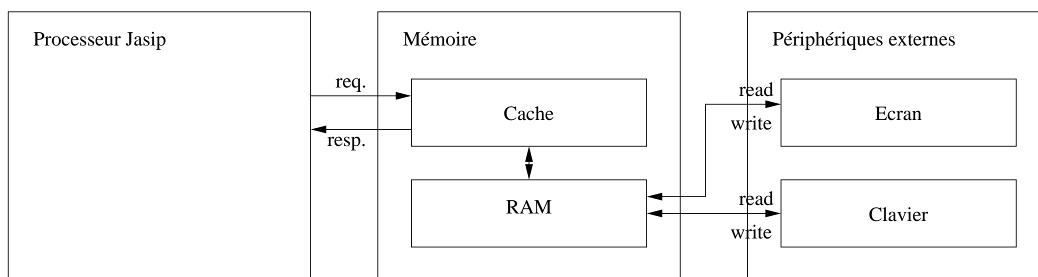
Un émulateur Jasip, écrit en C++, est téléchargeable à l'url :

`http ://www.derepas.com/csml/jasip_0.1.tar.bz2`

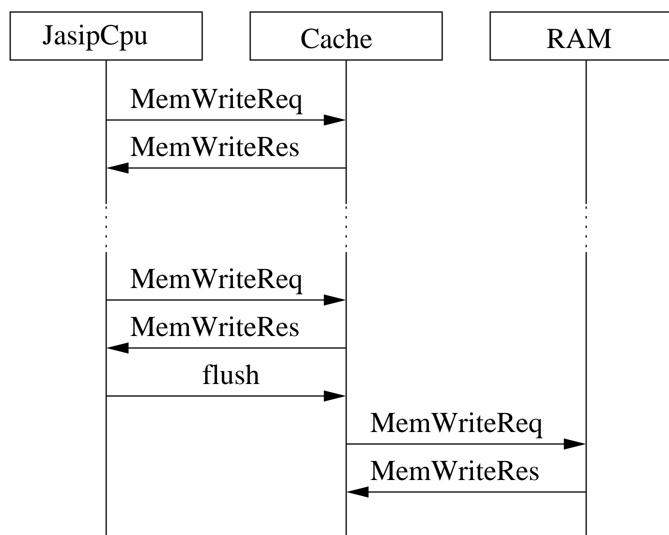
Un émulateur va permettre de faire tourner des exemples en bytecode java, il ne cherche pas à reproduire le comportement interne du microprocesseur, mais seulement ses actions sur la mémoire. En revanche le but des exercices autour de SystemC de ce chapitre est de réaliser un simulateur, c'est-à-dire de reproduire de la manière la plus fiable possible le comportement du microprocesseur, pour mieux en comprendre le fonctionnement.

2.3.1 Les éléments en jeux

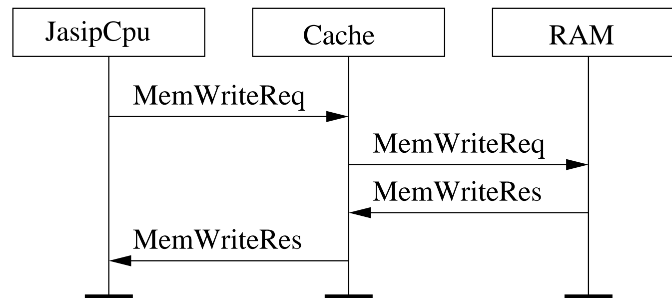
On va s'intéresser à la partie mémoire autour du processeur Jasip. Ci dessous figure l'architecture mémoire du processeur :



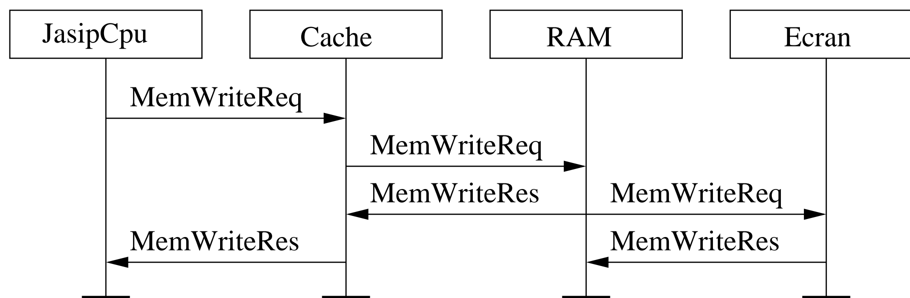
Les adresses mémoires au dessus de 0x1000 passent par le cache comme illustré ci-dessous :



En revanche les adresses plus petites que 0x1000 sont écrites directement en mémeoire :



Cet accès direct à la mémoire permet d'accéder rapidement à des périphériques. Les périphériques Jasip sont en effet accédés par des adresses mémoires. Ainsi le fait de lire à l'adresse 0x290 permet de récupérer la dernière touche tapée. L'écran de 20 lignes sur 40 colonnes est accessible à partir de l'adresse 0x300 jusqu'à l'adresse 0x620.



2.3.2 Description du cache

Le cache est une structure conservant des valeurs mémoire. L'acception du mot cache est la même que dans « la cache des pirates » : c'est là que l'on met les données qui sont précieuses.

L'image classique pour comprendre le rôle du cache est celle de la table à la bibliothèque : pour faire un exposé sur un thème précis on ramène plusieurs livres sur sa table et on travaille avec. Parfois on se relève pour aller en chercher de nouveaux et reposer les livres qui ne servent pas. Si il n'y avait pas de mémoire cache, c'est un peu comme si à chaque fois que l'on souhaite lire quelques lignes dans un ouvrage, il fallait perdre du temps à aller prendre un livre dans les rayons de la bibliothèque.

Un des rôles du simulateur est de pouvoir aider à dimensionner ces caches pour des applications particulières.

2.3.3 Description de la RAM

La RAM permet de conserver des informations mémoire ou bien de communiquer avec le monde extérieur.

2.3.4 Le besoin de modélisation

Nous avons ici décrit sommairement une architecture. De nombreux points doivent être précisés :

- quelles sont exactement les données échangées entre les différents blocs ?
- si on fait un simulation au cycle d'horloge près, quels vont-être les temps de réponse ?
- cette architecture va-t-elle être suffisamment performante pour exécuter des programmes ? Sinon pourquoi ?

Pour toutes ces raisons il y a un besoin important de simuler le microprocesseur. Nous allons voir comment SystemC va nous aider dans cette tâche.

2.4 Exercices sur C++

Ex 2.1 * Appel de fonction

Q1 On considère le code suivant :

```
1   void f(int i) {
2           i=3;
3   }
4   int main() {
5           int i=0;
6           f(i);
7           return i;
8   }
```

Quelle est la valeur retournée par la fonction `main` ?
Q2 On considère le code suivant :

```
1   void f(int & i) {
2           i=3;
3   }
4   int main() {
5           int i=0;
6           f(i);
7           return i;
8   }
```

Quelle est la valeur retournée par la fonction `main` ?
Q3 On considère le code suivant :

```
1   void f(int * i) {
2           *i=3;
3   }
4   int main() {
5           int i=0;
6           f(&i);
7           return i;
8   }
```

Quelle est la valeur retournée par la fonction `main` ?

Ex 2.2 * Opérateur et constructeur

Qu'affiche le programme ci-dessous ? Commentez le résultat.

```
1   #include <iostream>
2   using namespace std;
3   class C {
4   public:
5       C() { i=1;}
6       C(const C & c) { i=2;}
7       C & operator=(const C & c) { i=3; return *this;}
8       friend ostream & operator<<(ostream & os,const C & c) {
9           return os << c.i;
10      }
11  private:
12      int i;
13  };
14  int main() {
15      C c1;
16      C c2 = c1;
17      C c3;
18      c3=c1;
19      cout << c1 << " " << c2 << " " << c3 << endl;
20      return 0;
21  }
```

Ex 2.3 ** Validité de pointeur

On se donne la classe `Voiture` définie ci-dessous :

```
1   class Voiture {
2   public:
3           Voiture (int v) {
4                   valeur=v;
5           }
6           ~Voiture () {
7                   valeur=-1;
8           }
9           int valeur;
10  };
```

Q1 Que retourne la fonction `main` ci-dessous :

```
11  Voiture * donneAdresse() {
12          Voiture v(32);
13          return &v;
14  }
15  int main () {
16          Voiture * ptr = donneAdresse();
17          return ptr->valeur;
18  }
```

Q2 Si vous êtes sur un PC sous linux utiliser le logiciel `valgrind` pour diagnostiquer l'erreur dans le code précédent.

Ex 2.4 ** Efficacité du `switch`

On se donne le programme suivant `p.cc` :

```
1   int f(int i) {
2     switch (i) {
3     case 1: return 2;
4     case 2: return 3;
5     case 3: return 4;
6     case 4: return 5;
7     case 5: return 6;
8     case 6: return 7;
9     case 7: return 8;
10    case 8: return 9;
11    case 9: return 10;
12    default : break;
13    }
14    return 11;
15  }
```

Q1 Compiler le programme à l'aide de l'instruction `g++ -c -S p.cc`. Expliquer pourquoi dans le fichier assembleur `p.s` le temps d'accès aux instructions du `swicth` se fait en temps constant. Pour information, dans le fichier assembleur les valeurs explicites de nombres commencent par `$`, les registres commencent par le caractère `%`.

La documentation complète sur l'assembleur GNU se trouve à l'url :

`http://sourceware.org/binutils/docs-2.16/as/index.html`

Q2 Observez les différences et commentez l'assembleur du programme suivant où les cas d'entrée sont sur une plage d'entiers beaucoup plus grande :

```
1   int f(int i) {
2     switch (i) {
3     case 132: return 2;
```

```
4       case 232: return 3;
5       case 312: return 4;
6       case 4432: return 5;
7       case 542: return 6;
8       case 6234: return 7;
9       case 7234: return 8;
10      case 8543: return 9;
11      case 9364: return 10;
12      default : break;
13      }
14      return 11;
15    }
```

2.5 Exercices sur le système Jasip

Les exercices suivants servent à se familiariser avec le système Jasip. Ils n'ont pas de rapport avec SystemC. Ils nécessitent l'installation de l'émulateur Jasip, disponible à l'url :

`http ://www.derepas.com/csml/jasip_0.1.tar.bz2`

Pour installer Jasip dans `~/jasip`, effectuer les commandes suivantes :

```
tar xvfj jasip_0.1.tar.bz2
cd jasip_0.1
./configure
make
make install
```

Dans le répertoire `~/jasip` on a alors trois répertoires :

- `bin` qui contient l'éxécutable `jasip`,
- `lib` qui contient les fichiers byte code du système Jasip
- `src` qui contient le sources des fichiers dans `lib`.

Ex 2.5 * alphabet

Ecrire une classe java dont l'exécution affiche l'alphabet. On se servira de la méthode `java org.jasip.Jasip.term.setChar` pour l'affichage. Remarque : pour que le programme ne se termine pas tout de suite il faut faire une boucle infinie.

Une aide est fournie dans les commenataires de la classe org.jasip.Jasip située dans le fichier `src/org/jasip/Jasip.java`.

On pourra par exemple créer la solution dans la classe org.jasip.app.alphabet.Alphabet et l'écrire dans le fichier

```
~/jasip/src/org/jasip/app/alphabet/Alphabet.java
```

La compilation de la classe s'effectue alors à l'aide de la commande :

```
cd ~/jasip
javac -cp lib -g -sourcepath src/org/jasip/app/alphabet \
         -d lib src/org/jasip/app/alphabet/*.java
```

Les options de compilation utilisées sont décrite dans la fiche pratique 2 : compilation java. La classe peut alors s'exécuter avec la machine virtuelle par la commande :

```
~/jasip/bin/jasip org/jasip/app/alphabet/Alphabet.class
```

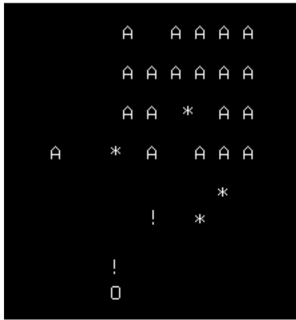
Ex 2.6 ** écho

Le but de l'exercice est d'écrire un programme java dans le système Jasip affichant sur l'écran les caractères frappés au clavier.

Q1 Nous allons tout d'abord afficher les caractères au même endroit. Ecrire un programme qui charge le registre `r1` avec le contenu de l'adresse 0x300. Si `r1` est différent de zéro alors, une touche a été enfoncée on peut écrire le contenu de `r1` à l'adresse 0x310 pour que l'affichage s'effectue.

Q2 Incrémenter l'adresse où `r1` est écrite, pour pouvoir écrire sur toute la ligne.

Ex 2.7 *** Space invader



Le but de l'exercice est d'écrire un jeu de type « space invader » en dans le système Jasip. Un vaisseau spatial (modélisé par la lettre `O`) en bas de l'écran est déplacé à droite ou à gauche par l'utilisateur. La touche espace permet de lancer des missiles (caractères `!`) sur les aliens (les caractères `A`). Les aliens bougent alternativement en groupe de gauche à droite de l'écran et lancent des bombes (caractères `*`).

Q1 Ecrire une classe `org.jasip.app.si.Item` qui va représenter un objet par un caractère à l'écran. On pourra utiliser une classe avec la struture suivante :

```
package org.jasip.app.si;
public class Item {
     public Item(char c,int x, int y);
     public void display(org.jasip.TerminalInterface t);
     public void moveLeft(org.jasip.TerminalInterface t);
     public void moveRight(org.jasip.TerminalInterface t);
     public void moveUp(org.jasip.TerminalInterface t);
     public void moveDown(org.jasip.TerminalInterface t);
}
```

Q2 Ecrire une classe `AlienSet` qui stocke l'ensemble des aliens à afficher à l'écran dans un tableau d'éléments de type `org.jasip.app.si`. Cette classe sera également en charge de déplacer les aliens de droite à gauche.

Q3 Ecrire une classe MissileSet affichant les missiles enovoyés par le vaisseau. Cette classe sera également en charge de déplacer les missiles vers le haut.

Q4 Ecrire une classe BombSet affichant les bombes enovoyées par les aliens. Cette classe sera également en charge de déplacer les bombes vers le bas.

Q5 Assembler les classes précédentes pour réaliser le space invader.

2.6 Exercices SystemC

Le but de cet exercice est de mettre en oeuvre un simulateur pour la mémoire du système Jasip.

Ex 2.8 ** Simulateur mémoire

Un squelette contenant le code du processeur et des périphériques est disponible à l'url :

`http ://www.derepas.com/csml/jasip_sim_0.1.tar.bz2`

Pour l'installation effectuer les commandes suivantes :

```
tar xvfj jasip_sim_0.1.tar.bz2
cd jasip_sim_0.1
./configure --with-systemc=/répertoire/vers/systemc
make
```

Pour le répertoire vers l'installation SystemC on pourra utiliser /users/profs/info/derepas/systemc

Le but est d'écrire les classes SystemC RAM et Cache décrites à la section 2.3.1. Il faut pour cela mettre à jour les fichers `ram.h` et `cache.h` dans le répertoire `jasip_sim_0.1`.

Une fois ces classes écrites on peut les tester à l'aide de la commande :

```
jasip_sim -l org/jasip/app/alphabet/Alphabet.class
jasip_sim -l org/jasip/app/si/SpaceInvaders.class
```

Chapitre 3

Test et intégration

Ce chapitre présente la librairie standard C++. Les problématiques de test et d'intégrations sont présentés.

3.1 Exemple d'utilisation de la STL

La STL (Standard Template Library) est une librairie importante qui fait partie de la norme C++ [iso03]. Elle définit une interface standard pour effectuer des manipulations communes : listes d'objets, tables de hachage. Une bonne documentation en ligne sur la STL se trouve à l'url :

`http://www.sgi.com/tech/stl`

Nous donnons tout d'abord quelques exemples d'utilisation de la STL avant d'en décrire l'architecture générale.

3.1.1 Chaînes de caractères

Les chaînes de caractères sont conservées dans les type `std::string`. Voici un usage de la classe string :

```
1   #include <string>
2   #include <iostream>
3   using namespace std;
4   int main() {
5     string s1 = "conception de systèmes";
6     cout << s1 << endl;
7     cout << "taille de s1=" << s1.size() << endl;
8     s1 += " matériels et logiciels";
9     cout << s1 << endl;
10    cout << "taille de s1=" << s1.size() << endl;
11    cout << "5eme catactère de s1 =" << s1[4] << endl;
12    cout << "la séquence 'sys' commence au caractère "
13         << s1.find("sys") << endl;
14    if (s1.find("blabla")==string::npos) {
15      cout << "séquence 'blabla' non trouvée" << endl;
16    }
17    return 0;
18  }
```

Voici le résultat produit par l'exécution du programme ci-dessous

```
conception de systèmes
taille de s1=22
conception de systèmes matériels et logiciels
taille de s1=45
5eme catactère de s1 =e
la séquence 'sys' commence au caractère 14
séquence 'blabla' non trouvée
```

Nous avons vu à la section 2.2.5 la structure des chaînes de caractères en langage C. Pour obtenir ce type de chaîne, terminée par le caractère zéro, il faut invoquer la méthode `c_str` :

```
string s = "foo";
char * cStyleString = s.c_str();
```

Ainsi `cStyleString [0]` vaut `'f'` et `cStyleString [3]` vaut zéro.

3.1.2 Ecrire dans une chaîne de caractères

Quand on souhaite mettre plusieurs données dans une chaîne de caractères et non les afficher à l'écran on peut utiliser un objet de type `std::ostringstream`. Voici un exemple écrivant dans une chaîne de caractères la valeur décimale et hexadécimale d'un entier `i` :

```
1  #include <string>
2  #include <iostream>
3  #include <sstream>  // contient le type ostringstream
4  #include <iomanip>  // contient setbase
5  using namespace std;
6  int main() {
7    int i=65;
8    ostringstream oss;
9    oss << "i vaut " << i << " en décimal, soit Ox"
10       << setbase(16) << i << " en hexa." ;
11   string s = oss.str();
12   cout << s << endl;
13   return 0;
14 }
```

On remarque que l'on peut écrire dans un objet de type `ostringstream` comme on écrit sur la sortie standard ou comme dans un fichier via un objet de type `ofstream`. Voici le résultat produit par l'exécution du programme :

```
i vaut 65 en décimal, soit Ox41 en hexa.
```

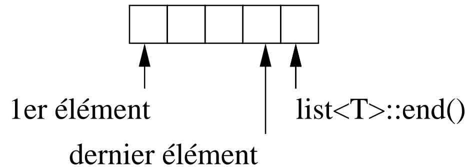
3.1.3 Listes

Une structure souvent utilisée est la liste. Les listes sont définies par le type template `list<class T>`. Voici un exemple où nous créons une liste contenant des éléments de type `std::string` :

```
1  #include <list>
2  #include <string>
3  #include <iostream>
4  int main() {
5    std::list<std::string> maListe;
6    maListe.push_back("Pomme");
7    maListe.push_back("Poire");   // la liste contient [Pomme, Poire]
8    maListe.push_front("Orange"); // la liste contient [Orange, Pomme, Poire]
9    // boucle pour afficher tous les éléments
10   for (std::list<std::string>::iterator i = maListe.begin();
11        i!=maListe.end();++i) {
12     std::cout << *i << std::endl; // affiche l'élément pointé par i
13   }
14   return 0;
15 }
```

Des nouveaux éléments sont ajoutés en tête de la liste avec la méthode `push_front` et en fin de liste à l'aide de la méthode `push_back`. Pour parcourir les éléments de la liste on a besoin d'un itérateur de type `list<T>::iterator`, obtenu à l'aide de la méthode `begin`. L'itérateur passe à l'élément suivant (respectivement précédent) à l'aide de l'opérateur `++` (respectivement à l'aide de l'opérateur `--`). Pour obtenir la valeur pointée par un itérateur il faut utiliser l'opérateur unaire `*`.

Un itérateur sur la liste est à la fin de la liste quand il vaut le résultat renvoyé par `list<T>::end()`. L'élément pointé est alors en dehors de la liste, et déréférencer l'itérateur n'a alors pas de sens. Voici un schéma résumant ce fonctionnement :



Pour supprimer un élément il suffit d'appeler la méthode `remove`, soit pour l'exemple précédent :

```
maListe.remove("Poire");  // supprime TOUTES les occurrences de "Poire"
```

3.1.4 Associations

Les associations sont effectuées à l'aide de la classe template `std::map<class Cle,class Valeur>`.

```
1  #include <map>
2  #include <string>
3  #include <iostream>
4  using namespace std;
5  int main() {
6    // donne le bureau dans lequel se trouve une personne
7    map<string,int> bureau;
8    bureau["anne"]=314;
9    bureau["chris"]=132;
10   bureau["laure"]=32;
11   bureau["léon"]=423;
12   bureau["didier"]=342;
13   cout << "taille de la table " << bureau.size() << endl;
14   map<string,int>::iterator i = bureau.find("laure");
15   if (i!=bureau.end()) { // un bureau a bien été attribué à laure.
16     cout << i->first << " est dans le bureau " << i->second << endl;
17   }
18   return 0;
19 }
```

Voici le résultat de l'exécution du programme ci-dessous :

```
taille de la table 5
laure est dans le bureau 32
```

Pour parcourir tous les éléments de l'association on utilise une syntaxe d'itérateur très proche de la liste :

```
for (map<string,int>::iterator i = bureau.begin();
     i!=bureau.end();++i) {
  cout << i->first << " est dans le bureau " << i->second << endl;
}
```

Pour supprimer le bureau de Didier il suffit d'effectuer les instructions :

```
map<string,int>::iterator i = bureau.find("didier");
if (i!=bureau.end()) { // didier a bien un bureau
  bureau.erase(i);
}
```

3.2 Architecture de la STL

3.2.1 Containers

Les containers sont des types pouvant conserver d'autres objets (ses éléments). La « vie » d'un élément contenu ne peut dépasser celle du contenant, pour cette raison on utilise souvent les containers avec des pointeurs. Pour tout objet `X` on va trouver entre autre les types suivants :

- X::iterator un type itérateur vers les éléments contenus.
- X::const_iterator un type permettant seulement d'observer les éléments du container.
- X::value_type le type des données conservées.

Voici les méthodes que l'on trouve souvent utilisées sur une instance a du container :

- a.begin() premier itérateur sur les éléments de a. Si a est constant alors le type renvoyé est X::const_iterator, sinon X::iterator.
- a.end() dernier itérateur sur les éléments de a. Si a est constant alors le type renvoyé est X::const_iterator, sinon X::iterator.
- a.size() renvoie le nombre d'éléments du container.

Les types suivant sont des containers : vector, deque, list, set, map, multiset multimap. L'idée derrière cette architecture est de pouvoir facilement passer d'un vecteur à une liste à un ensemble car les interfaces de ces containers sont relativement similaires.

3.2.2 Presque des containers

Le type std::string peut être vu presque comme un container. En effet un type de base basic_string constitue en fait l'implémentation des chaînes de caractères pour différents types de caractères possibles. Le type std::string est une instanciation de template particulier pour le type char :

```
template <class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
  // ....
};
typedef basic_string<char> string;
```

On peut ainsi définir d'autres types de chaînes de caractères :

```
struct CChar {
  //  définition des caractères chinois.
};
typedef basic_string<CChar> Cstring;
```

Cependant les basic_string ne sont pas des containers car ils imposent des contraintes sur les éléments qu'ils contiennent. Une classe dont nous ne développerons pas l'usage ici permettant d'implémenter des caractères est char_traits.

3.2.3 Itérateurs

On peut voir les itérateurs comme une généralisation des pointeurs. Ils sont souvent utilisés pour parcourir des groupes d'éléments. L'uniformité d'interface entre les itérateurs permet facilement de mettre en œuvre des algorithmes indépendamment des structures sur lesquelles ils s'appliquent. Ainsi pour écrire une fonction for_each_element qui applique un objet fonctionnel à une série d'éléments compris entre deux itérateurs il suffit d'écrire :

```
1  template<typename InputIter, typename Function>
2  void for_each_element(InputIter first, InputIter last, Function f) {
3      for ( ; first != last; ++first) f(*first);
4  }
```

Cette fonction template peut alors très facilement être appliquée à une liste ou tout autre objet disposant d'un itérateur :

```
5  #include <list>
6  class Times2 { // objet fonctionnel doublant la valeur pointée
7  public:
8      void operator() (int *i) { *i= 2*(*i); }
9  };
10 class DeleteInt { // objet fonctionnel libérant la mémoire
11 public:
12     void operator() (int *i) { delete i; }
13 };
14 int main() {
```

```
15      // création d'une liste de 100 éléments aléatoires
16      list<int*> l;
17      srand(0);
18      for (int i=0;i<100;++i) l.push_front(new int(rand()));
19      // doubler tous les éléments de la liste
20      Times2 times2;
21      for_each_element(l.begin(),l.end(),times2);
22      // libérer la mémoire allouée
23      DeleteInt di;
24      for_each_element(l.begin(),l.end(),di);
25      return 0;
26  }
```

La fonction que nous venons de créer : `for_each_element`, existe en fait déjà dans les algorithmes de la STL et est nommée `for_each`.

3.2.4 Algorithmes

Nous venons de constater comme les interfaces extrêmement structurées de la STL nous permettent d'écrire des algorithmes génériques. Les algorithmes fournis par la STL forment une partie importante et complémentaire des structures de données exposées jusqu'ici.

Nous avons déjà vu l'algorithme `for_each`.

Un autre algorithme est le tri. Il faut que les opérations d'addition et de soustraction entre itérateurs soient implémentées. Voici un exemple utilisant la classe `vector` et affichant 100 nombres tirés au hasard triés aléatoirement.

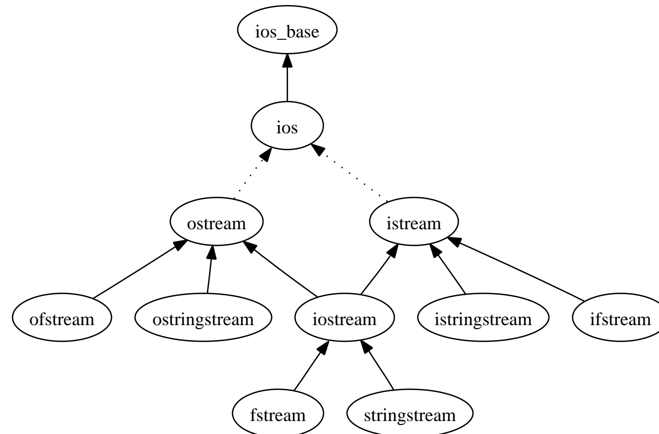
```
1   #include <vector>
2   #include <iostream>
3   using namespace std;
4   class Print {
5   public:
6     void operator() (int i) { cout << i << endl; }
7   };
8   int main() {
9     vector<int> l (100);
10    srand(0);
11    for (int i=0;i<100;++i) l[i]=rand();
12    sort(l.begin(),l.end());
13    Print p;
14    for_each(l.begin(),l.end(),p);
15    return 0;
16  }
```

La classe `list` possède une méthode `sort` qui lui est propre. On trouve d'autre types de méthode de tri comme `stable_sort` qui garantit que l'ordre des éléments égaux est préservé, ou `partial_sort` ne triant que les *n*-plus petits éléments.

On trouve également un algorithme nommé `binary_search` permettant de savoir en temps logarithmique si un élément est dans une séquence déjà triée.

3.2.5 Flux d'entrée/sortie

Voici la hiérarchie complète des classes standard utilisées pour les entrée sorties :



Voici les déclarations correspondant à la hiérarchie ci-dessus ainsi que les fichiers d'entêtes dans lesquels ces classes sont déclarées :

```
#include <iostream>
class ios;
class istream : virtual public ios;
class ostream : virtual public ios;
class iostream : public istream, public ostream;

#include <sstream>
class stringstream : public iostream;
class istringstream : public istream;
class ostringstream : public ostream;

#include <fstream>
class fstream : public iostream;
class ifstream : public istream;
class ofstream : public ostream;
```

Ces définitions ne sont pas exactes. En effet par exemple un flux `ostream` est un mécanisme qui permet de convertir des objets en une séquence de caractères. Il y a une façon normalisée de représenter les caractères en C++ : les `char_traits` déjà utilisée par le type `basic_string`. Ainsi les flux `ostream` sont, en fait, le résultat de la séquence de déclarations suivante :

```
template < class Ch, class Tr = char_traits<Ch> >
class std::basic_ostream : virtual public basic_ios<Ch,Tr> {
    // ....
};
typedef basic_ostream<char> ostream;
```

3.3 Style de codage modulaire

Nous avons vu à travers différents exemples dans la STL les bénéfices à utiliser une architecture modulaire. Nous allons décrire comment de manière générale essayer de respecter ce style en C++. Ceci sera ensuite appliqué à SystemC dans le chapitre suivant avec la mise en place des canaux hiérarchiques.

3.3.1 Interfaces

Une notion fréquemment utilisée présente dans Java et C# est la notion d'interface. Cette notion est souvent utilisée en particulier avec des interfaces graphiques : un composant représentant un bouton va par exemple posséder une interface lui permettant de réagir à un évènement du clavier mais également une autre interface lui permettant de réagir à un évènement de la souris.

Cette notion existe également en C++. Il suffit de créer des classes ne possédant que des méthodes virtuelles « pures », c'est à dire sans implémentation. L'attribut « pure » est dénoté par `=0` à la fin du prototype de la fonction (cela suggère que la valeur du pointeur associé à cette méthode n'est pas valide). Voici un exemple d'interface :

```
1   class TraiteurEvenementClavier {
2   public:
3     virtual void traiteClavier(char touche) = 0;
4   };
5
6   class TraiteurEvenementSouris {
7   public:
8     virtual void traiteClic(int x, int y) = 0;
9   };
```

3.3.2 Héritage multiple

Maintenant nous souhaiterions disposer d'une classe `Bouton` implémentant les deux interfaces précédentes. C++ possède alors une caractéristique que ne possèdent ni Java ni C# c'est l'héritage multiple.

Ainsi pour déclarer une classe `Bouton` mettant en œuvre ces deux interfaces on peut écrire :

```
10  #include <iostream>
11  class Bouton :
12    public TraiteurEvenementClavier,
13    public TraiteurEvenementSouris
14  {
15    void traiteClavier(char touche) {
16      std::cout << "Traitement touche " << touche << std::endl;
17    }
18    void traiteClic(int x,int y) {
19      std::cout << "Traitement clic x=" << x << " y=" << y << std::endl;
20    }
21  };
```

On peut alors utiliser cette classe partout où un `TraiteurEvenementClavier` ou un `TraiteurEvenementSouris` est attendu :

```
22  void appliqueEvenement(TraiteurEvenementClavier & tec, char touche) {
23    tec.traiteClavier(touche);
24  }
25
26  int main() {
27    Bouton b;
28    appliqueEvenement(b,'a');
29    return 0;
30  }
```

Le programme ci-dessus provoque l'affichage :

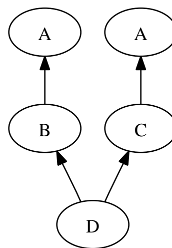
```
Traitement touche a
```

L'héritage multiple peut également avoir lieu pour des classes qui ne sont pas des interfaces mais possèdent bien des champs. Voici l'exemple :

```
1   #include <iostream>
2   using namespace std;
3   class A {
4   public :
5     A() { i=0;}
6     A(int _i) { i=_i;}
7     int i;
8   };
9   class B : public A {
10  public :
```

```
11    B(int _i) {i=_i;}
12  };
13  class C : public A {
14  public :
15    C(int _i) {i=_i;}
16  };
17  class D : public B, public C {
18  public:
19    D(int i, int j) : B(i), C(j) {}
20  };
21  ostream & operator<<(ostream & os, const D & d) {
22    return os << d.B::i << " " << d.C::i;
23  }
24
25  int main() {
26    D d(1,2);
27    cout << d << endl;
28    return 0;
29  }
```

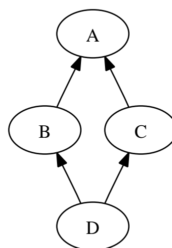
Le programme ci-dessus affiche `1 2`. Voici le diagramme de classes associé :



On remarque que les deux champs `i` d'une instance `d` de `D` sont accessibles par la syntaxe `d.B::i` et `d.C::i`.

3.3.3 Classes de base virtuelles

Pour certaines classes il peut être utile même si elles apparaissent plusieurs fois dans le diagramme d'héritage d'une classe, qu'il n'y ait qu'une seule instance, comme illustré à la figure ci-dessous :



Il faut alors rajouter le mot clé `virtual` au moment de l'héritage. Voici le code correspondant :

```
1   #include <iostream>
2   using namespace std;
3   class A {
4   public :
5     A() { i=0;}
6     A(int _i) { i=_i;}
7     int i;
8   };
9   class B : virtual public A {
10  public :
```

```
11      B(int _i) {i=_i;}
12    };
13    class C : virtual public A {
14    public :
15      C(int _i) {i=_i;}
16    };
17    class D : public B, public C {
18    public:
19      D(int i, int j) : B(i), C(j) {}
20    };
21    ostream & operator<<(ostream & os, const D & d) {
22      return os << d.B::i << " " << d.C::i;
23    }
24
25    int main() {
26      D d(1,2);
27      cout << d << endl;
28      return 0;
29    }
```

Le programme ci-dessus affiche `2 2`, contrairement à la section précédente où l'on obtenais `1 2` sans le mot clé `virtual`.

3.3.4 Application au codage modulaire

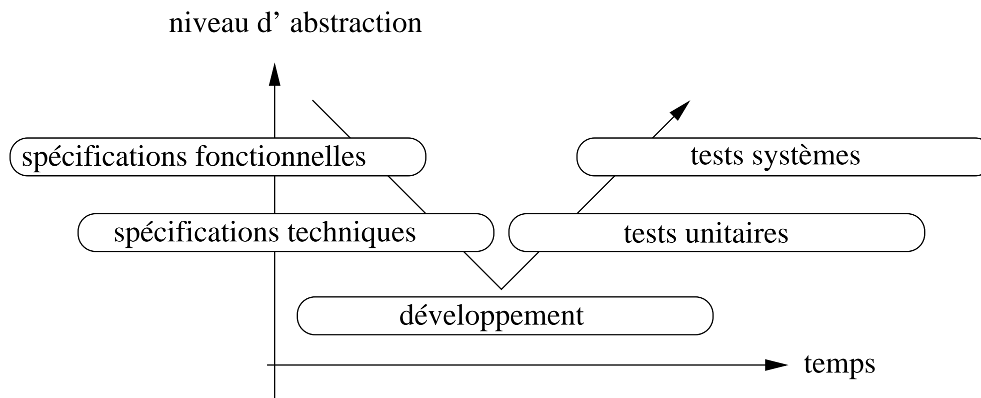
Voici une façon permettant de mettre en oeuvre ce mécanisme d'interface en utilisant l'héritage multiple :

```
class MaClasse :
        public monInterface,
        protected monImplementation
{
        // implémentation des fonctions requises par monInterface
        // à l'aide des méthodes de monImplementation.
};
```

3.4 Test des modèles SystemC

Il est important de vérifier que le modèle SystemC est conforme à l'idée que l'on s'en fait. De plus les modèles SystemC ne sont qu'une partie avant la création de modèles synthétisables VHDL ou Verilog. C'est donc dans un cycle de développement plus large qu'il faut considérer le modèle SystemC.

Considérons un cycle de développement de logiciel en « V » standard :



On part de spéficications fonctionnelles : l'idée est de décrire de manière la plus neutre technologiquement possible le travail qui doit être effectué. Il est saint à ce niveau de bannir toute ébauche ou suggestion de solution qui pourrait enfermer dans un carquant. C'est en général ce type de spécification qui est au début du cycle de développement.

Suite aux spécifications fonctionnelles on trouve des spécifications techniques qui décrivent les solutions techniques les mieux adaptées pour résoudre le problème exposé dans les spécifications fonctionnelles.

Ce découpage peut être répété sur plusieurs niveaux d'abstraction. Typiquement sur un projet d'une centaine d'hommes ans on trouvera tout d'abord des spécifications fonctionnelles puis techniques de haut niveau, puis des spécifications fonctionnelles et techniques de plus bas niveau pour chaque module identifié.

Une fois le développement effectué on va trouver des tests unitaires permettant de vérifier chaque module.

Puis surviennent des tests pour l'assemblage des différents modules. Le but de SystemC étant de faire communiquer facilement à haut niveau un nombre de composants pouvant être important, cet aspect d'intégration à haut niveau est important.

3.4.1 Test boite noire

Une boite noire est une boite dont on ne connaît pas le mécanisme interne. Typiquement si un automate peut modéliser la boite on ne s'attache pas à connaître les états de cet automate.

On va donc uniquement se focaliser sur les échanges avec l'extérieur pour savoir si l'implémentation à tester est conforme avec les spécifications fonctionnelles.

3.4.2 Test boite blanche

Au contraire du test boite noire, le test boite blanche va chercher à valider le fonctionnement interne d'un module SystemC. Ce type de démarche est soit utile pour valider que la simulation correspond bien à nos attentes (typiquement le module est bien passé par tel et tel état) ou pour valider le fonctionnement interne d'un module relativement critique.

3.5 Exercices C++

Ex 3.1 * Mauvaise utilisation des itérateurs

Considérons le programme suivant :

```
1   #include <list>
2   using namespace std;
3
4   template<class T>
5   void moveElementsInOtherList(list<T> & l,list<T> & otherList) {
6     for (typename list<T>::iterator i = l.begin(); i!=l.end() ;++i) {
7       otherList.push_back(*i);
8       l.erase(i);
9     }
10  }
11
12  int main() {
13    list<int> l1,l2;
14    l1.push_back(1);
15    l1.push_back(2);
16    l1.push_back(3);
17    moveElementsInOtherList(l1,l2);
18    return 0;
19  }
```

Il dispose d'une fonction template `moveElementsInOtherList` supprimant les éléments de `l1` et les ajoutant à la fin de `l2`.

Q1 Ce programme provoque une erreur d'exécution. Expliquer pourquoi. On pourra se servir du logiciel `valgrind` pour valider la réponse.

Q2 Rédiger une version correcte de la fonction `moveElementsInOtherList`.

Ex 3.2 ** Performance des containers associatifs

On se donne la fonction suivante renvoyant une chaine de caractères aléatoire :

```
#include <stdlib.h>
#include <string>

using namespace std;

string getRandomString(int size) {
  char buf [size+1];
  for (int i=0;i<size;++i)
    buf[i]=65+(int)(26.0*(rand()/(RAND_MAX+1.0)));
  buf[size]=0;
  return string(buf);
}
```

Q1 Ecrire un programme prenant en argument un entier *n*, qui crée un container de type `map<string,string>` ayant *n* entrées.
Q2 Effectuer 10 millions de fois des recherches dans le container définit à la question précédente.
Q3 Mesurer les résultats pour différentes valeurs de *n*. Pour mesurer la durée d'un bloc d'instruction on pourra utiliser la fonction `gettimeofday`:

```
#include <sys/time.h>

...
  struct timeval start;
  struct timeval end;
  gettimeofday(&start,NULL);
  {
    // bloc d'instruction à mesurer
  }
  gettimeofday(&end,NULL);
  // durée du bloc en micro secondes :
  int duree = (end.tv_sec-start.tv_sec)*1000000+(end.tv_usec-start.tv_usec);
```

3.6 Exercices SystemC

Ex 3.3 ** Test des modules ram et cache

Réaliser un module testant les modules créés à l'exercice 2.6.

Chapitre 4

Réseau sur puce

Après un bref complément sur C++, ce chapitre aborde les structures internes utilisées dans SystemC. Les réseaux sur puce sont présentés. Enfin les exercices visent à simuler un réseau sur puce pouvant servir à faire un processeur multicœur.

4.1 Complément C++

4.1.1 Retour sur la STL : compléxité des opérations

Collections ordonnées

- `vector` c'est comme un tableau C mais dynamique, c'est à dire dont la taille peut varier, et pour lequel l'accès à une case arbitraire se fait en temps constant. En revanche l'ajout ou la suppression d'un élément au milieu se fait en temps linéraire. L'ajout ou la suppression d'un élément au début ou la fin se fait en moyenne en temps constant.
- `list` il s'agit d'une liste doublement chaînée, les éléments n'étant pas conservés dans des zones adjacentes de la mémoire. L'ajout ou la suppression d'éléments à n'importe quel endroit se fait donc en temps constant. En revanche l'accès à un élément donné est en moyenne linéaire.

4.1.2 Exceptions

Un des traits du langage C++ qui n'a pas été abordé jusqu'ici sont les exceptions. Les exceptions sont, comme le nom l'indique des structures permettant de gérer les cas exceptionnels.

La sémantique est d'essayer un bloc d'instructions C++ et de « rattraper » des erreurs qui pourraient survenir pendant l'exécution de ces instructions. Les informations sur les erreurs survenues sont des classes arbitraires. Le bloc d'instruction à exécuter commence par le mot clé `try`. La récupération des erreurs s'effectue par le mot clé `catch` suivit du type d'exception levée. Voici un exemple : définissons tout d'abord un type `ErreurInterne` portant les informations que nous souhaitons remonter : la raison de l'erreur, la position dans le fichier où l'erreur s'est produite.

```
1  #include <iostream>
2  using namespace std;
3  class ErreurInterne {
4  public:
5    ErreurInterne(string _raison,string _nomFichier, int _ligne)
6          : raison(_raison), nomFichier(_nomFichier), ligne(_ligne) { }
7    friend ostream & operator<<(ostream & os, const ErreurInterne & e) {
8      os << e.nomFichier << ":" << e.ligne << ": " << e.raison << endl;
9    }
10 protected:
11   string raison;
12   string nomFichier;
13   int ligne;
14 };
```

Pour déclencher une exception il suffit alors d'utiliser l'instruction `throw` suivie de l'instance du type désiré. Voici un exemple avec une fonction nommée `doubleIfPositive` qui déclenche une exception quand son argument est négatif.

```
15  int doubleIfPositive(int i) {
16    if (i<0) throw ErreurInterne("i est négatif",__FILE__,__LINE__);
17    return 2*i;
18  }
19
20  int main() {
21    try {
22      int i=doubleIfPositive(-314);
23    } catch (ErreurInterne e) {
24      cerr << "Erreur " << e;
25    }
26    return 0;
27  }
```

L'exécution du programme ci-dessous déclenche une exception qui produit l'affichage suivant :

```
Erreur exception.cc:19: i est négatif
```

Dans l'exemple les instructions aux lignes 20 et 26 ne sont jamais exécutées car l'exception est déclenchée avant leur exécution.

4.1.3 Coût des méthodes virtuelles

Considérons le programme suivant qui effectue 4.10^9 appels à une méthode virtuelle :

```
1   #define VIRTUAL virtual
2   class A {
3   public:
4     A(int _i) : i(_i) {}
5     VIRTUAL int getValue() { return i; }
6   protected:
7     int i;
8   };
9   int f(A & a) {
10    return a.getValue()+ a.getValue()+ a.getValue()+ a.getValue();
11  }
12  int main() {
13    A a (314);
14    for (int i=0;i<1000000000;++i) {
15      int j=f(a);
16    }
17    return 0;
18  }
```

Si la macro `VIRTUAL` est définie à une valeur vide on compile le programme sans méthode virtuelle. Voici le résultat d'exécution :
- sans méthode virtuelle le temps d'exécution est de 23,8 secondes,
- avec une méthode virtuelle le temps est de 29,2 secondes.

Cette différence s'explique par l'implémentation utilisée pour mettre en oeuvre les méthodes virtuelles. En effet le fait de rajouter le mot clé `virtual` à une méthode de `A` fait que le compilateur va créer une table des méthodes virtuelles associée à `A`. Elle va contenir ici qu'une entrée, puisque `A` ne dispose que d'une méthode virtuelle, cette entrée étant un pointeur vers la méthode à exécuter.

Si une class `B` hérite de `A` , elle va disposer de sa propre table des méthodes virtuelles. Elle comporte également une entrée qui est un pointeur de méthode vers la méthode `getValue` surchargée dans `B`.

Lors de l'appel à la méthode `getValue` dans la fonction `f` il faut tout d'abord rechercher dans la table le pointeur de méthode puis l'appeler. Il y a donc un coût plus important avec une méthode virtuelle que sans.

4.1.4 Copie en ligne

Les fonctions ou les méthodes peuvent être déclarées `inline`. Cela signifie que quand elle sont appelées dans le code C++, au lieu d'être appelées dans le code assembleur, le code assembleur sera inséré au lieu de faire l'appel, ceci permet de gagner en performance pour des fonctions qui sont appelées souvent à partir du même endroit dans le programme.

D'un point de vue syntaxique il suffit de rajouter le mot clé `inline` devant la déclaration de la fonction. En voici un exemple :

```
1  inline int f(int i,int j, int k, int l, int m) {
2    return i+j+k+l+m;
3  }
4  int main() {
5    int j=0;
6    for (int i=0;i<1000000000;++i) {
7      j+=f(i,i+1,i+2,i+3,i+4);
8    }
9    return j;
10 }
```

Le programme a été compilé avec un niveau d'optimisation de 2. Le temps d'exécution du programme pour la version `inline` : 1.9 s. Le temps d'exécution du programme pour la version non `inline` : 7.9 s. Le gain important s'explique ici par le nombre d'arguments de la fonction (autant de recopies en moins) et les simplifications sur les opérations arithmétiques. La contrepartie de l'usage de méthodes `inline` est l'augmentation de la taille du code.

4.2 Structures internes de SystemC

Pour mieux comprendre la description d'un modèle SystemC il est important de comprendre ce que font les macros utilisées. Ceci est par exemple nécessaire si l'on souhaite disposer d'un module dont le constructeur a des arguments spécifiques : on ne peut utiliser la macro `SC_CTOR` mais uniquement une version expansée de celle-ci.

4.2.1 `SC_MODULE`

La définition d'un module se fait par `SC_MODULE(MonModule)`. Cette définition est équivalent à `struct MonModule : sc_module`. Ainsi les modules SystemC sont définis comme des classes SystemC héritant de `sc_module`.

On peut ne pas souhaiter utiliser la macro `SC_MODULE` si l'on souhaite que certains champs de la classe soient privés.

4.2.2 `SC_CTOR`

Le constructeur d'un module est déclaré par l'entête `SC_CTOR(MonModule)`. Cette déclaration est équivalente à :

```
typedef MonModule SC_CURRENT_USER_MODULE;
MonModule( sc_module_name)
```

Ainsi dans chaque module le nom du module est `SC_CURRENT_USER_MODULE`. Si on souhaite disposer d'un constructeur ayant différents paramètres, au lieu d'utiliser `SC_CTOR` il suffit de déclarer :

```
typedef MonModule SC_CURRENT_USER_MODULE;
MonModule(sc_module_name * param1, int param2) {
   // corps du constructeur du module
}
```

Omettre le `typedef` sur `SC_CURRENT_USER_MODULE` provoquerait une erreur, car cette définition de type est utilisée dans `SC_METHOD`, `SC_THREAD` et `SC_CTHREAD`.

4.2.3 `SC_METHOD`

Pour qu'une méthode d'un module SystemC soit invocable par l'ordonnanceur SystemC, on utilise l'instruction `SC_METHOD (maMethode)`. Ceci est équivalent à :

```
declare_method_process(maMethode_handle, "maMethode",
                       SC_CURRENT_USER_MODULE, maMethode)
```

Voici le code de la macro `SC_METHOD` :

```
#define SC_METHOD(func)                                              \
    declare_method_process( func ## _handle,                         \
                            #func,                                   \
                            SC_CURRENT_USER_MODULE,                  \
                            func )
```

On remarque l'usage de l'opérateur de concaténation du préprocesseur `##`, ainsi que l'opérateur de mise sous chaîne de caractère `#`. Cette macro appelle `declare_method_process` qui est elle même une macro :

```
#define declare_method_process(handle, name, host_tag, func)          \
    {                                                                 \
        sc_method_handle handle = simcontext()->register_method_process( name,\
                     SC_MAKE_FUNC_PTR( host_tag, func ), this );      \
        sc_module::sensitive << handle;                               \
        sc_module::sensitive_pos << handle;                           \
        sc_module::sensitive_neg << handle;                           \
    }
```

Les opérations effectuées par la fonction `simcontext()->register_method_process` consistent à mettre dans la table de toutes les `SC_METHOD` un pointeur vers la méthode passée en argument.

4.3 Canaux hiérarchiques

Il existe plusieurs types de canaux en SystemC. Nous avons présenté `sc_signal` que l'on peut voir comme un signal électrique sur un cable mais également `sc_fifo` est plus élaboré.

On peut définir ses propres types comme étant des canaux. Il faut pour cela respecter les règles suivantes :

- déclarer une interface (classe ne possédant que des méthodes purement virtuelles cf. section 3.3.1),
- écrire le code du module SystemC qui va implémenter l'interface,
- définir les ports permettant à un module de communiquer via le canal.

Nous allons dans cette section définir un canal de communication à l'aide d'un `sc_module`. Ce canal va implémenter une pile, c'est à dire une structure lifo (par opposition à fifo) pour « last in first out ».

4.3.1 Définition de l'interface

Nous allons définir deux interfaces sur la pile : une pour l'écriture et une pour la lecture. Les interfaces doivent hériter de manière virtuelle (cf. section 3.3.3) de la classe `sc_interface`.

```
1  #ifndef STACK_IF_H
2  #define STACK_IF_H
3
4  #include "systemc.h"
5
6  class StackWrite_if : virtual public sc_interface {
7  public:
8    virtual bool nb_write(char) = 0;
9  };
10
11 class StackRead_if : virtual public sc_interface {
12 public:
13   virtual bool nb_read(char&) = 0;
14 };
15 #endif
```

4.3.2 Implémentation du canal

Le canal est un module implémentant les interfaces de lecture et d'écriture.

```
1   #ifndef STACK_H
2   #define STACK_H
3   #include "systemc.h"
4   #include "stack_if.h"
5
6   #define STACK_SIZE 20
7
8   class Stack :
9     public sc_module,
10    public StackWrite_if,
11    public StackRead_if
12  {
13  public:
14    Stack(sc_module_name nm) : sc_module (nm), top(0) {}
15    bool nb_write(char c) {
16      if (top<STACK_SIZE) {
17        data[top++]=c;
18        return true;
19      }
20      return false;
21    }
22    bool nb_read(char& c) {
23      if (top>0) {
24        c=data[--top];
25        return true;
26      }
27      return false;
28    }
29    void register_port(sc_port_base & _port,
30                       const char * _if_typename)
31    {
32      cout << "binding "<< _port.name() << " to "
33           << "interface: " << _if_typename << endl;
34    }
35  private:
36    int top;
37    char data[STACK_SIZE];
38  };
39  #endif
```

4.3.3 Utilisation des ports

```
1   #ifndef CONSUMER_H
2   #define CONSUMER_H
3   #include "systemc.h"
4   #include "stack_if.h"
5   SC_MODULE(Consumer) {
6     sc_port<StackRead_if> in;
7     sc_in_clk clock;
8     void prcRead() {
9       while (true) {
10        char c;
11        wait();
12        if (in->nb_read(c)) {
13          cout << "Lecture de " << c << endl;
14        }
15      }
```

```
16      }
17      SC_CTOR(Consumer) {
18        SC_THREAD(prcRead);
19        sensitive_pos << clock;
20      }
21    };
22    #endif
```

Le code complet pour faire fonctionner l'exemple est disponible à l'url :

`http://www.derepas.com/csml/canaux_hierarachiques.tar.gz`

4.4 Ajout de périphériques

Deux périphériques sont présents dans le système Jasip : le clavier et le terminal. Nous décrivons ici comment rajouter un périphérique pour accéder au système de fichiers.

4.4.1 Opérations à effectuer

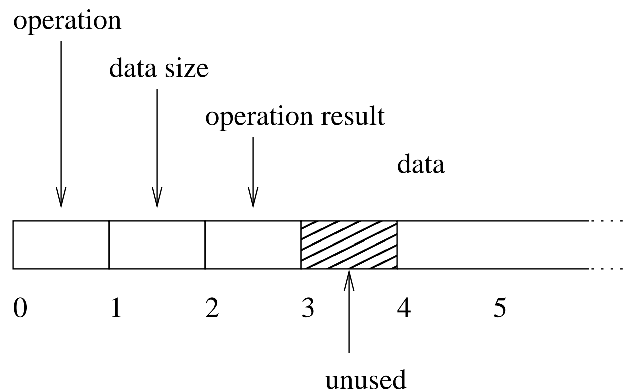
Déterminons d'abord les opéations à effectuer :
- ouvrir un fichier en écriture ou en lecture,
- lire ou écrire dans un fichier
- fermer un fichier ouvert

Il nous faut pour cela :
- une zone donnant les données nécessaires à l'opération
- une zone identifiant l'opération à effectuer
- une zone donnant le résultat de l'opération : s'est-elle bien déroulée ou non ?
- une zone donnant les données récupérées (quand on lit un fichier).

4.4.2 Mise en mémoire

Nous allons pour manipuler un fichier utiliser une zone mémoire de 128 bits comme suit :
- octet 0 : sert à choisir l'opération à effectuer (ouverture, écriture, lecture, etc...) . Le fait d'écrire à cette adresse déclenche l'opération. On dispose des opérations suivantes :
  - 1 : lecture dans le fichier ouvert
  - 2 : écriture dans le fichier ouvert
  - 4 : ouverture d'un fichier en lecture
  - 8 : ouverture d'un fichier en écriture
  - 16 : fermeture du fichier
- octet 1 : une fois l'opération effectuée cette adresse contient le résultat de l'opération.
- octet 2 : cet octet donne la taille en octets d'éléments dans le buffer.
- octet 3 : pas utilisé
- octet 4 à 127 : buffer de données

Ceci est résumé sur l'image suivante :



4.4.3 Exemples

Pour ouvrir un fichier il faut :

- écrire le nom du fichier dans octets 4 à 127.
- écrire le nombre de caractères dans le nom de fichier dans l'octet 2.
- écrire dans l'octet zéro l'opération à réaliser :
  - 8 pour une ouverture en écriture,
  - 4 pour une ouverture en lecture.
- maintenant que l'ouverture a eu lieu on peut lire l'octet numéro 1 pour savoir si l'opération s'est bien produite :
  - 0 si l'opération s'est bien effectuée.
  - 1 sinon.

Pour écrire dans un fichier :

- écrire les données à mettre dans le fichier dans octets 4 à 127.
- écrire le nombre d'octets écrits précédemment dans l'octet 2.
- écrire 2 à l'octet numéro zero.
- on peut alors consulter le résultat de l'écriture à l'octet 1.

Les autres opérations sont implémentées dans la classe `org.jasip.DiskDriver`. Le source de cette classe se trouve dans le répertoire `src` de la distribution jasip ou jasip_sim.

`http://www.derepas.com/csml/DiskDriver.java`

4.4.4 Zones mémoires

Le système jasip propose d'ouvrir 4 fichiers par microprocesseur :

- le premier fichier de `0x80` à `0xff`.
- le deuxième fichier de `0x100` à `0x17f`.
- le troisième fichier de `0x180` à `0x1ff`.
- le quatrième fichier de `0x200` à `0x27f`.

Ces plages mémoires viennent s'ajouter aux adresses `0x290` pour le clavier et la plage `0x300` à `0x61f` pour l'écran.

4.4.5 Une interface de plus haut niveau

La classe `org.jasip.DiskDriver` mentionnée précédemment permet également d'avoir un accès de plus haut niveau sans manipuler les adresses mémoires.

Voici comment écrire un fichier nommé `tmpFileName.txt` contenant la chaîne de caractères `"hello world!"`. Puis le relire pour vérifier son contenu :

```
import org.jasip.DiskDriver;

...

        // contenu du fichier
        String s = new String("hello world!");

        // écriture du fichier
        String fileName = new String("tmpFileName.txt");
        int fd = DiskDriver.open(fileName,DiskDriver.OPEN_MODE_WRITE);
        if (fd==-1) {
            System.out.println("Could not open file for writing.");
            return;
        }
        if (DiskDriver.write(fd,s)!=0) {
            System.out.println("Could not write to file.");
            return;
        }
        if (DiskDriver.close(fd)!=0) {
            System.out.println("Error closing file.");
            return;
```

```
}

// vérification du contenu du fichier
fd=DiskDriver.open(fileName,DiskDriver.OPEN_MODE_READ);
if (fd==-1) {
    System.out.println("Could not open file.");
    return;
}
String ss = DiskDriver.readWholeFile(fd);
if (ss==null) {
    System.out.println("Could not read file.");
    return;
}
if (!ss.equals(s)) {
    System.out.println("Error in the content of the file.");
    return;
}
if (DiskDriver.close(fd)!=0) {
    System.out.println("Error closing file.");
    return;
}
```

4.5 Réseaux sur puce

Nous avons déjà mentionné le fait que les composants électroniques devenaient de plus en plus intégrés. On trouve actuellement des systèmes sur puce ou SoC (System on Chip) dans une variété d'équipements (assistant numérique, voiture, ...). Utiliser un bus pour faire communiquer tous ces composants peut être mal adapté, en effet la capacité du bus est souvent limitée et le bus est mal adapté à une évolution rapide du nombre de composant connectés. Utiliser un système où chaque composant est relié à tous les autres est très coûteux et pas nécessairement adapté. Ainsi le réseau sur puce ou NoC (Network on Chip) tendent à être un compromis en coût et volutivit acceptable [GG00]. C'est en particulier important pour la réalisation d'un microprocesseur multicœur où le type de communication entre les microprocesseurs varie fortement en fonction de l'application considérée.

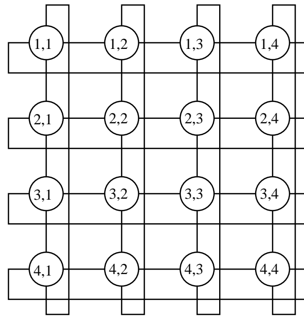
Il existe des environnements spécialisés basés sur SystemC pour modéliser les NoC comme par exemple OCCN `http://occn.sourceforge.net` [CCG+04]. Nous utiliserons dans les exercices suivant simplement les classes SystemC.

4.6 Exercices SystemC

Le but des exercices est de simuler un réseau de communication. Voici un rappel des différentes couches du modèle OSI :

- Couche physique : transport des zéros et des uns sur un support matériel.
- Couche de liaison physique : réalisation de trames. On suppose que la structure des trames est : noeud d'origine codé sur 8 bits, noeud de destination codé sur 8 bits, code correcteur d'erreur sur 8 bits, un paramètre nommé TTL (Time To Live) codé sur 8 bits qui est décrémenté de un à chaque passage dans un noeud, le paquet étant supprimé si ce champ vaut zéro, enfin des données codées sur 256 bits.
- Couche de réseau : acheminement des trames d'un noeud à l'autre, en tenant compte du noeud de destination.
- Couche de transport : s'assurer de l'intégrité des données, à l'aide d'un checksum sur 8 bits et renvoyer un acquittement de bonne ou mauvaise réception.

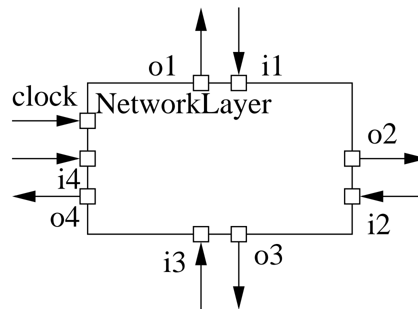
Le modèle OSI est un standard développé par l'ISO [DZ83], formalisant la notion de couche, aujourd'hui tombé quelque peu dans l'oubli par rapport à d'autres protocole comme Internet [Cla88] ou ATM [Kes97] mais dont la structure en couche se retrouve dans ces protocoles. Nous utilisons pour le réseau une topologie de tore comme représenté ci-dessous : les 16 noeuds de communication sont sur une grille de 4 par 4 et chaque noeud communique avec ses 4 voisins directs modulo 4.



Ex 4.1 ** Couche réseau du modèle OSI

Nous nous plaçons dans cet exercice au niveau de la couche réseau.

Q1 Ecrire la classe `NetworkLayer` possédant 4 entrées `i1`, `i2`, `i3` et `i4` et 4 sorties `o1`, `o2`, `o3` et `o4`, qui va être l'un des noeud du réseau présenté ci-dessous. On ajoutera de plus une entrée sur l'horloge du système permettant d'envoyer spontanément de manière aléatoire des paquets de données.



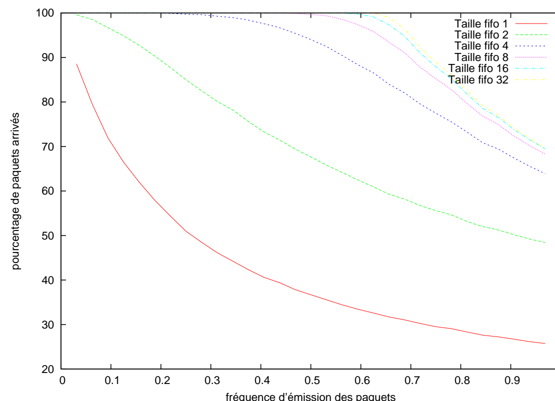
La classe `NetworkLayer` possédera les méthodes `setX(int)` et `setY(int)` permettant de fixer la place du noeud associé dans la topologie présentée dans l'énoncé, ou de manière alternative on pourra passer ces paraètres au constructeur du module en mettant à profit la section 4.2.2. On pourra télécharger la définition de la classe `Paquet` modélisant les paquets à l'url :

`http://wwww.derepas.com/csml/ex_modele_osi.tar.gz`

Si un nouveau paquet est émis alors une variable globale nommée `paquetSent` est incrémentée. Si un nouveau paquet se présente sur `i1`, `i2`, `i3` ou `i4` à destination du noeud alors une variable globale nommée `paquetReceived` est incrémentée. Si un nouveau paquet se présente sur `i1`, `i2`, `i3` ou `i4` à destination d'un autre noeud alors le paquet est envoyé vers `o1`, `o2`, `o3` ou `o4` en fonction de la destination.

Q2 Ecrire une fonction `sc_main` permettant d'instancier le réseau de 16 nœuds avec la topologie présentée dans l'énoncé. Afficher le nombre de paquets émis et le nombre de paquets reçus, en générant aléatoirement des paquets d'un nœud du réseau vers n'importe quel autre nœud.

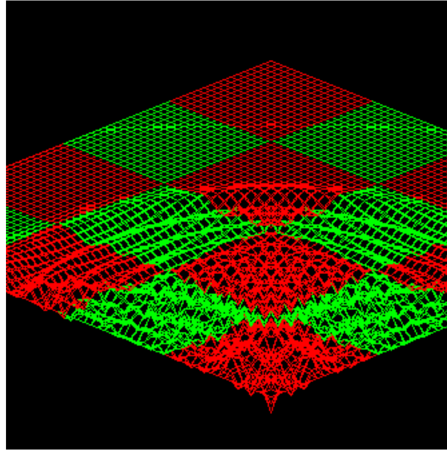
Ci-dessous figure les résultats de perte de paquets obtenus pour différentes valeurs de fréquence d'émission de paquets, et pour différentes tailles de fifo. On remarque qu'une taille de fifo de 8 permet d'avoir de très bon résultats, et que pour notre topologie il s'agit sans doute d'un bon compromis prix/performance.



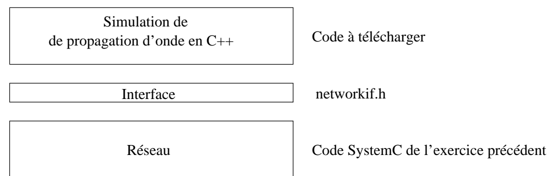
Expliquer la raison des pertes de paquets.

Ex 4.2 ** Utilisation du réseau

Dans l'exercice précédent nous avons développé un réseau pouvant comporter des pertes de paquets en cas d'engorgement. Ce type de réseau peut s'appliquer à des applications ayant un débit de communication constant ne générant pas d'engorgement, c'est ce que nous allons développer dans cet exercice en utilisant un code C++ effectuant du calcul de propagation d'onde, ce calcul étant distribué sur les différents nœuds du réseau. Ceci est illustré à la figure ci-dessous où les cases du damier vert et rouge représente les différents nœuds du réseau qui calculent la propagation de l'onde :



Nous allons utiliser la possibilité offerte par SystemC de reprendre un code existant. Voici l'architecture générale :



Le code de calcul de propagation d'onde est à télécharger à l'url :

`http://www.derepas.com/csml/ex_onde_ennonce.tar.gz`

Voici le contenu du fichier `networkif.h` définissant les interfaces à respecter :

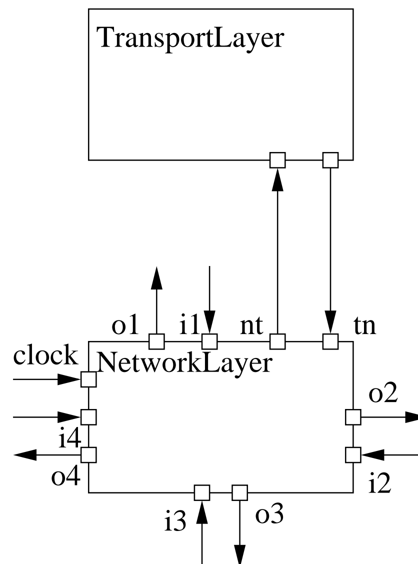
```
1  #ifndef NETWORKIF_H
2  #define NETWORKIF_H
3  #include "paquet.h"
4  class NetworkIf;
5  /**
6   * Définition de l'interface d'une classe
7   * pouvant se connecter sur le réseau.
8   */
9  class NetworkPlugableIf {
10 public:
11   virtual void setNetwork(NetworkIf*)=0;
12   virtual void readPaquet(Paquet)=0;
13   virtual void start()=0;
14 };
15
16 /**
17  * Interface de communication avec le réseau
18  */
19 class NetworkIf {
20 public:
```

```
21      virtual int getXPosition() =0;
22      virtual int getYPosition() =0;
23      virtual void sendPaquet(Paquet &) =0;
24    };
25    #endif
```

Q1 Implémenter l'interface `NetworkIf` sur le module `NetworkLayer` de l'exercice 4.6.
Q2 L'interface `NetworkPlugableIf` est implémentée par la classe `NodeCompute`. Utiliser une instance de `NodeCompute` par noeud de réseau pour obtenir une simulation de propagation d'onde. On veillera à utiliser dans la compilation les fichiers `paquet.cc`, `displaynodes.cc`, `nodecompute.cc`. Le lien avec les librairies graphiques se fera (comme indiqué dans le makefile fourni dans l'archive) par la commande `-L/usr/X11R6/lib -lX11`.

Ex 4.3 *** Couche transport du modèle OSI

Nous nous plaçons dans cet exercice au niveau de la couche transport. Nous réutilisons la couche réseau faite dans l'exercice précédent. Ainsi à chaque noeud de la couche réseau nous rajoutons un port nt (network vers transport) et un port tn (transport vers network) permettant de communiquer avec la couche de transport :



Au niveau de la couche transport nous utiliserons l'algorithme suivant : envoi d'un paquet numéro n vers le noeud d :

- Si un acquittement provenant de d pour le paquet n est reçu c'est bon, le paquet a été reçu.
- Si pas d'acquittement provenant de d pour le paquet n au bout d'un temps donné, alors réessayer encore deux fois d'envoyer le paquet.
- Si au bout de trois tentatives d'envoi du paquet aucune réception n'est possible alors il y a une rupture de la communication au niveau transport.

Q1 Modifier la structure des paquets de l'exercice précédent pour rajouter :

- un champ permettant d'identifier le paquet à envoyer.
- un champ permettant d'identifier le type de données : est-ce une émission de données ou un acquittement pour un paquet donné.

Q2 Modifier le module de la couche réseau de l'exercice précédent et lui adjoindre le module de la couche transport.
Q3 Produire les graphes affichant le taux de rupture de communication au niveau de la couche transport en fonction de la taille des fifos et de la fréquence d'émission des messages.

Ex 4.4 *** Interface réseau

Tout comme à la section 4.4 on a plaqué sur l'espace mémoire un périphérique de disque, définir et implémenter un périphérique réseau conforme au modèle présenté dans l'exercice 4.6.

Ex 4.5 *** Microprocesseur Multicœur

Utiliser le réseau des exercices 4.6 et 4.6 pour réaliser un microprocesseur multicœur basé sur le simulateur Jasip réalisé dans le chapitre précédent.

Il faut pour cela ajouter un nouveau périphérique mémoire au module ram.h de l'exercice 2.6. On déterminera donc des adresses mémoire spécifiques permettant à un microprocesseur de communiquer avec le réseau. Pour créer un programme Java testant le réseau il suffit d'utiliser les méthodes :

```
org.jasip.Jasip.writeByteAtAddress
org.jasip.Jasip.readByteAtAddress
```

Chapitre 5

Exemple de co-design

5.1 Complément C++ : relecture de code

Le complément C++ de ce chapitre vise à poser les bases d'une pratique fréquente : la relecture de code.

5.1.1 Qu'est-ce que la relecture de code ?

La relecture de code est une réunion de travail où l'on relit du code. Elle comprend en général :
- une autorité de type architecte, ou chef d'équipe / de projet.
- la personne ayant écrit le code
- un ou plusieurs relecteurs.

On veillera lors d'une revue de code à la présence de développeurs « sénior ». L'idée de la revue de code n'est pas de placer le concepteur devant un tribunal jugeant son travail mais de répandre les bonnes pratiques de conception et d'homogénéiser les méthodologies entre développeur. Aussi l'aspect relationnel est important aussi bien pendant la préparation de la réunion que lors de son déroulement.

5.1.2 Quand faire des relectures de code ?

Voici trois occasions de faire des relectures de code :
- de manière aléatoire et régulière pour encourager le fait de porter un regard critique sur son code et pour répandre les bonnes pratiques.
- sur des modules particulièrement critiques (d'un point de vue performance, sécurité, ...).
- lorsque les tests d'intégration révèlent un nombre d'anomalies plus élevé que de coutume.

5.1.3 Règles de codage

Tout d'abord la relecture de code vise à corriger le code dans sa forme. La forme est généralement donnée par un document nommé « règles de codages », et contenant généralement des informations de la forme :
- façon de nommer les variables et les fonctions `niveau_huile` ou `niveauHuile` par exemple.
- nombre de lignes maximum par fonction.
- style d'indentation
- structure des commentaires. Il est important de vérifier la structure des commmentaire, surtout si on utilise un outil comme doxygen. Ainsi la fonction ci-dessous dans jasip :

```
/**
 * Given a name of the class file returned its associated ClassFile.
 * If the class has not been loaded yet in the virtual machine it is then loaded.
 * If the class file can not be loaded NULL is returned.
 * @param className name of the class
 * @param noLoad tells if the class should be loaded each time
 * @return return the ClassFile instance representing the
 * class which name is in className.
 */
ClassFile * VM::getClassFile(string className,bool noLoad) {
```

```
    // ...
}
```

Donne le bout de page html ci-dessous après avoir été traité par doxygen :

ClassFile * VM::getClassFile (string *className*,
bool *noLoad* = *false*
)

Given a name of the class file returned its associated ClassFile.

If the class has not been loaded yet in the virtual machine it is then loaded. If the class file can not be loaded NULL is returned.

Parameters:
className name of the class
noLoad tells if the class should be loaded each time

Returns:
return the ClassFile instance representing the class which name is in className.

– abscence de certaines constructions (par exemple récursion qui peut donner des profondeur de pile importante, ou allocation mémoire qui peut ralentir le programme).

5.1.4 Mémoire

L'une des principales source de perte de temps dans un développement C++ mal maîtrisé est les **fuites mémoires**. Il faut vérifier qu'à chaque allocation (`new` ou `new[]`) correspond bien une desallocation (`delete` ou `delete[]`). L'idée n'est bien sur pas de couvrir tous les cas comme le ferait un programme de type Valgrind mais d'avoir pour chaque membre d'une classe une vision synthétique de la politique d'allocation/desallocation, c'est à dire qu'il soit clair à tout moment quel objet à la charge de desallouer la mémoire allouée.

Un dépassement de capacité dans l'allocation mémoire peut se produire par mégarde. Considérons ainsi le code suivant sur une machine 32 bits :

```
int size = 1073741824;
int *buffer = new int[size];
```

L'opération consiste en faite à allouer un nombre d'octets égal à `sizeof(int)*size`. Or ce nombre vaut 2^{32}, soit -1. Le tableau n'est donc finalement pas de la bonne taille.

5.1.5 Macros

Il est important de parenthéser les macros. Considérons ainsi le code suivant :

```
#define MUL(a, b) a*b
#define ADD(a, b) a+b
```

Est mauvais pour deux raisons

– Il manque le parenthésage externe, ainsi sans parenthésage externe dans `ADD`, `MUL(ADD(3,4),2)` est étendu en `3+4*2` soit une valeur de 11 et non 14.
– Il faut également effectuer un parenthésage interne, typiquement pour que `MUL(3+4,2)` soit correctement évalué.

Ainsi la « bonne »façon de définir ces macros est la suivante :

```
#define MUL(a, b) ((a)*(b))
#define ADD(a, b) ((a)+(b))
```

De manière générale il est facile de se tromper dans l'usage des macros en utilisant par exemple un objet du mauvais type (il n'y a pas de vérification de type dans les macros). Aussi il est important dès qu'une macro est de taille conséquente de se demander si on ne pourrait pas utiliser une fonction « inline ».

5.1.6 Logique

Dès que l'on utilise les opérateurs logiques `&&` et `||` ou les opérateurs bit à bit `&` et `|` il est important de vérifier qu'il n'y a pas eu de confusion entre les symboles.

Il faut se méfier de l'évaluation sur les opérateurs logiques : en effet dans l'expression `f() && g()` si `f` renvoie une valeur fausse alors aucun appel à la fonction `g` n'est effectué. De même dans l'expression `f() || g()` si `f` renvoie une valeur vraie alors aucun appel à la fonction `g` n'est effectué.

La sémantique précédente change complètement si l'opérateur `&&` ou `||` est redéfinit. En effet l'instruction `a && f()` est vue par le compilateur comme `a.operator&&(f())`. Dans ce cas la fonction f est toujours évaluée contrairement à ce qui a été dit dans le paragraphe précédent.

La précédence des opérateurs C ou C++ est souvent mal connue. Aussi vaut-il toujours mieux rajouter des parenthèses pour supprimer toute ambiguité. Ainsi les deux lignes suivantes sont équivalentes :

```
(x * 2) + 1;
(x << 1) + 1;
```

Alors que les deux suivantes ne le sont pas :

```
x * 2 + 1;
x << 1 + 1; // équivalent à (x*4)
```

5.1.7 Séquence d'appel

En C et en C++ plusieurs séquences d'appel ne sont pas spécifiées et varient d'un compilateur à l'autre (ou même pourraient varier pour un même compilateur).

Ainsi dans l'expression ci-dessous :

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

nous ne disposons d'aucune garantie pour savoir quelle fonction va être appelée en premier.

En revanche certains opérateurs nous garantissent que certains appels vont avoir lieu avant d'autres :

- `&&` ou `||` le membre de gauche est évalué avant le membre de droite
- `,` utilisé dans une expression (comme `int i=f(2), j=g(3);`) assure que le membre de gauche est évalué avant le membre de droite. En revanche pour les arguments d'appel à une fonction aucun ordre n'est spécifié, ainsi dans l'expression `f(g(),h())` on ne sait qui de `g` ou `h` est appelé en premier.
- Appel de fonction : tous les arguments sont évalués avant l'appel à une fonction.
- `;` tout ce qui est avant le point virgule est évalué avant ce qui suit.
- `if`, `while`, `switch`, `for`, `do while` : la partie conditionnel est toujours évaluée avant le corps de boucle.

Il est à noter que touche surcharge de ces opérateurs fait basculer la sémantique sous celle de l'appel fonctionnel.

Voici un exemple type de code à banir :

```
int x = 0;
f(x++, x++, x++);
```

On ne sait quel vont être les trois arguments de `f`.

De même si on dispose d'un fonction lisant un flux `lireFlux` l'expression :

```
x = lireFlux() * 256 + lireFlux();
```

Est à banir, on ne sait pas dans quel ordre les arguments du flux sont lus. Il faut écrire :

```
int tmp = lireFlux();
x = tmp * 256 + lireFlux();
```

Dans un switch l'absence de break (ou de return) entre deux case entraine une exécution du cas suivant :

```
switch (n) {
  case 1:
    f();
    break;        // quitter le switch
  case 2:
    g();
```

```
    case 3:
      h();           // exécuté pour n==2 et n==3
      break;         // quitter le switch
  }
```

Pour être certain que l'absence de break à la fin du cas 2 n'est pas une erreur c'est une bonne pratique d'imposer un commentaire :

```
  switch (n) {
    case 1:
      f();
      break;
    case 2:
      g();
      // Attention : pas de break ici
      // continuer maintenant sur le cas 3 en appelant h.
    case 3:
      h();
      break;
  }
```

5.1.8 Outils complémentaires

La relecture de code, même si elle est complémentaire à l'usage d'outils, peut tout de même s'appuyer sur les résultats donnés par :
- le compilateur avec le plus haut niveau de warning (`-Wall` avec `gcc`).
- d'outils d'analyse au runtime comme valgrind (`http://valgrind.org`) en open source ou purify (`http://www.ibm.com/software/awdtools/purify`) chez IBM.
- un analyseur statique en open source comme splint (`http://splint.org/`) e ou klocwork (`http://www.klocwork.com/`) commercial.

Le résultat de tels outils peut en effet aider à cibler la relecture sur certains points.

5.2 Cas d'étude pour le co-design

Dans ce chapitre nous allons nous intéresser aux problématiques de conception simultanée de matériel et de logiciel à travers un exemple sur la propagation des ondes sur une surface plane. Cet exemple a été utilisée dans l'exercice 4.6 pour valider la bonne marche de la simulation de réseau.

5.2.1 Equation des ondes

Voici l'équation de propagation des ondes en milieu homogène utilisée :

$$\frac{\partial^2 u}{\partial t^2} = c^2 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) - \rho \frac{\partial u}{\partial t}$$

La constante c représente la vitesse de propagation de l'onde, et ρ représente un facteur d'atténuation de l'onde que l'on pourra assimiler à un frottement visqueux.

5.2.2 Calcul approché

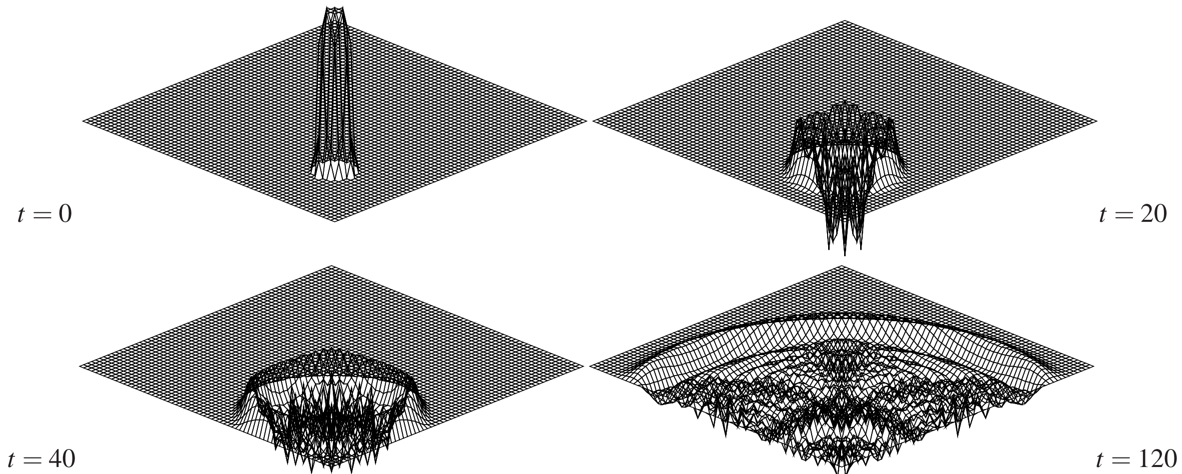
Le but est de calculer une solution approchée en discrétisant le problème sur une grille carrée. On se donne les tableaux ci-dessous contenant respectivement la valeur du niveau en un point de la grille, la vitesse de changement de ces valeurs et l'accélération de changement de ces valeurs :

```
int values[VAL_MAX][VAL_MAX];
int speed [VAL_MAX][VAL_MAX];
int accel [VAL_MAX][VAL_MAX];
```

La discrétisation de l'équation précédente pour $c^2 = 1/10$ et $\rho = 1/200$ plus un facteur proportionnel de 10 sur les valeurs de `values`, donnent les formules suivantes pour tout `i` et `j` :

```
accel[i][j]=values[i+1][j]+values[i-1][j]+values[i][j-1]+values[i][j+1]
           -4*values[i][j]-speed[i][j]/200;
speed[i][j]+=accel[i][j];
values[i][j]+=speed[i][j]/10;
```

Les itérations de cette série d'équations donnent un temps discret. Voici des exemples de résultats obtenus à l'aide de ces équations :



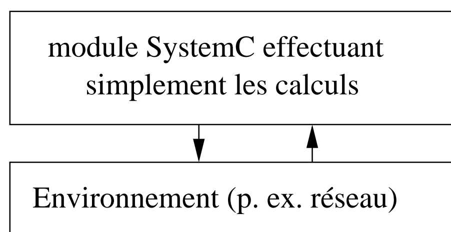
On a imposé la contrainte que les bords restaient à une hauteur constance. On remarque la réflexion de l'onde sur les bords au temps $t = 120$.

5.3 Exemples de co-design

Nous allons détailler comment un système matériel et logiciel en charger de calculer la propagation d'onde peut être modélisé à l'aide de SystemC.

5.3.1 Validation du principe

La première étape consiste tout d'abord à valider au sein d'une architecture plus large les actions faites par le module.

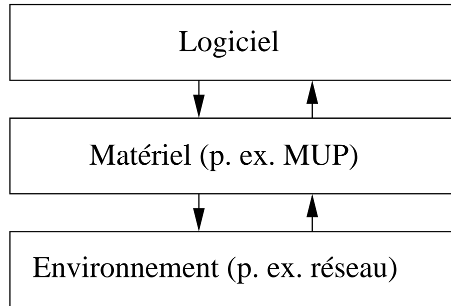


A ce stade les équations de la section 5.2.2 sont simplement écrites dans un module SystemC qui est connecté à son environnement, qui peut par exemple être un réseau comme à l'exercice 4.6.

Cette étape permet de configurer des paramètres importants de l'environnement (taille des fifos, validation du mécanisme de transport) ou de l'algorithme (précision suffisante). A ce niveau de détail on ne sait encore si le module SystemC sera implémenté par du matériel ou du logiciel.

5.3.2 Implémentation Logicielle

Si l'on dipose d'un modèle SystemC d'un microprocesseur comme Jasip produit par l'exercice 2.6 on peut écrire une version des équations de la section 5.2.2, puis lancer ce code sur le modèle du microprocesseur. On obtient alors l'architecture suivante :



Cette architecture nous permet de connaitre si un réseau de microprocesseurs MUP peut calculer pour le débit nécessaire, pour le coût nécessaire l'équation de propagation des ondes.

5.3.3 Ajout de nouvelles commandes

On remarque dans les formules de la section 5.2.2 trois fois on fait un ajout d'une d'valeur `v` multipliée par un certain facteur `f`. Il pourrait dès lors être intéressant de disposer d'une instruction `macc` multiply accumulate de la forme : `macc r1 v1 f` qui ajoute au registre `r1` la valeur `v` multipliée par le facteur `f`.

Disposer d'une telle instruction fait gagner du temps et permet une exécution plus rapide

5.3.4 Adjonction d'un co-processeur

Nous remarquons que l'on effectue des instructions répétitives sur des données différentes. Ainsi l'instruction *I* :

```
accel[i][j] = values[i+1][j] + values[i-1][j] + values[i][j-1] + values[i][j+1]
             -4*values[i][j] - speed[i][j]/200;
```

est effectuée pour de nombreux couples `i` et `j` : 65 536 si on a un carré de 256 par 256.

Dès lors si nous disposons d'une unité vectorielle qui effectue une même opération sur plusieurs instructions nous pourrions gagner plusieurs cycles d'horloge.

Interface de l'unité vectorielle

Un vecteur est une suite continue de cases mémoire. La taille en octets de chaque case doit être spécificiée par `vsets`. Le nombre d'éléments du vecteur doit être spécifié par `vsetn`.

- `vsets n` fixe la taille des cases mémoire du vecteur, `n` doit être compris en 1 et 4.
- `vsetn n` fixe le nombre d'éléments du vecteur, `n` doit être compris entre 1 et 8.
- `vset v1 v2` affecte le vecteur `v2` dans le vecteur `v1`.
- `vadd v1 v2 v3` ajoute les vecteurs `v2` et `v3` et écrit le résultat dans `v1`.
- `vsub v1 v2 v3` soustrait le vecteur `v2` au `v3` et écrit le résultat dans `v1`.
- `vmul v1 v2 r` multiplie le vecteur `v2` par le réel `r` et écrit le résultat dans `v1`.
- `vdiv v1 v2 r` divise le vecteur `v2` par le réel `r` et écrit le résultat dans `v1`.

Usage de l'unité vectorielle

L'instruction *I* donnée en exemple peut alors s'écrire dans l'assembleur utilisant le co-processeur :

```
# l'adresse de values[i][j] est dans r1
# l'adresse de accel[i][j] est dans r2
# l'adresse de speed[i][j] est dans r3
# r4 contient l'adresse d'une zone temporaire
add r5 r1 4
vset r2 r5      # accel[i][j] = values[i+1][j]
sub r5 4 r1
vadd r2 r5      # accel[i][j] += values[i-1][j]
add r5 1024 r1
vadd r2 r5      # accel[i][j] += values[i][j+1]
sub r5 1024 r1
```

```
vadd r2 r5      # accel[i][j] += values[i][j-1]
vmul r4 r1 4
vsub r2 r4 r2   # accel[i][j] -= 4*values[i][j]
vdiv r4 r3 200
vsub r2 r4 r2   # accel[i][j] -= speed[i][j]/200
```

Si l'unité vectorielle marche en parallèle de l'unité normale on peut même écrire les instructions du co-processeur en parallèle des instructions du microprocesseur :

```
add r5 r1 4
sub r5 4 r1    vset r2 r5
add r5 1024 r1 vadd r2 r5
sub r5 1024 r1 vadd r2 r5
nop            vadd r2 r5
nop            vmul r4 r1 4
nop            vsub r2 r4 r2
nop            vdiv r4 r3 200
nop            vsub r2 r4 r2
```

Ce type de processeur est nommé VLIW (Very Long Instruction Word).

Avec l'unité vectorielle on obtient un gain au moins égal au nombre d'éléments dans un vecteur.

5.3.5 Matériel dédié

Si le matériel est prévu pour être produit en un nombre très important d'exemplaires ou bien qu'un FPGA (Field Programmable Gate Array) est présent alors un circuit matériel dédié peut être construit.

Dans le cas présent les opérations les plus fréquentes sont les additions. Il faut donc pourvoir avoir un maximum d'additions. Il faut en outre des circuits matériels fixes pour multiplier par 4 et diviser par 200.

La méthode sera la même que dans le cas de l'unité vectorielle, mais sera moins généraliste et pourra donc comporter plus d'additionneurs pour une surface de silicium égale.

5.4 Exercice

Ex 5.1 *** Co-Processeur vectoriel

Implémenter le co-processeur définit à la section 5.3.3.

Chapitre 6

Dans le monde réel

Nous décrivons comment un cadre conceptuel comme Jasip peut être décliné en pratique. Nous présentons les initiatives basées sur Java et proches du matériel. Puis nous présentons Unisim un cadre d'utilisation de SystemC permettant de simuler des sytèmes complets.

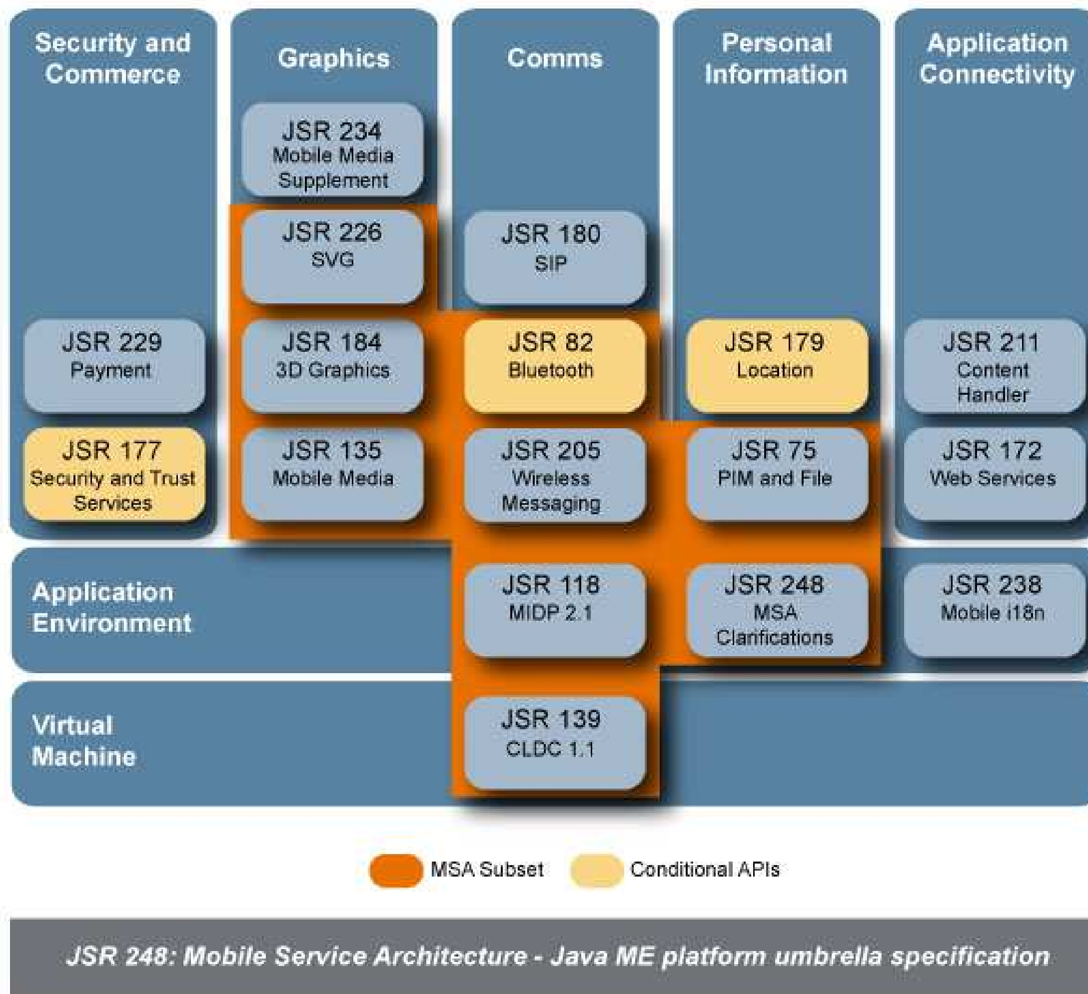
6.1 Approches à la Jasip

Jasip propose une API Java à un système. Cette approche n'est pas spécialement originale et on la retrouve dans plusieurs systèmes embarqués, en particulier les téléphones. L'avantage d'une telle approche est de pouvoir utiliser des développeurs standards pour la partie applicative.

6.1.1 J2ME

Très rapidement Sun a proposé une version embarquée de java nommée J2ME pour Java 2 Mobile/Micro Edition, par rapport à la version standard nommée J2SE Java 2 Standard Edition.

Tout comme dans Jasip on retrouve des classes de base réduites par rapport à J2SE, c'est la même chose dans J2ME. Il y a quelques bibliothèques en plus dans J2ME, par exemple pour gérer la mémoire si l'on ne dispose pas de la possibilité d'avoir un ramasse miette (garbage collector).



6.1.2 Android

Début novembre 2007, la société Google vient de sortir une plate-forme pour téléphones portables nommée Android (`http://code.google.com/android`). Tout comme Jasip elle se programme en Java.

L'architecture d'Android est bien sûr plus complexe que celle de Jasip.

La partie applicative ainsi que le cadre de base sont dans l'approche similaires à ceux de Jasip. On trouve simplement des libraires en plus, ce qui augmente considérablement la puissance du cadre. Il y a également plusieurs couches avec le matériel : linux joue le rôle de couche d'abstraction par rapport au matériel et une machine virtuelle (basée sur des registres, contrairement à la machine virtuelle de Sun) nommée Dalvik est utilisée.

6.1.3 Comparaisons

J2ME dispose d'une interface unique (à travers la machine virtuelle) pour accéder au matériel. Cela assure une plus grande portabilité.

Android en revanche utilise plus de couches d'abstraction avec le matériel. Cela peut conduire à une certaine pénalité à l'exécution mais permet de mieux réutiliser le code existant.

Android est une plate-forme en devenir dont la sortie opérationnelle est programmée mi 2008. Outre l'appreciation de l'architecture d'autre facteur sont à prendre en compte pour savoir laquelle des deux sortira gagnante dans quelques années. Ainsi Andoid est une plate-forme open-source alors que J2ME est une solution propriétaire actuellement largement dominante sur laquelle Sun a basé une partie important de ses recettes.

6.2 Unisim

Dans cette section nous détaillons unisim `unisim.org` un environnement de simulation basé sur SystemC et permettant de faire du prototypage virutel, c'est à dire permettant de facilement simuler des plate-formes matérielles à l'aide de logiciel. Unisim est distribué avec une licence open-source. Il existe deux versions : une version précise au cycle d'horloge près et un version dite « TLM »(Transaction Level Modeling) plus rapide mais moins précise sur les aspects temporels.

6.2.1 Rôle des plate-formes virtuelles

Les plate-formes virtuelles recouvrent les usages suivants :

– par exemple si on dispose de peu de plate-formes matérielles cibles pour le nombre de développeur en charge du logiciel.
– si le matériel n'est pas disponible.
– pour faciliter la mise en place de larges campagnes de tests.

6.2.2 Un exemple de réalisation

Ci-dessous figure un exemple de système réalisé à l'aide de la plate-forme unisim.

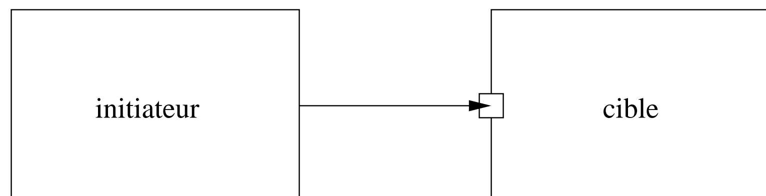
`\scalebox{.5}{\includegraphics{cours6/ptf.eps}}`

6.2.3 L'interface TLM

L'interface n'est implémentée qu'à travers un seule classe : `TlmSendIf`. Cette interface ne dispose que d'une méthode notifiant l'envoi d'un message.

```
1   template <typename REQ, typename RSP>
2   class TlmSendIf : public virtual sc_interface {
3   public:
4     virtual bool Send(TlmMessage<REQ, RSP> &message) = 0;
5   };
6
7   template <typename REQ, typename RSP>
8   class TlmMessage{
9   public:
10    REQ req;
11    RSP rsp;
12    sc_event event;
13  };
```

La classe `TlmSendIf` est une classe générique parmétrée par le type de la requête et le type de la réponse. C'est un initiateur qui va appeler la méthode `Send` sur la cible. Le message contient trois données : la requête, la réponse et un évènement. Si une réponse est attendue, l'initiateur doit mettre un évènement. Il doit alors attendre jusqu'à ce que la réponse soit écrite.



Pour être plus performant et donc éviter les recopies le message est partagé entre l'initiateur et la cible. Pour éviter de savoir à qui est la responsabilité de desallouer la mémoire du message, cette tâche est confiée à une classe nommée `Pointer` en charge de compter le nombre de référence à l'objet.

Ainsi l'interface devient :

```
1   #include "pointer.h"
2   template <typename REQ, typename RSP>
3   class TlmSendIf : public virtual sc_interface {
4   public:
5     virtual bool Send(Pointer<TlmMessage<REQ, RSP> > &message) = 0;
6   };
7
8   template <typename REQ, typename RSP>
9   class TlmMessage{
10  public:
11    REQ req;
12    RSP rsp;
13    sc_event event;
14  };
```

6.2.4 La gestion mémoire

Les auto pointeurs

Considérons le code suivant :

```
class A {} ;
void maFonction() {
    A * a = new A();
    // ...
    delete a;
}
```

Ce code est correct mais si une exception est lancée, ou si une autre personne rajoute un return au milieu de la fonction alors la libération mémoire n'est pas effectuée.

On peut déléguer la responsabilité de la libération mémoire à une structure nommée auto-pointeur (`auto_ptr`) dans la stl.

Ainsi le code résultant est

```
#include <memory>
using namespace std;
class A {} ;
void maFonction() {
  auto_ptr<A>  a (new A());
  // ...
} // la libération est faite automatiquement
  // au moment de la destruction de a
```

Prenons un exemple plus long :

```
#include <memory>
#include <assert.h>

using namespace std;

void g() {
  int* pt1 = new int;           // g gère le pointeur pt1
  auto_ptr<int> pt2( pt1 );     // g cède la gestion de pt1 à pt2
  *pt2 = 12;                    // équivalent à "*pt1 = 12;"
  assert( pt1 == pt2.get() );   // les deux pointeurs sont équivalents
  int* pt3 = pt2.release();     // g récupère la gestion de pt1
  delete pt3;                   // il faut maintenant détruire pt3
}  // pt2 ne possède plus de pointeur
   // il n'y a pas de double desallocation.
```

On remarque que les auto pointeurs se manipulent exactement comme les pointeurs. On peut reprendre la main sur la gestion du pointeur à l'aide de la méthode `release`.

On peut mettre les auto pointeurs dans les champs de classes, cela évite d'avoir à les détruire dans le destructeur

```
#include <memory>

using namespace std;

class B {};
class A {
public:
  A();
  /*...*/
private:
  auto_ptr<B> b;
};
```

Un pointeur n'est géré que dans un seul auto pointeur. Ainsi l'exemple ci-dessous conduit à une erreur :

```
void f() {
  auto_ptr<T> pt1( new T );
  auto_ptr<T> pt2;

  pt2 = pt1;  // pt2 gère maintenant le pointeur
              // et pt1 ne gère rien
  pt1->DoSomething(); // ERREUR ! PAS DE POINTEUR
}
```

Comme le fait de recopier un auto pointeur transfert sa valeur on peut utiliser des auto pointeurs en retour de fonction :

```
auto_ptr<A> monAllocateur() {
  // ...
  return auto_ptr<A> (new A());
}
```

Libre ensuite à la fonction qui récupère la valeur de prendre ou non la gestion du pointeur :

- si elle la prend elle devra alors libérer le pointeur ou bien l'assigner à un autre auto pointeur.
- si elle ne prend pas la valeur retournée par `monAllocateur`, ce n'est pas grave la desallocation se fera au moment de la fin de la portée de `monAllocateur`.

Il ne faut pas utiliser les auto pointeurs dans les containers, typiquement dans :

```
vector< auto_ptr<T> > v;  // A NE PAS FAIRE !!!
```

En effet il peut y avoir des recopie dans l'usage d'algorithmes (comme le tri par exemple), ce qui a pour effet de faire perdre la gestion du pointeur aux éléments du vecteur.

Les pointeurs partagés

Nous venons de voir les auto pointeurs. Leur limitation réside dans le fait qu'un pointeur ne peut être dans un seul auto pointeur à la fois. Ainsi l'usage des auto pointeurs n'est pas compatibles avec la STL et la sémantique de l'affectation n'est pas nécessairement intuitive.

L'utilisation des pointeurs partagés (*shared pointers*) est semblables à celle des auto pointeurs. Simplement le nombre de références d'un objet est compté.

Voici une utilisation simple des `shared_ptr` dans la librairie boost :

```
#include <boost/shared_ptr.hpp>

using namespace boost;

void f () {
  shared_ptr<int>  p (new int);
  *p = 4;
}
```

Les librairies boost sont disponibles à l'url `http://www.boost.org/`.

Le même pointeur pouvant être utilisé dans plusieurs shared pointers, cela permet une utilisation facile dans un contexte multi threadé :

```
#include <boost/shared_ptr.hpp>

using namespace boost;

class A {
public :
  int methode1() { return 1;}
  int methode2() { return 2;}
};
```

```
A * a = new A();

void thread1() {
  shared_ptr<A> p1(a);
  p1->methode1();
}
void thread2() {
  shared_ptr<A> p2(a);
  p2->methode2();
}
```

6.3 Exercice

Ex 6.1 *** Gestion des pointeurs

Q1 Proposer une implémentation pour les auto pointeurs.
Q2 Proposer une implémentation pour les pointeur partagés.

Chapitre 7

Fiche Pratique 1 : Compilation C++

Cette fiche pratique s'attache détailler les étapes de la compilation de fichiers C++ en ligne de commande sous linux. Nous supposons que l'utilisateur dispose d'un environnement linux comprenant les logiciels :

– gcc (`http://gcc.gnu.org`),
– gnu make (`http://www.gnu.org/software/make`).
– un éditeur de texte comme vi, emacs ou xemacs.

Nous supposons l'usage d'un shell bash. Le shell en cours s'obtient à l'aide de la commande `echo $SHELL`.

7.1 Compilation

7.1.1 Qu'est-ce que la compilation ?

La compilation c'est le fait de transformer un langage dit de haut niveau comme C ou C++ en une série d'instructions assembleur directement compréhensible par le microprocesseur de la machine.

Du point de vue de l'utilisateur la compilation C++ a lieu en deux passes :

– pour chaque fichier `*.cc` l'obtention d'un fichier traduit en assembleur, généralement avec un suffixe en `.o` pour fichier objet.
– le regrouppement des fichiers objets, cette étape est nommée édition de liens, ou génération de l'exécutable.

7.1.2 Compilation d'un fichier *.cc

Dans le même répertoire qu'un fichier `toto.cc` taper :

```
g++ -c toto.cc
```

Cela génère le fichier `toto.o` qui n'est pas un fichier lisible pour un humain. Si l'on souhaite voir le code assembleur (comme dans l'exercice 1.6) il faut taper `g++ -S toto.cc`, le fichier assembleur `toto.s` lisible par un humain est alors généré.

Pour voir tous les mises en gardes (warnings) possibles rajouter l'option de compilation `-Wall`, et pour garder dans le fichier objet des information donnant les numéros de lignes et de fichier (utiles au débuggage) utiliser l'option `-g` :

```
g++ -Wall -g -c toto.cc
```

Pour afficher l'ensemble du code ramené par les directives `#include` et remplacer les définitions `#define` par leur valeur, effectuer la commande suivante :

```
g++ -E toto.cc
```

7.1.3 Génération de l'exécutable ou édition de liens

Etant donné les fichiers objets `toto.o`, `tata.o` et `tutu.o`, si l'un de ces fichiers définit une fonction `main` on peut alors créer un exécutable nommé `monexecutable` grâce à l'instruction suivante :

```
g++ -o monexecutable toto.o tata.o tutu.o
```

Il suffit alors de taper `./monexecutable` pour lancer l'exécution.

7.1.4 Symboles dans l'exécutable

Considérons le programme suivant dans le fichier `main.cc` :

```
1  int i=0;
2  int main() {
3    return 0;
4  }
```

En le compilant par `g++ -c main.cc` on obtient le fichier `main.o`. Pour ensuite pouvoir effectuer l'édition de lien de `main.o` avec un programme `toto.o` utilisant la variable `i` faut que dans `main.o` on puisse retrouver une variable par son nom. La commande permettant d'inspecter les symboles (ou noms) est `nm`. Ainsi sur le programme `main.o` on obtient :

```
> nm main.o
00000000 B i
00000000 T main
```

Le premier nombre de la ligne représente l'adresse où se situe le symbole, ici la variable `i` ou la fonction `main`. Ici zéro est indiqué car le placement en mémoire n'est fait qu'au moment de l'édition de liens. La lettre détermine ensuite le type de symbole : `T` signifie un symbole définit dans le code et `B` un symbole dans la section des données non initialisées. Faire `man nm` pour obtenir la signification des autres lettres utilisées. Si on compile `main.o` en un programme exécutable on obtient alors une liste de symboles beaucoup plus longue (abrégée ici) :

```
> g++ -o monprog main.cc
> nm monprog
080494cc D _DYNAMIC
080495b0 D _GLOBAL_OFFSET_TABLE_
080484b0 R _IO_stdin_used
         w _Jv_RegisterClasses
080494bc d __CTOR_END__
080494b8 d __CTOR_LIST__
...
080495d0 B i
08048384 T main
080495c8 d p.0
```

On remarque que les variables ont été placées en mémoire à des adresses bien définies.

Tous ces symboles prennent de la place et peuvent être supprimés à l'aide de la commande strip. Il n'est alors plus possible d'utiliser la commande `nm`, mais on a gagné en taille :

```
> ls -lh monprog
-rwxr-xr-x  1 fabrice fabrice 12K May 18 10:10 monprog
> strip monprog
> ls -lh monprog
-rwxr-xr-x  1 fabrice fabrice 3.0K May 18 10:10 monprog
> nm monprog
nm: monprog: no symbols
```

7.2 Automatisation de la compilation

Taper ces lignes de commande à chaque fois que l'on veut tester à nouveau une modification est relativement fastidueux. Pour automatiser le processus de compilation existent les makefiles.

Un makefile est un fichier généralement nommé `Makefile` ou `makefile`. Il est lancé par la commande `make` exécutée dans le répertoire où il se trouve.

7.2.1 Exemple simple

Pour compiler le fichier C++ `toto.cc` en le fichier objet `toto.o` il faut écrire le makefile suivant :

```
toto.o : toto.cc
	g++ -c toto.cc
```

La première ligne signifie que l'on souhaite générer le fichier `toto.o` à partir du fichier `toto.cc`. La deuxième ligne commence par un caractère de tabulation suivit de la commande effective pour générer le fichier `toto.o`. En tapant `make` sur la ligne de commande on génère alors le fichier `toto.o`, la commande exécutée s'affiche alors. En tapant à nouveau `make` rien ne se passe, c'est normal, à cause de la dépendance de la première ligne, puisqu'aucune modification n'a été apportée à `toto.cc`, et que la date de création de `toto.o` est plus récente, il n'y a rien à faire.

```
>make
g++ -c toto.cc
>make
make: 'toto.o' is up to date.
>
```

Si la règle pour créer `toto.o` n'est pas la première règle du makefile, il faut alors taper `make toto.o` pour exécuter la règle correspondante. Si le fichier `toto.cc` inclus un fichier `tata.h` qui peut être modifié par l'utilisateur il faut alors le rajouter dans la liste de dépendances :

```
toto.o : toto.cc tata.h
	g++ -c toto.cc
```

De même pour créer un exécutable il suffit de rajouter au début du makefile les deux lignes suivantes :

```
monexecutable : toto.o tata.o tutu.o
	g++ -o monexecutable toto.o tata.o tutu.o
```

7.2.2 Utilisation de variables

Le compilateur peut parfois être g++, ou un autre. Si l'on veut tout changer d'un coup, il faut mieux utiliser une variable. De même pour les options :

```
CXX:=g++
CXXFLAGS:=-Wall -g

monexecutable : toto.o
	$(CXX) -o monexecutable toto.o
toto.o : toto.cc tata.h
	$(CXX) $(CXXFLAGS) -c toto.cc
```

Si l'on souhaite compiler tous les fichiers `*.cc` d'un même répertoire on peut définir la liste de fichiers sources dans `SOURCES` et la liste de fichiers objets dans `OBJ` à l'aide de la commande suivante :

```
SOURCES := $(wildcard *.cc)      # prend tous les fichiers *.cc du répertoire
OBJ := $(SOURCES:.cc=.o)         # remplace tous les .cc par .o
```

Cette syntaxe est spécifique à Gnu Make. Notons que les commentaires dans les makefile sont compris entre le symbole `#` et la fin de la ligne.

Il existe de plus deux variables définies à chaque règle : `$@` qui représent la cible et `$<` qui représente la première dépendance. Ainsi la règle :

```
monexecutable : toto.o
	$(CXX) -o monexecutable toto.o
```

est-elle équivalente à :

```
monexecutable : toto.o
	$(CXX) -o $@ $<
```

7.2.3 Règles génériques

On peut également écrire des règles génériques s'appliquant à un ensemble de fichiers définis par une extension. Par exemple au lieu d'écrire les deux règles suivantes :

```
toto.o : toto.cc
        $(CXX) $(CXXFLAGS) -c toto.cc
tata.o : tata.cc
        $(CXX) $(CXXFLAGS) -c tata.cc
```

Il suffit d'écrire la règle suivante :

```
%.o : %.cc
        $(CXX) $(CXXFLAGS) -c $<
```

Qui signifie : si à un moment donné un fichier ayant le suffixe `.o` doit être généré alors on appliquera la règle `$(CXX) $(CXXFLAGS) %.cc` si `%.cc` est plus récent que `%.o`.

7.2.4 Dépendance avec les fichiers d'entête

Un fichier `*.cc` inclus en général des fichiers d'entête `*.h`. Si un de ces fichiers `*.h` est modifié, il est en général sage de recompiler les fichiers `*.cc` qui l'incluent. Voici la règle à rajouer pour un fichier `toto.cc` incluant `toto.h` :

```
1  %.o : %.cc
2          $(CXX) $(CXXFLAGS) -c $<
3  toto.o : toto.cc toto.h
```

La liste des dependances peut être générée automatiquement à l'aide de l'option `-MM` de `g++`. Ainsi la ligne 3 dans le makefile précent peut être écrite dans le fichier `.deps` à l'aide de la commande en ligne :

```
g++ -MM toto.cc > .deps
```

Voici comment utiliser ces dépendances dans un makefile :

```
SOURCES := $(wildcard *.cc)      # prend tous les fichiers *.cc du répertoire
.deps : $(SOURCES)
        $(CXX) $(CXXFLAGS) -MM $(SOURCES) > .deps
-include .deps
```

7.2.5 Résumé

En résumé voici un makefile par défaut qui compile tous les fichiers `*.cc` d'un répertoire en un exécutable `monexe`, en tenant compte des dépendances sur les fichiers d'entête :

```
#
# définitions
#
CXX:=g++
CXXFLAGS:=-Wall -g
SOURCES := $(wildcard *.cc)      # prend tous les fichiers *.cc du répertoire
OBJ := $(SOURCES:.cc=.o)         # remplace tous les .cc par .o
EXE_NAME := monexe

#
# cibles
#
$(EXE_NAME) : $(OBJ)
        $(CXX) -o $@ $(OBJ)
%.o : %.cc .deps
        $(CXX) $(CXXFLAGS) -c $<
.deps : $(SOURCES)
```

```
	$(CXX) $(CXXFLAGS) -MM $(SOURCES) > $@
-include .deps

#
# supprimer les fichiers générés
#
clean :
	rm -f *.o $(EXE_NAME) *~ .deps
```

Chapitre 8

Fiche Pratique 2 : compilation Java

Cette fiche pratique s'attache à détailler la compilation de programmes Java en bytecode.

8.1 Compilation

8.1.1 Qu'est-ce que la compilation Java ?

Un ensemble de fichiers Java source est compilé vers un ensemble de fichiers byte code qui peut être exécuté sur une machine virtuelle. À l'inverse du C ou C++ il n'y a pas d'étape d'édition de lien puisque celle-ci est réalisée de manière dynamique au moment de l'exécution.

8.1.2 Compilation d'un ensemble de fichiers *.java

Pour compiler tous les fichiers dans le répertoire src vers le répertoire build :

```
javac -d build `find src -name \*.java`
```

Ainsi si on dispose du fichier `src/org/jasip/GraphTerminal.java` contenant les lignes suivantes

```
package org.jasip;
public class GraphTerminal {
    // ....
}
```

Alors la commande de compilation précédente créé le fichier `build/org/jasip/GraphTerminal.class`.

Pour que les symboles de debug soient présents il faut rajouter l'option `-g` :

```
javac -g -d build `find src -name \*.java`
```

8.1.3 Visualisation du bytecode

Pour visualiser le byte code avec l'exemple précédent il suffit d'invoquer la commande :

```
javap -classpath build -c org.jasip.GraphTerminal
```

Pour une description détaillée de la structure du bytecode se reporter à la version en ligne de [LY99].

8.2 Automatisation de la compilation

8.2.1 Makefiles

Comme pour les fichiers C ou C++ on peut faire des makefiles. Il suffit de faire une cible lançant la bonne commande :

```
compil_java :
        javac -g -d build `find src -name \*.java`
```

Cependant en développement Java on préfère utiliser le logiciel `ant` décrit à la section ci-dessous qui comprend plusieurs raccourcis permettant de facilier la mise en place de gros projets.

8.2.2 ant

La manière standard d des compilations en java est d'utiliser un script ant (`http://ant.apache.org`). Pour cela il suffit de créer le fichier `build.xml` suivant :

```
<?xml version="1.0"?>

<project name="myproject" basedir="." default="compile">

  <!-- initialise les constantes compile.dir et compile.src -->
  <target name="init">
   <property name="compile.dir" value="build"/>
   <property name="compile.src" value="src"/>
  </target>

  <!-- compile le contenu de ${compile.src} dans ${compile.dir} -->
  <target name="compile" depends="init">
    <mkdir dir="${compile.dir}"/>
    <javac srcdir="${compile.src}"
           destdir="${compile.dir}"
           debug="on"/>
  </target>

  <!-- cette cible efface les fichiers générés -->
  <target name="clean">
    <!-- supprime le répetoire build -->
    <delete dir="${compile.dir}"/>
    <!-- supprime dans tous les sous répetoires les fichiers *~ -->
    <delete>
      <fileset dir="." includes="**/*~" defaultexcludes="no"/>
    </delete>
  </target>
</project>
```

Puis d'invoquer la commande `ant`. Pour effacer les fichiers générés appeler `ant clean`.

Chaque cible est décrite dans une balise `<target name="..."> ... </target>`. Son nom est précisé par l'attribut `name`. Elle est invoquée par la commande `ant` *<nom de la cible>*. Si une cible dépend d'une autre cible il faut utiliser l'attribut `depends` pour expliciter la liste des dépendances. Ici la cible `compile` dépend de la cible `init`.

L'ensemble des syntaxes possibles sont décrites sur le site `http://ant.apache.org`.

Bibliographie

[CCG+04] Marcello Coppola, Stephane Curaba, Miltos D. Grammatikakis, Giuseppe Maruccia, and Francesco Papariello. OCCN : A Network-On-Chip Modeling and Simulation Framework. In *Proceedings of the conference on Design, automation and test in Europe - Volume 3*, 2004. `http://occn.sourceforge.net/occn_date04.pdf`.

[Cla88] D. Clark. The Design Philosophy of the DARPA Internet Protocols. *Proceedings of ACM SIGCOMM'88*, pages 106–114, August 1988.

[DZ83] J. Day and H. Zimmerman. The OSI reference model. *Proceedings of the IEEE*, 71(12), December 1983.

[GG00] Pierre Guerrier and Alain Greiner. A Generic Architecture for On-chip Packet-switched Interconnections. In *Proceedings of the DATE'2000 Conference*, pages 250–256, 2000. `ftp://asim.lip6.fr/pub/reports/2000/ar.gue.date00.pdf`.

[iso99] *ISO/IEC 9899 :1999 - Programming language C*. ISO/IEC, 1999.

[iso03] *ISO/IEC 14882 :2003 - Programming language C++*. ISO/IEC, 2003. `http://www.open-std.org/jtc1/sc22/wg21`. Le standard est payant, mais un bouillon est disponible : http ://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2315.pdf.

[Kes97] S. Keshav. *An Engineering Approach to Computer Networking*, chapter 4. Addison-Wesley, 1997.

[Knu81] Donald E. Knuth. *The Art of Computer Programming*, chapter 3. Addison-Wesley, 1981.

[LY99] Tim Lindholm and Frank Yellin. *The Java^{TM} Virtual Machine Specification*. Sun Microsystems, 1999. `http://java.sun.com/docs/books/vmspec/index.html`.

[osc05] *SystemC 2.1 Language Reference Manual*. www.systemc.org, 2005. `http://www.systemc.org/web/sitedocs/lrm_2_1.htm`.

[PE94] D. A. Pucknell and K. Eshraghian. *Basic VLSI Design, 3rd edition*. Prentice Hall, 1994.

[PFTV92] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C : The Art of Scientific Computing*, chapter 7. Cambridge University Press, 1992.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1997. ISBN 0-201-88954-4.

Liste des exercices

1.1 Hello World ! * 30
1.2 Inclusion de .h * 30
1.3 Simulation d'écosystème ** 30
1.4 Addition en temps logarithmique *** 31
1.5 Méta programmation *** 31
2.1 Appel de fonction * 54
2.2 Opérateur et constructeur * 55
2.3 Validité de pointeur ** 56
2.4 Efficacité du `switch` ** 56
2.5 alphabet * 57
2.6 alphabet ** 57
2.7 Space invader *** 57
2.8 Simulateur mémoire ** 58
3.1 Mauvaise utilisation des itérateurs * 68
3.2 Performance des containers associatifs ** 68
3.3 Test des modules ram et cache ** 69
4.1 Couche réseau du modèle OSI ** 79
4.2 Utilisation du réseau ** 80
4.3 Couche transport du modèle OSI *** 81
4.4 Interface réseau *** 81
4.5 Microprocesseur Multicœur *** 81
5.1 Co-Processeur vectoriel *** 89
6.1 Gestion des pointeurs *** 97

Index

Symbols
#, 74
##, 74

A
auto_ptr, 95

B
basic_string, 62
boost, 96

C
c_str, 60
cerr, 21
char, 11, 52
char_traits, 62
compilation, 99
const, 18, 21
constructeur, 13
par défaut, 37
par recopie, 45
cout, 20
cycle en V, 67

D
define, 23
delete, 44
destructeur, 13
do, 22

E
édition de liens, 99
else, 22
endif, 30
enum, 40
extern, 19

F
fifo, 50
__FILE__, 23
for, 22
for_each, 63
friend, 42

H
héritage, 16
multiple, 65

I
if, 21
ifndef, 30
include, 18, 19
inline, 72
int, 11
interface, 64
iterator, 60

J
jasip, 53

K
klocwork, 86

L
lifo, 74
__LINE__, 23
list, 60
long, 11

M
méthode, 13
macro, 23, 27
map, 61

N
namespace, 19
neg, 49
new, 43
npos, 59

O
ofstream, 20
operator, 43
(), 46
*, 43
* binaire, 43
+, 41
-, 42
/, 43
<<, 21, 43
=, 44
==, 45
[], 46
&&, 43
& binaire, 43
^, 43
ostringstream, 60

P
patron, 16
pointeur, 35
de fonction, 38
de méthodes, 39
pos, 49
préprocesseur, 23
private, 13
protected, 17
public, 13
pure virtuel, 64
purify, 86

R
référence, 36
rand, 28
return, 13

S
sc_clock, 48
SC_CTOR, 27, 73
sc_fifo, 50
sc_in_clk, 48
sc_interface, 74
sc_main, 52
SC_METHOD, 73
SC_MODULE, 27, 73
sc_module, 25
sc_start, 48
sc_stop, 48
SC_THREAD, 49
sensitive, 27
short, 11
simulation
SystemC, 47
spécifications, 67
splint, 86
std, 20
STL, 59
string, 59
struct, 31
surcharge, 14

T
tableau, 14
template, 16
de fonction, 62
this, 46
typedef, 27

U
unsigned, 11
using, 20

V
valgrind, 86
virtual, 17, 66

W
while, 22