LES GUIDES DE
GNU LINUX MAGAZINE / FRANCE
HORS-SÉRIE
N°83

France MÉTRO. : 12,90 € — CH : 18,00 CHF — BEL/PORT.CONT : 13,90 € — DOM TOM : 13,90 € — CAN : 18,00 $ CAD

DÉBUTEZ EN C++

LE GUIDE POUR APPRENDRE LES PRINCIPES CLÉS DU LANGAGE !

NIVEAU DÉBUTANT

Compatible Windows / Linux / Arduino

AVANT DE COMMENCER

Une rapide présentation du langage C++

VOTRE AGENDA

JOUR 1 : Installez vos outils et découvrez les bases du C++
JOUR 2 : Débutez votre projet
JOUR 3 : Abordez les types avancés et les pointeurs
JOUR 4 : Modélisez des objets
JOUR 5 : Écrivez des classes génériques
JOUR 6 : Finalisez votre programme

AIDE-MÉMOIRE

Les concepts clés du langage

Édité par Les Éditions Diamond
L 15066 - 83 H - F: 12,90 € - RD
www.ed-diamond.com

Retrouvez toutes nos publications
LES ÉDITIONS DIAMOND
sur **www.ed-diamond.com**

GNU/Linux Magazine Hors-Série
est édité par **Les Éditions Diamond**

10, Place de la Cathédrale - 68000 Colmar - France

Tél. : 03 67 10 00 20 / **Fax :** 03 67 10 00 21

E-mail : cial@ed-diamond.com
lecteurs@gnulinuxmag.com

Service commercial : abo@gnulinuxmag.com

Sites : www.gnulinuxmag.com
www.ed-diamond.com

Directeur de publication : Arnaud Metzler

Chef des rédactions : Denis Bodor

Rédacteur en chef : Tristan Colombo

Remerciements : Sébastien Chazallet, Yohan Andreotti

Responsable Service Infographie : Kathrin Scali

Conception graphique : Thomas Pichon

Responsable publicité : Tél. : 03 67 10 00 27

Service abonnement : Tél. : 03 67 10 00 20

Impression : pva, Druck und Medien-Dienstleistungen GmbH, Landau, Allemagne

Distribution France :
(uniquement pour les dépositaires de presse)

MLP Réassort :
Plate-forme de Saint-Barthélemy-d'Anjou.
Tél. : 02 41 27 53 12
Plate-forme de Saint-Quentin-Fallavier.
Tél. : 04 74 82 63 04

Service des ventes :
Distri-médias : Tél. : 05 34 52 34 01

IMPRIMÉ en Allemagne - PRINTED in Germany

Dépôt légal : A parution

N° ISSN : 0183-0864

Commission Paritaire : K78 976

Périodicité : Bimestrielle

Prix de vente : 12,90 Euros

Les articles non signés contenus dans ce numéro ont été rédigés par les membres de l'équipe rédactionnelle des Éditions Diamond.

PRÉFACE

Le développement logiciel est une science exacte et comme toute science, elle a ses principes inaliénables et ses lois incontournables. Développer un logiciel est un exercice qui requiert de multiples compétences qui vont bien au-delà de la simple connaissance d'un langage de programmation.

Mais cette connaissance est un atout indispensable, sa maîtrise essentielle pour prendre de la distance avec le projet. Il faut savoir ce qu'il est possible de faire et l'avoir pratiqué suffisamment pour être capable de concevoir la solution idéale, la plus simple, la plus directe, la plus élégante, puis organiser son projet en briques indépendantes, réutilisables, testables, en un mot : fiables.

Cette exigence de rigueur, cette hauteur de vue et cette recherche de perfection sont une des caractéristiques de C++. Il ne s'agit pas simplement d'un langage qui s'applique comme une recette de cuisine par des développeurs élevés en batterie – pour paraphraser Jean-Pierre Troll, il s'agit d'un langage qui façonne la pensée du développeur, qui lui inculque les bonnes pratiques, qui lui apprend à lever les yeux hors de son écran et à penser son code avant de l'écrire.

Il a repris de très nombreux concepts nés avant lui et s'est appuyé sur le langage C – langage intemporel, pérenne et ayant fait l'objet d'un précédent hors-série – qui faisait déjà preuve de nombre de qualités citées plus haut. Ce qui le distingue est la grande cohérence avec laquelle l'ensemble des concepts a été intégré et le fait qu'il a su poser un cadre permettant de répondre à des problématiques très complexes sans se dénaturer.

C'est la raison pour laquelle, aujourd'hui encore, il est toujours le langage le plus utilisé dans le milieu industriel (côte à côte avec Java, selon des classements dont la compréhension m'échappe) et qu'il est indétrônable dans des domaines exigeants, comme la programmation temps réel et les jeux vidéos.

Cette pérennité ne doit cependant rien au hasard. Non seulement le langage a été bien été conçu au départ, mais il continue d'évoluer régulièrement – cycle normatif de trois ans – en innovant constamment, en répondant aux exigences de ses utilisateurs, un introduisant de nouvelles pratiques mais surtout, en cherchant à améliorer sa cohérence à chaque cycle.

Ayant la connaissance de ce passif impressionnant, il est inutile de préciser que ce langage a un avenir radieux devant lui et qu'il sera encore, très probablement, un langage très important à mettre sur son CV. Autant dire que son apprentissage est indispensable.

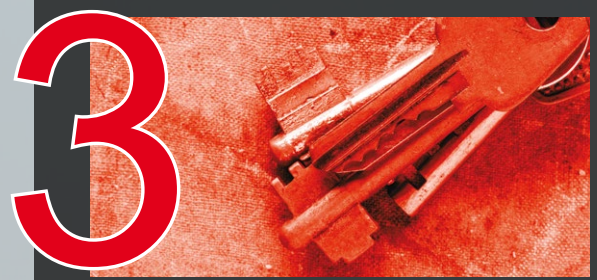
Ce hors-série sera votre guide pour vous permettre de prendre possession des aspects essentiels de C++, que vous soyez sous Linux ou Windows. Vous apprendrez les bases essentielles pour construire une application console, sur lesquelles vous pourrez vous appuyer dans un second temps pour développer des applications graphiques GNOME, KDE ou encore Raspberry PI, Arduino ou même Android ! En effet, le cœur du langage et les techniques que nous verrons resteront les mêmes partout.

Enfin, si vous connaissez déjà Python, vous apprendrez à maîtriser un des langages qui lui est le plus opposé du point de vue des concepts, mais aussi de leur application concrète, puisque les avantages de C++ sont les inconvénients de Python et vice-versa. D'où le fait que ces deux langages travaillent parfaitement bien ensemble et sont régulièrement associés. Autant dire que la connaissance de C++ vous apportera un complément indispensable et une nouvelle compréhension des concepts propres à chaque langage !

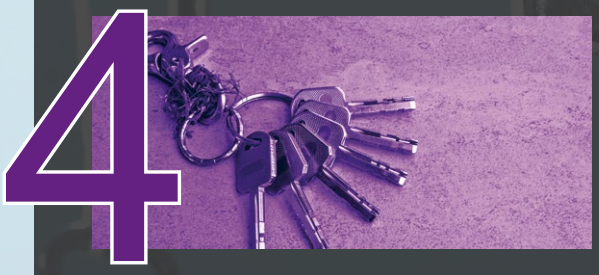
Sébastien CHAZALLET & Yohan ANDREOTTI

Sommaire

GNU/Linux Magazine Hors-Série N°83



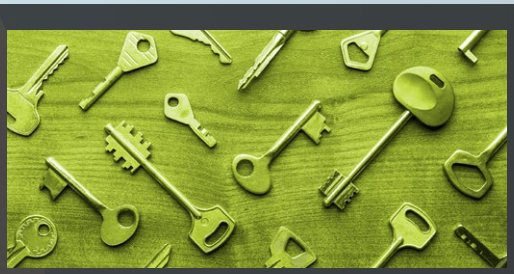
DÉBUTEZ EN C++



JOUR 4

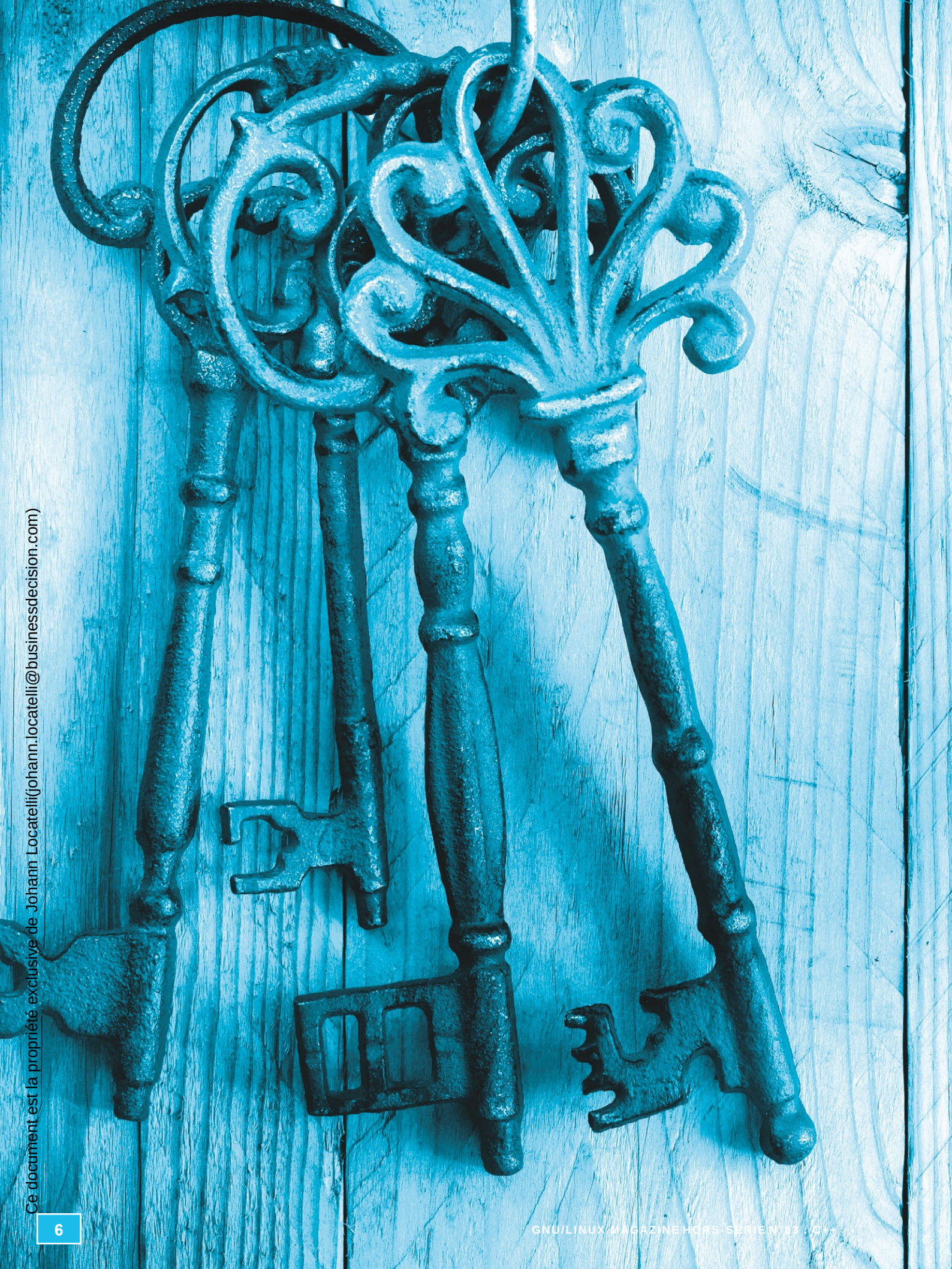
JOUR 5

JOUR 6



BONUS

Code source disponible sur : **https://github.com/GLMF/GLMFHS83**



INTRODUCTION

AVANT DE COMMENCER, UNE RAPIDE PRÉSENTATION DU C++

Depuis plus de 30 ans, la popularité de C++ ne s'est jamais démentie. À ce jour, il est l'un des langages les plus utilisés, en particulier dans des domaines très exigeants, comme le temps réel.

Il présente en effet de sérieux avantages : extrêmement cohérent, portant de nombreux concepts essentiels, reposant sur le C, référence absolue des langages statiques bas niveau, et permettant l'utilisation du paradigme objet, il a toutes les armes pour permettre le développement rapide de logiciels complexes.

Le C++ est un langage de programmation créé en 1983. Il est de *typage statique* et utilise les *paradigmes générique*, *orienté objet* et *procédural*. Son auteur est **Bjarne Stroustrup**.

1. LA GENÈSE

En 1972, **Dennis Ritchie** publiait le langage **C**. Il deviendra très rapidement extrêmement populaire et en 1980, il était déjà la référence des langages statiques à paradigme procédural. Performant, portable, il s'imposait dans de nombreux domaines, mais n'était pas adapté à des projets de taille et de complexité importantes. En effet, si le C est parfaitement adapté à des programmes courts ou de taille moyenne, il est plus complexe de l'utiliser pour les très gros projets, lesquels nécessitent par ailleurs beaucoup de temps de conception et de modélisation.

La même année sortait **Smalltalk**, premier langage objet en tant que tel (le paradigme objet avait été introduit dix ans plus tôt par **Simula**) et apportant nombre d'autres innovations. Le paradigme objet est une réponse idéale pour gérer cette augmentation en taille et en complexité des logiciels. Par contre, le défaut de ces langages restait leur lenteur.

Les langages **C** et **Simula** sont les deux principales influences de **Bjarne Stroustrup [1]** lorsqu'il sort en 1980 **C with classes**. Venant de terminer son doctorat, il travaille aux laboratoires Bell (tout comme **Dennis Ritchie** lorsqu'il a publié le **C**) sur le noyau UNIX et propose ainsi un premier jet dont l'idée consistait à ajouter le *paradigme objet* au langage **C**.

Ce langage deviendra **C++** dès 1983. Le ++ étant l'opérateur d'incrémentation, **C++** se présente donc comme une amélioration du langage **C**. En 1995, le livre de référence *The C++ Programming Language* écrit par l'auteur du langage paraît (il en écrira de très nombreux autres) permettant de servir de référence au langage.

2. LES DIFFÉRENTES VERSIONS

Le renommage du langage en **C++** est considéré comme la toute première version utilisable du langage. Par rapport à **C with classes**, elle avait introduit les notions très importantes de *fonctions virtuelles* ainsi que la *surcharge des opérateurs* et *des fonctions*. On notera aussi l'amélioration du *typage fort* et les commentaires en fin de ligne ainsi que les *références* et les constantes.

La version 2.0 apportera l'*héritage multiple* ainsi que les *classes abstraites* et l'amélioration de l'*encapsulation* de classe.

Le **C++** répond à différentes **normes** qui ont amélioré le langage et sa librairie standard au fil du temps. Un groupe de travail international travaille à la standardisation et à la définition des évolutions des différentes versions **[2]**. La première date de 1998 et correspond à la norme ISO/CEI 14882:1998 (aussi nommée **C++98**).

Les suivantes datent successivement de 2003, 2011 et la dernière date de 2014 :

- ISO/CEI 14882:2003 (**C++03**) ;
- ISO/CEI 14882:2011 (**C++11**) ;
- ISO/CEI 14882:2014 (**C++14**).

Les deux dernières mises à jour du langage sont intervenues après un long délai de près de 10 ans. Les différentes évolutions ont été réalisées avec différentes contraintes, dont :

- conserver la *compatibilité* avec les normes de C++ précédentes ;
- introduire des modifications qui simplifient les techniques de programmation. Les nouvelles techniques ne doivent pas avoir d'impact de performance tant qu'elles ne sont pas utilisées ;
- améliorer le typage, en proposant de nouveaux types plus robustes ; ajouter du contenu à la librairie standard plutôt que de modifier le langage.

En 2011, la mise à jour majeure a ajouté des fonctionnalités nombreuses et structurantes, dont nous ne pouvons citer que quelques-unes des plus importantes

- le type `auto` qui permet au compilateur de déduire automatiquement le type à utiliser, ce qui permet en général d'avoir à écrire à beaucoup moins de code, le rendant plus lisible ;
- le mot clé `nullptr` qui remplace avantageusement l'ancien `NULL` ;
- le support amélioré des boucles (paradigme `foreach`) ;
- les nouveaux mots-clés `final`/`override` pour mieux définir les mécanismes d'héritage ;
- les nouvelles classes de *smart pointers*, permettant de gérer la mémoire plus simplement ;
- les *lambda fonctions*, ou *fonctions anonymes* ;
- la *sémantique de déplacement de mémoire* (`&&`) pour optimiser la copie d'objet, particulièrement la copie en profondeur ;
- les *littéraux* définis par l'utilisateur, améliorant la lisibilité du nommage et des opérations sur les valeurs.

En 2014, nous avons vu l'arrivée d'une mise à jour mineure, introduisant de nouveaux concepts et généralisant des mécanismes introduits dans la mise à jour précédente :

- les *littéraux binaires*, *séparateurs de chiffres* améliorant la lisibilité des valeurs numériques ;
- les *recherches hétérogènes* dans les *containers*, permettant de faire des recherches sur des objets de types différents ;
- l'amélioration du support du mot clé `auto` pour les *fonctions lambdas* (*generic lambda*) et l'usage dans le *retour de fonction* (*return type deduction*).

3. PERSPECTIVES POUR LE FUTUR

Après la mise à jour mineure de 2014, il est attendu une mise à jour majeure pour 2017 [3]. Le contenu n'est pas encore figé, mais plusieurs améliorations importantes sont à l'étude. Nous pouvons les classer en plusieurs catégories.

En premier lieu, on constate toujours plus d'évolutions de la syntaxe du langage, améliorant sa cohérence, sa robustesse afin de simplifier son usage. Citons par exemple : l'ajout du mot clé `typename` dans la définition de *template*, le support des caractères UTF-8, la fiabilisation des types dans les *unions* ou la consolidation de la syntaxe pour l'appel des fonctions.

Un grand nombre de nouveautés sont liées à l'ajout du support natif de fonctionnalités, par exemple le *support du réseau*, la *manipulation de fichiers* ou le support de *co-routines*. Celles-ci pouvaient être disponibles dans différentes bibliothèques annexes, mais l'intégration de ces fonctionnalités permet de normaliser leur usage. Un certain nombre d'entre elles faciliteront l'écriture d'applications réparties, donc à destination des applications pour le *cloud*.

Enfin, il y aura une amélioration des performances, que ce soit par l'ajout du support d'instructions optimisées pour certains hardware (SIMD), un mappage plus direct avec le matériel (comme en C) ou l'amélioration du temps de compilation.

4. NOTIONS IMPORTANTES

Si vous n'êtes pas familier avec les principaux concepts objets, il est nécessaire de s'attarder un peu sur quelques-uns d'entre eux de manière à poser les bases qui vous permettront de lire ce hors-série sereinement. Nous aurons l'occasion de rentrer dans les détails dans les articles suivants.

Le premier concept est la **classification**. Lorsque l'on écrit un programme en utilisant le paradigme impératif, le plus connu, le plus accessible, on écrit son algorithme comme une succession d'instructions. On peut le factoriser par la création de fonctions et on peut classer ces fonctions dans des modules pour structurer son code. Lorsque l'on utilise le paradigme fonctionnel, on va s'attacher non pas à l'algorithme, mais à la donnée et on va décrire les différentes transformations et filtres appliqués sur cette donnée. Ce sont deux manières radicalement différentes de penser et donc d'écrire du code.

Le paradigme objet, est également une manière de penser et d'écrire du code. Il nécessite de modéliser nos données et de les classifier. Chaque donnée sera un objet qui sera une *instance* d'une *classe*. Cette classe permettra de décrire l'objet, par l'intermédiaire de ses *attributs* et de préciser comment il pourra évoluer ou être utilisé, par l'intermédiaire de ses *méthodes*. Écrire un programme consiste donc à manipuler des objets et les faire interagir entre eux.

La phase de *modélisation* détermine comment un programme sera structuré, donc comment seront définies les interfaces entre les différentes classes du programme et le contenu de chaque classe. Cette étape préliminaire est déterminante en C++, car si le paradigme objet permet de faire des économies en complexité de code et en temps de développement, c'est grâce à une modélisation réussie.

L'*encapsulation* consiste d'une part à positionner toutes les données utiles à un objet dans sa classe et d'autre part à gérer finement quelle classe peut voir les attributs ou méthodes de chaque classe. Pour la plupart des langages, tels que le langage C++, il existe trois niveaux de visibilité :

- ⇨ *public* : tout le monde peut accéder à l'élément ;
- ⇨ *protégé* : seuls les éléments de la classe et des classes filles peuvent accéder à l'élément ;
- ⇨ *privé* : seuls les éléments de la classe peuvent accéder à l'élément.

L'*héritage* est un des concepts clés du paradigme objet. Il permet la **réutilisabilité** et l'**adaptabilité** des objets. En effet, il est fréquent que des classes différentes partagent un certain nombre de comportements, mais divergent sur un certain nombre d'autres. Les comportements partagés sont alors décrits par une *super-classe* et les comportements spécialisés de chaque classe dans chacune des *classes dérivées*.

Cependant, lorsque l'on manipule des objets d'une classe dérivée, on présente la même interface que les objets de la super-classe ou que ceux d'une autre classe dérivée. On appelle ceci le *polymorphisme* : on peut manipuler de la même manière des objets de classes différentes.

L'*héritage multiple* permet à une classe d'hériter de plus d'une super-classe.

Dans le contexte d'héritage, on peut vouloir *spécialiser* une classe. Pour ce faire, on crée une nouvelle classe en la dérivant. La classe d'origine devient alors la super-classe de la nouvelle classe qui est la classe dérivée. Cependant, cette super-classe reste utilisable

directement, c'est à dire *instanciable* (on peut en créer des instances). Si la classe dérivée définit une fonction qui existe déjà dans la super-classe, cette dernière est dite *surchargée*. C'est-à-dire que les instances de la classe dérivée utiliseront la méthode de leur classe plutôt que de leur super-classe.

En aucun cas, il n'est interdit de définir plus de fonctions dans la classe dérivée que dans la super-classe et il est parfaitement possible d'avoir des attributs et des méthodes totalement différentes dans deux classes dérivées de la même super-classe.

On peut aussi souhaiter avoir plusieurs classes dérivées qui partagent des mêmes noms de méthodes, mais leur laisser l'initiative de l'*implémentation*. On créera alors dans la super-classe des *méthodes abstraites* (ou *retardées*). Les classes filles seront alors obligées de les implémenter pour être *effectives* (*instanciables*).

Une classe qui a au moins une méthode abstraite est une *classe abstraite*. Cette classe n'est pas instanciable.

On appelle une classe n'implémentant aucune méthode une *interface*. Ces interfaces sont très importantes, car elles définissent un modèle d'implémentation pour les objets voulant réaliser le fonctionnel attendu, quelle que soit la façon dont la classe implémente l'interface.

CONCLUSION

Plus de 30 ans après sa création, le C++ est un langage qui a fait ses preuves et qui est toujours en train d'évoluer. Il est cohérent, performant, populaire et très utilisé, en particulier dans des domaines très exigeants, comme la programmation temps réel ou le développement de grands jeux.

Il a suivi sa propre voie, le C restant privilégié pour le système et le C++ pour les logiciels à interface graphique hors système ou demandant un plus fort niveau d'abstraction (sans que cela soit une règle absolue). Il doit son succès à la présence de documentations de référence et surtout à sa bibliothèque standard très fournie et tout aussi performante.

Il est devenu une véritable référence et une source d'innovation. Nous allons vous le faire découvrir à travers la réalisation d'un projet amusant, mais néanmoins didactique, l'écriture d'un jeu de casse-briques, que nous allons présenter dès la prochaine page. ■

RÉFÉRENCES

[1] Site officiel de Bjarne Stroustrup : **http://www.stroustrup.com/**

[2] Site officiel du standard C++ : **https://isocpp.org/**

[3] Site officiel du standard C++ (planning des prochaines fonctionnalités) : **https://isocpp.org/std/status**

POUR ALLER PLUS LOIN

Pour en savoir plus sur le langage C++ et son auteur, vous pouvez consulter son site officiel **[1]** et surtout la FAQ : **http://www.stroustrup.com/C++11FAQ.html**.
Plusieurs sites contiennent des références très complètes de la librairie standard, nous pouvons citer par exemple : **http://www.cplusplus.com/reference/**.

JOUR 1

JOUR 1

INSTALLEZ VOS OUTILS ET DÉCOUVREZ LES BASES DU C++

Ce premier article a pour vocation de vous donner tous les éléments pour que vous soyez capables de travailler dans des conditions optimales.

Nous allons également introduire quelques notions élémentaires, préciser quelques points de vocabulaire et détailler ce qu'est un programme et comment l'on passe du code source au programme.

Le fil rouge de ce hors-série consiste à écrire un jeu de casse-briques. Tout au long des articles à venir, à commencer par celui-ci, nous introduirons différents concepts en utilisant ce fil rouge pour les mettre en application. Nous commencerons par le plus basique en construisant de toutes petites fonctionnalités, puis au fur et à mesure que l'on abordera de nouveaux concepts, le programme se structurera.

Bien que le résultat concret du programme ne soit véritablement testable qu'à la fin du hors-série, la progressivité des articles permettra de traiter beaucoup d'aspects très différents et de monter en puissance peu à peu, évitant de se noyer dans la complexité dès les premières lignes du premier article.

Si la problématique complexe de l'affichage sera traitée tout à la fin, les concepts essentiels comme la structuration d'un programme et la notion d'objet seront abordés assez rapidement. Avant cela, cet article se propose de présenter différents rudiments encore plus essentiels, tels que la définition des mots suivants : **type**, **littéral**, **variable**, **compilation**, **exécution**...

Aussi, nous n'allons pas encore commencer à écrire notre application, mais vous pourrez trouver des extraits de code issus de cet article dans le répertoire `jour1` du dépôt GitHub (**https://github.com/GLMF/GLMFHS83**) et faire vos propres manipulations à partir de là.

1. ENVIRONNEMENT DE DÉVELOPPEMENT

1.1 Choisir son environnement

La principale difficulté d'apprentissage consiste à mettre en pratique ce qui nous est proposé. Et le grand avantage d'un langage comme le C++ est que l'on dispose de tous les outils pour écrire son propre code et tester par soi-même. Ces outils sont capables de nous guider, de nous dire si l'on est sur le bon chemin ou non. En effet, si notre programme ne compile pas ou ne s'exécute pas correctement, c'est forcément une erreur de notre part : l'ordinateur ne se trompe jamais (en tout cas, à notre niveau, parce qu'un bug dans l'écriture d'un compilateur est toujours possible, mais pas sur une fonctionnalité très utilisée, grâce à une excellente communauté).

La première chose que nous allons faire va donc consister à installer ces outils, en commençant par un **environnement de développement**. En effet, étant donné que nous allons devoir taper de nombreuses lignes de codes dans différents fichiers que nous aurons à organiser dans une arborescence et que nous devrons être capables de rechercher, d'ouvrir et de sauvegarder rapidement, il nous faut un outil pour le faire le plus confortablement possible.

Reste à définir la notion de confort. À mon humble avis, il est impossible de se passer de certaines fonctionnalités basiques telles que la coloration syntaxique, le signalement des erreurs basiques, des outils de compilation rapides et la possibilité de visualiser le contenu de plusieurs fichiers en même temps.

Si vous utilisez déjà un tel outil pour développer dans un autre langage et que cet outil permet aussi de faire du C++, je vous encourage à le conserver. Vous connaîtrez ainsi déjà les raccourcis clavier et vous aurez la meilleure productivité possible. Si vous êtes un

inconditionnel du terminal, vous pourrez parfaitement utiliser **Vim** ou **Emacs** pour éditer les fichiers à condition de les associer aux bons greffons pour la coloration syntaxique, entre autres, ainsi que divers autres programmes de compilation qui sont utilisables directement depuis la ligne de commandes et que l'on présentera ultérieurement.

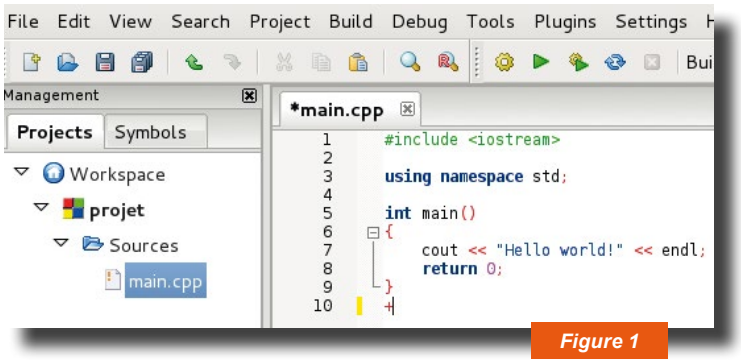


Figure 1

Si vous n'avez pas d'outils de référence, nous vous conseillons d'utiliser un environnement de travail graphique qui est relativement léger, très simple à utiliser et néanmoins très performant : **Code::Blocks**.

1.2 Installation sur Linux

Sur une distribution de type Debian, il s'installe ainsi (avec les greffons additionnels et en s'assurant que vous possédez bien le compilateur C++) :

Terminal

```
$ sudo aptitude install codeblocks codeblocks-contrib build-essential
```

1.3 Installation sous Windows

Sur Windows, il faut se rendre à la page de téléchargement du logiciel et choisir le bon fichier à télécharger, en fonction de vos besoins.

Si vous disposez déjà du compilateur C++, vous pouvez choisir l'installateur par défaut, nommé à l'instant où l'on écrit ces lignes **codeblocks-16,0,0-setup.exe**. Si vous ne disposez pas du compilateur, il faudra choisir la version **mingw** et si en plus vous n'avez pas non plus fortran et que vous en avez besoin (a priori, pas pour nous donc), vous aurez besoin de la version **mingw_fortran**.

De plus, il existe également une version **nonadmin** pour ceux qui ne disposent pas des droits d'administrateur sur leur machine, ce qui arrive souvent au travail, cette version ne peut cependant pas installer le compilateur. Il faudra trouver une autre solution pour cela.

Enfin, il existe aussi deux versions **nosetup** (une avec le compilateur et une sans) pour ceux qui sont allergiques aux installateurs (citation du site). Nous ne le conseillons pas, parce qu'il faut ensuite aller bidouiller dans les fichiers décompressés pour adapter le logiciel à ses besoins.

Une fois que vous avez votre installateur, vous pouvez le lancer et cliquer sur suivant autant de fois que nécessaire. Une seule page est réellement importante, celle qui vous permet de choisir les éléments à installer. Vous verrez ainsi au minimum que vous disposez comme pour linux d'une séparation entre le logiciel et des contributions externes et aussi qu'il existe une option **C::B Share Config** permettant de se donner les moyens de partager sa configuration (de l'exporter sur un autre PC facilement).

1.4 Utilisation du logiciel, premiers pas

Comme vous pouvez le voir sur la capture d'écran de la figure 1, vous disposez à gauche d'une arborescence de votre projet, à droite de vos fichiers ouverts et vous avez également une barre d'outils qui comprend entre autres l'icône en forme d'engrenage jaune qui permet de **compiler** son code, la flèche verte qui permet d'**exécuter** la toute dernière version compilée du code (pas forcément la version actuelle si vous n'avez pas compilé depuis vos dernières modifications) et l'icône avec une flèche verte au-dessus d'un engrenage jaune qui permet de **compiler**, **puis** d'**exécuter** (si la compilation a réussi).

Cette dernière icône est de loin la plus utilisée lorsque l'on développe et son raccourci clavier est <**F9**>. Enfin, l'éditeur de texte numérote les lignes et vous signalera le nouveau code que vous venez d'écrire par un trait vertical orangé et les erreurs de compilation par un rectangle rouge.

En bas de votre fenêtre (hors de la capture), vous apercevrez plusieurs onglets ; l'un d'entre eux est le résultat du compilateur, il vous affichera une ligne par erreur. Un autre relativement important est l'onglet présentant les résultats d'une recherche. On peut faire une **recherche** en utilisant le raccourci clavier <**Ctrl**> + <**F**> et un ***rechercher & remplacer*** par l'utilisation du raccourci clavier <**Ctrl**> + <**R**>.

Si vous débutez, vous familiariser avec cet environnement de travail vous permettra d'aller plus vite. Vous pourrez vous référer à la documentation du logiciel.

2. NOTION DE PROGRAMME

2.1 Le programme et son environnement

Un *programme* est un *fichier*, présent sur le disque dur, et qui a la particularité d'être *exécutable* (ce qui signifie concrètement que vous pouvez l'exécuter en double-cliquant dessus à l'aide de votre souris ou en appelant le programme depuis un terminal en tapant une commande.

Le terminal a un avantage : il permet de passer des *arguments* au programme. Voici un exemple :

Terminal

```
$ cp fichier1 fichier2
```

Dans cet exemple, `cp` est le nom du programme (il sert à copier un fichier) et `fichier1` et `fichier2` sont les deux *arguments*.

2.2 Point d'entrée

Un programme C++ se compose toujours d'au moins un fichier nommé `main.cpp`.
Ce nom est normalisé : il s'agit du fichier qui contient le point d'entrée du programme. Plus exactement, ce dernier est la fonction `main`, là encore un nom normalisé et que ce fichier doit absolument contenir. Voici le contenu minimal d'un code, permettant d'avoir une compilation qui fonctionne correctement :

Fichier

```
01: int main()
02: {
03: }
```

Cet exemple est un programme qui ne fait absolument rien : la syntaxe est minimaliste et néanmoins correcte, permettant un processus de compilation et une exécution qui fonctionne. Le point important est que le nom de la fonction qui est le point d'entrée du programme est **main** et qu'elle renvoie un entier. Pourquoi un entier ? Cela a trait au fonctionnement du système d'exploitation, encore et toujours.

En effet, tout programme renvoie un entier positif ou nul. Par convention, si le programme quitte proprement (sans erreur, en ayant fait son travail), la valeur de retour est **0**. Toute autre valeur signifie que le programme a terminé en erreur. Le fait de retourner un entier est obligatoire (**void main** engendrerait une erreur de compilation), mais le choix de retourner telle ou telle valeur est celui du développeur.

2.3 Passage d'arguments

L'exemple plus haut est strictement équivalent à celui-ci, plus verbeux :

Fichier

```
01: int main(int argc, char* argv[])
02: {
03:     return 0;
04: }
```

D'une part, à la ligne 3, on voit ici le retour explicite de l'entier **0** (qui est renvoyé de manière implicite dans l'exemple précédent). Plus important, on voit surtout à la ligne **1** une *signature* plus complète pour la fonction **main**. La signature d'une fonction est sa première ligne et se compose du type de la valeur qui sera retournée, du nom de la fonction et des paramètres, entre parenthèses. Ceci s'oppose au *corps de la fonction* qui est l'ensemble des instructions se trouvant entre les accolades.

Le premier paramètre de la fonction, **argc**, pour *argument count*, est un entier qui donne le nombre d'arguments en entrée de la fonction. Le second paramètre de la fonction, **argv**, pour *argument values*, est un tableau de pointeurs vers des chaînes de caractères (qui seront présentées ultérieurement).

Attention ici à ne pas confondre la notion d'argument et celle de paramètre. Toute fonction C++ peut prendre zéro, un ou plusieurs paramètres, ces derniers sont alors utilisables dans le corps de la fonction. Au contraire, les arguments sont les valeurs qui sont passées à une fonction lorsqu'elle est appelée :

Fichier

```
01: int f(int a, int b) { // a et b sont des paramètres.
02:     g(b, a); // a et b sont des arguments.
03: }
```

La plupart du temps, cette distinction est uniquement d'ordre sémantique. Dans ce cas précis, il faut être minutieux, car les arguments passés par le système d'exploitation ne correspondent pas un à un à ceux de la fonction **main**.

En effet, le premier paramètre va nous indiquer combien d'arguments ont été passés et donc combien de valeurs sont à aller chercher dans le second paramètre. Ainsi, si `argc` vaut `4`, cela signifie que `argv` contient `4` chaînes de caractères qui sont les `4` arguments qui ont été passés en paramètre.

C'est le cas si le programme a été appelé ainsi :

Terminal
```
$ programme arg1 arg2 arg3 arg4
```

En effet, le *séparateur* entre les arguments (ainsi qu'entre le nom du programme et le premier argument) est l'espace. Il faut donc faire attention à ne pas lire trop vite et savoir qu'il est possible de mettre un espace dans une valeur d'un argument soit en l'échappant, soit en utilisant des guillemets droits :

Terminal
```
$ programme test v1,v2,v3  a\ b\ c "a b c"
```

Dans ce cas, la première valeur est `test`, la deuxième est `v1,v2,v3`, et les deux dernières sont `a b c` (espaces compris). Encore une fois, cette règle est due au fonctionnement du système d'exploitation ; l'exécution d'un programme en est indissociable.

De la même manière, c'est lui qui se chargera de renseigner `argc` et de construire la donnée `argv`. En ce qui vous concerne, vous devez simplement savoir appeler votre programme en lui transmettant des arguments et savoir les récupérer, ce que l'on verra ultérieurement.

Pour information, une chaîne de caractères peut aussi être représentée sous forme de pointeur au lieu de tableau (on présentera cette notion plus tard). Pour ce qui nous intéresse aussi, il faut savoir que l'on peut utiliser indifféremment l'une ou l'autre forme. D'autre part, les noms `argc` et `argv` sont des conventions. Ces variables peuvent prendre le nom que vous souhaitez (cependant, respecter les conventions peut vous permettre de vous y retrouver plus facilement dans votre code). Ainsi, ceci est tout aussi valide :

Fichier
```
01: int main(int ac, char** av)
02: {
03:     return 0;
04: }
```

Un dernier détail d'importance : on ne déclare pas les arguments de `main` dans sa signature si l'on n'en a pas besoin, c'est-à-dire si l'on ne les utilise pas dans le corps de la fonction `main`. La raison en est simple : ne pas les utiliser signifie que l'on n'a pas besoin d'argument et que le programme refusera tout argument, ce qui est plus cohérent que le fait de laisser l'utilisateur passer des arguments qui ne seront jamais pris en compte.

Si ceci n'est pas respecté, un avertissement (`-Wunused-parameter`) sera émis lors de la phase de compilation, détaillée plus loin.

2.4 Hello world

Maintenant que l'on sait comment s'initie un programme en C++, on peut passer à l'indémodable programme hello world! :

Fichier

```
01: #include <iostream>
02:
03: int main()
04: {
05:     std::cout << "Hello world!" << std::endl;
06:     return 0;
07: }
```

On retrouve divers éléments déjà introduits. D'une part, on a gardé notre fonction **main** avec sa signature sans argument, puisque l'on n'en utilise pas. D'autre part, elle renvoie de manière explicite la valeur **0**, ce qui signifie que le programme quitte en indiquant au système d'exploitation que tout s'est bien passé.

Cet exemple nous permet aussi de voir plusieurs nouveaux éléments : on réalise en effet un affichage. Là encore, cette fonctionnalité fait intervenir le système d'exploitation : il y a trois flux d'entrées/sorties qui sont le flux d'entrée standard **in**, la sortie standard **out** et la sortie d'erreur **err**.

Le premier nous permet de demander une saisie à l'utilisateur du programme, les deux derniers nous permettent de lui communiquer des informations. Le fait de disposer de deux canaux de sortie permet de séparer l'affichage standard de celui des erreurs.

Afficher un message dans la sortie standard est l'une des opérations les plus communes. Pour ce faire, on utilise la fonction **cout**. Et cette dernière ne vient pas du néant, elle fait partie du module **iostream** qui est inclus dans notre programme par l'instruction de la première ligne. Le fait d'inclure un module nous permet d'utiliser tout ce qu'il contient, en particulier dans ce cas, les fonctions **cout** et **endl**.

On notera que **#include** est une directive pré-processeur. On y reviendra plus loin. Mais on notera également que ces fonctions sont préfixées par **std::**. En effet, les deux points permettent de préciser par un préfixe le nom de l'espace de nommage auquel la fonction appartient. Dans ce cas, il s'agit de l'*espace de nommage* de la bibliothèque de fonctions standard qui se nomme **std**.

Les notions de module et d'espace de nommage sont les deux nouveautés sur lesquelles nous allons maintenant nous pencher.

2.5 Espaces de nommage

À quoi sert un espace de nommage ? La réponse est multiple. En effet, si vous écrivez un petit programme, vous n'avez pas besoin d'utiliser les espaces de nommage. Vous arriverez toujours à différentier les noms de vos fonctions et vous saurez vous y retrouver sans trop de problèmes.

Dans ce cas, vous travaillerez alors toujours dans l'espace de nommage **std** et vous pourrez simplifier votre programme :

Fichier

```
01: #include <iostream>
02:
03: using namespace std;
04:
05: int main()
06: {
07:     cout << "Hello world!" << endl;
08:     return 0;
09: }
```

Par contre, si vous écrivez un programme d'une taille conséquente, les espaces de nommage vont vous permettre de mieux organiser votre code. En créant vos propres espaces de nommage et en regroupant vos fonctions à l'intérieur de ceux-ci selon des règles de bon sens, vous pourrez éviter des conflits de nom, mieux identifier le rôle et l'action de ces fonctions et au final vous permettre de simplifier grandement votre code, en l'organisant mieux, en le clarifiant et en améliorant sa lisibilité.

Il faut cependant savoir qu'au sein d'un espace de nommage donné il est toujours possible d'utiliser des fonctionnalités d'autres espaces de nommage et qu'il est aussi possible de n'utiliser que quelques-unes des fonctions d'un espace de nommage :

Fichier

```
01: #include <iostream>
02:
03: using std::cout;
04:
05: int main()
06: {
07:     cout << "Hello world!" << std::endl;
08:     return 0;
09: }
```

Dans ce dernier exemple, la fonction **cout** est utilisable sans avoir à la préfixer par le nom de son espace de nommage grâce à l'instruction de la ligne 3, au contraire de **endl**. On approfondira cette notion d'espace de nommage dans l'article suivant.

2.6 Modules

Un *module* est un ensemble de fonctionnalités dont le code a été regroupé dans un seul et même fichier. La bibliothèque standard de C++ est une immense collection de fonctionnalités qui est livrée avec le langage C++ qui permet de réaliser la plupart des opérations essentielles. Elle est organisée selon un ensemble de modules, chaque module regroupant des fonctionnalités d'un thème spécifique.

Pour inclure un module appartenant à la bibliothèque standard, la syntaxe est la suivante :

Fichier

```
#include <nom_du_module>
```

Mais la notion de module ne se limite pas à la bibliothèque standard. Nous pouvons nous aussi créer nos propres modules et nous avons même intérêt à le faire, toujours pour des raisons d'organisation de code.

Ainsi, chaque fichier de code peut ainsi être considéré comme une bibliothèque personnelle et aura vocation à contenir l'ensemble du code relatif à un thème particulier, de la même manière que les fonctionnalités de la bibliothèque standard elle-même.

Dans notre exemple fil rouge qu'est le casse-briques, on peut ainsi voir plusieurs thèmes et ainsi créer plusieurs fichiers. Chacun de ces fichiers pourra contenir du code, c'est-à-dire essentiellement des fonctions et des classes et ce code sera isolé au sein du fichier. En effet, il y a une notion de visibilité : les fonctions, variables ou classes ne sont visibles qu'au sein du même fichier. Il n'est pas possible d'accéder à ce qui a été écrit dans un autre fichier **cpp**.

Ceci, à moins, bien entendu, d'utiliser une directive pré-processeur **include** qui va nous permettre de pouvoir accéder aux codes contenus dans les fichiers inclus, exactement comme nous l'avons vu lorsque nous avons inclus un module de la bibliothèque standard :

Fichier

```
#include <nom_bibliotheque>
#include "nom_fichier"
#include "nom_repertoire/nom_fichier"
```

On se sert donc simplement du nom du fichier pour l'inclure, mais ce n'est pas le fichier CPP qui est réellement inclus, c'est son fichier d'en-têtes (qui porte le même nom, mais avec l'extension **hpp** ou **h**). En effet, pour chaque fichier de code **cpp**, il faut un fichier d'en-têtes **hpp**. Ce dernier ne va contenir que les signatures des fonctions, ainsi que la déclaration des constantes et autres directives pré-processeur qui ont plus leur place dans le fichier d'en-têtes que dans le corps du fichier **cpp**. Et ce fichier d'en-têtes ne sert pas uniquement en prévision d'une éventuelle inclusion, mais aussi pour le fichier **cpp** auquel il est lié :

Fichier

```
01: int main()
02: {
03:     f();
04:     return 0;
05: }
06:
07: void f()
08: {
09:     return;
10: }
```

Ce code ne compilera pas ; en effet, lorsque le compilateur lit la fonction **main**, il voit un appel à la fonction **f** (ligne 3) et cette dernière n'est pas encore déclarée (elle l'est à la ligne 7).

Par défaut, tous les fichiers incluent leurs propres en-têtes. Ainsi, la solution pour faire en sorte que le code ci-dessus fonctionne est de disposer d'un fichier d'en-têtes contenant ceci : **void f();**.

Il s'agit des signatures des fonctions. Au moment de la compilation, la fonction **f** ne sera toujours pas déclarée lorsqu'il passera à la ligne 3, mais sa signature le sera et cela suffira pour que la compilation fonctionne proprement.

Il est nécessaire de rappeler que la fonction **main** est le point d'entrée du programme. En cela, elle est une fonction spéciale, car elle ne peut être appelée par aucune autre fonction, y compris par elle-même. Sa signature n'a donc pas sa place dans un fichier d'en-têtes.

2.7 Procédure de compilation

2.7.1 Avec Code::Blocks

Comme on l'a vu au début de ce tout premier chapitre, la compilation est une étape extrêmement importante. En utilisant Code::Block, compiler est aussi simple que de cliquer sur le bouton à l'icône d'engrenage orange. Cependant, avant la première

utilisation, il est nécessaire de configurer cet outil correctement, en particulier par rapport aux avertissements, il faut aller dans le menu ***Settings*** > ***Compiler and debugger...*** puis cocher les cases souhaitées.

Si à la compilation, il ne vous trouve pas un entête qui appartient à la bibliothèque C ou C++, il est inutile de chercher plus loin : si vous êtes sous Windows, vous auriez du choisir l'installateur avec **MingW** et si vous êtes sous Linux, il vous faut installer **g++**. Nous vous renvoyons pour cela plus haut dans l'article.

2.7.2 Compiler en ligne de commandes

Outre la différence entre les possibilités des terminaux linux et windows, l'expérience utilisateur de la compilation en ligne de commandes sera identique. Les mêmes commandes seront utilisées des deux côtés. On peut donc se retrousser les manches et se lancer.

Comme l'on a généralement plusieurs fichiers de code, accompagnés de leurs fichiers d'en-têtes, il faut tout d'abord compiler séparément chaque fichier :

Terminal

```
$ g++ -c fichier1.cpp
$ g++ -c fichier2.cpp
```

Ces commandes vont créer respectivement **`fichier1.o`** et **`fichier2.o`**. Lors de cette étape, la commande **`g++ -c`** va exécuter les directives pré-processeur, puis compiler le fichier de code C++ en un fichier machine d'extension **`.o`** (pour *object file*), qui sera adapté à la machine sur laquelle la compilation a été effectuée.

Enfin, il est important de noter que le compilateur peut être plus ou moins bavard. En cas d'erreur, il va forcément échouer et ne pas compiler. On verra à ce moment-là les différentes erreurs. Mais en cas de succès, il est important de regarder malgré tout s'il n'y a pas d'avertissements, ces derniers pouvant cacher de véritables problèmes. Pour ce faire, on peut utiliser l'option **`-Wall`**, avec **W** pour *warning* et **all** pour tous.

Il existe aussi une autre option, utile pour le développement. Il s'agit de l'option **`-g`** qui permet de rajouter des éléments particuliers sur lesquels va s'appuyer le debugger **`gdb`**. L'option **`-g`** couplée avec l'option **`-O0`**, qui désactive toutes les optimisations du processeur dont le réordonnancement des instructions, nous permettra de faire de l'exécution en mode pas à pas dans notre programme afin de comprendre une erreur, par exemple.

La dernière option que nous allons présenter est la norme de C++ utilisée dans notre programme. En effet, le compilateur GNU supporte les différentes normes C++ existantes, mais n'utilise pas encore la dernière par défaut. Il faut ajouter l'option **`-std=c++14`** à la compilation de nos fichiers source :

Terminal

```
$ g++ -std=c++14 -Wall -g -O0 -c fichier2.cpp
```

Si vous utilisez Code::Blocks, il sera éventuellement nécessaire d'ajouter manuellement certaines options, disponibles dans **`g++`**, mais pas encore dans Code::Blocks, par exemple la définition du standard à la version C++14. Pour ce faire, vous trouverez facilement des tutoriels sur la toile.

Une fois ce travail réalisé, on peut passer à l'assemblage et l'édition de liens, ce qui permet de capitaliser le code des différents fichiers et rendre l'ensemble exécutable :

Terminal
```
$ g++ -o nom_executable fichier1.o fichier2.o
```

On notera aussi qu'il est possible potentiellement de rajouter des liens vers des bibliothèques externes qui sont des modules qui ne font pas partie de la bibliothèque standard, mais qui sont, de la même manière des collections de fonctionnalités qui peuvent êtres utiles. Pour se faire, on utilisera l'option **-shared**.

On peut aussi réaliser cette étape en une seule fois, si on ne désire pas particulièrement garder une trace des fichiers intermédiaires :

Terminal
```
$ g++ -o nom_executable fichier1.cpp fichier2.cpp
```

Cette commande est particulièrement utile lorsque l'on ne travaille qu'avec un seul fichier. Sinon, on utilisera des outils plus complets, comme **make**.

3. NOTIONS ÉLÉMENTAIRES

3.1 Bas niveau

Il n'est pas faux de considérer que le langage C++ est le langage C augmenté de nouvelles fonctionnalités. C'est ainsi qu'il a été pensé lors de sa création et même si les aspects propres au C++ en font un langage à part entière qui par beaucoup d'aspects est très éloigné du C, certains principes fondamentaux restent identiques.

Le premier de ces principes fondamentaux est que C++ est un langage de bas niveau. Un tel langage se distingue par le fait que le développeur doit impérativement avoir quelques connaissances de base sur la manière dont la machine fonctionne et connaître les contraintes à la fois matérielles et liées au système d'exploitation.

Il doit gérer par lui-même les ressources matérielles, en particulier la mémoire, ainsi que les contraintes auxquelles il doit faire face (mémoire, architecture processeur, système de fichiers tel que FAT32 ou ext4).

L'inconvénient principal réside dans la complexité inhérente à cette gestion, principale raison pour laquelle la maîtrise de tels langages de programmation demande plus de temps et d'expérience que pour des langages de haut niveau. Elle réside aussi dans la spécialisation de certaines parties du programme qu'il convient d'adapter pour qu'elles puissent fonctionner sur des machines à l'architecture différente.

D'un autre côté, le principal avantage est que le développeur expérimenté saura tirer parti du langage pour optimiser les temps de calcul et/ou la consommation mémoire, selon son besoin et ses contraintes. Cet état de fait souligne que C++ est idéal à la fois pour la programmation temps réel, (programmes gourmands en mémoire, mais demandant des temps de réponse extrêmement faibles), mais aussi pour la programmation embarquée (programmes fonctionnant avec une quantité de mémoire limitée et d'importantes autres contraintes matérielles).

On pourra toujours argumenter que, pour les programmes de haut niveau, les compilateurs sont capables d'optimisations extrêmement performantes, il est

certains domaines où ces derniers ne permettent pas encore d'égaler l'inestimable expérience de très bons développeurs en langage bas niveau.

À notre niveau, il faut surtout retenir un et un seul fait : le programme C++ va être compilé et transformé en instructions lisibles par un processeur et ce dernier ne fait faire qu'une seule chose : manipuler des octets. Donc peu importe ce que l'on va voir dans tout ce qui suit, du point de vue du processeur, on ne manipule pas des nombres, des chaînes de caractères, des tableaux, des pointeurs ou tout ce que vous imaginerez d'autre, on manipule des octets.

Après, libre à nous de donner la signification que l'on souhaite à ces octets.

3.2 Typage statique

C'est donc tout naturellement que l'on est amené à parler de type. En C++, toute donnée a un type. Il peut s'agir d'un nombre (il y en a plusieurs types), d'une chaîne de caractères, d'un tableau, d'un pointeur, ou de beaucoup d'autres choses.

C'est grâce au type que l'on va savoir ce qu'un ou plusieurs octets vont signifier. En effet, le même octet peut représenter à la fois un nombre ou un caractère, par exemple. En sachant son type, on peut savoir ce que l'on va faire avec et on va pouvoir réaliser des opérations.

Par contre le processeur, lui, va simplement prendre des octets et faire une opération binaire dessus, sans avoir la moindre notion de la signification que nous portons à cette opération. C'est le langage de programmation qui se charge de dire que telle opération vue par un développeur est équivalente à telle opération vue par le processeur. À notre niveau, savoir le détail des opérations réalisées par le processeur n'est pas pertinent.

Mais savoir que le processeur ne peut pas penser à notre place ou anticiper des exceptions qui, de notre point de vue en tant que développeur, nous semblent logiques, est quelque chose qu'il faut avoir en tête pour comprendre pourquoi on rencontre certaines erreurs.

Une fois que l'on voit toute l'importance des types, il convient de s'attarder sur la qualification de **statique**. Cette qualification signifie que toute donnée manipulée par le programme est typée et que le type est fourni dès la première mention de la donnée.

En effet, si vous vous souvenez de la signature de la fonction `main`, les deux paramètres sont typés. Si vous déclarez n'importe quelle variable à n'importe quel moment de votre programme, vous devez impérativement préciser le type de cette variable et durant toute la durée de vie du paramètre ou de la variable, son type ne pourra pas changer. Il est statique.

Le **typage statique** s'oppose au **typage dynamique**. En effet, certains langages permettent aux variables de changer de type au cours de leur existence, ce qui n'est pas du tout une hérésie, cela est une façon différente de programmer et peut avoir des avantages indéniables. Mais comme pour la problématique du niveau de programmation, l'un n'est pas mieux que l'autre. C'est juste que les avantages ou inconvénients sont différents.

En ce qui nous concerne, l'avantage certain est que le fait de connaître le type d'une donnée permet d'anticiper des problématiques d'allocation mémoire et d'optimiser un certain nombre de comportements. D'autre part, cela permet aussi au compilateur de faire des vérifications supplémentaires et de détecter plus d'erreurs potentielles.

Pour autant, il est faux de penser que toutes les erreurs potentielles dues à une mauvaise utilisation d'une donnée d'un type particulier seront détectées à la compilation. À ce niveau-là, le typage statique est un plus, mais pas une garantie absolue.

3.3 Qu'est-ce que C++ ?

À ce stade, on a vu ce qu'était un programme, on a entraperçu la manière dont ce programme s'exécutait, les rôles du disque dur, de la mémoire et du processeur et on commence à entrevoir ce qu'est du code C++.

Mais si on devait définir le langage C++, qu'en est-il ? De quoi est-il fait concrètement ? La réponse n'est pas forcément évidente, il peut y avoir plusieurs manières de présenter la notion. Pour rester simple, on dira que C++, c'est d'abord une norme qui définit un certain nombre de règles. Cette norme s'exprime essentiellement sous la forme d'une grammaire et de cette dernière, on en tire des règles syntaxiques.

Il existe par exemple un certain nombre de **mots-clés** et des **opérateurs**. Chacun peut être utilisé en respectant des règles de construction strictes et est interprété d'une manière déterministe. Par exemple, on a vu comment on devait déclarer une fonction : une signature suivie d'un bloc contenant les instructions. Chaque paramètre porte un nom et est précédé par son type.

La syntaxe permet aussi de décrire comment définir des classes, manipuler des objets, permettre à ces objets d'utiliser des opérateurs, ce que nous verrons dans le jour 4. Cette grammaire est le premier pilier du langage. En maîtrisant ces règles de syntaxe, on se donne les moyens d'écrire des instructions, lesquelles pourront être exécutées par le programme et ont un but spécifique, celui que nous leur donnons.

Mais ce n'est pas tout. Comme nous l'avons vu précédemment, C++, c'est aussi une immense collection de fonctionnalités. Il s'agit de la bibliothèque standard. C'est là l'autre pilier du langage et la maîtriser est essentiel pour utiliser C++ avec dextérité.

Nous allons donc nous attacher à découvrir ces deux aspects progressivement et en parallèle.

Il reste un dernier élément à avoir à l'esprit. Faisant abstraction de l'historique et en regardant les choses comme si on les découvrait ce jour, C++ est d'abord un ensemble de normes et il existe un certain nombre de compilateurs (GNU, Borland, Microsoft Visual...). Ces différents compilateurs implémentent les normes, mais peuvent se comporter différemment. En effet, certains points sont laissés libres d'interprétation et chacun a fait ses choix. De plus, le support des différentes normes est bien entendu différent en fonction des versions du compilateur, à l'exemple du compilateur Visual C++.

Aussi, en fonction de l'architecture de votre machine, de votre système d'exploitation ainsi que de votre compilateur, la compilation peut réagir de manière différente.

Pour terminer, on soulignera que vous devriez être familier avec la signification de la plupart des termes employés dans ce paragraphe. Dans la dernière partie de cet article, nous allons écrire notre premier programme et présenter les ultimes notions élémentaires.

Précisons que chaque compilateur C++ est différent, donnera un programme différent, avec une empreinte mémoire et des performances différentes et qu'il détectera certaines erreurs de manière différente. Par contre, un programme C++ est portable et le résultat concret visible par l'utilisateur sera toujours identique, quel que soit le compilateur.

4. PREMIERS PAS

4.1 Commentaires

Au sein de notre code, il va être possible de définir des zones de commentaires. Un commentaire est tout simplement une note écrite que le développeur va laisser à l'attention de toute personne pouvant relire le code (y compris lui-même). Ce commentaire a pour vocation de préciser des points de détail du code, par exemple pourquoi tel ou tel choix a été fait ou encore pour souligner un reste à faire.

La manière dont vous utilisez les commentaires est laissée à votre appréciation. Vous devez simplement savoir qu'il est possible d'écrire un commentaire à l'aide de `//` (deux slashs ou barres obliques) auquel cas le commentaire commence après ce symbole et se termine à la fin de la ligne de code. Il est aussi possible d'utiliser `/*` et `*/`. Dans ce dernier cas, le premier symbole permet de commencer le commentaire et le dernier de le terminer. Un tel commentaire peut ainsi courir sur plusieurs lignes consécutives.

4.2 Littéraux

Un **littéral** est tout simplement une valeur écrite directement dans une portion de code.

Fichier

```
01: int main(int argc, char* argv[])
02: {
03:     return 0;
04: }
```

À la ligne 3 de cet exemple, `0` est une valeur littérale. Il s'agit d'un nombre entier écrit en dur dans le code. L'utilisation des littéraux est un passage obligé dans toute écriture de code. Ces littéraux ont un type qui leur est automatiquement assigné au moment de la compilation. Ce type est déterminé par la syntaxe.

En effet, sans trop entrer dans le détail, lorsque l'on a que des chiffres, il s'agit de nombres entiers, comme `42`. Si on a des chiffres et un point, il s'agit d'un nombre réel, comme `42.42`. Il existe aussi deux éléments particuliers que sont `true` et `false`, les deux valeurs booléennes. Les derniers éléments que nous verrons sont le simple caractère délimité par une apostrophe de part et d'autre, comme dans `'a'` et la chaîne de caractères, délimitée par, cette fois-ci, des guillemets droits, tels que `"abcde"`.

Attention, à ne pas confondre caractère et chaîne de caractères. Ce sont les délimiteurs qui importent. Pour être parfaitement clair, `"a"` est bien une chaîne de caractères de longueur `1` et `'abcde'` n'existe tout simplement pas.

4.3 Constantes

Par opposition à une variable dont la valeur va pouvoir changer au cours de l'exécution du programme, une **constante** aura une valeur qui restera toujours identique :

Fichier

```
01: int main(int argc, char* argv[])
02: {
03:     [... algorithme ...]
04:         return 42;
05:     [... algorithme ...]
06:         return 0;
07: }
```

Dans cet exemple, on voit que le programme peut quitter en renvoyant les valeurs **42** ou **0**. La signification de cette dernière est parfaitement claire pour toute personne connaissant le fonctionnement d'un programme : on quitte en expliquant au système d'exploitation que tout s'est bien passé.

Par contre, à quoi correspond **42** ? Il est impossible de le savoir sans un commentaire approprié. Mieux encore, on peut utiliser un commentaire et une constante :

Fichier

```
01: // Sortie du programme en erreur : il est impossible de se connecter à
la base de données
02: #define SORTIE_ERREUR__CONNEXION_BDD_KO 42
03:
04: // Sortie du programme sans erreur
05: #define SORTIE_OK 0
06:
07: int main(int argc, char* argv[])
08: {
09:     [... algorithme ...]
10:         return SORTIE_ERREUR__CONNEXION_BDD_KO;
11:     [... algorithme ...]
12:         return SORTIE_OK;
13: }
```

Dans ce cas-là, le code est exactement identique au précédent sur le principe. Pourtant, sans avoir à consulter sa documentation technique, mais simplement en le lisant, on sait pourquoi on quitte le programme. Tout ceci parce que le nom de la constante est explicite.

On peut noter que définir une constante consiste à l'associer à un littéral et que cela est fait par une directive pré-processeur. Celle-ci va parcourir le code et remplacer toutes les occurrences de la constante par sa valeur, avant le processus de compilation. Donc, encore une fois, le code sera exactement le même que le code précédent.

Le commentaire associé a son importance ; il est simplement une précision supplémentaire, mais cette dernière est toujours utile et appréciée. D'ailleurs, tous les littéraux sont des constantes. Leur valeur ne change pas au cours de l'exécution du programme.

Si l'utilisation des constantes est un excellent moyen de permettre d'avoir un code explicite, c'est surtout un moyen de centraliser leurs valeurs. En effet, ces dernières peuvent être utilisées plusieurs fois et pour des raisons de maintenance du code, il faut se laisser la possibilité de changer ces valeurs simplement.

Par exemple, pour notre casse-briques, nous pouvons définir le fait que le joueur commence avec 3 balles.

Fichier

```
#define NOMBRE_BALLES_INITIAL 3
```

Si après avoir testé le jeu on décide que c'est trop ou pas assez, on peut aller changer facilement cette valeur sans avoir à rechercher tous les littéraux **3** dans le code et se poser la question de savoir si ce **3** correspond bien au nombre initial de balles ou à une autre notion. Ceci est un moyen simple, mais néanmoins extrêmement efficace pour avoir un code plus facile à maintenir. Diminuer les coûts de maintenance d'un programme commence par des choses aussi simples que celles-là.

Il faut aussi noter qu'il existe une autre manière de déclarer des constantes, qui n'a aucun rapport avec ce que l'on vient de voir et qui fonctionne via l'utilisation du mot-clé **const**. On l'abordera dans la section sur les variables. Vous verrez alors pourquoi...

4.4 Nombres

4.4.1 Introduction

Il est temps de détailler ce que l'on entend par nombre. En effet, il existe différents types de nombres en C++. Chaque type utilise une représentation mémoire différente, ce qui permet de traiter de petits nombres avec peu de mémoire et ainsi rester plus performant ou au contraire d'aller utiliser plus de mémoire pour utiliser de très grands nombres entiers ou des nombres réels avec une grande précision derrière la virgule.

Comme précisé plus haut, lorsque l'on utilise des littéraux, le compilateur se charge seul de choisir le type à utiliser :

Fichier

```
01: int main(int argc, char* argv[])
02: {
03:     std::cout << 123 << std::endl;
04:     std::cout << 12345 << std::endl;
05:     std::cout << 123456789 << std::endl;
06:     std::cout << 123456789123456789 << std::endl;
07:     std::cout << 12345678912345678912345678912345678 << std::endl;
08:     return 0;
09: }
```

Par contre, il faut savoir que C++ ne définit pas précisément la taille de la représentation mémoire de chaque type, mais seulement des minimas. C'est un des éléments qui diffère selon l'architecture, le système d'exploitation et le compilateur choisi. Ainsi, ce qui suit est susceptible de varier.

4.4.2 Représentation des nombres entiers

Les différentes tailles des **nombres entiers** vont du **char** (1 octet minimum) au **long long** (8 octets minimum) en passant par le **short**, le **int** (2 octets minimum) et le **long** (4 octets minimum).

Au-delà de cette notion de taille, il y a aussi une notion de signe. En effet, sans entrer dans les détails de leur représentation sous forme d'octet, on peut utiliser des **nombres entiers signés** (pouvant être positifs ou négatifs) ou des **nombres entiers non signés** (uniquement positifs).

Suivant les compilateurs, la compilation du code précédent pourra générer une erreur ou bien un simple avertissement (penser à vérifier la compilation du compilateur

pour l'affichage des avertissements). C'est ce dernier cas qui se produit avec mon implémentation et le résultat du dernier affichage est celui-ci : `5020797143238336277`.

Que s'est-il passé ? Dans mon implémentation, le nombre a été représenté sous la forme d'un `signed long long` qui a donc une limite maximale de `9223372036854775808`. Le résultat affiché est tout simplement le reste de la division entière du nombre écrit dans le code par la limite maximale.

En d'autres termes, le bon nombre a bien été représenté en mémoire, mais comme l'espace mémoire lui étant alloué est insuffisant, il y a eu **dépassement** (en gros, on a perdu des retenues).

4.4.3 Représentation des nombres réels

Les nombres réels peuvent être représentés de deux manières : soit par des nombres flottants, soit par des nombres doubles.

Les **nombres flottants** sont représentés en mémoire par une mantisse et un exposant. Pour faire simple, restons en décimal et rappelons le concept de *notation scientifique* : la *représentation scientifique* du nombre `12345.6789` est `1.23456789 x 10^4`. Dans cette notation, le chiffre `1.23456789` est la **mantisse** et la puissance de `10` (le nombre `4`) est l'**exposant**.

De plus, le nombre `0.0000000123456789` serait noté `1.23456789 x 10^-8`. L'exposant négatif permet de stocker des nombres très petits. L'avantage de cette notation est que l'on garde une très bonne précision au fur et à mesure des calculs en gardant toujours un nombre maximal de chiffres significatifs. Pour des calculs mathématiques, cet avantage est indéniable.

Au niveau informatique, c'est exactement les mêmes définitions. Sauf qu'un processeur classique ne sait pas traiter du décimal, mais seulement du binaire. En conséquence, on n'utilise pas des puissances de `10`, mais des puissances de `2`. Mais à part ceci, tout le principe reste identique.

On peut donc assez aisément visualiser le fait que la représentation d'un nombre flottant est très différente de ce à quoi l'on pourrait s'attendre au premier abord. L'avantage de cette représentation est qu'elle est très précise, puisqu'elle permet d'utiliser toute sa taille pour représenter n'importe quel nombre.

Le nombre flottant est idéal pour stocker des nombres réels purs, tels que `pi` ou `e`. Par contre, pour stocker en flottant un nombre tel que `0.1` qui semble pourtant être un nombre tout bête, on va devoir gérer des approximations. En effet, un tel nombre n'a pas une représentation finie lorsqu'il est écrit en base `2`. Ainsi, on peut vouloir stocker un nombre `0.1` puis relire `0.09999999999999`.

Pour cette raison, il existe un autre type qui est le **nombre double**. Toujours en restant en décimal, le principe consiste à stocker la partie entière d'un côté et la partie flottante de l'autre. Dans ce cas, le nombre `12345.6789` serait stocké sous la forme de deux entiers. D'une part `0000012345` et d'autre part `6789000000`.

L'avantage est que l'on n'a plus de soucis d'arrondis. L'inconvénient est que l'on perd en précision. En effet, le nombre `0.0000000123456789` serait stocké sous la forme de deux entiers que sont `0000000000` et `0000000123`. On voit tout de suite que d'une part l'on perd ici énormément d'informations puisque l'on ne conserve que trois chiffres significatifs et que d'autre part on perd plein d'espace mémoire qui n'est utilisé que pour stocker des zéros.

4.5 Variables

Une variable est l'association entre un nom et une valeur. La déclaration d'une variable est obligatoire pour pouvoir l'utiliser et elle impose de déclarer le nom de la variable. Sa valeur, par contre, peut être affectée plus tard. Pour déclarer une variable : `int univers;`.

La déclaration est le processus qui consiste à réserver une zone mémoire pour pouvoir stocker ultérieurement la valeur de la variable. La taille de cette zone mémoire va dépendre du type de la variable. Dans le cas d'un `int`, la taille est généralement de quatre octets comme on vient de le voir.

Une fois que la variable a été déclarée, il peut y avoir affectation : `univers = 42;`.

L'affectation est le processus qui consiste à mettre une valeur dans la zone mémoire réservée pour la variable. La déclaration et l'affectation peuvent être réalisées de concert, et ceci par trois méthodes distinctes, mais exactement identiques :

Fichier

```
int univers = 42;  // Opérateur d'affectation
int univers (42);  // Parenthèses
int univers {42};  // Accolades
```

La première méthode est issue du langage C, la suivante est introduite par le langage C++ dès son origine : elle fait appel à la notion de constructeur. La dernière est introduite par la norme C++11 et recouvre la notion d'initialisation uniforme. Ces différences sont dues à l'évolution du langage et ont des utilisations que nous expliquerons en traitant les objets.

Il faut savoir que dans le cas où l'on a une déclaration sans affectation, la valeur de la variable est indéterminée. Si des données avaient été écrites précédemment en mémoire à cet endroit-là, ces octets pourront être traduits en une valeur, mais celle-ci est totalement aléatoire. C'est pourquoi faire l'affectation en même temps que la déclaration est une bonne pratique. Si vous n'avez pas de valeur à positionner, mettez une valeur par défaut : `int univers = 0;`.

Il reste un dernier élément à introduire, le mot clé `const` : `int const univers = 42;`.

Ce mot clé signifie que l'élément syntaxique qui le précède (ou qui le succède si rien ne le précède) ne peut être modifié par la suite dans le programme. Ainsi, par rapport à notre définition précédente, le type `int` sera `const`, donc l'univers ne pourra jamais être différent de 42. Toute tentative de changer sa valeur sera considérée comme une erreur par le compilateur.

On note que si le mot clé `const` s'applique à ce qui se trouve à sa gauche il est souvent utilisé ainsi : `const int univers = 42;`.

Dans ce cas, comme il n'y a rien à gauche, alors il s'applique à ce qui se trouve à sa droite. La notation puriste est donc `int const`, mais celle que l'on trouvera le plus souvent est `const int` et ceci n'est pas un problème.

Ce mot-clé est très important et nous allons l'utiliser tout au long du projet, sur des types simples, des objets et même des fonctions :

Fichier

```
const int univers = 42;
void maFonction(Objet const &o); // le paramètre Objet ne peut pas être
modifié dans maFonction
void maFonction(const Objet &o); // plus courant et identique au précédent
void maClasse::maMethode() const; // maMethode ne peut pas modifier le
contenu de l'objet manipulé
```

4.6 Caractères et chaînes de caractères

4.6.1 Précisions sur les littéraux

Comme nous l'avons déjà vu, un littéral caractère est délimité par une apostrophe et il est de type **`char`**, tandis que le littéral chaîne de caractère l'est par le guillemet double et il est de type **`const char *`**.

Voici l'affichage d'un caractère, puis d'une chaîne de caractères :

Fichier

```
01: #include <iostream>
02:
03: using namespace std;
04:
05: int main()
06: {
07:     cout << '*' << std::endl;
08:     cout << "Hello world!" << std::endl;
09:     return 0;
10: }
```

Le type caractère est en réalité, comme tous les types un octet. On a créé une table de caractères – identique pour tous les langages – qui associe à un octet particulier un caractère. Or, cet octet peut aussi s'interpréter comme un nombre. Pour déclarer une variable de type char, on pourra donc aussi bien utiliser un caractère délimité par deux simples guillemets droits qu'utiliser un nombre.

Il est à noter que, tel un nombre, on peut réaliser des opérations arithmétiques dessus, mais cela ne présente que peu d'intérêt.

Un premier élément important est que si l'on écrit plusieurs caractères délimités par une simple chaîne, cela va fonctionner, mais pas comme on pourrait l'attendre :

Fichier

```
01: #include <iostream>
02:
03: using namespace std;
04:
05: int main()
06: {
07:     cout << "Hello world!" << std::endl;
08:     cout << 'Hello world!' << std::endl;
09:     return 0;
10: }
```

Le résultat est :

Terminal

```
Hello world!
1919706145
```

Lors de la compilation, le caractère mal écrit a été interprété comme un entier, mais un avertissement a été émis. En effet, comme précisé plus tôt, le programme que vous avez

écrit ne va manipuler que des octets. C'est parce que vous avez précisé un type que ces octets pourront être interprétés comme, par exemple, des nombres ou des caractères. Mais si vous écrivez quelque chose d'incohérent, il est possible que le compilateur trouve ce qu'il considérera comme une solution acceptable, mais qui de votre point de vue n'a tout simplement pas de sens.

Le second élément important est qu'il existe un autre type de littéral pour noter les chaînes de caractères :

Fichier

```
01: #include <iostream>
02:
03: using namespace std;
04: using namespace std::string_literals;
05:
06: int main()
07: {
08:     cout << "Hello world!"s << std::endl;
09:     return 0;
10: }
```

Dans cet exemple, le *littéral* est postfixé par un **s**, ce qui est permis par la norme C14. Ceci permet de définir des littéraux de type **string** et non plus de type **const char ***. Ceci évite des problèmes pénibles de conversion dans pas mal de cas. Il faut cependant penser à introduire la ligne 4 pour les utiliser.

4.6.2 Types de chaînes de caractères

Détaillons un peu cette histoire de type de chaînes de caractères :

Fichier

```
01: #include <iostream>
02: #include <string>
03:
04: using namespace std;
05:
06: int main()
07: {
08:     char s1[] ="Hello world!";
09:     char *s2 ="Hello world!";
10:     string s3 = "Hello world!";
11:     cout << s1 << endl << s2 << endl << s3 << endl;
12:     return 0;
13: }
```

Dans cet exemple, on voit trois types de chaînes de caractères. En premier lieu, à la ligne 7, on a un tableau de caractères. Ensuite, à la ligne 8, on a un pointeur vers un tableau de caractères. Tous les deux sont des héritages directs du langage C.

Ces types sont assez rudimentaires, mais néanmoins très puissants. Il faut savoir qu'il existe un certain nombre de fonctions de la bibliothèque standard (**string.h**) qui sont utilisables sur ces chaînes de caractères, tout comme en C. Pour plus de détails, je vous renvoie vers le hors-série de *GNU/Linux Magazine* n°80 de septembre/octobre 2015 traitant de C.

Enfin, à la ligne 9, on a une chaîne de caractères C++. Et c'est ce type de chaîne de caractères, décrite dans l'espace de nom standard (`std::string`) que l'on va préférer utiliser. On introduira ce nouveau type dans le prochain article.

4.7 Fonctions

Une fonction est une brique élémentaire d'un programme. Une autre manière de présenter ceci consiste à dire que tout programme est un ensemble de fonctions. Ce découpage en fonctions permet d'organiser le code en transformant un algorithme compliqué en plusieurs petits algorithmes simples.

Chaque fonction a un rôle simple : elle fait une tâche simple en exécutant une suite d'actions. Elle peut appeler d'autres fonctions ou s'appeler elle-même (récursivité).

Au point de vue syntaxique, nous avons déjà présenté l'essentiel : chaque fonction est définie par sa signature (type de retour, nom de la fonction, paramètres et leurs types) et par un bloc qui contient les instructions. Le nom de la fonction doit expliciter ce que réalise la fonction.

Une fonction doit être générique dans le sens où elle doit fonctionner, quels que soient les arguments qui lui sont donnés lors de son appel. S'il y a des pré-conditions ou des post-conditions, elles doivent être écrites en tant que parties de la fonction.

4.8 Classes

Une **classe** est la définition d'un modèle de données. Elle permet d'expliciter ce que la donnée représente, comment elle est interprétée, comment elle peut être modifiée.

Une classe définit un certain nombre d'attributs permettant de qualifier la donnée ainsi que des méthodes qui permettent de récupérer des informations ou d'en altérer. Ces dernières sont également appelées dans la terminologie C++ des **fonctions membres**.

Une **instance** d'une classe est une variable qui est représentée par la classe. Chaque instance possède les attributs définis au niveau de la classe, mais leurs valeurs leur sont propres. Chaque instance permet aussi d'utiliser les méthodes définies au niveau de la classe.

CONCLUSION

Ce premier article de fond a permis de poser les bases du langage C++. Certains points ont été assez détaillés, car ils correspondent à des erreurs que font souvent les débutants et il est nécessaire de les comprendre pour les corriger rapidement.

On a pu également définir de très nombreuses notions de manière précise et ceci va nous permettre d'entrer maintenant dans le détail des choses tout en ayant déjà en tête une vue d'ensemble.

Vous avez donc maintenant installé votre environnement de travail, vous êtes prêts à taper du code et vous avez éveillé votre curiosité. Il est temps de nourrir la bête ! ▮

JOUR 2

JOUR 2

DÉBUTEZ VOTRE PROJET

Cet article a pour vocation d'introduire le langage C++, en commençant par présenter deux éléments de base – le nombre et la chaîne de caractères – et en vous donnant les premières notions d'algorithmique tout en développant notre jeu de casse-briques.

JOUR 2

1. MANIPULER DES NOMBRES

1.1 Nombres, bases et chiffres

Comme nous l'avions précisé dans l'article précédent, le processeur ne fait rien d'autre que manipuler des octets. Tout comme n'importe quel type de données, les nombres sont donc des octets.

En informatique, on utilise la base 2. On aura donc 2 chiffres qui sont, par convention **0** et **1**. De la même manière que le chiffre **42** se décompose en **4 x 10^1 + 2 x 10^0** dans la base 10 et donc s'écrit avec un **4** suivi d'un **2**, ce même chiffre se décompose en **1 x 2^5 + 0 x 2^4 + 1 x 2^3 + 0 x 2^2 + 0 x 2^1 + 0 x 2^0** dans la base 2 et s'écrira donc **101000**.

Mais il est aussi possible d'utiliser une représentation octale (base 8), c'est-à-dire en n'utilisant que les chiffres **0** à **7** ou encore une représentation hexadécimale (base 16), c'est-à-dire en utilisant tous les chiffres **0** à **9** puis les lettres **a** (**10**) à **f** (**15**) pour compléter.

Il est donc possible de représenter un nombre en bases 2, 8 ou 16 simplement par préfixant ces chiffres par, respectivement, **0b**, **0** ou **0x** :

Fichier

```
01: #include <iostream>
02:
03: int main(int argc, char* argv[])
04: {
05:     std::cout << 69 << std::endl;
06:     std::cout << 0x45 << std::endl;
07:     std::cout << 0b1000101 << std::endl;
08:     std::cout << 0105 << std::endl;
09:     return 0;
10: }
```

L'exemple précédent écrit ainsi le même nombre 69 de manière décimale, hexadécimale, binaire et octale, comme le montre l'affichage de cet extrait de code.

De la même manière, on peut configurer la fonction **cout** pour lui permettre d'afficher un nombre sous une représentation particulière :

Fichier

```
01: #include <iostream>
02:
03: int main(int argc, char* argv[])
04: {
05:     std::cout << std::hex;
06:     std::cout << "0x" << 69 << std::endl;
07:     std::cout << "0x" << 0x45 << std::endl;
08:
09:     std::cout << std::oct;
10:     std::cout << "0" << 0b1000101 << std::endl;
11:     std::cout << "0" << 0105 << std::endl;
12:
13:     std::cout << std::dec;
14:     std::cout << 0x45 << std::endl;
15:     std::cout << 0b1000101 << std::endl;
16:
17:     return 0;
18: }
```

Attention, les instructions à la ligne 5 et 9 modifient la manière dont les entiers sont représentés et restent valables pour toutes les instructions suivantes. Le seul moyen de revenir à l'affichage décimal est par l'utilisation de l'instruction ligne 13. On notera aussi que l'affichage ne produit que les chiffres et non le préfixe, c'est la raison pour laquelle nous l'ajoutons explicitement dans cet exemple, pour faciliter la lecture du résultat.

Pour terminer à propos de la manipulation de nombres, il faut savoir qu'il est possible grâce au C++14 de les représenter (quelle que soit la base) de façon plus lisible grâce à l'usage d'un séparateur, qui n'a pas d'impact sur le code généré. Ainsi, les deux lignes suivantes produisent le même résultat :

Fichier

```
01: std::cout << 1234567890 << std::endl;
02: std::cout << 1'234'567'890 << std::endl;
```

On voit cependant que le second nombre est plus aisé à lire que le premier.

1.2 Opérateurs mathématiques

Une variable de type entier peut subir tout type d'opérations mathématiques :

Fichier

```
01: void operations()
02: {
03:     int a = 7, b = 1, c = 42;
04:     a = 2 * (1 + b * a) * b + 5 - c / 2;
05:     std::cout << "valeur de a après calcul: " << a << std::endl;
06:     a++;
07:     c--;
08:     b += 3;
09:     b *= 4;
10:     b /= 2;
11:     b -= 1;
12:     std::cout << a << " / " << b << " = " << a / b << std::endl;
13: }
```

À la ligne 5, on utilise les opérations mathématiques les plus élémentaires. Leur notation correspond à une écriture mathématique qui nous est familière. Aucune difficulté ici.

Il est important de préciser que les règles mathématiques de priorité sont respectées. Ainsi, la multiplication ou la division a priorité sur l'addition ou la soustraction et l'utilisation des parenthèses permet de donner explicitement une priorité différente en demandant à ce que les opérations à l'intérieur de celles-ci soient résolues avant de réaliser celles à l'extérieur.

La ligne 6 montre l'opérateur d'incrémentation qui permet de rajouter **1** à la valeur d'une variable. C'est cet opérateur qui a inspiré son nom au langage. Il existe aussi l'opérateur de décrémentation, visible à la ligne 7.

Les lignes 8 à 11 utilisent des opérateurs d'auto-modification. Il s'agit en réalité d'une affectation qui donne à une variable une nouvelle valeur calculée à partir de cette dernière. Ainsi, ces lignes sont équivalentes à :

Fichier

```
b = b + 5
b = b * 4
b = b / 2
b = b - 1
```

Ce qui évite de répéter la variable deux fois.

Au passage, la ligne 6 est aussi équivalente aux deux possibilités suivantes :

Fichier

```
a += 1
a = a + 1
```

Notons enfin que le résultat de ce programme est le suivant :

Terminal

```
valeur de a après calcul: 0
1 / 7 = 0
```

La dernière ligne peut paraître surprenante. Il n'en est rien. En effet, une opération entre deux entiers aura pour résultat un entier. On a donc le vrai résultat, mais tronqué.

Si vous souhaitez avoir un résultat réel, il faut utiliser un nombre réel, c'est-à-dire une variable de type flottant ou de type double. Je vous renvoie à l'article précédent pour savoir comment déterminer le meilleur choix.

Ainsi, en reprenant exactement le même programme, mais en changeant **`int`** par **`float`** à la ligne 3, on obtient ce résultat :

Terminal

```
1 / 7 = 0.142857
```

Pour terminer, sachez qu'il existe aussi l'opérateur modulo **`%`** et son modifieur **`%=`**, mais qu'il ne fonctionne qu'entre deux entiers :

Fichier

```
std::cout << 42 << " % " << 5 << " = " << 42 % 5 << std::endl;
c %= 5;
std::cout << 41 << " % " << 5 << " = " << c << std::endl;
```

1.3 Bibliothèque standard

La bibliothèque standard dispose de nombreuses fonctionnalités mathématiques, la plupart disponibles dans le module **cmath [1]**. On y trouve entre autres les fonctions trigonométriques dont nous allons avoir besoin. Il existe aussi quelques fonctions usuelles comme **`abs`** ou **`div`** dans **`cstdlib`** et **`cinttypes`**.

Ce que l'on ne trouve pas, ce sont les définitions de quelques constantes, telles que le nombre pi. Certains compilateurs la proposent, mais elles ne sont pas dans la norme. Il faut donc calculer cette constante, ce qui est fait dans le code suivant, à la ligne 4 :

Fichier

```
01: #include <iostream>
02: #include <cmath>
03:
04: const double PI = acos(-1);
05:
06: void print_pi()
07: {
08:     std::cout << "PI = " << PI << std::endl;
09: }
```

Le résultat de cet extrait de code n'est cependant pas satisfaisant, puisque l'on ne voit que peu de décimales. Mais rassurez-vous, ce n'est qu'un problème d'affichage (il faut pouvoir choisir la précision lors de l'affichage, ce que l'on introduira lorsque l'on présentera **cout** en détail).

Voici un petit pot-pourri de quelques fonctions utiles :

Fichier

```
01: #include <iostream>
02: #include <cmath>
03:
04: void fonctions()
05: {
06:  double valeur = 42.42;
07:
08:  std::cout << "Valeur: " << valeur << std::endl;
09:  std::cout << "Arrondi supérieur: " << ceil(valeur) << std::endl;
10:  std::cout << "Arrondi inférieur: " << floor(valeur) << std::endl;
11:  std::cout << "Racine: " << sqrt(valeur) << std::endl;
12:  std::cout << "Carré: " << pow(valeur, 2) << std::endl;
13:  std::cout << "Puissance: " << pow(valeur, 2.1) << std::endl;
14:  std::cout << "Exponentielle: " << exp(valeur) << std::endl;
15:  std::cout << "Logarithme népérien: " << log(valeur) << std::endl;
16:  std::cout << "Logarithme base 10: " << log10(valeur) << std::endl;
17:
18:  std::cout << "-----------------------------------------------" <<
std::endl;
19:
20:  std::cout << "Valeur de pi/4: " << PI / 4 << std::endl;
21:
22:  std::cout << "-----------------------------------------------" <<
std::endl;
23:
24:  std::cout << "Cosinus pi/4: " << cos(PI / 4) << std::endl;
25:  std::cout << "Sinux pi/4: " << sin(PI / 4) << std::endl;
26:  std::cout << "Tangeante pi/4: " << tan(PI / 4) << std::endl;
27:
28:  std::cout << "Acosinus pi/4: " << acos(7.07107) << std::endl;
29:  std::cout << "Asinux pi/4: " << asin(7.07107) << std::endl;
30:  std::cout << "Atangeante pi/4: " << atan(1) << std::endl;
31:
32:  std::cout << "-----------------------------------------------" <<
std::endl;
```


```
33:
34:  std::cout << "Opération incorrecte " << asin(42) << std::endl;
35:
36:  std::cout << "-----------------------------------------------" <<
std::endl;
37: }
```

Notez que la fonction **pow** permet d'utiliser indifféremment des entiers ou des réels et surtout que si une opération est incorrecte, comme à la ligne 34, le résultat est **NAN**, ce qui correspond à une constante particulière de la bibliothèque **cmath** de C++ permettant d'indiquer un résultat incorrect.

À titre d'illustration de l'utilisation de ces fonctions, voici un cas pratique pour notre fil rouge : calculer le déplacement de la balle. Tout d'abord, il faut poser quelques conventions. On va dire que la balle se déplace dans un plan qui est le tableau du casse-briques. On va ensuite choisir un repère avec l'origine étant en haut à gauche du tableau, l'axe X étant dirigé vers la gauche et l'axe Y étant dirigé vers le bas. Cette convention est un choix que l'on fait dès à présent et ce choix particulier est motivé par le fait que c'est de cette manière que la plupart des bibliothèques graphiques orientent leurs plans.

On va ensuite considérer que la balle se déplace tout le temps et, pour l'instant, à vitesse constante. La seule chose qui change est sa direction, ce dont on ne se préoccupe pas encore.

Calculer son déplacement revient à calculer ses prochaines coordonnées. Contrairement à ce qui se passe dans la vraie vie où le temps s'écoule de manière ininterrompue, au niveau informatique on va diviser le temps en un nombre fini d'instants. À chacun de ces instants, un déplacement a lieu, mais entre deux instants, rien ne se passe.

Cette approximation est largement suffisante. En effet, à partir du moment où l'on utilise plus de 10 de ces instants par seconde, la persistance rétinienne fait que le mouvement nous semble parfaitement fluide. La plupart des jeux fonctionnent ainsi en 60 *frames* par seconde, c'est-à-dire qu'ils font 60 mouvements par seconde.

Du coup, cela allège les lois de la physique, puisque l'on peut dire que le déplacement c'est ajouter une vitesse à la position précédente (la vitesse étant alors un ordre de grandeur exprimé en fonction du nombre de *frames* par secondes).

Voici un extrait de la solution, que vous trouverez dans le code fourni avec cet article :

Fichier

```
01: double centre_x = 50, centre_y = 0;
02: int rayon = 5;
03:
04: int vitesse = 4;
05: double angle = PI / 4;
06:
07: void deplacer()
08: {
09:     centre_x += vitesse * cos(angle);
10:     centre_y += vitesse * sin(angle);
11: }
```

1.4 Opérateurs binaires

Les opérateurs binaires permettent de travailler sur des octets et sont définis par l'algèbre de Boole. On distingue les opérateurs NON (**~**), ET (**&**), OU (**|**, implicitement

inclusif) et le OU exclusif (^). Il ne faut surtout pas les confondre avec les opérateurs logiques que sont !, && et ||. Ces derniers renvoient comme résultat vrai ou faux et sont utilisés uniquement pour la logique tandis que les opérateurs binaires prennent des octets en entrée et renvoient des octets. On peut ajouter à la liste les opérateurs de décalage (gauche ou droite).

Voici un exemple complet qu'il suffit d'exécuter pour visualiser ce que font ces opérateurs :

Fichier

```
01: #include <iostream>
02: #include <cmath>
03: #include <limits>
04:
05:
06: void operations()
07: {
08:    short x = 42, y = 38;
09:    std::cout << "x      = " << x << ",  représenté 0b00101010" <<
std::endl;
10:    std::cout << "y      = " << y << ",  représenté 0b00100110" <<
std::endl;
11:    std::cout << "~x     = " << ~x << ", représenté 0b11010101" <<
std::endl;
12:    std::cout << "x >> 2 = " << (x >> 2) << ",  représenté 0b00001010"
<< std::endl;
13:    std::cout << "x << 2 = " << (x << 2) << ", représenté 0b10101000" <<
std::endl;
14:    std::cout << "x & y  = " << (x & y) << ",  représenté 0b00100010" <<
std::endl;
15:    std::cout << "x | y  = " << (x | y) << ",  représenté 0b00101110" <<
std::endl;
16:    std::cout << "x ^ y  = " << (x ^ y) << ",  représenté 0b00001100" <<
std::endl;
17: }
```

Pour l'opérateur NON, on voit que tous les 1 ont été transformés en 0, et vice-versa. Pour les deux opérateurs suivants, on voit que le nombre sur un octet (soit 8 bits) a été décalé à droite ou à gauche de 2 bits.

Pour la ligne 14, on voit qu'il n'y a des 1 qu'aux endroits où à la fois x et y avaient des 1. Tout le reste est à 0.

Pour la ligne 15, on voit qu'il y n'a des 1 qu'aux endroits où il y en avait soit dans x, soit dans y. Tout le reste est à 0.

Enfin, pour la ligne 16, on voit qu'il y n'a des 1 qu'aux endroits où il y en avait dans x, mais pas dans y ou dans y, mais pas dans x. Tout le reste est à 0.

2. MANIPULER DES CHAÎNES DE CARACTÈRES

2.1 Présentation de String

Les chaînes de caractères sont un type de données qu'il faut absolument maîtriser. En effet, un logiciel doit régulièrement, d'une manière ou d'une autre,

interagir avec son utilisateur. La principale manière pour un logiciel de restituer de l'information est l'affichage de chaînes de caractères. Et lorsqu'il reçoit l'information depuis l'utilisateur par des événements, entre autres la saisie clavier, cela fait encore intervenir des chaînes de caractères.

Comme on l'a vu dans l'article précédent, nous allons utiliser le type **`string`**. Il s'agit d'un objet et nous allons rapidement présenter quelques fonctionnalités, lesquelles nous permettront d'avoir un avant-goût du paradigme objet.

En effet, en C++, on va préférer utiliser un objet pour représenter une chaîne de caractères, afin d'éviter, entre autres, d'avoir à gérer par nous-mêmes la problématique de l'allocation mémoire. L'objet gère ça en interne et cela est plus simple pour tout le monde.

Pour rappel, les objets de la classe **`string`** sont décrits dans l'espace de nom standard (**`std::string`**), et nécessitent l'entête **`string`** pour pouvoir être utilisés. Il existe certaines fonctions de la bibliothèque standard qui seront utilisables sur une telle chaîne, comme sur les autres chaînes héritées de C, mais on préférera utiliser des méthodes :

Fichier

```
01: #include <iostream>
02: #include <string>
03:
04: using namespace std;
05:
06: int main(int argc, char* argv[])
07: {
08:     string s = "Hello world!";
09:     cout << s << " : (" << s.size() << ")" << endl;
10:
11:     string s1 = "Hello";
12:     string s2 = "world";
13:     s = s1 + " " + s2 + '!';
14:     cout << s << " : (" << s.size() << ")" << endl;
15:
16:     s.clear();
17:     s += s1;
18:     s += ' ';
19:     s.append(s2);
20:     s.append(",;:!?./§", 3, 1);
21:     cout << s << " : (" << s.size() << ")" << endl;
22:
23:     return 0;
24: }
```

Quelques mots pour expliquer ce que l'on voit aussi. Aux lignes 9, 14 et 21, on affiche la chaîne ainsi que sa taille, que l'on obtient par l'utilisation de la méthode **`size`**. À la ligne 13, on fait une concaténation entre deux chaînes de type **`string`**, une chaîne de type **`char*`** et un caractère (de type **`char`**).

Au passage, il existe aussi une méthode **`length`**, mais elle est exactement identique à la méthode **`size`**. Pour des raisons de cohérence, il vous est recommandé d'en choisir une et de s'en tenir tout au long de votre projet.

L'opérateur **`+=`** permet de modifier par concaténation, mais il existe aussi une méthode **`append`** qui permet de concaténer une chaîne (quel que soit son type), mais pas un

caractère. La méthode **append** permet aussi de concaténer avec une sous-chaîne de caractères (ligne 20).

Il y a beaucoup d'autres choses à voir, mais nous n'irons pas beaucoup plus loin dans cet article, nous avons besoin de voir d'autres notions avant.

2.2 Encodage et Unicode

Avant d'entrer plus en détail sur les chaînes de caractères, il faut dire quelques mots sur les caractères eux-mêmes. Lorsque l'on déclare qu'un type de données est un caractère, c'est que sa représentation sera celle d'un caractère. En réalité, chaque caractère est représenté sous la forme d'un ou plusieurs octets. Or, les nombres le sont aussi. Tout octet peut donc potentiellement être interprété comme un nombre ou comme un caractère.

L'interprétation de l'octet d'un caractère en tant qu'entier est appelée son **ordinal**. Aussi, on peut créer un tableau allant de **1** à **255** et pour chaque case assigner un caractère. C'est ce que l'on nomme table de caractères. Mais il n'y a que 256 octets différents.

C'est plus qu'il n'en faut pour représenter les encodages des langues latines, par exemple. Ces encodages reposent sur la table ASCII qui détermine les 127 premiers caractères. Cela inclut toutes les lettres majuscules et minuscules, les chiffres, les éléments de ponctuation les plus importants et quelques caractères spéciaux comme le saut de ligne, le guillemet simple et double, etc.).

Les autres caractères sont utilisés pour des éléments propres au langage encodé, comme le C cédille pour le français, le N tilde pour l'espagnol ou l'eszett pour l'allemand.

Ces encodages sont toujours très utilisés et sont pratiques puisque chaque caractère se code sur un seul octet. Mais ils ne permettent pas facilement de manipuler d'autres langues plus complexes, comme celles fondées sur les idéogrammes et ne permettent pas d'écrire un texte avec à la fois des mots français, allemands et espagnols, par exemple.

À cet effet, on travaille depuis quelques années avec l'Unicode. L'Unicode est un encodage des caractères sur un, deux, trois ou quatre octets et il représente absolument tous les caractères existants, qu'il s'agisse bien entendu des lettres latines, spécifiques à certaines langues ou non, mais aussi de toutes les autres, telles que le cyrillique, l'hébreu, le copte, mais aussi les idéogrammes japonnais ou chinois, le hindi et même les hiéroglyphes.

Ainsi, ce code-ci est parfaitement compris :

Fichier

```
01: #include <iostream>
02:
03: using namespace std;
04:
05: int main(int argc, char* argv[])
06: {
07:     cout << "Grec : Ελληνικά" << endl;
08:     cout << "Hébreux : עִבְרִית" << endl;
09:     return 0;
10: }
```

L'Unicode n'est pas une norme spécifique au C++, mais est partagé par tous les langages de programmation (<troll>enfin, ceux qui ont réussi à l'implémenter</troll>).

Il faut savoir que pour utiliser l'Unicode, il faudra utiliser des variables de type **`wstring`** au lieu de **`string`**. De plus, pour que les caractères relatifs à une langue s'affichent, il faudra que ce jeu soit installé sur la machine. Le détail de l'affichage des caractères est quelque chose qui se joue au niveau du système d'exploitation.

2.3 Caractères spéciaux

Nous utilisons la plupart des caractères lorsque nous écrivons, quelle que soit leur langue. Ces derniers se trouvent donc logiquement encodés et sont disponibles. Y compris les caractères comme la ponctuation. Il ne s'agit là que de caractères usuels.

La notion de caractère spécial recouvre les caractères qui ont un sens au niveau informatique et qui ne sont pas des caractères à proprement parler. C'est le cas par exemple du saut de ligne. En effet, avec un papier et un stylo, la plupart des gens passent assez simplement à la ligne suivante. Avec une chaîne de caractères, on indique que l'on passe à la ligne suivante par l'utilisation d'un caractère spécial.

Ces caractères spéciaux sont notés à l'aide d'un caractère d'échappement **`\`** suivi d'un caractère usuel. En réalité, il ne s'agit là que d'une représentation destinée à nous, développeurs, parce que ce qui est réellement stocké, c'est l'**ordinal** de ce caractère, comme c'est toujours le cas.

Parmi eux, il est nécessaire de connaître au minimum : la fin de ligne **`\n`**, le retour en début de ligne **`\r`**, et la tabulation **`\t`**.

Si l'on n'échappe pas un guillemet double, il est confondu avec le délimiteur de fin de caractère par le compilateur, ce qui donne des situations fâcheuses. De même, si l'on n'échappe pas un antislash, il est confondu avec le caractère d'échappement. Pour le reste, il s'agit de conventions partagées, quel que soit le langage de programmation.

Voyons un exemple, sous un environnement Linux, avec un code volontairement obscur :

Fichier

```
01: #include <iostream>
02:
03: using namespace std;
04:
05: int main(int argc, char* argv[])
06: {
07:     cout << "abcde\nabcde\rABC\na\ta\t\ta\n\"1\\2\"" << endl;
08:     return 0;
09: }
```

Le résultat est celui-ci :

Terminal

```
abcde
ABCde
a       a               a
"1\2"
```

On voit simplement que chaque `\n` correspond à un changement de ligne et que l'on a pu afficher des tabulations, des guillemets droits et un antislash en les échappant.

Ce qui peut être déroutant est l'utilisation du `\r`. En effet, dans la seconde ligne, on écrit `abcde`, puis le `\r`, ce qui fait revenir le curseur en début de ligne. Le reste de la chaîne `ABC` va donc s'écrire en remplacement de ce qui avait déjà été écrit au fur et à mesure du déplacement du curseur. Le résultat est donc `ABCde`.

2.4 Conversions

Il est possible de convertir des nombres en chaînes de caractères et inversement. D'abord, la partie simple : pour convertir n'importe quel type de nombre en une chaîne de caractères, il faut utiliser la fonction `to_string` [4] qui fait partie de la bibliothèque `string` et de l'espace de nommage standard.

Pour convertir une chaîne de caractères vers un nombre, il faut d'abord savoir quel est le type du nombre que l'on souhaite obtenir. On utilisera alors la fonction `stoX` où `X` est l'abréviation du type souhaité (obtenu en utilisant la première lettre des mots composant le type, par exemple `i` pour `int` ou `ull` pour `unsigned long long`) [5] :

Fichier

```
01: #include <iostream>
02: #include <string>
03:
04: using namespace std;
05:
06: int main(int argc, char* argv[])
07: {
08:
09:     string = to_string(42);
10:     cout << s << " : (" << s.size() << ")" << endl;
11:
12:     int i = stoi(s);
13:     long int l = stol(s);
14:     unsigned int ul =stoul(s);
15:     long long ll = stoll(s);
16:     unsigned long long ull = stoull(s);17:     float f = stof(s);
18:     double d = stod(s);
19:     long double ld= stold(s);
20:
21:     return 0;
22: }
```

Ces dernières fonctions sont très pratiques pour récupérer une saisie de l'utilisateur et la convertir en un nombre, ce que nous ne manquerons pas de faire, en temps et en heure.

2.5 Opérations de recherche

Pour rechercher un caractère particulier dans une chaîne de caractères, on utilise la méthode `find`. Cette méthode va renvoyer la position du premier caractère

trouvé, sachant que la première position est **0**. On appelle cette position *index*. Si le caractère n'est pas trouvé, le résultat de la fonction **find** est **-1**.

La fonction **find** permet également de trouver la position du caractère suivant. En effet, si on a trouvé un résultat à l'index **i**, on peut relancer la recherche à partir de l'index **i + 1**.

Il faut également savoir que l'on peut chercher également une sous-chaîne de caractères. Pour cela, il faudra passer une chaîne de caractères en argument et non plus un seul caractère, tout simplement.

Enfin, sachez que l'on peut chercher plusieurs caractères à la fois, à l'aide de la fonction **find_first_of**. S'il n'y a qu'un caractère ou une chaîne de caractères de longueur **1** en argument, cette fonction se comportera exactement comme la fonction **find**. Sinon, elle permet de s'arrêter dès qu'un des caractères présents en argument est trouvé.
En résumé :

Fichier

```
01: void recherche()
02: {
03:   string s = "** abcba **";
04:   cout << "b est dans \"** abcba **\" ? " << s.find('b') << endl; // 4
05:   cout << "b est dans \"** abcba **\" ? " << s.find('b', 5) << endl;
// 6
06:   cout << "b est dans \"** abcba **\" ? " << s.find("ba") << endl;
// 6
07:   cout << "a ou b sont dans \"** abcba **\" ? " << s.find_first_
of('b') << endl; // 4
08:   cout << "b est dans \"** abcba **\" ? " << s.find("za") << endl;
// -1
09:   cout << "a ou b sont dans \"** abcba **\" ? " << s.find_first_
of("za") << endl; // 3
10: }
```

Le commentaire de fin de ligne donne l'affichage attendu. Au passage, en ligne 8, vous pourriez voir un grand nombre positif au lieu de **-1**. Pas de panique, c'est le même nombre (dépassement de tampon, une histoire de représentation). Pour vous en convaincre, vous pouvez tester l'égalité avec **-1**.

3. ALGORITHMIQUE BASIQUE

3.1 Opérateurs logiques et de comparaison

Les **opérateurs logiques** permettent de tester deux variables et de renvoyer **1** pour vrai ou **0** pour faux. On distingue :

⇨ l'opérateur ET (**&&**) : donne vrai si les *opérandes* sont tous deux vrais ;

⇨ l'opérateur OU (**||**) : donne vrai si au moins l'un des deux *opérandes* est vrai ;

⇨ l'opérateur NON (**!**) : renverse la valeur de l'*opérande*.

Les autres opérateurs usuels (OU exclusif, de NAND ou de NOR) sont absents (et le sont rarement dans la plupart des langages informatiques, étant surtout utilisés dans l'électronique), mais ils peuvent s'écrire en combinant les trois qui sont présents.

Les **opérateurs de comparaison** (`==`, `!=` , `>`, `<`, `>=`, `<=`) permettent d'exprimer des conditions en renvoyant `1` ou `0` en fonction du fait que l'*expression* écrite est vérifiée ou non.

Ainsi, une expression telle que `42 > 0` renverra `1` (vrai) tandis qu'une même expression, réalisée sur des objets de type `string`, telle que `"petit" <= "grand"` renverra `0`. Ces opérateurs ne fonctionnent donc pas que sur des nombres. Cependant, il faut qu'ils aient un sens. Autant pour les nombres, ce sens est évident, autant pour les autres types, ce n'est pas forcément le cas.

Les opérateurs de comparaison n'existent pas toujours, pour tous les types et pour tous les objets. Tout dépend en fait de l'implémentation écrite sur chaque type ou objet. Cependant, s'ils existent, et si les règles tacites sont respectées, on dispose soit uniquement des opérateurs d'égalité et de différence, soit de tous les opérateurs de comparaison.

Dans notre projet, les occasions d'écrire des *expressions* ne vont pas manquer. C'est d'ailleurs le cas de tous les projets. Le premier exemple que l'on peut prendre nous permet de situer la balle dans l'espace de jeu.

En effet, dès lors que, après un mouvement, la balle sort de l'espace de jeu par le haut, la droite ou la gauche, cela signifie qu'elle aurait dû rebondir sur le mur. On va donc changer l'angle de la balle en assumant qu'elle rebondit symétriquement par rapport à la norme de la surface et on va recalculer sa nouvelle position. On appelle cela la détection de collision.

Si la balle sort de l'espace de jeu par le bas, c'est que l'on vient de perdre une vie. Et si l'on n'a plus de vie, on vient de perdre le jeu.

Ce ne sont que quelques-unes des situations que l'on peut trouver, mais ce sont celles qui seront abordées dans le code livré avec cet article. Dans le futur, on devra en gérer beaucoup d'autres. Par exemple, on doit également faire de la détection de collision entre la balle et la raquette ainsi qu'entre la balle et chacune des briques.

3.2 Notion de bloc

Pour bien comprendre ce qui va suivre, il est essentiel de bien saisir la notion de **bloc de code**. Un bloc de code est tout simplement des lignes de code qui font partie d'un ensemble. Pour mieux comprendre, voici un extrait de méta-code qui utilise plusieurs blocs que nous avons déjà vu ou que nous allons voir :

Fichier

```
01: ESPACE DE NOMMAGE
02: {
03:     // Bloc de code de l'espace de nommage
04:     FONCTION ()
05:     {
06:         // Bloc de code de la fonction.
07:         TANT QUE EXPRESSION 1 VRAIE
08:         {
```

```
09:             // Bloc de code exécuté autant de fois que nécessaire
10:             // Tant que l'expression 1 est vraie
11:             SI EXPRESSION 2 VRAIE
12:             {
13:                 // Bloc exécuté si l'expression 2 est vraie
14:             } SINON SI EXPRESSION 3 VRAIE
15:             {
16:                 // Bloc exécuté si l'expression 2 est fausse et 3
vraie
17:             } SINON
18:             {
19:                 // Bloc exécuté si les expressions 2 et 3 sont fausses
20:             }
21:         }
22:         SI EXPRESSION 4 VRAIE 23:         {
24:             // Bloc exécuté si l'expression 4 est vraie
25:         }
26:         // FIN du bloc de la fonction
27:     }
28:     // FIN du bloc de l'espace de nommage
```

Ainsi, les blocs peuvent être adjacents ou hiérarchiques. Ils commencent et terminent par une accolade. Le bloc de l'espace de nommage ne contient qu'une seule fonction dans notre exemple, mais il pourrait en contenir plusieurs, plus des définitions de variables…

En découpant notre code en blocs, on va se donner la possibilité de faire exécuter certains blocs seulement si une expression est vraie ou de le faire exécuter plusieurs fois tant qu'une expression est vraie, par exemple. On écrit donc un code dont l'exécution n'est pas linéaire.

Attention ! Dans le cas où le bloc de code n'est composé que d'une seule instruction, il est possible de ne pas mettre d'accolades pour isoler la ligne de code. Bien que la syntaxe soit autorisée par le langage, elle est interdite dans de nombreuses normes de codage pour des raisons principalement de maintenabilité : elle facilite l'introduction d'erreurs lors de modifications successives de cette partie de code par différents auteurs [2].

3.3 Blocs conditionnels

3.3.1 L'instruction si

Un **bloc conditionnel** est initié par un test si. La syntaxe impose d'écrire le mot `if`, suivi d'une expression écrite entre parenthèses et suivie d'un bloc contenant la suite des instructions à n'exécuter que si l'expression est vraie.

Dans le cas contraire, le bloc de code lié à l'instruction si est simplement ignoré et la suite du code réalisée.

Voici un exemple extrait du projet final, permettant de tester si la position de la balle reste entre certaines bornes :

Fichier

```
01: if (max_pos_x > m_x_max || max_pos_y > m_y_max) {
02:   // exécuté uniquement si l'expression est vraie
03:     return false;
04: }
05: // code exécuté quel que soit le résultat du test
```

3.3.2 L'instruction sinon

L'utilisation de l'instruction si suffirait potentiellement à gérer toutes les situations, mais pour plus de simplicité, il est aussi possible de faire appel à une instruction sinon pour pouvoir gérer de concert les deux cas ; l'expression est vraie et l'expression est fausse :

Fichier

```
01: if (m_sprites[y][x]) {
02:     // bloc de code exécuté uniquement si l'expression est vraie
03:     out << "#";
04: } else {
05:     // bloc de code exécuté uniquement si l'expression est fausse
06:     out << ".";
07: }
08: // code exécuté quel que soit le résultat du test
```

Ce code est extrait de l'algorithme permettant de visualiser la position des éléments dans le jeu. Il serait exactement équivalent à celui-ci :

Fichier

```
01: if ( 42 > 0 ) {
02:     // bloc de code exécuté uniquement si l'expression est vraie
03: }
04: if (!( 42 > 0 )) {
05:     // bloc de code exécuté uniquement si l'expression est fausse
06: }
07: // code exécuté quel que soit le résultat du test
```

On voit que si elle n'est pas strictement indispensable, cette instruction apporte une certaine élégance et surtout facilite la maintenabilité du code et sa lisibilité.

3.3.3 L'instruction sinon si

Mais dans la vie, tout n'est pas blanc ou noir. Il y a des niveaux de gris, voire en fait trois canaux de couleurs... Tester seulement une seule expression n'est pas toujours suffisant. Aussi, il est possible de tester plusieurs expressions et de les chaîner de la manière suivante :

Fichier

```
01: if ( condition1 ) {
02:     // bloc de code exécuté si la première expression est vraie
03: } else if (condition2) {
04:     // bloc de code exécuté si l' expression 1 est fausse et la
seconde vraie
```


```
05: } else {
06:     // bloc de code exécuté si les expressions 1 et 2 sont  fausses
07: }
08: // code exécuté quel que soit le résultat du test
```

Il n'est pas besoin d'aller chercher très loin pour imaginer à quel point l'instruction *sinon si* permet de simplifier son code dans le cas où il y ait plusieurs cas à mettre en évidence, surtout lorsque l'on sait qu'il peut y avoir autant d'instructions *sinon si* que souhaité, à condition qu'elles suivent une instruction *si*.

Bien entendu, l'instruction *si* est toujours au début et l'instruction *sinon* est toujours à la fin et les instructions *sinon si* sont toujours entre les deux. De plus, l'instruction *sinon* n'est pas obligatoire.

3.3.4 Les branchements

Bien qu'il soit possible de réaliser tous les blocs conditionnels seulement par l'utilisation des instructions que l'on vient de présenter, il y a un cas particulier qui peut s'avérer peu élégant. En effet, il est souvent utile de tester plusieurs valeurs d'une variable particulière.

Pour ceci, nous pouvons utiliser l'instruction **`switch`** sur la variable et les instructions **`case`** pour chaque valeur à tester. Sous chaque instruction **`case`**, on pourra mettre un bloc, mais attention, car toutes les instructions **`case`** sont adjacentes, mais non exclusives. Il faut donc, pour les rendre exclusives les terminer par l'utilisation de l'instruction **`break`** qui sert à terminer le bloc de code **`switch`**.

Fichier

```
01: switch (key) {
02:   case KEY_RIGHT:
03:   {
04:     //bloc de code exécuté uniquement si la variable key vaut 'C'
05:     deplacer_raquette_a_droite();
06:     break;
07:   }
08:   case KEY_LEFT:
09:   {
10:     //bloc de code exécuté uniquement si la variable vaut valeur2
11:     deplacer_raquette_a_gauche();
12:     break;
13:   }
14:   default:
15:   {
16:     //bloc exécuté si la variable ne vaut ni KEY_RIGHT ni KEY_LEFT
17:   }
18: }
```

Cet exemple est adapté de l'algorithme permettant d'exécuter une action à partir de l'écoute du clavier. Il faut noter que l'usage du **`break`** n'est pas obligatoire, il permet simplement d'arrêter le flux de traitement afin d'éviter que les blocs de code des **`case`** suivants soient exécutés. On peut, par exemple souhaiter que pour deux valeurs distinctes le même bloc de code s'exécute :

Fichier

```
01: switch (key) {
02:   case KEY_ENTER:
03:   case PADENTER:
04:   {
05:     //bloc de code exécuté si la variable vaut valeur1 ou valeur2
06:     démarrer_le_jeu();
07:     break;
08:   }
09:   default:
10:   {
11:     //bloc exécuté si la variable ne vaut ni KEY_ENTER ni PADENTER
12:   }
13: }
```

3.4 Blocs itératifs

3.4.1 Les différents types d'itérations

Itérer permet de répéter un même bloc de code. Ce dernier peut l'être 0, 1 ou n fois. Il existe différents cas d'utilisation.

Parfois, on veut exécuter un code autant de fois que nécessaire, jusqu'à ce que l'on arrive à un objectif particulier. Dans ce cas-là, on va utiliser l'instruction **`while`**. Ce qui caractérise cette forme d'itération, c'est qu'il est impossible de prédire a priori combien d'itérations seront nécessaires. D'ailleurs, rien n'empêche que l'expression soit fausse dès le premier test et le bloc ne soit jamais exécuté.

C'est pour cela qu'il existe aussi une forme particulière de cette instruction, **`do while`**, qui permet de s'assurer que le bloc soit toujours exécuté au moins une fois. La différence réside dans le fait que l'expression est testée après que le bloc soit exécuté au lieu d'être testée avant.

D'un autre côté, il y a l'instruction **`for`** qui est utilisée soit pour exécuter un nombre constant de boucles, soit pour itérer sur un nombre de boucles prédictible : par exemple, itérer sur toutes les lignes d'un tableau.

Enfin, on peut signaler aussi le **`for_each`** qui est spécifiquement désigné pour les **itérateurs**. Ces derniers sont des objets spécialement conçus pour itérer sur des objets conteneurs, tels que les chaînes de caractères, par exemple. On peut aussi écrire ses propres itérateurs. Tout ceci sera présenté dans le hors-série.

3.4.2 L'instruction tant que

Pour développer notre jeu, nous avons déjà vu que nous allions déplacer la balle et tester si elle rebondit sur les murs ou si elle sort en bas de l'écran, signifiant la fin de la partie en cours. On va donc commencer la mise en place de la structure de notre programme en créant une variable booléenne **`game_over`** qui serait initialisée à **`false`**, valeur qu'elle conserverait pendant toute la partie et serait mise à **`true`** dès l'instant où l'on détecterait que la balle est sortie de l'écran.

Ensuite, on peut modéliser l'action de jouer ainsi : à chaque instant, on va regarder les événements (actions du joueur sur le clavier ou la souris) à l'aide d'une fonction

handle_events, puis on va en tirer les conséquences à l'aide d'une fonction **update** et finalement redessiner l'écran avec la fonction **draw** pour montrer à l'utilisateur la nouvelle position des divers éléments.

La fonction clé est la fonction **update**, puisqu'elle va gérer le déplacement de la raquette en fonction des événements capturés, celui de la balle en fonction de son angle, de sa vitesse et d'une éventuelle collision et la détection des conditions de fin de jeu (elle mettra la variable **game_over** à **true** si nécessaire). Sachant cela, « jouer » s'écrit tout simplement comme suit :

Fichier

```
01: void play_one_game()
02:     while (!game_over) {
03:         handle_events();
04:         update();
05:         draw();
06:     }
07: }
```

L'instruction **while** est ici le bon choix, parce qu'il est impossible de déterminer a priori combien de temps le jeu va durer (tout dépend de l'adresse du joueur). Ceci dit, rien n'interdit d'utiliser une variable de boucle dans un bloc de l'instruction **while**. Elles ne sont pas forcément réservées aux instructions **for**. Pour montrer cela, on présente la fonction qui va permettre de calculer l'évolution du score :

Fichier

```
01: int set_score(int score, short niveau, short max)
02: {
03:     short bonus = 1, i = 0;
04:     do
05:     {
06:         score += bonus;
07:         bonus *= 2;
08:         i ++;
09:     } while(bonus <= max || i <= niveau)
10:     return score;
11: }
```

Dans cet exemple, on veut forcément que le score augmente au moins de **1**, mais que suivant certains paramètres, il puisse évoluer plus (et de plus en plus vite). L'expression d'arrêt vérifie deux choses : d'une part, le bonus ne doit pas être trop grand et d'autre part, le nombre d'itérations est limité par le niveau du tableau joué. Autrement dit, il n'y a pas de moyen d'anticiper le nombre de boucles, on est bien sur une instruction de type **while**. Pour que le score évolue toujours au moins de **1**, il faut que la boucle soit exécutée au moins une fois. On va donc utiliser **do while** pour s'en assurer. À partir de là, il suffit de gérer la modification de la ou des variables de boucle et de trouver l'expression juste pour arrêter la boucle au bon moment.

Pour information, le score évoluera plus vite avec le niveau (qui est le numéro du tableau en train d'être joué), mais il évoluera aussi plus vite avec la rapidité entre deux briques cassées. Ainsi, à chaque brique cassée, la valeur **max** sera mise au plus haut et avec le temps qui passe, elle diminuera graduellement. Ce qui fait que si le joueur met trop de temps entre deux briques cassées, cela lui rapportera moins de points. Cette valeur **max** sera calculée par la fonction **update**, le cœur du jeu.

3.4.3 L'instruction pour chaque

L'instruction **for** utilise généralement une variable de boucle, puisqu'elle permet de réaliser une itération sur un nombre déterminé de boucles. Elle a une syntaxe atypique, puisqu'elle se compose de trois parties, séparées par des points-virgules. La première partie contient une instruction d'initialisation. Elle est exécutée une fois seulement, avant que le bloc ne soit exécuté. La deuxième partie contient l'expression de fin de boucle, elle est évaluée au début de chaque itération. Si elle est vraie, le bloc de code sera exécuté. Il est possible qu'elle soit fausse dès le premier test, auquel cas le bloc de la boucle **for** ne sera jamais exécuté. Enfin, la troisième partie est une instruction de fin d'itération. Elle est exécutée à la fin de chaque itération et immédiatement suivie par une nouvelle évaluation de l'expression de fin de boucle qui va déterminer s'il y aura une boucle supplémentaire. Voyons un exemple concret :

Fichier

```
01: void play()
02: {
03:     for (int i = 0; i < NOMBRE_BALLES_INITIAL; i++)
04:     {
05:         play_one_game();
06:     }
07: }
```

Jouer consiste à lancer autant de jeux que ce que l'on a de balles. Pour rappel, **NOMBRE_BALLES_INITIAL** est la constante que nous avions définie dans l'article précédent, indiquant le nombre de balles initial. L'écriture telle qu'on la voit ci-dessus est la manière usuelle d'écrire une boucle dont on peut prévoir le nombre d'itérations (dans ce cas, autant de boucles que la valeur de la constante).

On peut aussi jouer sur la variable de boucle en modifiant la ligne 3 ainsi :

Fichier

```
for (int i = NOMBRE_BALLES_INITIAL; i > 0; i--)
```

Le résultat sera identique, mais cette fois-ci, **i** ne représente plus seulement une quelconque variable de boucle, mais le nombre de balles restant à jouer. Il aurait d'ailleurs fallu la renommer pour qu'elle soit plus explicite, ce que l'on ne fait pas ici volontairement pour que les différents extraits de codes restent plus facilement comparables.

Et comme l'on est très joueur, on peut imaginer qu'il y ait des bonus dans le jeu et que l'on puisse gagner des balles (autrement dit, que le nombre de balles puisse être supérieur au nombre de balles initial). Si l'on s'amuse à faire cela, on n'a plus du tout un nombre d'itérations prédictible. Il faut donc préférablement utiliser à nouveau un **while** et non plus un **for**.

Fichier

```
01: void play()
02: {
03:     int i = NOMBRE_BALLES_INITIAL;
04:     while (i > 0)
05:     {
06:         play_one_game();
```


```
07:         // Incrémenter i ici si l'on a gagné des balles.
08:         i++;
09:     }
10: }
```

Il faut ensuite trouver un moyen simple pour mettre à jour le nombre de balles (représenté par la variable **i**). Toujours est-il que l'on touche ici du doigt la différence entre les deux instructions **while** et **for** et que l'on voit que, sur le plan technique, elles sont interchangeables. Cependant, il faut toujours faire l'effort d'utiliser la bonne instruction pour chaque cas d'utilisation.

3.4.4 Casser une boucle

Nous avons déjà vu le mot clé **break**, utilisé dans une boucle **switch** – **case** ; il nous permettait d'interrompre la boucle. Ce mot est également utilisable dans les boucles (quel que soit leur type) et permet de les interrompre également.

Reprenons l'exemple de la fonction **set_score**. Nous utilisions une condition complexe, à savoir **bonus <= max or i > level**. Ce faisant, on posait une limite par rapport au bonus qui ne devait pas dépasser un certain maximum et une autre par rapport au nombre d'itérations qui ne devaient pas dépasser la valeur du niveau. Il est possible de séparer cette expression composée en deux distinctes en utilisant l'une pour l'itération et l'autre comme garde-fou :

Fichier

```
01: int set_score(int score, short niveau, short max)
02: {
03:     short bonus = 1, i = 0;
04:     do
05:     {
06:         score += bonus;
07:         bonus *= 2;
08:         i ++;
09:         if (i <= niveau)
10:         {
11:             break;
12:         }
13:     } while(bonus <= max)
14:     return score;
15: }
```

Sur le plan sémantique, la boucle est maintenant centrée sur la variable **bonus** : c'est cette variable qui est mise à jour, elle sert à faire évoluer le score et c'est elle qui est utilisée dans l'expression de fin de boucle. On introduit ensuite une variable accessoire qui sert de compteur de boucle et qui permet de s'assurer que l'on ne peut pas augmenter le score au-delà d'une certaine limite, imposée par le niveau. C'est une façon de faire centrée sur le besoin fonctionnel, mais on peut aussi renverser le raisonnement et utiliser cette variable compteur de boucle en tant qu'élément central et la limite sur le bonus en tant que garde-fou. Cela va nous permettre de réaliser une itération simple à l'aide d'une boucle **for**, travaillant sur le niveau et de casser cette boucle en travaillant sur le bonus.

Fichier

```
01: int set_score(int score, short niveau, short max)
02: {
03:     short bonus = 1
04:     for (i=0; i<=niveau; i++)
05:     {
06:         score += bonus;
07:         bonus *= 2;
08:         if (bonus > max)
09:         {
10:             break;
11:         }
12:     }
13:     return score;
14: }
```

L'inconvénient de ce code est qu'il utilise un élément secondaire comme variable de boucle, mais ce n'est pas si important que cela. Le fait est que l'on quittera la boucle plus souvent par le **break** que par une évaluation négative de l'expression de fin de boucle. Ce n'est pas un problème.

3.4.5 Sauter une boucle

Dans certains cas d'utilisation, on réalise des boucles, mais on souhaite que selon certaines conditions, le bloc ne soit pas exécuté ou soit interrompu pour cette boucle, sans que l'itération s'arrête pour autant. Le mot clé **continue** va interrompre la boucle, tout comme le mot clé **break**. La différence est que si ce dernier continuait à la ligne suivant la fin du bloc **for**, le mot-clé **continue** va retourner au début de la boucle, incrémenter la variable de boucle puis évaluer à nouveau l'expression de fin de boucle pour éventuellement recommencer celle-ci.

CONCLUSION

Cet article a permis d'aborder tous les principes les plus élémentaires de C++, introduisant les nombres et les chaînes de caractères. Il est absolument impératif de vraiment bien maîtriser tout ceci avant d'aller plus loin et d'aborder des choses plus complexes, telles que celles que nous allons aborder dès l'article suivant : les tableaux, les algorithmes, puis les classes, etc. Si vous jetez un œil au code tel qu'il est à ce stade de hors-série, vous verrez surtout un squelette vide et quelques exemples déjà implémentés. Il vous est recommandé de prendre en main ce code et de l'altérer pour rajouter vos propres tests, et ainsi visualiser le résultat. ■

RÉFÉRENCES

[1] Documentation de *cmath* :
http://www.cplusplus.com/reference/cmath/

[2] Blocs de codes et accolades :
http://www.possibility.com/Cpp/CppCodingStandard.html#brace

JOUR 3

JOUR 3

ABORDEZ LES TYPES AVANCÉS ET LES POINTEURS

Cet article va revenir sur certaines notions déjà abordées et vous donner des clés supplémentaires pour les exploiter pleinement.

Il donnera également des explications détaillées sur les pointeurs et les références, notions qu'il faut impérativement maîtriser dès maintenant, puisqu'on les utilisera de manière abondante dès le prochain article.

Nous introduirons ensuite les conteneurs et en présenterons une sélection de manière détaillée.

1. ESPACES DE NOMMAGE

On a vu dans l'article précédent ce qu'était un espace de nommage et comment on pouvait l'utiliser. On va maintenant en créer un. Dans l'exemple suivant, nous allons créer un espace de nommage pour isoler la partie de code qui concerne une balle. Cette balle peut être modélisée comme un cercle, soit par un centre et un rayon. Le centre est un point qui sera représenté comme une coordonnée sur un plan à deux dimensions. Il ne s'agit là de rien d'autre qu'un peu de géométrie, mais la géométrie est un des piliers qu'il faut maîtriser pour créer des jeux, puisque tout jeu nécessite d'afficher des éléments qu'il faut savoir correctement placer.

La ligne 3 crée l'espace de nommage en lui donnant un nom et en lui associant un bloc de code, délimité par les accolades lignes 4 et 18. Tout ce qui se trouve à l'intérieur de ce bloc est donc le corps de l'espace de nommage. Ses éléments sont appelés les membres de l'espace de nommage et ne sont donc accessibles au reste du monde que par l'utilisation du nom de cet espace, comme c'est le cas lignes 22 à 24 :

Fichier

```
01: #include <iostream>
02:
03: namespace balle
04: {
05:  double centre_x = 50, centre_y = 0;
06:  int rayon = 5;
07:
08:  void deplacer(double x, double y)
09:  {
10:   centre_x += x;
11:   centre_y += y;
12:  }
13:
14:  void afficher()
15:  {
16:   std::cout << "<Balle (" << centre_x << ", " << centre_y << ")>" <<
std::endl;
17:  }
18: }
19:
20: void test_balle()
21: {
22:     balle::afficher();
23:     balle::deplacer(1.2, 3);
24:     balle::afficher();
25: }
```

Il n'y a donc pas de difficultés majeures dans la création d'un espace de nommage. On peut en créer plusieurs dans le même fichier, les uns à la suite des autres et du code situé dans différents fichiers peut déclarer le même espace de nommage. Les fonctionnalités d'un espace de nommage seront tout simplement l'ensemble de toutes celles déclarées à travers tous les fichiers dans l'ensemble des blocs. Il est important de mettre le doigt sur la notion de portée des variables. En effet, les variables déclarées lignes 5 et 6 seront accessibles partout, dans tout l'espace de nommage, y compris à l'intérieur des fonctions. C'est ce qui se passe lignes 10, 11 et 16 : on accède à ces variables directement.

On pourrait y accéder depuis le reste du code, par exemple via un code tel que celui-ci : **`std::cout << balle::rayon << std::endl;`**. Cet exemple nous permet aussi de créer nos deux premières fonctions (outre la fonction particulière **`main`**). La fonction **`deplacer`** permet de modifier les deux variables permettant de localiser le centre de la balle et on notera que sa signature précise qu'elle attend deux nombres réels (de type **double**). Lors de l'appel de cette fonction, ligne 23, on lui passe des littéraux. Ces littéraux sont alors interprétés avec le bon type. Si on avait passé des variables, il aurait fallu s'assurer qu'elles soient du bon type, sans quoi il y aurait eu une conversion automatique. Comme on peut le constater, le second paramètre est un entier. Il est tout naturellement converti en double. Dans l'autre sens, la conversion aurait entraîné une perte d'information (troncature du nombre).

La fonction **`afficher`** nous permet de visualiser la balle à l'aide d'une représentation particulière et nous permet surtout de vérifier que la fonction **`déplacer`** s'est correctement exécutée. Cet extrait de code nous donne pour résultat **`<Balle (50, 0)>`** et **`<Balle(51.2, 3)>`**.

Pour terminer sur les espaces de nommage, sachez qu'il est possible de les créer l'un dans l'autre :

Fichier

```
01: #include <iostream>
02:
03: namespace chose {
04:     namespace truc {
05:         int machin = 42;
06:     }
07: }
08:
09: void test_namespaces()
10: {
11:     std::cout << "machin=" << chose::truc::machin << std::endl;
12: }
```

Le résultat de ce programme est : **`machin=42`**.

Comme vous le voyez, les mêmes règles s'appliquent. Cependant, il n'est pas nécessaire de multiplier ce genre de choses au risque de rendre le code trop compartimenté. Il faut utiliser les espaces de nommage avec parcimonie, uniquement lorsqu'ils présentent un vrai intérêt pour l'organisation de son code.

Enfin, dernière précision, il est possible de créer des espaces de nommage anonymes :

Fichier

```
namespace
{
    // Code
}
```

Tous les membres de l'espace de nommage anonyme d'un fichier ne seront accessibles que depuis le fichier lui-même, contrairement aux membres déclarés directement à sa racine. La fonction **`main`**, par exemple, doit impérativement être déclarée dans le corps du fichier et non pas dans un espace de nom, qu'il soit anonyme ou non, sans quoi vu de l'extérieur, c'est comme si elle n'avait pas été déclarée.

2. TYPES AVANCÉS

2.1 Tableaux

Un **tableau** est une collection ordonnée de données homogènes. Il ne s'agit pas d'un type particulier de donnée, mais les données qu'il contient sont toutes du même type.

Voici comment déclarer un tableau d'entiers : **`short tableau[7];`**.

On peut aussi l'initialiser directement : **`short tableau[7] = {1, 2, 3, 4, 5, 6, 7};`**.

Il est à noter que dans ce cas, la longueur du tableau est déductible des données, il n'est donc pas obligatoire de la donner.

On peut avoir des tableaux contenant ce que l'on souhaite, y compris d'autres tableaux, ce qui donne des tableaux de tableaux : **`short tableau2[6][7];`**.

Pour itérer sur les tableaux à une dimension, il faut procéder ainsi :

Fichier

```
01: for (i = 0 ; i < 7 ; i++)
02: {
03:   // Code à exécuter, pour chaque case du tableau, par exemple:
04:   cout << "tableau[" << i << "] = " << tableau[i];
05: }
```

Et pour des tableaux de tableaux (tableaux à deux dimensions), il faudra imbriquer les boucles d'itération :

Fichier

```
01: for (i = 0 ; i < 6 ; i++)
02: {
03:   for (j = 0 ; j < 7 ; k++)
04:   {
05:     // Code à exécuter pour chaque case du tableau:
06:     cout << "tab2[" << i << "][" << j << "]=" << tab2[i][j];
07:   }
08: }
```

On notera qu'il existe des objets conteneurs très intéressants en C++ qui rendent l'utilisation du tableau moins intéressante, alors que c'est un passage obligé en C. On présentera ces objets dans la section dédiée.

2.2 Structures de données

La notion de structure de données est également un héritage du C. L'idée d'une structure consiste à décrire un nouveau type de donnée, à partir d'un ensemble d'autres types ou structures :

Fichier

```
struct vector
{
    int x;
    int y;
};
```

Les composantes de la structure sont ses champs. On vient ici de définir un vecteur, en utilisant deux entiers, identifiés par **x** et **y**. Initialiser un tel vecteur est relativement simple : **`struct vector a = {4, 2};`**. Cette syntaxe est compatible C, mais en C++, le mot clé **`struct`** n'est plus nécessaire. De plus, déclarer des vecteurs sans leur assigner de valeurs est aussi possible : **`vector b, c;`**.

Voici comment assigner les valeurs après coup, soit une par une, soit par ensemble :

Fichier

```
b.x = 4;
b.y = 2;
c = {4, 2};
```

Enfin, voilà comment restituer ces valeurs :

Fichier

```
cout << "a=(" << a.x << ", " << a.y << ")" << endl;
```

Voici un exemple plus complet utilisant plusieurs structures :

Fichier

```
struct field
{
    vector point;
    vector direction;
    int intensity;
};
field f = {{0, 0}, {1, 2}, 5};
cout << "f={(" << f.point.x << ", " << f.point.y << "), (" <<
f.direction.x << ", " << f.direction.y << "), " << f.intensity << "}" <<
endl;
```

La déclaration de la structure reste relativement simple, il suffit d'énumérer ses champs, en les préfixant par leurs types. La déclaration et l'utilisation de la structure se comprennent aussi relativement aisément, lorsque l'on sait ce qu'elle contient.

On peut également utiliser des structures anonymes, ce qui est utile lorsque l'on a qu'une seule variable qui va utiliser la structure :

Fichier

```
struct
{
    int x;
    int y;
} my_vector = {4, 2};
```

C'est au niveau de la syntaxe que cela se joue. On peut également déclarer la structure et les variables l'utilisant dans la même ligne. La place d'un identificateur est importante. S'il se trouve juste après le mot clé **`struct`**, il est alors le nom de la structure. S'il se trouve après le bloc c'est le nom d'une variable (il peut aussi y en avoir plusieurs à ce niveau-là, si l'on déclare plusieurs variables).

Comme en C, C++ dispose d'un mécanisme permettant de donner un nom à des types en en faisant des alias : **`typedef unsigned short nombre;`**.

Une fois ceci fait, on peut écrire ceci : **`nombre a = 42;`**.

Ce qui est strictement équivalent à ceci : **`unsigned short a = 42;`**.

Dans le cas présent, l'utilité est très limitée. Elle est un peu plus visible lorsque l'on souhaite définir des tableaux particuliers : **`typedef int canvas[30][20];`**.

Ainsi, on vient de définir la notion de canevas, représentant un tableau à deux dimensions, de taille 30 par 20, ce qui peut représenter notre espace de jeu dans lequel on pourrait positionner les briques, par exemple. Le fait de centraliser la déclaration de cette notion de canevas et de la réutiliser ailleurs permet d'avoir un code plus simple et plus facilement maintenable. Voici en effet comment déclarer un tel canevas : **`canvas niveau1;`**. Enfin, il faut aussi savoir que les structures sont le plus souvent déclarées sous la forme d'un alias, pour être réutilisées ensuite :

Fichier

```
struct _vector
{
    int x;
    int y;
} vector;
```

Dans cet exemple, **`_vector`** est le nom de la structure tandis que **`vector`** est celui de l'alias. Pour déclarer un nouveau vecteur, nous ferons ceci : **`vector v {4, 2}`**.

2.3 Les classes de stockage

2.3.1 Portée d'une variable

La portée d'une variable est toute la portion de code pour laquelle la variable existe. Jusqu'à présent, nous avons utilisé de nombreuses variables, voici un exemple :

Fichier

```
01: void f()
02: {
03:     cout << "Hello World !" << endl;
04:     int i = 42;
05:     for (int j=0; j < i; j++)
06:     {
07:         cout << j + 1 << endl;
08:     }
09:     cout << "Bye World." << endl;
10: }
```

Dans cet exemple, les variables **`i`** et **`j`** sont toutes deux des variables locales. La variable **`i`** est une variable locale à la fonction **`f`**. Sa portée va de la ligne ligne 2 à la ligne 10, c'est-à-dire tout le bloc de la fonction f. Peu importe qu'elle ait été déclarée uniquement à la ligne 4. Bien évidemment, on ne peut pas utiliser cette variable avant sa déclaration, car elle n'existe pas encore, mais ceci est la notion de durée de vie – présentée au chapitre suivant – et non de portée qui intervient.

Dans le cas de la variable **`j`**, il s'agit d'une variable qui est locale à la boucle **`for`**. Sa portée va de la ligne 5 à la ligne 8. Elle n'est pas valable en dehors, même si c'est après la boucle, comme en ligne 9. Faites le test et vous aurez un message du type *« j was not declared in this scope »*. Ce qui est vrai : un bloc contenu peut voir toutes les variables locales de ses blocs contenants, mais un bloc contenant ne peut pas voir les variables locales d'un bloc contenu.

Notons un dernier point d'importance : une variable globale n'est pas déclarée à l'aide d'un quelconque mot clé supplémentaire, elle est simplement déclarée hors de tout bloc – soit ni dans une fonction ni dans une classe – mais directement à la racine d'un fichier, tout simplement :

Fichier

```
01: int reponse = 42;
02: void g()
03: {
04:     cout << reponse << endl;
05: }
```

Dans ce dernier exemple, la variable **reponse** est une variable globale et elle est utilisée par la fonction **g**. Il faut aussi noter que si la variable globale est déclarée à l'intérieur d'un espace de nommage, sa portée est celle de l'espace de nommage, il faudra utiliser l'opérateur de résolution de portée **::** pour pouvoir y accéder.

En aucun cas la portée d'une variable ne peut être modifiée par une classe de stockage.

Il existe une autre classe de stockage, **extern**, qui permet de déclarer une variable qui est définie dans une portée englobante ou dans un autre fichier. Ceci est utile pour la compilation séparée.

2.3.2 Durée de vie d'une variable

La durée de vie d'une variable globale est celle du programme. Ces dernières sont initialisées dès le lancement du programme (avant l'appel à la fonction **main**) et sont supprimées lorsque le programme s'éteint (après la fin de la fonction **main**).

La durée de vie d'une variable locale est celle de son bloc. Si le bloc est une boucle, la variable locale est supprimée à la fin de la boucle. Si le bloc est une fonction, la variable locale est supprimée à la fin de l'appel de la fonction. Il est possible de changer la durée de vie d'une variable locale par l'utilisation de la classe de stockage **static**. Si l'on utilise **static**, la valeur de la variable n'est plus détruite à la fin du bloc. Si le bloc est rejoué, alors dès le début du bloc cette variable reprendra la valeur qu'elle avait la dernière fois qu'elle l'a quitté.

En revanche, la classe de stockage **static** sur une variable globale aura une autre signification (sa durée de vie est déjà celle du programme). Cette nouvelle signification est de rendre cette variable inaccessible à l'extérieur du fichier où elle a été déclarée (même si on tente d'y accéder avec la classe de stockage **extern**). Déclarer les variables globales avec la classe de stockage **static** est une très bonne pratique, car cela les protège d'accès extérieurs et force l'utilisation d'accesseurs pour les manipuler.

Voici comment mettre ceci en évidence :

Fichier

```
01: void h(int increment ) {
02:     static int variable = 0;
03:     variable += increment;
04:     cout << variable << endl;
05: }
06:
07: int main()
08: {
```

```
09:     for (int i; i < 5; i ++)
10:     {
11:         h(2 * i + 1);
12:     }
13:
14:     return 0;
15: }
```

L'initialisation de la variable à **0**, réalisée ligne 2, n'est effectuée que la toute première fois que la fonction est exécutée. Le résultat de cet extrait de code est les cinq premiers nombres carrés (somme de la suite **2n+1**).

2.3.3 Emplacement mémoire d'une variable

Lorsqu'une variable est créée, elle va occuper un emplacement dans la mémoire. Or, le processeur ne sait pas utiliser directement la mémoire. Lorsqu'il a besoin d'une donnée, cette donnée est chargée dans un de ses registres (après avoir traversé ses caches L1,L2 et L3 s'il est présent). Le résultat de son calcul est également positionné dans ses registres, puis lorsque l'opération est terminée, le résultat est recopié du registre vers la mémoire. On précise ici que l'on ne fait qu'effleurer le fonctionnement réel de tout ceci, les puristes me pardonneront.

Bien que la mémoire soit rapide et les registres ultra rapides, si l'on doit répéter cette opération de multiples fois sur une variable très utilisée, c'est une perte de temps. Il est donc possible de *recommander* au compilateur de conserver une variable dans un registre pour éviter de la charger systématiquement depuis la mémoire. Pour ceci, on utilise la classe de stockage **register**. On a bien noté qu'il ne s'agit que d'une recommandation et que le compilateur fait ce qu'il veut, en particulier si vous avez demandé des optimisations. Il faut cependant savoir que les compilateurs sont largement meilleurs que les humains pour optimiser un programme et qu'il est largement recommandé de ne pas utiliser cette classe de stockage et de faire confiance à son compilateur.

La classe de stockage **volatile** permet d'indiquer au compilateur qu'aucune optimisation ne peut être réalisée sur cette variable, car sa valeur peut avoir été modifiée par un programme extérieur, sans qu'elle ait l'air d'avoir changé. Ce mot clé n'est réellement utile que dans certains cas particuliers comme la programmation de pilotes matériels ou de communication inter-processus. Le programme devra donc toujours aller relire la valeur avant de l'utiliser, car elle a peut-être changé entre deux lectures.

2.3.4 Constance d'une variable

On a déjà vu cette partie dans l'article du jour 1. Il s'agit ici de déterminer que la valeur d'une variable ne puisse être modifiée par l'utilisation de la classe de stockage **const**. On peut aussi l'utiliser à la fin du prototype d'une méthode pour déclarer que cette méthode ne modifie pas l'instance, ce que l'on reverra dans l'article suivant.

Il faut savoir qu'il existe aussi une classe de stockage **mutable** qui n'est utile que pour les structures : lorsque l'on a une structure **const**, on peut toujours déclarer un champ **mutable** pour passer outre.

On peut cependant noter également une nouveauté, **constexpr**, apparue avec C++11 et améliorée par C++14, qui peut d'être appliquée aux fonctions et méthodes,

permettant de préciser que la valeur de retour est constante. À ce moment-là, cette valeur de retour est calculée à la compilation, ce qui fait gagner du temps d'exécution et de la mémoire lors de l'exécution.

Une fonction **`constexpr`** doit retourner une valeur dont le type est celui d'un littéral. Si la fonction a des arguments et produit un calcul qui fait que le résultat n'est pas toujours prédictible au moment de la *compilation*, alors la fonction s'*exécutera* tout de même comme une fonction classique et générera son propre résultat.

3. POINTEURS ET RÉFÉRENCES

3.1 Pointeurs

Comme on a pu le voir, tout élément (variable, constante, structure, fonction, classe...) d'un programme est stocké en mémoire. Cette mémoire peut être modélisée par une rangée de cases numérotées – le numéro d'une case est son adresse – et chaque élément du programme utilise un certain nombre de ces cases (contiguës ou non) – ce que l'on appelle la taille de l'empreinte mémoire de l'élément. L'adresse mémoire d'un élément est quelque chose de très important, puisque la connaître permet d'aller le retrouver. Précisons que les divers éléments ne changent pas d'adresse mémoire à part si on le demande explicitement (réallocation mémoire) ou implicitement (appel d'une méthode d'un objet qui fait une réallocation mémoire de l'un de ses attributs, par exemple).

Une adresse mémoire étant le numéro de la case mémoire, elle est assimilable à un nombre entier. On peut par conséquent créer une variable dont le contenu est l'adresse d'une case mémoire. Une telle variable est ce que l'on appelle un pointeur. L'élément correspondant à cette adresse est dit pointé par le pointeur. Reste à savoir quelle est la taille de l'empreinte mémoire de cet élément pointé ; c'est la raison pour laquelle les pointeurs sont typés, de la même manière que les variables qu'ils pointent : **`int *p1;`**. Dans cet exemple, il faut comprendre que **`p1`** est un pointeur vers un entier. On accède au contenu de l'entier pointé en utilisant **`*p1`** (**`*p1`** est donc un entier). Lors de la déclaration, l'étoile est la caractérisation du fait que la variable **`p1`** est un pointeur, mais ce qu'il faut retenir est que si l'on veut travailler sur le pointeur, on utilisera **`p1`**, alors que si l'on veut travailler sur l'entier, on travaillera sur **`*p1`** : **`*p1 = 42;`**. L'utilisation de l'étoile dans l'instruction précédente est appelée déréférencement. Attention, lors de la déclaration, l'étoile ne porte pas sur le type, mais sur la variable. Ainsi, ceci n'est pas la déclaration de trois pointeurs vers des entiers : **`int* p1, p2, p3`**. Mais ceci l'est : **`int *p1, *p2, *p3`**. Ainsi, on peut déclarer plusieurs variables de type entier et pointeurs d'entiers sur une même ligne : **`int *p2, i, *p3;`**.

Notons qu'il est obligatoire de déclarer la variable entière avant de déclarer une autre variable lui pointant dessus, en utilisant l'indirection. Voici un exemple simple :

Fichier

```
double d = 42.34;
double *p4 = &d;
```

À partir de ce moment-là, on peut utiliser **d** ou ***p4** indifféremment. L'utilisation de ***p4** sans l'avoir préalablement fait pointer vers un emplacement mémoire à toutes les chances de très mal se passer : cela se traduit en lecture et écriture à des endroits aléatoires de la mémoire, ce qui est l'une des nombreuses erreurs possibles de manipulation de pointeur.

Pour terminer, voici une autre information que l'on peut tirer des exemples précédents : l'espace mémoire pointé par **p2** est celui d'un **double** alors que celui pointé par les autres pointeurs est celui d'un **int**.

Ce qui nous amène aux tableaux C (statique ou dynamique) : ces derniers allouent des cases mémoire contiguës et on passe d'un élément du tableau au suivant en allant à la case suivante (ou précédente), ce qui se fait en incrémentant ou décrémentant le pointeur. La valeur du pointeur, c'est-à-dire l'adresse mémoire qu'il contient n'est pas incrémentée ou décrémentée de un ou de deux, mais de la taille mémoire d'un élément du tableau et ceci sans qu'on ait à le penser :

Fichier

```
01: void test_pointeurs()
02: {
03:     int tableau[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
04:     int *p = tableau;
05:     cout << p << endl;
06:     *p += 42;
07:     ++p;
08:     cout << p << endl;
09:     *p += 33;
10:     p += 2;
11:     cout << p << endl;
12:     *p += 51;
13:     *(p+2) = 68;
14:     for (int i=0; i<10; ++i)
15:     {
16:         cout << ' ' << tableau[i];
17:     }
18:     cout << endl;
19:     double tableau2[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
20:     double *p2 = tableau2;
21:     cout << p2 << endl;
22:     *p2 += 42;
23:     ++p2;
24:     cout << p2 << endl;
25:     *p2 += 33;
26:     p2 += 2;
27:     cout << p2 << endl;
28:     *p2 += 51;
29:     *(p2+2) = 68;
30:     for (int i=0; i<10; ++i)
31:     {
32:         cout << ' ' << tableau2[i];
33:     }
34: }
```

Voici un résultat de cette fonction :

Terminal
```
0x7fffd2cfb3a0
0x7fffd2cfb3a4
0x7fffd2cfb3ac
 42 35 2 57 4 68 6 7 8 9
0x7fffd2cfb3d0
0x7fffd2cfb3d8
0x7fffd2cfb3e8
 42 35 2 57 4 68 6 7 8 9
```

Vous pouvez constater que l'adresse mémoire va de 4 en 4 pour les entiers et de 8 en 8 pour les doubles. Précisons également que travailler sur des pointeurs ne change absolument rien au contenu de la mémoire – les variables ne vont pas se déplacer – mais comme on l'a vu, cela permet de se déplacer dans cet espace mémoire.

3.2 Références

Les références n'existent qu'en C++ et non pas en C. Elles sont des synonymes d'identificateurs, c'est-à-dire qu'elles permettent de manipuler une variable sous un autre nom que celui sous lequel elle a été déclarée.

Voici comment déclarer une référence :

Fichier
```
int reponse = 42;
int &r = reponse;
```

À ce moment-là, les variable **reponse** et **r** sont exactement identiques. Modifier l'une modifie l'autre :

Fichier
```
reponse /= 2; // r et reponse valent 21
r += reponse; // r et reponse valent 42
```

Les références sont un moyen de simplifier la manipulation de variables en cachant les problématiques liées aux pointeurs. En effet, une référence doit être assignée dès sa déclaration. Elle n'est pas réassignable dans la suite de l'exécution du programme. Elles sont donc beaucoup plus faciles à manipuler que les pointeurs et améliorent la sûreté du code.

4. CONTENEURS

4.1 Typologie

Un conteneur est simplement un objet permettant de stocker plusieurs autres objets. Les présenter maintenant nous permettra de nous familiariser avec l'utilisation de certains types d'objets, avant de voir comment créer les nôtres.

L'objet stocké peut être appelé un contenu et en C++, ce contenu doit être typé. Par conséquent, les différents conteneurs sont tous homogènes : ils stockent tous des objets de même type. Lors de la déclaration d'un conteneur, on doit donc déclarer à la fois le type du conteneur ainsi que le type des contenus : `std::array<string> mots;`.

On vient de déclarer ici un tableau de chaînes de caractères. On aurait parfaitement pu utiliser un tableau C tels que ceux présentés plus haut, mais il y a plusieurs avantages à employer les conteneurs. Le premier est que les conteneurs disposent de méthodes de plus haut niveau, permettant de manipuler leur contenu assez facilement, sans avoir à s'inquiéter de problématiques basiques. Il est en effet assez facile de faire une erreur de programmation qui entraîne une *segmentation fault* lorsque l'on utilise des tableaux C. Le second avantage est qu'il existe des conteneurs adaptés à chaque cas d'utilisation et qu'en fonction de votre besoin, vous pourrez aller chercher le bon conteneur et ainsi optimiser votre code sans trop d'efforts.

Pour cela, il est nécessaire de faire un tour d'horizon de ces conteneurs. On peut les diviser en deux grandes familles. La première est la famille des **séquences**. Il s'agit tout simplement de ranger les éléments dans une liste, chacun de ces éléments étant identifié par sa position dans la séquence. Cette position est simplement un numéro nommé indice et permet d'adresser directement chaque élément. Si un élément est supprimé de la séquence, tous les éléments d'indice supérieur voient leur indice diminuer de un, de telle manière à ce qu'il n'y ait pas d'espace vide. La séquence de référence est le vecteur : elle permet de répondre à toutes ces exigences et dès lors que l'on ne se pose pas spécifiquement la question de l'optimisation de notre programme, elle peut être utilisée de manière exclusive. On la présentera en détail.

Parmi les autres séquences on peut trouver :

- ⇨ `array` : identique au vecteur, mais de taille fixe (ne pouvant évoluer après initialisation) ;
- ⇨ `list` : identique au vecteur, mais plus performant pour les insertions et suppressions à tout endroit de la liste, meilleur également pour l'itération quel que soit le sens de parcours **[1]** ;
- ⇨ `stack` : spécialement conçu pour traiter les piles (dernier entré, premier sorti), soit optimisé pour rajouter à la fin et sortir à la fin ;
- ⇨ `queue` : spécialement conçu pour traiter les files d'attente (premier entré, premier sorti), soit optimisé pour rajouter à la fin et sortie au début.

Les séquences les plus spécialisées contiennent moins de méthodes que les vecteurs et sont plus contraignantes. Par exemple, pour une pile, seul le dernier élément est accessible et peut être récupéré et il n'est possible d'en rajouter qu'à la fin. Cette limitation fait que l'on utilise forcément le type de séquence sélectionné de la bonne manière.

La seconde grande famille de conteneurs est les **associations**. Cette fois-ci, les valeurs ne sont pas référencées via un indice, mais via une clé, il n'y a donc pas de notion de position. Par conséquent, l'insertion ou la suppression d'un élément n'aura aucun impact sur les clés des autres éléments de l'association. Ces associations répondent à un besoin totalement différent de celui des séquences, celui d'associer une clé à une valeur. Il faut ainsi préciser à la fois le type du conteneur, celui de la clé et celui de la valeur, ce qui se fait ainsi : `std::map<string, int> occurences;`.

Dans cet exemple, on a choisi un conteneur de type **map**, les clés sont des chaînes de caractères et les valeurs des entiers. Cette association particulière a deux propriétés : ses éléments sont automatiquement triés entre eux selon leur propre relation d'ordre et ses clés sont forcément uniques. Il existe plusieurs variantes :

- **multimap** : les clés ne sont pas forcément uniques ;
- **unordered_map** ; il n'y a pas de relation d'ordre entre les éléments de l'association ;
- **unordered_multimap** : les clés ne sont pas uniques et il n'y a pas de relation d'ordre.

Nos présenterons en détail les conteneurs **vector**, et **map** dans cet article, cependant, si vous allez jeter un œil au code livré avec ce hors-série, vous verrez des cas d'utilisation pour tous les types.

Enfin, pour terminer, sachez que pour utiliser un conteneur, il faut ajouter à votre code l'en-tête correspondant à son nom. Pour notre part, nous aurons besoin de ceci :

Fichier

```
#include <vector>
#include <map>
#include <unordered_set>
```

En ce qui concerne le projet de jeu, nous utiliserons une liste de pointeurs vers les éléments à afficher, pour pouvoir itérer dessus lors de la routine d'affichage : **std::list<Sprite*> m_sprites;**.

Mais comme nous n'avons pas encore vu les classes, nous nous contenterons ici de présenter des exemples simples, pour que cette première approche des conteneurs reste abordable.

4.2 Vecteurs, première partie

La maîtrise d'un objet tel que le vecteur est un passage obligé pour pouvoir profiter pleinement du langage C++. Il est très complet et permet de faire beaucoup d'opérations de manière très sécurisée et très efficace. Lors de la déclaration d'un vecteur, on déclare le type de ses contenants (qui sont donc homogènes) et les éléments sont identifiés par leur position : **std::vector<int> nombres = {34, 42, 54, 16, 27, 42};**. Signalons le fait que cette manière d'initialiser un vecteur est possible depuis C++11. On peut aussi tout simplement créer un vecteur vide, en précisant sa taille ainsi : **std::vector<int> nombres (10);**.

Le vecteur sera initialisé avec la valeur par défaut du type contenu, soit **0** pour les entiers.

Il est possible de réaliser de nombreuses opérations sur ces vecteurs. Tout d'abord, on peut récupérer les divers éléments via l'utilisation de l'opérateur crochet et d'un indice. Ce dernier commence à **0**.

Fichier

```
cout << "Premier élément : " << nombres[0];
```

Il est à noter que si l'on demande un indice qui est plus grand ou égal à la taille du vecteur, cela génère une erreur. Au contraire, la méthode suivante permet de faire une vérification préalable et génère une exception le cas échéant (nous présenterons les exceptions dans le jour 5) :

Fichier
```
cout << "Premier élément : " << nombres.at(0);
cout << "Premier élément : " << nombres.at(100);
```

On peut accéder également très facilement au premier ou au dernier élément du vecteur :

Fichier
```
cout << "Premier élément : " << nombres.front();
cout << "Dernier élément : " << nombres.back();
```

L'ensemble de ce que l'on vient de voir est également utilisable pour l'affectation :

Fichier
```
nombres.back() = nombres.at(1);
nombres.at(4) = nombres.front();
```

On peut ensuite récupérer la taille du vecteur, c'est-à-dire son nombre d'éléments :

Fichier
```
cout << "Nombre d'éléments : " << nombres.size();
```

Il est aussi possible de tester le fait que le vecteur soit vide :

Fichier
```
cout << "Aucun élément ? " << nombres.empty();
```

Ainsi que vider le contenu d'un vecteur par **`nombres.clear();`**.

On peut modifier la taille du vecteur, soit en le diminuant (première ligne), soit en l'augmentant (si on ne précise pas la valeur des nouveaux emplacements en utilisant le second paramètre, ils seront initialisés à la valeur par défaut, soit **`0`** pour un entier) :

Fichier
```
nombres.resize(3);
nombres.resize(5);
nombres.resize(7, 42);
```

On peut aussi déterminer sa capacité (espace réellement utilisé, plus grand ou égal à la taille) et sa taille maximale (la taille maximale théorique qu'il pourrait avoir étant donné l'implémentation et les limites de la machine, il n'est pas garanti que l'on puisse effectivement l'atteindre) :

Fichier
```
cout << "Capacité : " << nombres.capacity();
cout << "Taille maximale : " << nombres.max_size();
```

Au fur et à mesure de l'utilisation du vecteur, la capacité s'ajuste en augmentant lorsque cela est nécessaire. Cependant, il est possible de modifier la capacité (en prévision d'un certain nombre d'éléments à venir, pour éviter d'avoir plusieurs réallocations (lesquelles

sont faites par le vecteur lui-même, pas par nous) : **`nombres.reserve(100);`**.
Il est à noter que la capacité n'est jamais ajustée à la baisse, ni automatiquement, ni via l'utilisation de la méthode **`reserve`**. On peut cependant demander au vecteur de réduire sa capacité pour l'ajuster à sa taille courante : **`nombres.shrink_to_fit();`**.

Voici comment rajouter des valeurs à la fin du vecteur : **`nombres.push_back(58);`**.
Et pour supprimer la dernière valeur : **`nombres.pop_back();`**.

On peut également signaler qu'il est possible d'initialiser (ou réinitialiser) un vecteur en précisant sa taille et la valeur de ses éléments. Si le vecteur avait un contenu, ce dernier est remplacé. Par exemple, si on veut que **`nombre2`** soit un vecteur de 12 éléments 42, on peut procéder ainsi : **`nombres2.assign(12, 42);`**.

Enfin, il est aussi possible d'échanger deux vecteurs (quelles que soient leurs tailles respectives) à partir du moment où ils sont homogènes :

Fichier

```
std::vector<int> nombres2 = {42, 58};
nombres2.swap(nombres);
```

Nous avons vu ici l'ensemble des fonctionnalités du vecteur qui ne nécessitent pas l'utilisation d'itérateurs. En effet, pour utiliser les autres fonctionnalités, il faut avoir au préalable découvert cet outil, ce que nous allons nous empresser de faire.

4.3 Itérateurs

Itérer sur un vecteur est relativement simple et ne requiert a priori pas de faire appel à une quelconque nouveauté ; il suffit simplement de travailler sur la position et de la faire varier de **`0`** à la taille « moins un » du conteneur :

Fichier

```
for (int i=0; i < nombres.size(); i++)
{
    std::cout << ' ' << nombres[i];
}
```

Seulement cette manière de faire n'est pas idéale, et ceci pour plusieurs raisons. Tout d'abord, si jamais dans la boucle **`for`** on supprime ou rajoute des éléments, on peut se trouver potentiellement face à un problème (même dans cet exemple où la taille est recalculée à chaque itération, ce qu'il ne faut usuellement pas faire). Ensuite, certaines séquences ne sont pas adressables à l'aide d'un indice, tout comme les **`map`**.

Enfin, il est appréciable d'avoir un moyen de parcourir les conteneurs qui soit homogène et qui fonctionne de la même manière, quel que soit son type.

Un itérateur est un objet particulier qui est associé au conteneur qu'il permet d'itérer : on itère un vecteur avec un itérateur de vecteur. L'information permettant de le créer est donc contenue dans son type :

Fichier

```
it =  std::vector<int>::iterator it;
```

Une fois que l'on a notre itérateur, il est relativement simple de passer à l'objet suivant : **`++it;`**. La seule chose qui nous manque est de déterminer le début et la fin du conteneur :

Fichier

```
it_debut =  std::vector<int>::iterator it = nombres.begin();
it_fin =  std::vector<int>::iterator it = nombres.end();
```

L'objet **it_debut** est un itérateur pointant vers le premier élément du conteneur alors que **it_fin** est un itérateur pointant vers l'élément théorique qui serait après le dernier élément (*past-the-end element*). Ainsi, l'algorithme de parcours précédent pourrait avantageusement être réécrit ainsi :

Fichier

```
for (std::vector<int>::iterator it = nombres.begin() ; it != nombres.
end(); ++it)
{
    std::cout << ' ' << *it;
}
```

Il est à noter que l'on peut aussi parcourir les conteneurs à l'envers, en utilisant **rbegin** et **rend**. Ainsi, le premier pointe vers le dernier élément de la séquence et le second vers l'élément théorique qui se trouverait avant le premier élément. C'est la raison pour laquelle **rbegin** n'est pas interchangeable avec **end** et **rend** ne l'est pas avec **begin**.

Enfin, on peut également vouloir préciser que l'on veut itérer en lecture seule, c'est-à-dire sans modifier les divers éléments du conteneur, auquel cas on dispose de **cbegin**, **cend**, **crbegin** et **crend**.

Dernier point, il ne faut pas confondre une méthode comme **front** ou **back** renvoyant une référence vers le premier ou le dernier objet du conteneur avec **begin** et **end** qui renvoient des itérateurs pointant vers lui [2].

4.4 Vecteurs, seconde partie

Maintenant que l'on sait ce qu'est un itérateur, il est temps de présenter les fonctionnalités du vecteur qui le requièrent. Pour commencer, voici comment insérer un élément au début d'un vecteur :

Fichier

```
std::vector<int>::iterator it = nombres.begin();
it = myvector.insert(it , 42);
```

La méthode renvoie un nouvel itérateur qui a pris en compte l'insertion du nouvel élément. Si nous ne faisons pas cela, il faut alors créer un nouvel itérateur en appelant à nouveau la méthode **begin**.

Si on souhaite insérer un élément à la position **n**, il faut procéder ainsi :

Fichier

```
it = myvector.insert(it + n , 42);
```

Enfin, il est possible d'insérer plusieurs éléments, qu'ils soient en provenance d'un tableau C ou d'un autre conteneur compatible (dans ce cas, un autre vecteur, un tableau ou encore une liste d'entiers) :

Fichier
```
it = myvector.insert(it + n, nombres2.begin(), nombres2.end());
```

Dans ce cas, nous insérons l'ensemble des éléments du vecteur **nombres2** dans le nôtre. Voici comment insérer tous les éléments de **nombre2**, sauf le premier et le dernier :

Fichier
```
it = myvector.insert(it + n, nombres2.begin() - 1, nombres2.end() + 1);
```

Il est aussi important de voir la syntaxe pour rajouter un tableau C :

Fichier
```
int tableau[] = {1, 2, 3};
it = myvector.insert(it + n, tableau, tableau + 3);
```

On voit donc que la méthode et son utilisation des itérateurs est un peu verbeuse, mais permet de répondre à tous les cas d'utilisation.

Voici maintenant comment supprimer le **n**-ème élément d'un vecteur : **nombres.erase(nombres.begin() + (n - 1));**. Pour supprimer les éléments **n** à **m** : **nombres.erase(nombres.begin() + (n – 1), nombres.begin() + (m - 1));**. Et on peut bien évidemment supprimer le dernier élément : **nombres.erase(nombres.rbegin());** . Ou l'antépénultième : **nombres.erase(nombres.rbegin() – 2));** . Ou encore en supprimer plusieurs à partir de la fin : **nombres.erase(nombres.rbegin() – n, nombres.rbegin() - m);**.

Il faut noter que les itérateurs peuvent s'utiliser partout. Si par exemple on veut remplacer les valeurs actuelles de **nombre2** par les 5 premières valeurs de **nombres**, on peut simplement procéder ainsi : **nombre2.assign(nombres.begin(), nombres.begin() + 5);**. Et si l'on souhaite recopier tous les nombres sauf le premier et le dernier, il faut procéder ainsi : **nombre2.assign(nombres.begin() + 1, nombres.end() - 1);**. Enfin, si on veut la même chose, mais en inversant le sens, il faut procéder ainsi : **nombre2.assign(nombres.rbegin() + 1, nombres.rend() - 1);**.

Un dernier point est à retenir : il est possible de récupérer les données brutes, sous la forme d'un pointeur vers le premier élément du tableau utilisé en interne par le vecteur pour stocker les données :

Fichier
```
nombres2.assign(4, 0); // Réinitialisation du tableau (4 éléments)
int* p = nombres2.data();
```

C'est l'occasion idéale pour s'entraîner à utiliser les pointeurs. Commençons par modifier la valeur du premier élément du tableau : ***p = 42;**. Ou encore : **p[0] += 100;**.
Et pour modifier les suivants, on peut soit utiliser l'opérateur crochet :

Fichier
```
p[1] = 16;
p[2] = 34;
p[3] = 54;
```

Soit déplacer le pointeur :

Fichier

```
++p;
*p += 18;
++p;
*p += 20;
++p;
*p += 22;
```

Le résultat de cet exemple est un tableau contenant les chiffres **142**, **34**, **54** et **76**.

4.5 Maps

Le second conteneur qu'il est impératif de maîtriser est l'association simple (**map**). Sa présentation sera plus rapide, parce qu'un certain nombre de choses sont communes avec le vecteur. Citons tous les itérateurs de **begin** à **crend**, ainsi que **empty**, **size**, **max_size**, **swap** et **clear**. On ne reviendra donc pas dessus (des exemples d'utilisation se trouvent cependant dans le code livré avec ce hors-série). Ensuite, il faut se rappeler que, par rapport au vecteur, les indices sont remplacés par les clés. Sachant cela, on peut tout de suite visualiser le fonctionnement de l'opérateur crochet : entre crochets, nous aurons une clé et non un indice et l'opérateur crochet nous permet de récupérer une référence vers la valeur souhaitée.

Il faut préciser que si la clé n'existe pas, alors un nouvel élément est créé et que la référence de cette nouvelle valeur est alors envoyée.

Il existe également une méthode **at** qui permet aussi de récupérer une valeur de l'association, sauf que si la clé n'existe pas, alors une exception est générée.

Sur le principe, ces deux éléments fonctionnent comme avec le vecteur, sauf qu'ils utilisent les particularités de l'association.

Fichier

```
01: std::map<string, int> occurrences;
02:
03: occurrences["mot"] = 42;
04: occurrences["python"] = 34;
05:
06: std::cout << "NB d' occurrences de \"mot\": " << occurrences["mot"] <<
std::endl;
07: std::cout << "NB d' occurrences de \"python\": " <<
occurrences["python"] << std::endl;
08: std::cout << "NB d' occurrences de \"rugby\": " <<
occurrences["rugby"] << std::endl;
09:
10: std::cout << "occurences contients " << occurences.size() << " mots."
<< std::endl;
```

Cet exemple montre qu'il y a bien 3 éléments dans l'association, le troisième ayant été créé à la ligne 8.

Pour rajouter une valeur, on peut aussi utiliser ceci : **occurrences.emplace("ovale", 34);**. Ou encore : **occurrences.emplace_hint(occurrences.end(), "ovale", 34);**. Si on utilise cette dernière méthode plusieurs fois, il ne faut pas oublier de récupérer l'itérateur modifié :

Fichier

```
it = occurrences.end();
it = occurrences.emplace_hint(it, "rond", 31);
it = occurrences.emplace_hint(it, "carré", 54);
it = occurrences.emplace_hint(it "point", 68);
```

On peut aussi utiliser la méthode **insert**, qui nécessite de faire appel à un nouvel objet, **pair**, qui doit être cohérent avec notre association : **occurrences.insert(std::pair<string,int>("ovale", 34));**. Cette méthode est plus verbeuse, mais est avantageuse si on dispose déjà de l'objet **pair** construit auparavant. Si la clé existe déjà, elle ne réalise pas l'insertion, mais retourne une référence vers l'objet déjà existant.

Pour rechercher un élément, on dispose de la méthode **find** :

Fichier

```
std::map<string, int>::iterator it1;
std::map<string, int>::iterator it2;
if1 = occurrences.find("rond");
if1 = occurrences.find("point");
```

Cette méthode retourne également un objet itérateur que l'on peut alors utiliser exactement comme ceux de **begin** ou **end** (en additionnant ou soustrayant un nombre).

Pour supprimer un élément, on peut se baser sur sa clé : **occurrences.erase("carré");**. On peut faire exactement la même chose en utilisant un itérateur : **occurrences.erase(it1);**. On peut également supprimer plusieurs éléments en utilisant deux itérateurs : **occurrences.erase(it1 ,it2);**.

Il existe encore un certain nombre de fonctionnalités, mais à ce stade, maîtriser celles que l'on vient de voir est largement suffisant pour survivre.

CONCLUSION

Avec ce que nous venons de présenter ici, nous entrons de plain-pied dans les particularités du langage C++. Si vous connaissez déjà C, vous venez de découvrir un pan totalement différent de ce à quoi vous êtes habitués et si vous faites l'effort de prendre en main ces outils, vous verrez rapidement que vous ne pourrez plus vous en passer.

On peut aussi noter, pour les pythonistes (désolé, on ne se refait pas) que la sémantique est très différente. En effet, en Python, la notion de liste est très différente. On notera que les piles et les files gardent le même esprit.

Comme cela a déjà été dit dans l'article, nous vous invitons vraiment à aller découvrir le code qui est livré avec ce hors-série et à le tester pour découvrir tous les conteneurs et leurs différences, puis à modifier ce code pour faire vos propres essais et vraiment vous familiariser avec des objets incontournables. Et en parlant d'objet, maintenant que l'on en a vu quelques-uns à l'œuvre, il est temps d'approfondir le paradigme objet en tournant la page. ■

RÉFÉRENCES

[1] Différences entre liste et vecteur :
http://stackoverflow.com/questions/2209224/vector-vs-list-in-stl

[2] Random access iterator :
http://www.cplusplus.com/reference/iterator/RandomAccessIterator/

JOUR 4

JOUR 4

MODÉLISEZ LES OBJETS

Dans cette partie, nous allons nous attaquer à la force principale de C++ : la Programmation Orientée Objet (POO). Nous allons découvrir par étapes successives les différentes facettes de ce paradigme particulier.

Notamment, nous allons y découvrir l'essentiel sur la définition des classes et comment utiliser l'héritage et les interfaces. Nous allons créer des opérateurs de classes et des fonctions virtuelles, puis découvrir les détails du polymorphisme et du typage dynamique.

1. LA PROGRAMMATION ORIENTÉE OBJET (POO)

1.1 Le paradigme objet

Le paradigme objet a été pensé vers la fin des années 50/début des années 60 au MIT. Le premier langage à l'utiliser fut **Simula 67**. Il a inspiré de nombreux autres langages, parmi lesquels nous pouvons citer **LISP**, **Object Pascal**, **Objective-C** et bien sûr le C++. Aujourd'hui, la plupart des langages modernes supportent ce paradigme, à un niveau plus ou moins avancé.

Ce fut dans les années 90 que ce paradigme devint de plus en plus populaire, car les langages le supportant atteignirent alors un haut niveau de maturité. Le paradigme objet répondait à des problématiques nouvelles, qui étaient entre autres la définition d'interfaces homme-machines (IHM) plus évoluées, l'augmentation de la complexité des applications, le besoin d'abstraction et de modélisation des données (utilisant entre autres la méthode UML) et surtout le développement d'applications modulaires.

1.2 Applications modulaires

Ce dernier point est fondamental, car il marque une rupture avec les méthodes de conception des programmes alors en usage ; ces dernières souvent très monolithiques, basées sur de longs cycles en V, avaient pour conséquence le fait que les grosses applications devenaient rapidement difficiles à faire évoluer et que leur code devenait très complexe et peu réutilisable. Cycle après cycle, la maintenance associée à ce type de logiciel devenait considérable, ayant pour conséquence la croissance de la dette technique, allant de pair avec la disparition des compétences et le maintien des contraintes opérationnelles qui imposent l'ajout continu de fonctionnalités, pour lesquelles les temps de test sont insuffisants. Aussi, pouvoir découper une grande application en petits modules permettait de diminuer grandement la complexité et d'améliorer la réutilisabilité entre parties de la même application comme d'une application à une autre, lorsqu'elles ont des parties communes.

Un langage tel que C++ a permis cette modularité et l'a encouragée, puisque cette façon de procéder fait partie des bonnes pratiques. Mais programmer de manière modulaire n'est pas une sinécure, elle impose un certain nombre de contraintes et demande pas mal d'efforts, en particulier au regard de la qualité.

1.3 Patrons de conception

Les normes de qualité de développement logiciel furent bousculées avec l'arrivée du livre du *Gang of Four* (Abrégé *GoF*, composé de Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides) intitulé « *Design patterns – Elements of Reusable Object-Oriented Software* » en 1995. L'idée fondatrice des patrons de conception est assez simple : avec l'expérience acquise en utilisant le paradigme objet, on s'est aperçu qu'il y avait un certain nombre de problématiques qui revenaient tout le temps. Et pour chacune d'entre elles, on pouvait déterminer une façon idéale de la résoudre en utilisant au mieux les

capacités du paradigme objet. Chacune de ces solutions est alors décrite sous la forme d'un patron de conception.

Lorsque nous-mêmes nous rencontrons une problématique, nous devons donc l'identifier et utiliser le bon patron de conception, plutôt que d'inventer une nouvelle solution exotique. L'avantage est que ces patrons sont reconnaissables, leur connaissance est partagée par tous les développeurs et leur implémentation sera donc forcément mieux comprise par tout le monde. Pour cette raison, les patrons de conception sont les bases fondamentales du développement modulaire et réutilisable ; ils font maintenant partie des critères de qualité essentiels pour le développement d'une architecture logicielle de qualité.

1.4 Un peu de vocabulaire

Reste à définir la notion d'objet : il s'agit de la donnée élémentaire d'un programme en POO et il est important de comprendre ce que recouvre cette notion. Sur le plan conceptuel, il est facile de se représenter simplement ce qu'est un objet : nous en avons partout autour de nous ! Une table, une chaise ou encore une voiture ; ce sont là des objets que nous pourrions chercher à représenter dans nos applications. Ensuite, il faut chercher à comprendre comment on peut les modéliser. En effet, ces objets vont posséder des caractéristiques (que l'on va nommer **attributs**) et vont pouvoir réaliser des actions grâce à des procédures particulières (que l'on nommera **méthodes**). La présence de ces attributs est commune à tous les objets d'un même type (c'est ce qui fait que deux objets sont semblables), mais leurs valeurs diffèrent (c'est ce qui fait qu'un objet est différent d'un autre). Les méthodes sont communes : tous les objets semblables peuvent réaliser les mêmes actions.

Pour prendre un exemple concret relatif à notre fil rouge, les attributs d'une brique sont : sa position, sa taille, sa couleur, le nombre de fois qu'il faut la toucher pour qu'elle disparaisse, le nombre de fois qu'elle a déjà été touchée. L'ensemble de ces attributs permet de caractériser une brique, autrement dit toutes les briques auront ces attributs. Par contre, d'une brique à l'autre, leurs valeurs seront potentiellement différentes.

Concernant les méthodes, chaque brique devra faire les actions suivantes : déterminer s'il y a collision avec une balle, dire si la brique est active ou non (elle n'est plus active à partir du moment ou elle a été touchée le nombre de fois nécessaire), afficher la brique. Ces méthodes sont communes à toutes les briques, mais lorsqu'elles seront appelées, elles répondront en fonction des caractéristiques de l'objet lui-même.

Une fois que l'on a décrit notre classe, on peut créer un objet en instanciant la classe : un objet est simplement une instance de la classe. Deux objets similaires sont deux instances de la même classe (on dit qu'ils sont du même type). Lorsque, dans un algorithme, l'on crée une instance, on va donc initialiser tous les attributs nécessaires. Une fois cette initialisation faite, on va pouvoir appeler les méthodes de l'instance, ce qui va permettre de faire travailler l'objet ou de lui demander des informations à propos de lui.

2. MON PREMIER OBJET : LE VECTEUR

Nous allons continuer sur notre fil rouge, le casse-briques. Nous avons pris l'exemple d'une brique, mais il faudra aussi, entre autres, une classe pour représenter une

balle et une autre pour représenter un plateau (ou raquette). Le point commun de toutes ces classes est qu'elles auront une position, cette dernière étant exprimée par rapport à un repère commun à tous les objets et fixé sur l'espace de jeu lui-même (qui sera également modélisé par une classe et que l'on nommera canevas).

Étant donné que l'on va travailler sur un espace en deux dimensions, la position va tout simplement s'exprimer par un jeu de coordonnées : une abscisse et une ordonnée. Nous allons représenter cette position par un vecteur.

2.1 Définition d'une classe

Notre première classe sera donc un vecteur, composé de deux coordonnées **`x`** et **`y`** qui sont tout simplement des nombres réels. La définition de chaque classe se fait dans un fichier d'en-tête qui prend son nom et portant l'extension **`.hpp`**. Ici, on crée donc le fichier **`Vecteur.hpp`** dont voici le contenu :

Fichier

```
namespace cassebrique
{
  class Vecteur {
    public:
      /* constructeur */
      Vecteur(double x, double y);

      /* méthodes */
      void nouvellesCoordonnees(double x, double y);
      void recupererCoordonnees(double &x, double &y) const;

    private:
      double m_x;
    double m_y;
  };
};
```

En premier lieu, nous pouvons noter l'utilisation d'un espace de nommage pour encapsuler notre code. Ce dernier sera commun à tous les éléments caractéristiques de notre jeu.

Analysons maintenant le code de la classe. Une nouvelle classe se déclare le mot clé **`class`** suivi de son nom, ici **`Vecteur`** et d'un bloc (délimité par les accolades) contenant les divers éléments de la classe. Ces éléments sont d'une part les attributs de notre classe. Ils sont au nombre de deux **`m_x`** et **`m_y`** et il se déclarent tout simplement à la manière de variables, c'est-à-dire un type de donnée (qui peut éventuellement être une autre classe) suivi du nom de la variable. D'autre part, on distingue les **prototypes** de toutes les méthodes. La notion prototype d'une méthode est identique à celle que nous avions présentée pour la fonction. N'oublions pas que nous sommes dans un fichier d'en-tête !

Parmi ces prototypes, il y a ce que l'on nomme le constructeur. Il porte le même nom que la classe (y compris la majuscule) et n'a pas de type de retour. Il sert à initialiser l'objet avec les paramètres fournis. Ensuite, il y a deux autres méthodes particulières qui permettent d'accéder aux coordonnées et sur lesquelles nous reviendrons.

2.2 Visibilité

Dans l'exemple précédent, on peut remarquer la présence des mots clés **`public`** et **`private`**. En première approche, nous nous contenterons de dire que le premier permet de déclarer des attributs ou méthodes publiques, c'est-à-dire lisibles et modifiables depuis n'importe quel endroit dans le code tandis que le second permet de définir des attributs ou méthodes privés, c'est-à-dire lisibles ou modifiables depuis n'importe quel endroit dans la classe.

Il existe aussi le mot-clé **`protected`** qui peut limiter la visibilité d'un attribut ou d'une méthode à la classe elle-même et à celles qui en sont dérivées, nous en reparlerons lorsque nous présenterons l'héritage.

2.3 Encapsulation

D'un point de vue générique, le concept d'encapsulation est un des fondamentaux de la programmation par objet. Il stipule que tous les éléments caractérisant un objet sont encapsulés, c'est-à-dire décrits au sein de sa classe. Ainsi, lorsque l'on crée une instance de notre objet, on dispose de tous les éléments qui nous permettent de le connaître et de le faire vivre. En C++, cette notion va plus loin : elle empêche l'usage, à l'extérieur de la classe, des caractéristiques propres de l'objet et impose l'utilisation d'actions pour les connaître ou les modifier. Autrement dit, en C++, les bonnes pratiques imposent de déclarer les attributs en tant qu'éléments privés.

Si on veut que la valeur d'un attribut soit lue, on doit donc écrire une méthode particulière qui va renvoyer la valeur d'un attribut (potentiellement sous conditions). Il s'agit d'un accesseur en lecture, souvent désigné par l'anglicisme *getter*. Si l'on veut que la valeur d'un attribut puisse être modifiée, il faut également écrire une méthode spécifique (qui va potentiellement décrire des pré-conditions et/ou des post-conditions) pour faire ce travail. Il s'agit d'un accesseur en écriture, ou *setter*. On doit donc impérativement utiliser des méthodes pour manipuler l'objet, ce dernier se chargera lui-même de modifier ses attributs si besoin.

Cela a deux avantages majeurs. Le premier est la fourniture d'une abstraction aux données, autorisant par exemple une modification ultérieure des attributs sans aucun impact sur les autres classes. Le second est la garantie de l'intégrité et la cohérence des données d'un objet. Avec cette méthode, il n'est en effet pas possible de modifier les données de façon inattendue et offre un cadre plus rigide qui est en particulier très utile dans le cas d'un projet réalisé à plusieurs, en obligeant tout le monde à suivre les mêmes règles.

Dans notre cas, la valeur d'un seul attribut ne nous intéresse pas, on souhaite gérer les deux attributs simultanément. Aussi, nous avons créé deux méthodes particulières : **`nouvellesCoordonnees`** pour mettre à jour des coordonnées (c'est un *setter*) et **`recupererCoordonnees`** pour récupérer la valeur des attributs (c'est un *getter*).

2.4 Constructeurs

Le constructeur est une méthode spéciale de la classe. C'est celle qui est appelée automatiquement à chaque fois qu'une nouvelle instance est créée. Il est chargé d'initialiser les différents attributs de l'instance. Il est obligatoirement de visibilité **`public`** et n'a pas de type de retour. L'écriture d'un constructeur est optionnelle.

Si la classe n'en contient pas de manière explicite, le compilateur en créera un automatiquement, de manière implicite. Ce dernier ne prendra aucun paramètre et laissera tous les attributs non initialisés. Il est donc la plupart du temps impossible de s'en affranchir. Toutefois, dès qu'un constructeur est défini explicitement dans une classe, le constructeur par défaut disparaît pour laisser sa place au nouveau, même si le prototype du nouveau constructeur (nombre et types des arguments) est différent. Il faut donc l'ajouter explicitement s'il était requis.

Il est à noter qu'au-delà du constructeur usuel, il existe un constructeur de copie qui lui aussi peut être explicite ou implicite. Cependant, dans les deux cas, il a le même prototype : il prend en paramètre une référence constante vers une instance de la même classe. Il est chargé de la cloner. Le processus de clonage est réalisé en copiant la valeur de l'ensemble des attributs vers la nouvelle instance. Lorsque certains attributs ne sont pas copiables, il est alors impératif de définir de manière explicite ce constructeur de copie, ce que nous aurions pu faire ainsi : `Vecteur(const Vecteur &v);`. Comme dans notre cas tous les types sont copiables, nous pouvons laisser ce constructeur de copie être géré de manière implicite.

2.5 Objet copiable

En C++, un objet copiable est simplement un objet dont la classe possède soit un constructeur de copie, soit un opérateur d'affectation. L'objectif de cette copie est de dupliquer tous les attributs de l'objet copié vers un nouvel objet. Lors de la copie, certains attributs d'un objet doivent être traités avec la plus grande considération : les pointeurs. En effet, un pointeur vers un objet est facilement copié, pour autant la mémoire pointée vers cet objet ne l'est pas automatiquement. Le résultat naturel est l'obtention de deux objets contenant des pointeurs vers un unique objet, ce qui peut entraîner de graves problèmes si ce n'était pas le but recherché.

2.6 Destructeur

Le destructeur, tel que son nom l'indique, est la méthode inverse du constructeur. Il est appelé lors de la destruction de l'objet. Celle-ci arrive toujours, que l'objet soit **statique** (il est alors détruit lorsque l'on quitte la fonction où il a été déclaré) ou **dynamique** (dans ce cas, l'objet doit être détruit avec l'opérateur `delete`). Il n'est pas toujours nécessaire de déclarer un destructeur. Dans ce cas, le compilateur ajoutera un destructeur par défaut à la classe, tel qu'il le faisait pour le constructeur. La plupart du temps, le destructeur est chargé de libérer la mémoire qui aurait été allouée dans l'objet, mais pas encore relâchée. Comme le constructeur, le destructeur prend le même nom que la classe, précédé du symbole `~`. Il n'accepte jamais de paramètres (qu'il serait de toute façon impossible de lui fournir).

Pour la classe `vecteur`, le destructeur déclaré de manière explicite serait : `~Vecteur() { }` (à positionner sous la déclaration du constructeur.

2.7 Implémentation

Maintenant que nous avons défini notre classe dans son fichier d'en-tête, c'est-à-dire défini les attributs et le prototype des méthodes, il faut en définir une implémentation,

c'est-à-dire définir le corps de ces méthodes. Ceci est usuellement fait dans un fichier portant le nom de la classe avec l'extension **.cpp**. Nous allons donc retrouver dans le fichier **Vecteur.cpp** un code de notre constructeur et nos deux méthodes qui pourraient ressembler à cela :

Fichier

```
01: #include <Vecteur/Vecteur.hpp>
02:
03: using namespace cassebrique;
04:
05: Vecteur::Vecteur(int x, int y)
06: {
07:         m_x = x;
08:         m_y = y;
09: }
10:
11: void Vecteur::nouvellesCoordonnees(int x, int y)
12: {
13:         m_x = x;
14:         m_y = y;
15: }
16:
17: void Vecteur::recupererCoordonnees(int &x, int &y) const
18: {
19:         x = m_x;
20:         y = m_y;
21: }
```

Sur le plan technique, il n'y a pas grand-chose à expliciter ici. Notre *getter* consiste à recopier les valeurs des attributs **m_x** et de **m_y** dans **x** et **y**, tandis que notre *setter* et le constructeur consistent à faire l'inverse. Rien de très bien compliqué. La seule chose à retenir, c'est la manière d'implémenter une méthode, qui utilise le prototype de la méthode suivie d'un bloc, à ceci près que le nom de la méthode est préfixé par le nom de la classe à laquelle elle appartient et qu'entre les deux, se trouve l'opérateur de résolution de portée **::**.

On peut noter tout de même l'importance du mot-clé **const**, situé à la fin du prototype de la méthode **recupererCoordonnees**. Ce dernier est absolument nécessaire, puisqu'il sert à clamer que quel que soit ce que fait la méthode, cela ne modifiera pas l'état de l'objet (aucun des attributs ne sera modifié durant son exécution). On verra que dans certains cas, le compilateur, par mesure de sécurité, refusera d'appeler une méthode si elle n'a pas précisé cet état de fait.

On peut tout de même améliorer un peu notre code. En effet, lorsque le constructeur réalise des opérations basiques (ce qui est usuellement le cas), on doit utiliser la liste d'initialisation pour initialiser les données membres. Celle-ci présente la syntaxe suivante : **Vecteur::Vecteur(int x, int y) : m_x(x), m_y(y) {}**. Le code effectif qui ne consiste qu'à de l'initialisation d'attributs est écrit après un deux-points et le bloc de code est vide. Cette liste d'initialisation n'est disponible que pour les constructeurs des classes.

2.8 Constructeurs des objets membres

Il est important de noter que puisque nous sommes dans le paradigme objet, tout type est défini comme un objet. Cela a pour conséquence que les objets membres

d'une classe doivent avoir leur constructeur appelé lors de la création de l'objet les contenant. Par convention, les constructeurs des objets membres sont appelés avant le constructeur de la classe et dans l'ordre de leur déclaration dans la classe. Dans le cas précédent, sans utiliser la liste d'initialisation, `m_x` puis `m_y` ont vu leur constructeur par défaut appelé de manière implicite. Leur initialisation dans le corps du constructeur consistait en une deuxième initialisation. C'est pour cela qu'il est nécessaire d'initialiser les variables présentes dans la liste d'initialisation (qui est exécutée avant l'appel au constructeur de la classe) et dans le même ordre que leur déclaration dans la classe, pour respecter l'ordre d'initialisation des objets membres.

2.9 Associations

Le paradigme objet permet de définir des objets, mais surtout de définir leurs relations, c'est-à-dire la manière dont ceux-ci interagissent. La principale manière consiste à les associer. Cette notion d'association entre deux objets peut également s'étendre, sur le plan conceptuel à la notion d'association entre un objet et un type. Ainsi, on notera que notre classe `Vecteur` dispose de deux attributs, qui sont de type `double`. On dispose donc de deux associations entre notre classe et le type `double`. Il existe deux sortes d'associations : une association forte, nommée composition et une association faible, nommée agrégation. Sur le plan conceptuel, il y a une grande différence entre les deux. Dans le cas de la composition, l'objet associé n'existe que si l'objet principal existe. Il n'a pas de sens sans cela. Dans le cas d'une agrégation, l'objet associé existe en dehors de l'objet principal. D'où la notion de fort et faible. Lors de la destruction d'un objet principal, tous ses objets composés sont également détruits au contraire de ses objets agrégés qui ont une vie par eux-mêmes.

2.10 Utilisation d'une classe

Maintenant que nous avons fait tout le travail préparatoire qui consiste à écrire notre classe, nous pouvons enfin en créer des instances et les manipuler. Commençons par initialiser quelques variables : `double x1{3.0}, x2{4.0}, y1{4.0}, y2{2.0};`.

Voici comment créer un vecteur (instanciation) : `Vecteur v(1, 2);`.

Comme nous l'avions déjà vu au jour 1 (dans la section intitulée « Variables »), on utilise ici les parenthèses pour l'initialisation. On peut maintenant récupérer les coordonnées en utilisant notre *getter* : `v.recupererCoordonnees(x1, y1);`. Et faire un test unitaire : `CPPUNIT_ASSERT(x1 == 1 && y1 == 2);`. Nous pouvons également utiliser notre *setter*, et vérifier que cela a fonctionné par un nouvel appel au *getter* et un test unitaire :

Fichier

```
v.nouvellesCoordonnees(x2, y2);

// Test unitaire
v.recupererCoordonnees(x1, y1);
CPPUNIT_ASSERT(x1 == 4 && y1 == 2);
```

Nous avons donc ici testé nos trois méthodes (nous pourrions également tester le constructeur de copie `Vecteur v2(v)` qui peut s'écrire également `Vecteur v2 = v`).

2.11 Objet courant

La notion d'objet courant est une notion essentielle pour la POO et la comprendre est toujours un peu un challenge lorsque l'on débute. Il est en effet important de voir que lorsque l'on décrit une action, on la décrit dans l'implémentation d'une méthode, décrite au sein d'une classe. Mais cette méthode ne s'exécute que sur un seul objet à la fois, l'objet courant, c'est-à-dire l'objet sur lequel la méthode est en train d'être appliquée.
Il faut savoir qu'il existe toujours, en C++, un pointeur vers ce fameux objet courant, nommé **`this`**. On peut alors utiliser la référence **`*this`** pour manipuler cet objet courant.

3. HÉRITAGE

Après avoir abordé les détails de la création d'une classe et l'instanciation d'objets, nous allons pouvoir entrer dans le vif du sujet. La programmation objet sans l'héritage et les fonctions virtuelles seraient bien vides de sens, alors nous allons voir comment exploiter au mieux ces mécanismes.

Pour cela, reprenons notre projet. Dans le casse-briques, nous allons avoir beaucoup d'objets présents sur le plateau de jeu. Nous avons le bord définissant les limites de l'aire de jeu, la raquette, une ou plusieurs balles et surtout plusieurs briques ! Un casse-briques avec une seule brique serait relativement peu ludique. Sachant que tous ces objets vont avoir une position sur l'aire de jeu, nous allons définir notre première classe ainsi :

Fichier

```
01: #include <Vecteur/Vecteur.h>
02:
03: namespace cassebrique
04: {
05:     class Element {
06:         public:
07:             Element(const Vecteur &p);
08:
09:             void nouvellePosition(const Vecteur &p);
10:             void recupererPosition(Vecteur &p);
11:
12:         private:
13:             Vecteur m_position;
14:     };
15: };
```

Cette première classe **`Element`** est très générique et peut représenter n'importe quel type d'objet. Elle met en commun ce que nous avons dit précédemment, c'est-à-dire que tout objet dans l'aire de jeu a une position connue. Et comme nous l'avons vu pour la classe **`Vecteur`**, nous avons un constructeur qui définit la position de notre objet et des accesseurs pour accéder aux données de notre objet.

Enfin, il reste à définir la manière dont les objets interagiront entre eux : en effet, on peut imaginer à ce stade que l'on va créer une classe **`Canevas`** pour représenter l'espace de jeu, une classe **`Balle`** et une classe **`Raquette`** pour des raisons évidentes. Les balles vont interagir avec le canevas, les raquettes et les briques pour gérer la détection de collision, entre autres ; il faudra que l'on définisse comment. Il faudra aussi que l'on se

penche sur la question de la réutilisabilité pour voir s'il est possible de ne pas doublonner le code à ce niveau-là, par exemple, par l'utilisation des patrons de conception. On va aussi devoir créer une classe **Jeu** pour gérer les aspects de l'animation du jeu, ce qui est un peu moins facile à imaginer de prime abord. Mais rassurez-vous, nous allons faire tout ceci très progressivement.

3.1 Concept d'héritage

L'héritage est un mécanisme offert par les langages objets qui permet de faire bénéficier à d'autres classes des propriétés de la classe de laquelle on hérite. On dit qu'une classe hérite d'une autre : la classe fille hérite de la classe mère. Ce mécanisme est fondamental en POO. C'est ce qui permet de réaliser des architectures modulaires et évolutives sans avoir à dupliquer du code lors des évolutions successives du programme, même inattendues.
Le concept est très simple ! En effet, lorsqu'une classe hérite d'une autre, les méthodes et attributs de la classe parent sont transférés à la classe fille. Toutefois, les règles de l'encapsulation s'appliquent toujours : si les fonctions et attributs déclarés **public** sont hérités, les fonctions et attributs **private** ne sont pas accessibles, car ils ne sont disponibles que pour la classe mère. Mais comment faire pour que les classes filles puissent utiliser les attributs de la classe mère, sans que le reste du monde puisse y accéder ? C'est la raison d'être du mot-clé **protected**, qui s'ajoute à la liste des mots-clés **public** et **private**, permettant de répondre à cette problématique. Ainsi, toutes les méthodes et les attributs déclarés sous le mot-clé **protected** et jusqu'au prochain mot-clé (ou la fin de la classe), ne sont accessibles que par la classe mère et ses classes filles.

3.2 Classe Brique

Nous allons réaliser notre première classe utilisant l'héritage : la classe **Brique**, qui hérite de la classe **Element**.

Fichier

```
01: #include <Element/Element.h>
02:
03: namespace cassebrique
04: {
05:     enum class BriqueCouleur { BLEU, BLANC, ROUGE };
06:
07:     class Brique : public Element {
08:         public:
09:             Brique(const Vecteur &, BriqueCouleur);
10:         private:
11:             BriqueCouleur m_couleur;
12:     };
13: };
```

Syntaxiquement, il n'y a pas de grosse différence par rapport à la déclaration d'une classe simple. La seule différence est à la ligne 7, lors de la définition du nom de la classe fille, à laquelle on ajoute le mot clé **public** (ou **private**) suivi du nom de la classe mère. Nous utilisons ici le mot-clé **public**, car nous voulons hériter des propriétés de la classe mère, ce qui aurait été impossible si le mot-clé utilisé avait été **private**. Ce dernier cas est principalement utilisé pour les interfaces (ce que nous

verrons plus bas dans cet article). En faisant ceci, toutes les méthodes publiques de la classe **Element** sont disponibles, ce qui nous permet de récupérer la position d'une brique via la méthode **recupererPosition** tel que le montre le code suivant :

Fichier

```
Brique b(Vecteur(1,2), BriqueCouleur::BLANC);
CPPUNIT_ASSERT(b.recupererPosition() == Vecteur(1,2));
```

Mais nous voulons que notre classe **Brique** soit plus qu'un simple élément positionnable, sans quoi cet héritage serait inutile. On doit pouvoir lui rajouter des propriétés qui lui sont propres, ce que nous réalisons aux lignes 8 à 11. Nous avons ajouté la couleur en tant qu'attribut privé dans la classe (ligne 11) et modifié le constructeur (ligne 9) pour permettre l'initialisation de cette couleur. Ainsi, nous permettons à nos briques d'avoir des couleurs différentes, améliorant ainsi l'esthétisme du jeu. Cette couleur est choisie parmi une énumération de couleurs prédéfinies (ligne 5).

3.3 Héritage et constructeurs

Un des constructeurs de la classe mère doit toujours être appelé lors de l'instanciation de tout objet de la classe fille. Cela implique qu'il doit exister un constructeur compatible lors de l'initialisation du nouvel objet. Dans notre cas, comme nous n'avons pas défini de constructeur par défaut pour les objets de la classe **Element** donc le compilateur ne pourra pas l'appeler automatiquement. Il faudra préciser quel est le constructeur que nous voulons utiliser (code de **Brique.cpp**) :

Fichier

```
01: #include <Element/Brique.h>
02: using namespace cassebrique;
03:
04: Brique::Brique(const Vecteur &v, BriqueCouleur couleur) : Element(v),
m_couleur(couleur) {}
```

Nous précisons à la ligne 4, dans la liste des initialisations (que nous avons expliqué précédemment) que le constructeur de la classe fille **Brique** appelle le constructeur **Element** pour initialiser la partie de l'objet relative à la classe **Element**. On en profite pour aussi initialiser la couleur de cette brique.

Attention, l'appel au constructeur de la classe mère se fait préalablement à l'exécution du code contenu dans le corps du constructeur de la classe fille. Il est donc obligatoire d'utiliser la liste des initialisations de la méthode, pour appeler le constructeur de la classe mère que nous souhaitons utiliser (ici **Element(const Vecteur &)**). Tenter de le faire dans le corps de la fonction sans passer par la **liste d'initialisation** produira une erreur de compilation, car le compilateur cherchera de toute façon un constructeur de la classe mère préalablement à celui de la classe fille et il sera bien incapable de le trouver (car notre classe **Element** ne possède pas de constructeur par défaut). L'ordre des initialisations est donc le suivant : d'abord les objets membres de la classe mère **Element**, le constructeur de la classe mère **Element**, puis les objets membres de la classe fille **Brique** et enfin le constructeur de la classe fille **Brique**.

3.4 Héritage imbriqué (*inception*)

Pour encore améliorer l'expérience du joueur, c'est-à-dire lui rendre la tâche plus difficile, nous voulons définir des briques incassables. Celles-ci doivent avoir une couleur spécifique, par exemple rouge, afin de maximiser l'impact psychologique sur le joueur lorsqu'il découvre un niveau contenant beaucoup de rouge. Pour cela, nous allons repartir de notre classe **`Brique`**, qui définit déjà les propriétés d'une brique classique, à laquelle nous allons apporter une nouvelle propriété qui est d'être incassable. Plutôt que de modifier la classe brique, ce qui reviendrait à la complexifier avec un cas particulier, nous allons là aussi utiliser l'héritage. Une nouvelle classe, qui héritera de la classe **`Brique`**, va affiner les caractéristiques d'une brique classique :

Fichier

```
01: class BriqueIncassable : public Brique {
02:     public:
03:         BriqueIncassable(const Vecteur &);
04: };
```

Dont le constructeur est le suivant : **`BriqueIncassable::BriqueIncassable (const Vecteur &v) : Brique(v, BriqueCouleur::ROUGE) {}`**. Nous constatons ici que la classe **`BriqueIncassable`** va à nouveau préciser quel est le constructeur de la classe mère à utiliser dans sa liste d'initialisation, ce qui rajoute un nouveau niveau dans l'ordre d'initialisation des constructeurs et des objets membres.

3.5 Casting

Comme nous venons de le voir, nous avons déclaré deux types de briques : les normales et les incassables (cette dernière en tant que classe fille des briques normales). Or lorsque nous allons manipuler toutes les briques du niveau, nous ne voulons pas avoir à gérer les différences entre elles, mais plutôt considérer toutes les briques de la même manière, quelle que soit leur classe réelle, comme si elles partageaient un même type. Cela est possible en utilisant une déclaration telle que celle-ci : **`Brique &brique = Brique Incassable(Vecteur(0,0));`**. Bien que notre brique soit incassable, nous pouvons l'assigner à une référence du type de sa classe mère, plus générique que notre classe fille. Cela ne pose pas de problème, car notre objet hérite de sa classe mère, donc il détient toutes les propriétés de celle-ci, comme nous l'avons vu précédemment. Cette opération de conversion d'un objet d'une classe fille vers sa classe mère est un *cast* implicite (*upcasting* en anglais).

L'opération inverse, c'est-à-dire convertir une référence vers une brique générique en une brique incassable (nommé *downcasting* en anglais), est beaucoup moins évidente (il est communément admis qu'avoir recours à ce type de conversion relève d'une erreur de conception).

3.6 Gestion des collisions

Il reste maintenant à définir comment se comporte la balle lorsqu'elle rencontre une brique. En effet, pour qu'une brique soit incassable, il ne suffit pas qu'elle soit rouge. Il faut que lorsque la balle rencontre une de ces briques incassable, elle se contente de

rebondir dessus, sans la détruire. Dans le modèle objet, cela signifie que nos briques vont devoir se comporter (via un appel de méthode) différemment lorsqu'une balle rencontre une brique, en fonction de sa friabilité (caractère incassable). Nous allons donc tout naturellement ajouter une méthode de **collision** à nos deux classes **Brique**.

Toutefois, lorsque nous allons construire notre niveau, nous n'aurons qu'une liste de références vers des objets de la classe **Brique**, car nous aurons utilisé l'*upcasting* sur tous nos objets **BriqueIncassable** pour les changer en **Brique**. La question est : comment permettre à ces deux types de briques de se comporter différemment alors que nous les traitons comme des briques similaires en tout point. Comment allons-nous, à partir d'un objet **Brique**, réussir à appeler la bonne méthode **collision**, c'est-à-dire celle de la classe réelle de la brique ? Pour réaliser ceci, nous devons utiliser les fonctions virtuelles.

3.7 Fonctions virtuelles

Il est important de s'attarder un peu sur le fonctionnement des fonctions virtuelles. Il s'agit d'un des concepts fondateurs de la POO et qui va de pair avec l'héritage. Leur fonctionnement est le suivant : une méthode d'une classe mère déclarée comme virtuelle peut être surchargée par une méthode déclarée dans une des classes filles si celle-ci détient un prototype identique. C'est-à-dire que la méthode de la classe fille pourra être appelée à la place de la méthode de la classe mère, y compris si l'objet a été *downcasté*. Les deux méthodes coexistent, le choix de celle qui sera effectivement exécutée dépend uniquement du type de l'objet manipulé.

Dans le cas de nos briques, nous allons déclarer une méthode **collision** dans les deux classes **Brique** et **BriqueIncassable**, dont la méthode de la classe mère est virtuelle.

Fichier

```
01: class Brique : public Element {
02:     public:
03:         Element();
04:         virtual ...lement();
05:         [...]
06:
07:         virtual ObstacleResultat collision();
08:         [...]
09: };
10:
11: class BriqueIncassable : public Brique {
12:     public:
13:         ...
14:         ObstacleResultat collision() override;
15: };
```

Nous voyons ici apparaître deux nouveautés : les mots-clés **virtual** et **override** qui typent nos fonctions **collision**. C'est le mot-clé **virtual** qui va préciser que la fonction **collision** de la classe mère peut être surchargée par une fonction d'une classe fille. De la même manière, le mot-clé **override** après la fonction **collision** de la classe fille précise que cette fonction doit obligatoirement surcharger une fonction de la classe mère. Bien que ce mot clé ne soit pas strictement nécessaire (il a été introduit en C++11), il permet de détecter à la compilation si une erreur d'héritage ou de prototype est présente. Le compilateur va rechercher une fonction virtuelle à surcharger et lever une erreur s'il n'en trouve pas.

Le contenu de ces deux fonctions est particulièrement simple :

Fichier

```
ObstacleResultat Brique::collision() { return ObstacleResultat::DETRUIT; }
ObstacleResultat BriqueIncassable::collision() { return
ObstacleResultat::INTACT; }
```

Tout ceci va ensuite se comporter de la manière suivante :

Fichier

```
01: /* test de la fonction virtuelle collision */
02: Brique           normale(Vecteur(2,2), BriqueCouleur::BLANC);
03: BriqueIncassable incassable(Vecteur(4,4));
04:
05: Brique  copie_incassable  = incassable;  /* copie non polymorphique */
06: Brique &ref_incassable    = incassable; /* la référence est
polymorphique */
07:
08: /* collisions avec brique normale : destruction */
09: CPPUNIT_ASSERT(normale.Collision() == ObstacleResultat::DETRUIT);
10:
11: /* collisions avec brique incassable : pas de détruction */
12: CPPUNIT_ASSERT(incassable.Collision()     == ObstacleResultat::INTACT);
13: CPPUNIT_ASSERT(ref_incassable.Collision() == ObstacleResultat::INTACT);
14:
15: /* le résultat n'est pas celui attendu : utiliser des refs
16: /* pour la résolution dynamique de la fct virtuelle */
17: CPPUNIT_ASSERT(copie_incassable.Collision() ==
ObstacleResultat::DETRUIT);
```

Nous avons expliqué que la résolution de la méthode à exécuter était dynamique et dépendante du type de l'objet utilisé. Nous voyons que lorsque nous appelons la méthode d'un objet déclaré sur la pile, il n'y a pas de résolution dynamique de la fonction. Celle de la classe respective est appelée. En revanche, lorsque nous utilisons une référence du type **`Brique`**, dans laquelle nous avons assigné un objet de type **`BriqueIncassable`** (avec *upcasting*), la méthode appelée est celle de la **`BriqueIncassable`** et non de l'objet **`Brique`**. Cela est rendu possible, car la référence (comme le pointeur) détient des propriétés polymorphiques et permet de manipuler la brique incassable en tant que brique générique, tout en gardant la propriété incassable de l'objet.

Le polymorphisme est la capacité d'un objet à pouvoir prendre différents types. Notre référence vers une brique peut être une brique simple ou une brique incassable. Le choix est déterminé pendant l'exécution du programme et non à la compilation. Le dernier test est particulièrement important : il montre qu'il est indispensable d'utiliser des références ou des pointeurs pour manipuler l'objet, car dans le cas d'une copie simple, l'objet est réellement dupliqué, mais avec une perte des caractéristiques de la classe fille. On dit qu'il s'agit d'une copie non polymorphique.

3.8 Héritage multiple

En C++, il est possible pour une classe d'hériter de plusieurs classes parentes. Pour ce faire, la syntaxe est assez simple. Voici un exemple d'école, non relié à notre projet :

Fichier

```
01: class A
02: {
03:     [...]
04: }
05: class B
06: {
07:     [...]
08: }
09: class AB : public A, public B
10: {
11:     [...]
12: }
```

La partie intéressante est à la ligne 9 : on y déclare les deux classes parentes ainsi que leur caractère **public** ou **private**. L'intérêt de procéder ainsi est assez élevé. Si on a plusieurs comportements identifiables, on peut les traiter dans des classes nommées *mixins*. Ces dernières sont implémentées, potentiellement fonctionnelles, mais pas faites pour être instanciées directement, mais pour être héritées. Lorsque l'on a plusieurs *mixins*, on peut alors créer des classes en définissant leur comportement par l'héritage d'une sélection de *mixins*. Cela demande cependant un effort de modélisation de ses classes assez important, sachant que l'utilité réside dans le fait que les comportements sont identiques. Si vous devez surcharger les méthodes d'un *mixin* dans votre classe, cela ne présente que peu d'intérêt et cela pourrait signifier que la notion d'interface est peut-être mieux adaptée à votre besoin. Ici, l'idée est vraiment de capitaliser du code commun à plusieurs classes dans un *mixin* et de déclarer dans les classes utiles que l'on utilise le comportement défini tel quel dans le *mixin*.

Nous avons vu précédemment que les constructeurs des classes mères sont appelés avant le constructeur de la classe fille, mais comment se comporte l'appel aux constructeurs des classes mères dans le cas de l'héritage multiple ? Les classes mères sont simplement initialisées dans l'ordre de leur déclaration, de gauche à droite, c'est-à-dire d'abord la classe héritée la plus à gauche, puis celle à sa droite et ainsi de suite. Rappelons que le constructeur ne peut pas être surchargé.

4. SURCHARGE DE MÉTHODE

4.1 Redéfinition et polymorphisme paramétrique

Pour permettre d'introduire un certain nombre de concepts et présenter des cas dont nous n'aurons pas besoin, on va s'éloigner quelques instants de notre projet et reprendre des exemples plus scolaires. La notion de surcharge recouvre essentiellement une seule notion : surcharger une méthode est un moyen de la redéfinir dans une classe fille.

Voici la classe parente dont nous souhaitons redéfinir ou surcharger les méthodes :

Fichier

```
01: class Base
02: {
03:     public:
04:         void a(int i);
05:         virtual void b(int i);
06:         void c(int i);
07:         virtual void d(int i);
08:         void e(int i);
09:         virtual void f(int i);
10: };
```

Et voici la classe fille qui va nous permettre de visualiser les différents cas possibles :

Fichier

```
01: class Derived : public Base
02: {
03:     public:
04:         void a(int i);              // Redéfinition
05:         virtual void b(int i);      // Redéfinition
06:         void a(double i);           // Surcharge
07:         virtual void b(double i);   // Surcharge
08:         using Base::e;              // Déplacement
09:         using Base::f;              // Déplacement
10: };
```

Dans cet exemple, les fonctions **a** et **b** sont toutes deux redéfinies, puisque l'on retrouve leur prototype à l'identique incluant le mot clé **virtual**. Chaque exemple est proposé avec et sans ce mot-clé pour montrer que, dans la partie qui nous concerne, ce mot-clé n'a aucune influence.

Pour revenir aux deux premières fonctions, il s'agit d'une forme de surcharge qui éclipse totalement la méthode parente, on peut parler de redéfinition. Pour autant, il est toujours possible d'y faire référence dans l'implémentation :

Fichier

```
void Derived::a(int i)
{
    [...]
    Base::a(i);
    [...]
}
```

Ainsi, on réussit à créer une surcharge qui va réutiliser la méthode parente et y rajouter de nouvelles choses.

Autre point important, dans la classe dérivée, on ne fait pas référence aux méthodes **c** et **d** définies dans la classe parente. Lorsque l'on va appeler cette méthode depuis une instance de la classe **Derived**, on va d'abord chercher dans cette classe, puis remonter dans un second temps à la classe parente. On finira par trouver la bonne méthode.

Enfin, en ce qui concerne les méthodes **e** et **f**, l'utilisation du mot-clé **using** permet de les déplacer dans la classe fille, à l'identique. Notons que nous ne précisons que le nom de la fonction et non son prototype complet. L'utilisation de ce mot-clé permet

de signifier que la méthode référencée sera directement accessible. Par contre, avec ce seul changement au code précédent **`class Derived : private Base`**, alors l'utilisation de **`using`** serait obligatoire pour permettre à la classe dérivée de proposer les méthodes **`e`** et **`f`** tandis que les méthodes **`c`** et **`d`** ne seraient tout simplement pas utilisables. À noter que le mot-clé **`using`** peut aussi servir à changer la visibilité d'une méthode : on peut déclarer en visibilité **`public`** une méthode qui au niveau du parent est en visibilité **`protected`**.

Pour terminer, on s'attardera sur les lignes 6 et 7 de la classe **`Derived`** pour observer que nous avons ici deux nouvelles méthodes qui sont en réalité des méthodes existantes, mais avec une signature différente. Dans le langage C++, on appelle cela surcharge, alors qu'il s'agit de ce que l'on appelle ailleurs du polymorphisme paramétrique.

Enfin, signalons que, dans le cas où il existe plusieurs fonctions exploitant le polymorphisme paramétrique dans une classe parente, mais qu'une seule d'entre elles est surchargée dans la classe fille, alors C++ ne trouve que la méthode surchargée. Pour prévenir cela, il faut utiliser le mot-clé **`using`** pour déplacer toutes les méthodes portant le même nom – en effet, on ne précise que le nom de la fonction derrière ce mot-clé et non un prototype – puis surcharger la méthode de notre choix dans un second temps. La notion de surcharge en C++ recouvre donc plusieurs notions distinctes et d'une source à l'autre, le vocabulaire utilisé pour en parler peut varier.

4.2 Principes de la surcharge d'opérateur

Le langage C++ dispose d'un nombre conséquent d'opérateurs **[1]**. Une des forces du langage est que vous pouvez faire en sorte que vos propres objets puissent être manipulés grâce à ces opérateurs, grâce à la surcharge d'opérateur. L'avantage de cette technique est indéniable. Elle n'est pas plus complexe que le fait d'écrire une méthode standard, mais utiliser un objet avec des opérateurs est beaucoup plus simple et lisible que de l'utiliser avec des méthodes.

Il faut savoir qu'il existe principalement deux catégories d'opérateurs : les opérateurs unaires et les opérateurs binaires. Les premiers ne travaillent que sur un objet (un opérande) et les seconds travaillent sur deux objets :

Fichier

```
int i(42);
-i;   // Opérateur soustraction unaire, i est l'opérande
i-7;  // Opérateur soustraction binaire, i et 7 sont les opérandes
```

Reprenons notre exemple de vecteur et améliorons-le pour qu'il soit aussi simple à utiliser qu'un type **`double`**, en introduisant la possibilité d'utiliser les opérateurs usuels **`+`**, **`-`**, **`+=`**, **`-=`**, **`==`**, **`!=`**. Nous les écrirons en tant que fonctions membres de la classe. Ce choix est motivé d'une part par des raisons de cohérence avec ce qui a été écrit jusque-là, car certains opérateurs ne peuvent être définis qu'ainsi et d'autre part parce que ce type d'implémentation est plus aisé.

Commençons par le commencement : le type des opérandes et du résultat sera identique. En effet, on va pouvoir additionner un vecteur avec un autre vecteur et le résultat sera un troisième vecteur. Ensuite, il faut comprendre que lorsque nous utilisons un opérateur, nous allons en réalité utiliser notre méthode spéciale. L'objet courant sera l'opérande de gauche et (pour les opérateurs binaires), l'objet en paramètre sera l'opérande de droite.

Ainsi, on peut écrire cette ligne de code : **`v3 = v1 + v2;`**. Elle sera équivalente à ce pseudo-code : **`v3 = v1.methode_plus(v2);`**.

4.3 Implémentation de l'opérateur +

Nous sommes maintenant prêts à écrire notre premier opérateur. Nous allons commencer par implémenter l'opérateur **+** comme fonction membre de classe. Nous définissions ainsi, dans notre fichier d'en-tête, le prototype de cet opérateur de cette manière :

Fichier

```
Vecteur operator+(const Vecteur &other) const;
```

C'est la présence du mot-clé **`operator`** qui fait de cette méthode une méthode spéciale. Notons que l'opérateur **+** ne modifie ni l'objet courant (d'où la présence du mot clé **`const`** à la fin du prototype) ni le paramètre (d'où la présence du mot clé **`const`** avant celui-ci). L'opérateur va seulement retourner un nouvel objet qui sera le résultat : la somme des deux objets manipulés. Nous pouvons maintenant réaliser son implémentation, dans le fichier **`Vecteur.cpp`** :

Fichier

```
01: Vecteur Vecteur::operator+(const Vecteur &other) const
02: {
03:     /* on retourne un nouvel objet qui est la somme
04:      * des deux objets *this et other */
05:     return Vecteur(m_x + other.m_x,
06:                    m_y + other.m_y);
07: }
```

4.4 Implémenter l'opérateur +=

Cet opérateur est sensiblement différent, car il modifie l'objet courant. Il va être utilisé de la manière habituelle, soit **`v1 += v2;`**. Ceci est équivalent à **`v1 = v1 + v2;`** et l'on peut assimiler cela au pseudo code **`v1.methode_plus_egal(v2);`**. Pour fournir un tel opérateur, nous avons besoin d'écrire le prototype suivant :

Fichier

```
Vecteur& operator+=(const Vecteur &other)
```

On note l'absence du mot clé **`const`** à la fin de son prototype justement parce que l'objet courant est modifié. On note aussi que le résultat retourné est une référence vers cet objet courant. Ceci peut paraître surprenant, puisque si l'objet courant a été modifié, il n'y a a priori aucune raison de renvoyer un résultat. En réalité, ceci est utile dans le cas où plusieurs opérations seraient chaînées. Par contre, comme précédemment, l'objet passé en paramètre ne doit pas être modifié.

L'implémentation de cette méthode peut se faire de plusieurs manières. L'une d'entre elles consiste à mettre à jour les attributs de l'objet courant par rapport à ceux de l'objet en paramètre. Une autre consiste à réutiliser l'implémentation de l'opérateur **+** et de créer un nouvel objet courant :

Fichier

```
Vecteur& Vecteur::operator+=(const Vecteur &other)
{
    /* on affecte dans l'objet courant la somme des deux objets
     * avec réutilisation de l'opérateur + */
    *this = *this + other;

    /* on retourne une référence vers l'objet modifié  */
    return *this;
}
```

Là encore, on modifie l'objet courant grâce au pointeur **this**. Nous réutilisons l'opérateur **+** vu précédemment pour calculer la somme des deux objets puis l'affecter à l'objet manipulé, soit exactement l'opération équivalente **v1 = v1 + v2**.

4.5 Opérateurs de comparaison

Pour terminer cette introduction au modèle objet de C++, nous allons présenter les opérateurs d'égalité **==** et de différence **!=**. Voici leur prototype :

Fichier

```
bool Vecteur::operator==(const Vecteur &other) const;
bool Vecteur::operator!=(const Vecteur &other) const;
```

Cette syntaxe est toujours identique et ceci doit toujours se placer dans le fichier d'en-tête. Il ne faut pas oublier l'utilisation des mots-clés **const**, car la comparaison ne doit modifier aucun des deux opérandes. L'implémentation en tant que fonctions membres est là aussi assez directe :

Fichier

```
bool Vecteur::operator==(const Vecteur &other) const
{
    return (m_x == other.m_x) && (m_y == other.m_y);
}

bool Vecteur::operator!=(const Vecteur &other) const
{
    return !(*this == other); /* on réutilise l'opérateur == */
}
```

En effet, comparer deux vecteurs consiste à comparer leurs coordonnées respectives. Par contre, on notera que l'on implémente pas les opérateurs **>**, **<**, **>=** ou **<=**. En effet, pour des vecteurs, de tels opérateurs n'auraient pas de sens. Leur en donner embrouillerait l'esprit de ceux qui seraient amenés à lire le code.

5. INTERFACE

Un héritage est utilisé lorsqu'un objet est une spécialisation d'un autre objet (héritage simple avec de nouvelles méthodes et/ou des surcharges) ou lorsque l'on définit cet objet comme étant la combinaison de deux autres (héritage multiple). En ce cas, toute l'implémentation est réalisée dans la ou les classes parentes, potentiellement ajustées dans la classe fille.

Une interface est utilisée lorsque différents objets doivent réaliser certaines opérations similaires alors qu'ils sont hétérogènes et que l'implémentation de ces opérations est propre à chaque objet. C'est aussi la base du patron d'architecture « injection de dépendance ». Au lieu qu'une méthode accepte en paramètre un objet d'un type prédéterminé, ce qui est une dépendance dure, elle accepte un objet implémentant une certaine interface. L'interface doit alors décrire tout ce qu'il est nécessaire d'implémenter pour que cet objet soit utilisable dans cette méthode. Quelle que soit son utilisation, la définition d'une interface est celle-ci : une interface est un mécanisme d'héritage qui permet de passer un contrat entre l'objet appelant et l'objet appelé. Ce contrat porte sur les prototypes des méthodes que l'objet doit implémenter et éventuellement sur les préconditions et les postconditions [2] que l'implémentation des méthodes doit respecter. En d'autres termes, une interface est une classe contenant les prototypes des méthodes, mais n'étant pas implémentée, ce que l'on précise explicitement. Lorsque l'on veut que les objets d'une classe respectent les prototypes d'une interface afin d'être utilisables de la manière définie, il lui suffit d'implémenter cette interface et d'implémenter les méthodes définies par celle-ci.

5.1 Interfaces multiples

Il est possible d'implémenter autant d'interfaces que nécessaire. La classe fille devra alors pour ce faire implémenter toutes les méthodes de toutes les interfaces qu'elle implémente. Il est également possible d'hériter d'une ou plusieurs classes et d'implémenter une ou plusieurs interfaces en même temps.

Elle devra aussi appeler les constructeurs des différentes classes dont elle hérite, comme nous pouvons le voir sur cet exemple de **`Sprite`** qui hérite de deux classes :

Fichier

```
01: /* SpriteObservé est utilisé pour les objets suivis  */
02: class SpriteObserved : public Sprite, public SpriteObservable {
03:     public:
04:         SpriteObserved(SpriteObservateur &o,
05:                        const Vecteur<double> &p,
06:                        const Vecteur<int> &t,
07:                        Element &r) : Sprite(p, t, r),
SpriteObservable(o, *this) {}
08:
09:         virtual bool afficher(Ecran &) = 0;
10:
11:         /* observable */
12:         void nouvellePosition(const Vecteur<double> &position)
override {
13:             m_position = position;
14:             notify();
15:         }
16:
17:         void deplacer(const Vecteur<double> &mouvement) override {
18:             m_position += mouvement;
19:             notify();
20:         }
21: };
```

La classe **`SpriteObserved`** hérite de sa classe mère **`Sprite`**, mais implémente aussi de la classe **`SpriteObservable`**.

Signalons pour terminer qu'il n'y a pas de distinctions entre classes parentes et interfaces dans la déclaration de la classe fille. Comme vu précédemment, que ce soit une classe ou une interface, les classes mères et les interfaces sont simplement initialisées dans l'ordre de leur déclaration, de gauche à droite. Donc, dans notre exemple, la classe `Sprite` sera initialisée en premier puis la classe `SpriteObservable`.

Encore une fois, sur le plan syntaxique et sur le fonctionnement interne, il n'y a pas de différence fondamentale entre une classe parente et une interface. C'est au niveau conceptuel que les deux doivent être utilisés différemment, car les deux notions servent des buts différents.

5.2 Héritage ou interface ?

Il n'est parfois pas toujours facile de déterminer si une classe doit être une interface ou non. Il existe un principe mathématique qui permet de savoir comment un objet doit hériter d'un autre. Cela s'appelle le principe de substitution de Liskov [3] et il donne la définition de ce qui doit être un sous-type d'un objet et des propriétés que cela doit respecter. Dans le cas où le sous-type contrevient à ce principe, il est nécessaire d'utiliser une interface plutôt qu'un héritage.

La différence entre l'héritage et l'interface peut aussi se formaliser ainsi : pour l'héritage on dit que **A** est une partie de **B** alors que pour l'implémentation d'une interface, on dit que **A** est implémenté selon les règles de **B**.

CONCLUSION

Nous venons d'étudier les fondamentaux de la programmation objet en C++. Ce vaste sujet nous a amenés à créer de nombreuses classes et découvrir de nombreux concepts : l'héritage, les méthodes virtuelles, les interfaces ; nous avons vu comment *caster* nos objets dans des types polymorphiques ; nous avons évité les pièges liés aux constructeurs et destructeurs ; nous avons surchargé les opérateurs pour rendre l'usage de nos classes aussi simple que s'il s'agissait d'un `int` ! Et dans tout ceci, l'encapsulation nous a garanti l'abstraction de nos données privées et l'usage d'accesseurs pour accéder au contenu de nos objets. Nous allons maintenant pouvoir utiliser nos classes dans les autres composants de l'application sans risque d'erreur !

Bien entendu, nous sommes loin d'avoir entièrement couvert le sujet, il existe beaucoup d'autres voies à explorer pour parfaitement maîtriser les classes en C++. Nous vous laissons les découvrir au fur et à mesure de votre pratique du langage, mais sachez que l'essentiel a été couvert ici. En effet, nous en savons assez pour nous pencher sur le prochain thème d'importance, qu'il est indispensable de maîtriser, car il est utilisé dans tous les projets C++ et qu'il structure l'ensemble de la bibliothèque standard : les *templates*. ▮

RÉFÉRENCES

[1] Liste de référence des opérateurs :
http://en.cppreference.com/w/cpp/language/operators

[2] Présentation de la programmation par contrat :
https://fr.wikipedia.org/wiki/Programmation_par_contrat

[3] Définition du principe de substitution de Liskov :
https://fr.wikipedia.org/wiki/Principe_de_substitution_de_Liskov

JOUR 5

JOUR 5

ÉCRIVEZ DES CLASSES GÉNÉRIQUES

Depuis les articles précédents, vous avez eu le temps de digérer les itérateurs, les chaînes de caractères, les ensembles, les pointeurs et les références ?

C'est tant mieux, car nous allons en rajouter une petite couche pour vous donner définitivement toutes les clés pour vous permettre de prendre en main la bibliothèque C++ par vous-même.

Nous terminerons par vous montrer comment réaliser un code générique et comment le spécialiser dans un second temps, le dernier des grands points incontournables de C++.

JOUR 5

1. EXCEPTIONS

Le mécanisme des exceptions est un ajout de C++, mais il date de la parution d'un langage pionnier dans beaucoup de domaines, Smalltalk, paru la même année que le C. La problématique est la suivante : il s'agit de gérer les problèmes de responsabilité lors de la survenue d'une erreur. Imaginons que je sois une fonction qui doit se connecter à une base de données. On me donne une URL et lorsque je tente de joindre le serveur, il ne répond pas. Que dois-je faire ? Je ne peux pas ne rien faire et arrêter le programme semble un peu rude. Mais que faire ? Tenter d'accéder à un autre serveur, afficher un message d'erreur, envoyer un courriel au responsable ? La bonne réponse est : rien de tout cela. En effet, ceci n'est pas ma responsabilité. Si on me demande de me connecter à un serveur et que je n'y arrive pas, car le serveur n'est pas en ligne, je vais lancer une alerte en donnant ces informations. À charge à celui qui est responsable de savoir que faire avec cette alerte. On dit que je lève une exception.

Les exceptions, c'est cela, c'est la manière de communiquer sur ces alertes. Ensuite, il faut suivre le cheminement de l'alerte et c'est assez simple. Pour arriver à la fonction qui a lancé l'alerte, on a exécuté la fonction **`main`**, qui, à son tour, a appelé une autre fonction, qui... vous avez compris ce qu'il se passe, on a construit une pile de fonctions qui part de **`main`** et qui finit à la fonction de connexion à la base de données. Et bien l'alerte remonte toute cette pile jusqu'à rencontrer un responsable. Ce dernier est celui qui dit, textuellement : essaie de faire ceci et s'il y a un problème, reviens me voir, on trouvera une alternative. On dit qu'il capture l'exception et qu'il la traite. Et si par hasard, il n'y a aucun responsable, eh bien l'exception finit son chemin au bout de la pile et le programme *crashe*, provoquant une erreur. Ce qui, au final n'est pas si différent que s'il n'y avait pas d'exceptions du tout : au moins, on s'est donné les moyens de faire quelque chose.

Lorsque le développeur voit un tel *crash*, il se doit de déterminer ce qui s'est mal passé et de désigner un responsable qui doit surveiller un risque particulier et engager une réponse alternative en cas d'exception.

1.1 Lancer une exception

On a donc bien compris que l'on lève une exception lorsqu'on a identifié un problème auquel on ne sait pas comment faire face :

Fichier

```
if (reponse!= 42)
{
    throw 42;
}
```

Le mot clé **`throw`** est celui qu'il faut utiliser pour lever l'exception. Mais une exception, cela peut être n'importe quoi... un nombre, une chaîne, une classe... Cependant, il faut savoir que les bonnes pratiques actuelles ne peuvent pas se contenter d'un simple nombre pour gérer les exceptions. Il faut au minimum un message clair sous la forme d'une chaîne :

Fichier

```
if (reponse!= 42)
{
    throw "H2G2, Dude!";
}
```

L'idéal reste quand même de se référer à des exceptions existantes, c'est-à-dire un objet héritant de la classe **`std::exception`** **[1], qu'il soit déjà implémenté ou que vous dériverez par vous-même** :

Fichier

```
if (reponse!= 42)
{
    throw std::logic_error("H2G2, Buddy!");
}
```

Ce qui laisse le choix d'un message d'erreur et qui a l'avantage d'être plus précis. Le plus difficile reste le fait de désigner les conjectures dont la fonction n'est pas responsable, de déterminer leurs conditions. Pour le reste, lancer le bon type d'exception et mettre un message adapté n'est que de la finition.

1.2 Intercepter et traiter une exception

Avant d'intercepter une exception, il faut déterminer à quel moment elle peut arriver. Et une exception arrive lorsque l'on utilise ce que l'on nomme une fonction critique, c'est-à-dire une fonction susceptible de lever une exception :

Fichier

```
try
{
    fonction_critique();
}
catch (const std::logic_error &e)
{
    std::cerr << "L'exception " << e.what() << " a été levée." <<
std::endl;
}
```

Il est important de noter que l'on passe l'exception par référence dans la boucle **`catch`**, ce qui permet d'avoir accès à la classe réelle de l'exception et non la classe exception elle-même, laquelle aurait des choses moins intéressantes à dire (on perdrait le sous-type de l'exception levée, voir le jour 4 sur les fonctions virtuelles et le polymorphisme). Il faut savoir que tous les objets déclarés dans le bloc après **`try`** sont détruits et voient donc leur destructeur appelé. Il faut que ces derniers soient capables de libérer leur mémoire quel que soit leur état, pour pouvoir être détruits correctement lorsqu'une exception est levée. Même l'objet exception déclaré après **`throw`** est détruit, mais un mécanisme le copie (un constructeur de copie doit être disponible) pour le rendre disponible dans l'objet exception de **`catch`**.

On note aussi que l'exception dispose d'une méthode **`what`** qui permet de retrouver le message d'erreur donné au **`throw`**. Une fois que l'on sait cela, on sait tout ce qu'il y a à savoir sur les exceptions. On peut éventuellement rajouter que l'on peut mettre plusieurs blocs **`catch`** les uns à la suite des autres pour capturer des exceptions différentes (dans le cas où la fonction critique puisse lever plusieurs exceptions de nature différente) comme le montre le code suivant :

Fichier

```
try
{
    fonction_critique();
}
```

```
catch (const std::logic_error &e)
{
    std::cerr << "L'exception " << e.what() << " a été levée." <<
std::endl;
}
catch (const std::bad_alloc &)
{
    std::cerr << "Nous n'avons plus de mémoire." << std::endl;
}
```

Notez qu'il n'est pas possible de capturer différents types d'exceptions dans le même bloc **`catch`**. Au passage, si aucune exception capturée ne correspond à l'exception levée, la pile continue d'être remontée comme si l'on n'avait pas eu ce bloc de traitement d'exception.

Finalement, il existe un gestionnaire d'exception universel, qui permet de capturer n'importe quelle exception, quelle que soit sa classe. Il utilise une syntaxe qui ne précise pas d'objet, mais le remplace par des points de suspension :

Fichier

```
catch (...)
{
     cout << "Exception inattendue !" << endl;
}
```

Il ne permet pas de récupérer d'information sur l'exception, car le type d'objet n'est pas défini.

1.3 Liste des exceptions autorisées pour une fonction

Comme nous l'avons vu, pour écrire un gestionnaire d'exception, il faut connaître les différents types d'exceptions qui vont être levées par les fonctions appelées dans le bloc **`try`**. Mais sans plus de précisions, il est difficile de définir la liste des types d'exceptions à implémenter dans les blocs **`catch`**. Il faut lire tout le code appelé à la recherche des différentes exceptions qui peuvent être levées. Aussi, pour faciliter la définition de la liste des exceptions levées, il est possible d'ajouter à chaque fonction la liste des exceptions qu'elle est susceptible de lever. L'intérêt est à la fois de préciser l'interface de chaque fonction et d'interdire à une fonction de lever des exceptions qu'elle n'a pas déclarées dans son prototype. Par « interdire », cela signifie que lever une exception non déclarée va mettre fin au programme en appelant **`std::terminate`**.

Par exemple, pour définir qu'une fonction ne peut lever que des exceptions de type **`std::logic_error`**, nous écrirons le prototype de la fonction comme ceci : **`int fonction_critique() throw(std::logic_error);`**.

1.4 Usage des exceptions

Les exceptions permettent donc de simplifier le code de gestion des erreurs, tout en fournissant une information de meilleure qualité sur l'origine de l'erreur. En effet, elles permettent d'éviter d'avoir à gérer les codes d'erreur dans chaque fonction d'un composant et de les centraliser à un seul endroit qui sera le gestionnaire des exceptions. C'est aussi la seule méthode pour signaler qu'une opération a échoué dans

un constructeur, car les constructeurs n'ont pas de valeur de retour. Un constructeur qui lève une exception ne finit pas la construction de l'objet et n'appellera donc pas son destructeur. Bien que très utile, l'exception dans un constructeur pose deux problèmes : la manipulation des objets partiellement initialisés et la récupération des exceptions pour les objets déclarés de manière globale. Il existe donc une syntaxe particulière permettant à un constructeur d'appeler son propre gestionnaire d'exception.

2. FONCTIONS ET CLASSES GÉNÉRIQUES

La fonction ou classe générique, ou *template* en anglais, est une fonctionnalité extrêmement puissante en POO. Elle apporte au langage la possibilité d'utiliser le paradigme de programmation générique. Ce dernier consiste à permettre au développeur d'utiliser des paramètres génériques dans les fonctions ou les classes.

Pour une fonction générique, lorsque l'on parle de paramètre générique, on parle du fait qu'elle doit pouvoir fonctionner avec des paramètres et/ou des valeurs de retour de type différent. Par exemple, imaginez devoir concevoir une fonction permettant l'addition de deux nombres homogènes (deux `int` entre eux, deux `double` entre eux...), il faudrait écrire autant de fonctions que de types pouvant représenter ces nombres. La programmation générique vous permet de n'écrire qu'une et une seule fonction qui fonctionne, quel que soit le type réellement utilisé pour les paramètres et la valeur de retour.

Pour une classe générique, lorsque l'on parle de paramètre générique, on parle du fait qu'elle doit pouvoir fonctionner avec des attributs de type différent et que ses méthodes puissent également prendre des paramètres et des valeurs de retour de ce même type. Par exemple, imaginez que vous deviez créer un vecteur (tel que nous l'avons présenté dans le jour 4) qui puisse fonctionner pour des `double` ou des `int` ou encore des `float` ou des `short`, en gardant exactement les mêmes fonctionnalités. Attendez une seconde... En fait non, n'imaginez pas, nous allons le faire ensemble. Au lieu de créer ces quatre classes, nous allons utiliser la programmation générique pour n'en créer qu'une.

Mais attention, ce n'est pas pour autant que l'on abandonne l'idée du typage statique pour le typage dynamique : le typage reste statique. En réalité, la classe générique ou la fonction générique n'est jamais utilisée en tant que telle, car elle n'existe pas dans le programme final. À la compilation, C++ va détecter leurs utilisations avec des types particuliers. Il va donc créer automatiquement toutes les fonctions ou classes standards nécessaires à partir de leur version générique et des types réellement utilisés. Ce sont eux qui seront concrètement utilisés par le programme. C'est la raison pour laquelle on parle également de modèle générique de code. Ce modèle générique est utilisé pour créer de nouvelles fonctions ou de nouvelles classes, spécialisées pour un type particulier. La création d'une de ces fonctions ou classes spécialisées, réalisée à partir de son modèle, s'appelle spécialisation : on spécialise le modèle générique pour un type donné et ce fameux type est le paramètre du modèle. Pour dire les choses d'une autre manière, un modèle est donc créé quand nous voulons qu'une classe ou une fonction puisse s'adapter à différents types, sans avoir besoin de dupliquer le code, ce qui est toujours une mauvaise idée et complique grandement la maintenance du code.

2.1 Application à la classe Vecteur

Pour notre projet, nous avons besoin de placer des éléments à des coordonnées précises. Nous avions donc créé une classe Vecteur et lui avions donné la possibilité d'être manipulée aussi simplement qu'un type en lui permettant de gérer des opérateurs. Mais en réalité, cette classe a un souci, qui n'est pas des moindres : elle est implémentée pour utiliser le type **`int`** uniquement. Or, cela ne nous suffit pas. En effet, pour placer un élément à l'écran, nous avons besoin du **`int`**, le pixel étant indivisible, mais pour calculer le déplacement réel, il nous faut des nombres réels, quitte à les arrondir au moment de l'affichage. Nous pourrions donc vouloir utiliser des **`double`** ou éventuellement, si l'on change d'avis, ou pour d'autres circonstances, des **`float`**. Voici le contenu de cette classe, avant modélisation :

Fichier

```
01: class Vecteur {
02:     public:
03:         Vecteur(int x, int y);
04:
05:         void nouvellesCoordonnees(int x, int y);
06:         void recupererCoordonnees(int &x, int &y) const;
07:
08:         Vecteur& operator+=(const Vecteur &other);
09:         Vecteur& operator-=(const Vecteur &other);
10:         Vecteur operator+(const Vecteur &other) const;
11:         Vecteur operator-(const Vecteur &other) const;
12:
13:         bool operator==(const Vecteur &other) const;
14:         bool operator!=(const Vecteur &other) const;
15:
16:     private:
17:         int m_x;
18:         int m_y;
19: };
```

Nous allons ensemble faire la démarche consistant à transformer le code ci-dessus pour être capable de le faire fonctionner avec différents types de nombres.

2.2 Écrire une classe générique, partie 1

Pour déclarer une classe générique ou *template* de classe, nous utilisons tout naturellement le mot-clé **`template`**, suivi de près par le bloc **`<typename T>`**. La notation **`T`** est ici un nom arbitraire choisi pour représenter le paramètre du *template*, c'est-à-dire le type qui sera utilisé pour spécialiser le modèle. Ayant vu simplement cela, nous sommes en état de réécrire la classe **`Vecteur`** précédente, à l'exception des lignes 8 à 14 : nous allons simplement remplacer toutes les occurrences du type **`int`** par **`T`**.

Fichier

```
01: template <typename T>
02: class Vecteur {
03:     public:
04:         Vecteur(T x, T y);
05:
06:         void nouvellesCoordonnees(T x, T y);
07:         void recupererCoordonnees(T &x, T &y) const;
08:
09:     private:
10:         T m_x;
11:         T m_y;
12: };
```

On change donc tout autant les types des attributs de la classe, des paramètres de ses méthodes, ainsi que ceux de leurs valeurs de retours (ce qu'il n'y a pas dans notre exemple). Vous trouverez parfois le mot **`class`** au lieu de **`typename`**. Ils sont strictement interchangeables, mais il semble un peu plus logique d'utiliser **`typename`** pour désigner un type que **`class`**.

2.3 Spécialisation de notre classe Vecteur

Si vous vous souvenez bien, dans le jour 3, nous avons utilisé des **`vector`**, ou encore des **`unsorted_map`**. Il s'agissait de conteneurs et ces derniers pouvaient recevoir n'importe quel type, tous les objets qu'ils contenaient étant homogènes (comme **`vector<int> mon_vecteur_d_entiers;`**). La spécialisation se fait en introduisant cet élément de syntaxe consistant à faire suivre le nom de la classe générique par celui du paramètre type ainsi : **`<type>`**. Cette écriture entre chevrons est similaire à celle **`<typename T>`** utilisée lors de la déclaration du modèle. Ainsi, pour spécialiser notre vecteur avec le type **`int`**, tel qu'il était écrit précédemment, nous utiliserons **`Vecteur<int> mon_vecteur(0, 0);`**.

Mais maintenant, nous pouvons aussi spécialiser notre modèle avec de nouveaux types, comme le **`double`** que nous voulions utiliser : **`Vecteur<double> mon_vecteur_double(0.0, 0.0);`**. Notons que, dans notre cas, nous allons surtout utiliser des vecteurs d'entiers. Et nous l'avons déjà fait, puisque nous utilisons déjà abondamment cet objet. Sachez qu'il est possible d'assigner au paramètre du *template* un type par défaut.

Étant donné ces circonstances, et le fait que nous ne souhaitons pas effectuer un *refactoring* de notre code pour utiliser cette nouvelle syntaxe **`Vecteur<int>`**, nous allons préférer assigner **`int`** comme type par défaut à notre *template*, ce qui se fait avec la syntaxe suivante, en début de déclaration : **`template <typename T=int>`**.

2.4 Écrire une classe générique, partie 2

De la même manière que nous avons déclaré une classe spécialisée **`Vecteur<int>`**, nous pouvons désigner notre classe générique **`Vecteur<T>`**. La prise en compte de cette nouvelle syntaxe nous permet de finaliser l'écriture de notre modèle :

Fichier

```
01: template <typename T=int>
02: class Vecteur {
03:     public:
04:         Vecteur(T x, T y);
05:
06:         void nouvellesCoordonnees(T x, T y);
07:         void recupererCoordonnees(T &x, T &y) const;
08:
09:         Vecteur<T>& operator+=(const Vecteur<T> &rhs);
10:         Vecteur<T>& operator-=(const Vecteur<T> &rhs);
11:         Vecteur<T> operator+(const Vecteur<T> &rhs) const;
12:         Vecteur<T> operator-(const Vecteur<T> &rhs) const;
13:
```

```
14:         bool operator==(const Vecteur<T> &rhs) const;
15:         bool operator!=(const Vecteur<T> &rhs) const;
16:
17:     private:
18:         T m_x;
19:         T m_y;
20: };
```

Nous voyons au passage que les règles s'appliquant à **`const`** ne changent pas, on peut toujours l'utiliser et il a toujours la même signification.

2.5 Implémentation des méthodes du modèle de classe

Il existe une contrainte lors de l'usage des modèles de classe : les méthodes du modèle doivent être définies dans les fichiers d'en-têtes (**`.h`** / **`.hpp`**) et non dans un fichier objet (**`.cpp`**). En effet, le modèle seul ne suffit pas. Son code intégral doit être entièrement chargé et analysé par le préprocesseur, sans quoi des références indéfinies à des méthodes du modèle apparaîtront à l'édition de lien.

Lorsque l'on souhaite conserver une séparation claire entre les en-têtes et le code, pratique similaire à l'écriture d'une classe standard, il est courant d'implémenter le code des *templates* dans un fichier contenant une nouvelle extension (**`.tpp`** – pour fichier template C++) et de l'inclure à la fin du fichier d'en-têtes décrivant le modèle.

Outre cette contrainte, les méthodes des modèles sont implémentées à l'identique des méthodes de classe, à l'exception de l'ajout devant chaque méthode de la définition du modèle : **`template <typename T>`** ou **`template<class T>`** . Ainsi le constructeur du modèle **`Vecteur`** s'écrit tout simplement :

Fichier

```
01: template <typename T>
02: Vecteur<T>::Vecteur(T x, T y) : m_x(x), m_y(y) {}
```

2.6 Utilisation de plusieurs paramètres de template

Si vous vous rappelez toujours le jour 3, vous vous souviendrez aussi des associations du type **`map<int, int> mon_association_entier_entier;`**. Il y avait en fait deux paramètres de *template*, ce qui nous donne accès à plus de combinaisons. Nous aussi, nous pouvons créer des modèles à plusieurs paramètres, il suffit de connaître la syntaxe appropriée : **`template <typename T1, typename T2, typename T3, typename T4>`**.

2.7 Fonction générique

Nous avons commencé ce chapitre par la face nord, les classes. Pour les fonctions, les principes restent exactement identiques, mais la complexité est légèrement moindre. Voici, par exemple, une fonction utilisée pour sommer deux entiers :

Fichier

```
01: template <typename T>
02: T somme(const T & a, const T & b) {
03:     return a + b;
04: }
```

Grâce à cette définition de modèle, nous allons pouvoir utiliser la fonction très facilement avec la syntaxe que nous connaissons bien, par exemple pour sommer des entiers de type **int** :

Fichier

```
01: int a = 2, b = 1;
02: cout << "Résultat: " << somme<int>(a, b) << endl;
```

Et on peut même laisser le compilateur trouver tout seul le bon type !

Fichier

```
01: double a = 4, b = 3.1;
02: cout << "Résultat: " << somme(a, b) << endl;
```

Toutefois, il n'est pas possible d'avoir des types différents pour **a** et **b**, car le modèle n'est basé que sur un seul paramètre de *template*. Les deux *paramètres* de la fonction et la *valeur de retour* doivent être du même *type*.

2.8 La spécialisation pour surcharger des fonctions

Les fonctions génériques que nous venons de présenter permettent de réaliser des opérations sans avoir besoin de déclarer différentes fonctions surchargées, dont l'objectif serait de supporter tous les types utilisés dans un programme. Toutefois, il peut se trouver des exceptions pour lesquelles il est nécessaire d'adapter le comportement générique. En effet, dans le cadre de l'utilisation sur des chaînes de caractères, l'opération de somme va concaténer les deux morceaux de chaîne, pour transformer naturellement **"a" + "b"** en **"ab"**, puisque c'est la manière de fonctionner de l'opérateur **+** appliqué aux chaînes de caractères. Nous allons donc spécialiser par nous-mêmes le modèle pour l'adapter au cas des chaînes de caractères :

Fichier

```
01: template <>
02: string somme<string>(const string & a, const string & b) {
03:         return a + "," + b;
04: }
```

On remarque que le mot-clé **typename** et le nom du type a disparu, car il n'est plus paramétrable. À la place, nous avons remplacé toutes les occurrences de ce type générique par le type spécialisé **string**.

3. DÉPLACEMENT ET DÉTECTION DE COLLISION

Après avoir construit un modèle pour notre classe vecteur, nous allons pouvoir l'utiliser pour le déplacement et la détection de collision. La gestion de collision peut être d'une

complexité très variable, en fonction de la précision de la modélisation que l'on souhaite apporter à notre programme. Plus la modélisation se veut réaliste et plus elle fera intervenir des formules provenant de la mécanique des solides (déformables ou non) et de la physique des chocs (élastiques ou non).

Dans notre cas, un casse-briques reste un modèle extrêmement simple, aussi nous allons garder les mécanismes qui le régissent les plus simples possible. Quelle que soit la finesse que nous apportons dans notre moteur, un algorithme de collision ressemble toujours peu ou prou à ceci [2] : on commence par le déplacement des objets (calcul de positions), on recherche des collisions, puis on gère les collisions et enfin on met à jour les objets impactés. Nous allons voir dans la suite comment réaliser les différentes étapes pour notre projet.

3.1 Calcul de position

L'algorithme de déplacement a la charge de manipuler les différents objets mobiles du jeu. Dans un souci de modularité, les objets graphiques ont été dissociés des objets du jeu. Nous avons appelé **`Sprite`** l'objet graphique qui correspond à un objet **`Element`** du jeu. Nous vous avons fait grâce de la traduction française qui aurait demandé à ce que nous traduisions **`Sprite`** en **`Lutin`**. En réalité, on parle plus volontiers de calque. Nous détaillerons dans le jour prochain comment nous avons construit ce **`Sprite`**. Pour travailler avec le moteur de collisions, nous avons simplement besoin de savoir que cet objet doit avoir quelques méthodes et attributs :

- Sa position sur l'aire de jeu, qui est différente de la position logique de la brique. Cette position dépend aussi de la taille de la surface d'affichage (et donc de la résolution de l'écran). On utilisera un **`Vecteur<double>`** pour bénéficier d'une précision inférieure à 1 lors du déplacement des objets.
- Le mouvement de l'objet, qui représente le déplacement effectué par l'objet donné à chaque itération de la boucle principale du jeu. Il est constitué lui aussi d'un **`Vecteur<double>`** pour que nous disposions toujours d'une précision inférieure à 1 lors du calcul du déplacement des objets.
- La taille de l'objet, qui est représentée sous forme d'un vecteur d'entiers. Ce vecteur représente le nombre de points en largeur et en hauteur occupé par l'objet dessiné sur l'écran.

Le calcul du déplacement que nous avons mis en place est très simple. Il consiste à calculer le prochain emplacement que le calque va occuper, suite à quoi nous testerons la case de destination avec le moteur de collision. Cela se réalise efficacement avec le code suivant :

Fichier

```
Vecteur<double> position  = sprite->recupererPosition();
Vecteur<int> prochaine_position = VecteurDoubleVersInt(position +
mouvement);
```

Il faut noter que nous utilisons une conversion pour passer de la somme de la position et du déplacement avec un **`Vecteur<double>`** vers un **`Vecteur<int>`**. En effet, les bibliothèques graphiques 2D usuelles ne fonctionnent qu'avec des nombres entiers, car elles sont conçues pour travailler sur l'unité de mesure qu'est le pixel et que ce dernier est indivisible.

Nous avons donc besoin d'utiliser le type **int** pour indiquer leur position en bout de processus. Cependant, leur position réelle et leur déplacement doivent être des types plus précis que **int**, sans quoi nous verrions des erreurs d'approximation, en particulier lors de déplacements avec un angle très faible.

3.2 Recherche des collisions

3.2.1 Modélisation des objets

Cette étape est la plus complexe de toutes, car nous mettons ici en œuvre les algorithmes physiques sous-jacents. Heureusement pour nous, il existe différents modèles assez classiques que nous pouvons utiliser. Le plus standard étant **AABB** : *Axis Aligned Bounding Box*, autrement dit un rectangle qui va entourer l'objet et qui est aligné sur les axes **X**, et **Y**. Il est parfois plus facile, bien que sa réalisation soit plus difficile, d'utiliser un modèle à base d'**OBB**, pour *Oriented Bounding Box*, c'est-à-dire le même rectangle que précédemment, mais qui n'est pas aligné sur les axes du repère. Pour réaliser des formes plus complexes en 2D, il suffit la plupart du temps de combiner plusieurs AABB ou OBB pour un seul objet.

Nous parlons ici de modélisation, c'est-à-dire que nous cherchons un compromis entre une représentation fidèle de la réalité et être assez haut niveau pour rester dans des contraintes acceptables en temps de développement et temps de calcul. Pour le casse-briques, nous utiliserons un modèle très simple à partir de AABB. Nous utilisons la position du calque ainsi que sa taille pour le définir.

3.2.2 Identification des collisions

Maintenant que nous avons modélisé un objet dans notre moteur de collision, comment allons-nous calculer une collision ? Dans un moteur générique, il est nécessaire de réaliser une boucle sur tous les objets présents avec un simple test d'inclusion dans la AABB que nous avons défini précédemment.

Ici se pose très souvent un gros problème de performances, car itérer sur une liste trop longue d'objets peut très souvent ralentir tout le programme. En effet, l'étude mathématique de cet algorithme nous indique que cette recherche à une complexité algorithmique en **O(n)**, ce qui signifie que le temps de recherche dépend du nombre d'éléments présents dans la liste. Dans un jeu, si le nombre d'objets devient trop grand par rapport aux capacités de l'ordinateur, le nombre d'images par seconde calculé peut tomber assez bas pour que le joueur s'en rendre compte et s'exclame : « l'affichage rame ! ». Il faut donc accorder un soin tout particulier à diminuer au maximum cette liste. Une méthode classique est par exemple de partitionner l'écran en zones et de créer des listes par zone.

La nouvelle classe **Physique2D** va être responsable de la détection des collisions. Cette classe prend la forme suivante :

Fichier

```
01:     class Physique2D : public SpriteObservateur {
02:           public:
03:                Physique2D() {}
04:                virtual ~Physique2D() {}
05:                virtual bool canevas(Sprite *canevas, int min_x, int
min_y, int max_x, int max_y) = 0;
06:                virtual bool collision(const Sprite &sprite, const
Vecteur<int> &v, Sprite **rencontre) const = 0;
```

```
07:                virtual bool ajouter(Sprite &s) = 0;
08:                virtual bool detruire(Sprite &s) = 0;
09:
10:                /* observable */
11:                bool mise_a_jour(SpriteObservable &) override;
12:
13:           protected:
14:                /* dimensions autorisées et objet associé */
15:                int m_x_min;
16:                int m_y_min;
17:                int m_x_max;
18:                int m_y_max;
19:                Sprite *m_canevas;
20:          };
```

Cette classe a de nouveau le profil d'une interface. Nous ne voulons pas implémenter le code de notre algorithme à l'intérieur, car nous voulons pouvoir implémenter différents algorithmes en fonction de nos besoins. Nous voyons tout d'abord qu'elle hérite de **`SpriteObservateur`** et qu'elle implémente la fonction **`mise_a_jour`**, car nous voulons être avertis des changements de position des objets sur le plateau de jeu.

Ensuite il y a 4 méthodes principales :

- **`canevas`** : initialise le moteur de collision avec les dimensions du plateau de jeu et le *sprite* de **`canevas`** ;
- **`collision`** : calcule si un *sprite* entre en collision avec un objet après un déplacement du **`vecteur V`** ;
- **`ajouter`** et **`détruire`** : modifient la liste des objets qui sont suivis par le moteur de collision.

Attardons-nous un instant sur la méthode **`collision`**. La fonction de collision du moteur physique prend en entrée le calque testé (pour connaître sa position et sa taille) ainsi que la prochaine position à laquelle nous voulons le placer. Il nous retourne **`true`** s'il y a une collision, **`false`** sinon.

Le troisième paramètre de la fonction est un pointeur vers un pointeur ! À quoi cela peut-il bien servir ? En réalité, c'est relativement simple : nous avons déjà vu (jour 3, fin de la section 3.1) que lorsque l'on souhaite qu'une fonction puisse modifier un type simple, il faut un pointeur vers le type dans son prototype.

Dans notre cas, c'est la même chose, sauf que l'on ne manipule pas un type simple, mais un pointeur. On va donc avoir besoin de manipuler un pointeur vers un pointeur. De cette manière, si le moteur de collision rencontre un objet, la fonction retourne **`true`** et modifie le contenu du pointeur vers le pointeur **`rencontre`**, ce qui fait que ce pointeur **`rencontre`** sera modifié pour pointer vers l'objet rencontré.

3.2.3 Implémentation simplifiée

Dans notre cas, nous allons prendre le contre-pied de la méthode générique, pour implémenter un autre type d'algorithme, qui est beaucoup plus gourmand en mémoire, mais qui a l'immense avantage d'avoir une complexité en **`O(1)`**, c'est-à-dire que le temps de recherche est constant et ne dépend pas du nombre d'objets. Ce mécanisme consiste à simuler le contenu du plateau du jeu dans le moteur de collision. On associe à chaque point affichable un objet, ou rien s'il n'y a rien à cet endroit. Cela suppose que la surface affichable n'est pas trop grande (sinon cette méthode consommerait trop de mémoire) et que le nombre d'objets mobiles n'est pas très important (sinon le temps de remise à jour

du plateau de jeu dans le moteur de collision serait trop long). Mais cela est bien adapté à notre casse-briques, car nous n'avons que la balle et la raquette qui se déplacent.

Pour simuler l'écran de jeu, nous utilisons un tableau à deux dimensions. Plutôt que de créer de la mémoire dynamiquement, nous utilisons un vecteur de vecteurs : **`std::vector<std::vector<Sprite*>> m_sprites;`**. Ce conteneur stocke des pointeurs vers des **`Sprites`**. Cela nous permet de savoir immédiatement si un objet se trouve à des coordonnées données, de la façon suivante : **`Sprite *rencontre = m_sprites[y][x];`**.

Nous initialisons le conteneur de la manière suivante :

Fichier

```
m_sprites.resize(max_y);
for (int y = 0; y < max_y; y++) {
    m_sprites[y].resize(max_x);
}
```

Les variables **`max_x`** et **`max_y`** étant la taille de l'écran du jeu en nombre de pixels. Ensuite, il suffit d'assigner à chaque coordonnée du tableau le **`Sprite`** présent à l'écran avec la méthode **`ajouter`**. Finalement, il suffit d'appeler la méthode de recherche de **`collision`** pour qu'elle nous retourne l'objet rencontré suite à la tentative de déplacement du *sprite*.

3.2.4 Utilisation du moteur

Pour utiliser notre nouveau système de collision, nous implémentons la fonction suivante, qui réutilise notre calcul de position et qui va tester si un mouvement est réussi ou si un objet est rencontré lors du déplacement :

Fichier

```
template <typename P>
bool testerDeplacement(const P &physique, Sprite *sprite, Vecteur<double>
mouvement, Sprite **rencontre)
{
        /* test si il n'y a pas de collision */
        Vecteur<double> position  = sprite->recupererPosition();

        Vecteur<int> prochaine_position = VecteurDoubleVersInt(position +
mouvement);

        if (physique.collision(*sprite, prochaine_position, rencontre) ==
false) {
                return false;
        }

        /* on a une collision, le pointeur rencontre pointe vers l'objet
rencontré */
        return true;
}
```

Nous avons utilisé un modèle pour abstraire le moteur physique. En effet, nous ne voulons pas lier ce mécanisme de déplacement à un moteur de collision particulier. Il suffit que le moteur possède une fonction **`collision`** et nous pourrons les utiliser pour notre algorithme. Nous retrouvons les mêmes types de paramètres que pour la fonction de collision, c'est-à-dire le **`Sprite`**, le **`Vecteur`** de mouvements à tester et le **`pointeur`** vers l'objet éventuellement rencontré.

3.3 Gestion des collisions

La gestion de la collision consiste à décider de la manière de traiter le choc entre les objets qui ont été identifiés précédemment. Dans notre modèle, nous cherchons simplement à faire rebondir la balle sur l'obstacle rencontré. En effet, nous n'avons pas de déformation de nos objets ni de transfert de l'énergie cinétique. Ce jeu reste centré sur la balle, la raquette et les briques immobiles.

Le moteur de collision va donc s'intéresser principalement au calcul du rebond de la balle. Nous avons simplifié le modèle classique en deux points car :

- nous testons la collision avant le déplacement et arrêtons le déplacement s'il peut produire une collision. En effet, cela signifie que sur l'itération dont le déplacement produit une collision, l'objet ne se déplacera pas. Cela réduit drastiquement les possibles problèmes d'objets qui dépassent les bornes en disparaissant dans le décor, même si cela est un peu moins réaliste. Toutefois, à nos vitesses d'objets, ce comportement ne se remarquera pas.
- nous ne gérons pas non plus la collision au point de contact. Nous considérons que si l'objet en rencontre un autre à la suite de son prochain déplacement, c'est qu'il est déjà en contact avec cet objet. L'objet rebondira donc à partir du point où l'objet se situe, c'est-à-dire proche de l'objet touché, mais pas en contact ! Comme l'approximation faite ci-dessus, cela est acceptable, car nous utilisons de faibles vitesses de déplacement et l'objet sera très proche de son point de contact lorsqu'il rebondira, mais cela pourrait ne plus être le cas à de plus fortes vitesses : l'objet rebondirait bien avant de toucher l'objet !

Ces deux hypothèses posées, nous constatons que lors d'un déplacement de la balle, il est possible de changer les coordonnées **`x`** ou **`y`**, selon la surface sur laquelle on rebondit. En effet, lors d'un rebond, la trajectoire après rebond est symétrique à celle avant rebond par rapport à la normale à la surface rencontrée. Or, il se trouve que dans notre jeu, outre la balle, nous n'avons que des surfaces verticales ou horizontales. Par conséquent, les normales aux surfaces seront respectivement horizontales ou verticales. Autrement dit, seule l'une des deux coordonnées du déplacement sera inversée tandis que l'autre restera inchangée. Pour réaliser ceci, on ne travaille plus directement avec le vecteur de déplacement initial, mais avec ses projections sur les axes **`x`** et **`y`**.

De la même manière que pour la méthode **`testerDeplacement`**, nous allons utiliser un modèle pour implémenter le corps de la fonction **`faireDeplacement`**, mais cette fois-ci un peu plus élargi, car le type **`Sprite`** a été rendu générique :

Fichier

```
template <typename P, typename S>
Vecteur<double> faireDeplacement(const P &physique, S *sprite,
std::list<Sprite*> &collisions)
```

Le prototype du modèle abstrait le moteur physique et le *sprite*. Cela nous permettra de spécialiser le modèle plus tard pour gérer des cas particuliers. La fonction retourne le nouveau vecteur de mouvement pour le **`Sprite`** déplacé. S'il y a eu des collisions, les objets rencontrés sont ajoutés dans la liste **`collisions`** fournie en argument. Ensuite, le système se comporte comme nous l'avons décrit, en testant les coordonnées une par une :

Fichier

```
01: {
02:         S *rencontre;
03:         Vecteur<double> mouvement = sprite->recupererMouvement();
04:
```

```
05:         /* si on a une collision, il faut inverser le mouvement */
06:         /* il faut tester quelle coordonnée a généré le rebond : X ou
Y */
07:         double mv_x, mv_y;
08:         mouvement.recupererCoordonnees(mv_x, mv_y);
09:
10:         /* on teste d'abord X */
11:         Vecteur<double> deplacement = Vecteur<double>(mv_x, 0);
12:         if (testerDeplacement(physique, sprite, deplacement,
&rencontre) == false) {
13:                 sprite->deplacer(deplacement);
14:         } else {
15:                 /* on inverse le mouvement et on ajoute l'objet dans
la liste de collisions */
16:                 mv_x = -mv_x;
17:                 collisions.push_back(rencontre);
18:         }
19:
20:         /* puis Y */
21:         ...
22:
23:         return Vecteur<double>(mv_x, mv_y);
24: }
```

Enfin, le code du moteur principal s'intéresse à la liste des objets rencontrés. Il va vérifier chaque collision pour savoir si une brique a été effectivement détruite, ce que nous verrons plus tard dans l'article suivant.

CONCLUSION

Vous venez de découvrir la dernière grande brique d'importance de C++, les *templates*. Il vous faudra probablement un tout petit peu de temps pour les maîtriser totalement, mais une fois que vous les aurez implémentés dans quelques situations concrètes, vous ne pourrez plus vous en passer.

Ils sont un outil indispensable qui permet au langage à typage statique qu'est C++ de bénéficier de certains avantages qu'ont les langages à typage dynamique qui peuvent proposer des fonctions ou des classes pouvant utiliser n'importe quel type, mais par des recettes qui respectent toujours ce côté statique et la sécurité de programmation qui va avec.

Vous avez également pu voir un cas concret d'utilisation de ces *templates* dans le moteur de collision ainsi que quelques algorithmes simples, mais très orientés.

Cet article a également servi à terminer de vous donner les clés indispensables à la maîtrise de la bibliothèque de C++, vous saurez maintenant utiliser toutes les fonctions qu'elle propose et vous saurez tirer profit des exemples de la documentation que vous trouverez en ligne et les appliquer rapidement.

Le dernier article de ce hors-série va mettre tout ceci à profit et se focaliser plus sur les algorithmes spécifiques au jeu. ■

RÉFÉRENCES

[1] Liste d'exceptions : **http://www.cplusplus.com/reference/exception/exception/**.

[2] Tutoriel sur la conception d'un moteur physique : **http://gregorycorgie.developpez.com/tutoriels/physic/**

JOUR 6

JOUR 6

FINALISEZ VOTRE PROGRAMME

Nous arrivons maintenant dans la dernière ligne droite du projet : l'essentiel des classes du jeu est prêt, il ne nous manque plus que la capacité d'afficher les objets et de terminer l'implémentation de la classe principale du jeu, celle qui va orchestrer l'ensemble.

Pour ce faire, nous allons poursuivre notre quête consistant à respecter les grands principes de l'objet en C++ afin de garder un code modulaire et évolutif (ce qui pourrait nous permettre de faire appel à différents moteurs de rendu sans effort ou encore à réutiliser nos classes dans d'autres objets).

1. CRÉATION DE L'IHM

Avant de finir par la classe **Jeu**, nous devons définir la façon dont nous allons contrôler l'écran pour afficher le jeu. Dans un premier temps, nous allons définir des interfaces, afin que le jeu ne soit pas lié à un seul moteur de rendu : l'idée est de proposer des contrats qui pourront être remplis d'autres manières que les choix que nous ferons. En effet, comme tout ce que nous faisons en C++, nous voulons le rendre le plus générique possible pour permettre un maximum de réutilisabilité et d'évolutivité. Cela constitue évidemment un travail itératif en apportant de plus en plus d'abstractions à chaque étape.

1.1 Classe Écran

Nous allons donc commencer par écrire une classe **Ecran** qui sera une interface et qui définira ce que nous devons mettre en œuvre pour afficher le contenu du jeu.

Fichier

```
01: enum class Action { DROITE, GAUCHE, QUITTER, DEMARRER, AUCUNE };
02:
03: class Ecran {
04:     public:
05:         Ecran() {};
06:         virtual ...cran() {};
07:         virtual void effacer() = 0;
08:         virtual void rafraichir() = 0;
09:         virtual Action lireAction() = 0;
10:         virtual bool recupererDimensions(int &x, int &y) const = 0;
11:         virtual bool ajouterPixel(const Vecteur<int> &v, char pix,
Couleur c) = 0;
12: };
```

À travers cette classe requise par notre jeu, nous définissons les actions possibles suivantes : effacer et afficher le contenu de l'écran, récupérer les entrées du clavier converties à notre besoin, obtenir les dimensions de l'écran utilisable, nécessaire à la définition du canevas, dessiner sur une case/pixel de l'écran. Ainsi, cette interface va servir de support pour l'affichage des objets qui composent notre jeu, même si le contenu n'est pas encore implémenté (nous le ferons juste après avoir défini les autres interfaces et ainsi défini tous nos contrats).

La définition de l'écran est le socle autour duquel va s'articuler les différentes autres classes du projet : l'algorithme du jeu et les objets d'affichage, aussi nommés **Sprite**.

1.2 Définition d'un objet affichable

Après notre écran, nous allons maintenant nous attaquer aux différents éléments graphiques du projet. Nous avons déjà évoqué la classe **Sprite** dans le jour précédent, lorsque nous avons traité le moteur de collisions. Voici maintenant le détail de cette classe :

Fichier

```
01: class Sprite {
02:   public:
03:     Sprite(const Vecteur<double> &p, const Vecteur<int> &t, Element
&e) :
04:       m_position(p), m_mouvement(Vecteur<double>(0,0)),
```

```
05:       m_taille(t), m_element(e) {}
06:     virtual ~Sprite() {}
07:
08:     /* taille du sprite */
09:     void nouvelleTaille(const Vecteur<int> &taille) { m_taille =
taille; }
10:     Vecteur<int> recupererTaille() const { return m_taille; }
11:
12:     /* mouvement du sprite */
13:     void nouveauMouvement(const Vecteur<double> &mouvement)
{ m_mouvement = mouvement; }
14:     Vecteur<double> recupererMouvement() const { return m_mouvement; }
15:
16:     /* position */
17:     Vecteur<double> recupererPosition() const { return m_position; }
18:     /* observé */s
19:     virtual void nouvellePosition(const Vecteur<double> &position)
{ m_position = position; }
20:     virtual void deplacer(const Vecteur<double> &mouvement)
{ m_position += mouvement; }
21:
22:     /* affichage, dépend de l'écran */
23:     virtual bool afficher(Ecran &) = 0;
24:
25:     /* recupération de l'objet du jeu */
26:     Element &element() { return m_element; }
27:
28:   protected:
29:     Vecteur<double> m_position;
30:     Vecteur<int> m_taille;
31:     Vecteur<double> m_mouvement;
32:     Element &m_element;
33: };
```

Cette classe est utilisée pour manipuler des objets graphiques, avec des caractéristiques différentes en fonction de l'élément qu'elles doivent représenter.

Le **Sprite** a donc 4 attributs (dont certains ont déjà été décrits lors du jour précédent) : sa position sur l'aire de jeu (qui est différente de la position logique de la brique), le mouvement de l'objet qui représente le déplacement effectué par l'objet donné à chaque itération de la boucle principale du jeu, la taille de l'objet qui est représentée sous forme d'un vecteur d'entiers (ce vecteur représente le nombre de pixels en largeur et en hauteur occupé par l'objet dessiné sur l'écran), et la référence vers l'objet **Element** qui est représenté par ce **Sprite**.

Au niveau des méthodes, on a tout ce dont nous avons besoin pour accéder aux données de position, mouvement et taille, plus l'élément original. Nous n'avons que quatre méthodes virtuelles, dont une pure, pour des besoins particuliers.

1.3 Définition des calques observés

La classe **SpriteObserve** va hériter de la classe **Sprite**, car il s'agit de la brique élémentaire que nous allons spécialiser. Elle va aussi implémenter l'interface **Observable**, pour que ce type de calque puisse être suivi par un **SpriteObservateur**.

Fichier

```
01: /* SpriteObserve est utilisé pour les objets suivis  */
02: class SpriteObserve : public Sprite, public SpriteObservable {
03:   public:
04:     SpriteObserve(SpriteObservateur &o,
```

```
05:       const Vecteur<double> &p,
06:       const Vecteur<int> &t,
07:       Element &r) : Sprite(p, t, r), SpriteObservable(o, *this) {}
08:
09:     /* observable */
10:     void nouvellePosition(const Vecteur<double> &position) override {
11:       m_position = position;
12:       notify();
13:     }
14:
15:     void deplacer(const Vecteur<double> &mouvement) override {
16:       m_position += mouvement;
17:       notify();
18:     }
19: };
```

Le constructeur doit appeler tous les constructeurs, de la classe mère dont il hérite et de l'interface qu'il implémente, implicitement ou explicitement. Cette classe surcharge les deux méthodes virtuelles de la classe **Sprite**, afin d'appeler la méthode **notify** de la classe **SpriteObservable** lorsque les coordonnées du **Sprite** changent. Ces deux classes finalisent la définition générique de l'écran et d'un **Sprite**. Nous allons poursuivre par l'écriture du cœur du jeu.

2. CLASSE JEU

La classe **Jeu** est la classe de plus haut niveau de l'application. Elle manipule les différents composants que nous avons vus tout au long de ce projet. Elle a très peu de méthodes **public** :

Fichier

```
01: class Jeu {
02:     public:
03:         Jeu(Ecran &ecran, Physique2D &physique) : m_ecran(ecran), m_
physique(physique), m_jeu_en_cours(false) {}
04:         bool demarrer(Niveau &niveau);
05:         bool afficher();
06:         bool executer(Action action);
```

Ces méthodes servent à contrôler l'ensemble du jeu à haut niveau. La méthode **démarrer** va charger un nouveau niveau, la méthode **afficher** va dessiner ce qui doit être présenté à l'écran et la fonction **exécuter** va prendre en compte les entrées clavier pour modifier les variables du jeu. Ces fonctions sont exécutées dès le démarrage de l'application, dans la fonction **main** suivante.

Au niveau des attributs privés, la classe **Jeu** contient :

Fichier

```
    private:
       Balle m_balle;
       Canevas m_canevas;
       Raquette m_raquette;

       Sprite *m_sprite_balle;
       Sprite *m_sprite_canevas;
       SpriteObserve *m_sprite_raquette;

       std::list<Sprite*> m_sprites;
```

Nous voyons que le jeu crée la balle, le canevas et la raquette. Les briques ne sont pas présentes ici, car elles sont définies dans la classe **Niveau**, pour que nous puissions faire varier leur type et leur disposition pour chaque niveau. D'autre part, les calques sont aussi référencés dans la classe **Jeu**. Nous retrouvons ceux de la balle, du canevas et de la raquette ainsi qu'une liste de calques destinée à contenir ceux créés pour chaque brique. Étant donné que leur nombre est variable d'un niveau à l'autre et n'est pas connu à l'avance, il est nécessaire d'utiliser un conteneur pour nous permettre d'avoir la souplesse nécessaire pour manipuler ces éléments créés dynamiquement.

En ce qui concerne le choix d'avoir utilisé une liste, il est déterminé par le fait qu'il nous faut une séquence et non une association. On doit pouvoir itérer sur les éléments dans l'ordre (donc pas de pile ou de file) et on doit pouvoir ajouter les briques à la fin rapidement. Pour ce faire, on a le choix entre un vecteur, une liste ou un « deque ». Le choix s'est porté sur la liste, car elle ne réalise pas de réallocation lors de l'ajout d'un seul élément et elle se comporte mieux avec des éléments de grande taille (> 128 bits), cependant un vecteur ou un deque aurait aussi bien pu faire l'affaire.

Nous allons maintenant détailler quelques unes des principales fonctions du jeu.

2.2 Affichage des calques

La fonction **afficher** utilise massivement les objets **Ecran** et **Sprite**. Elle utilise un mécanisme effacer/rafraîchir et elle demande à chaque **Sprite** de se dessiner à l'écran :

Fichier

```
01: bool Jeu::afficher()
02: {
03:     m_ecran.effacer();
04:
05:     for (Sprite *s : m_sprites) {
06:         s->afficher(m_ecran);
07:     }
08:
09:     m_sprite_balle->afficher(m_ecran);
10:     m_sprite_raquette->afficher(m_ecran);
11:     m_sprite_canevas->afficher(m_ecran);
12:
13:     m_ecran.rafraichir();
14:
15:     return true;
16: }
```

Cela permet à chaque objet de se gérer lui-même et de respecter ainsi l'encapsulation : ce qui concerne un objet est décrit dans sa classe. Ce n'est pas au jeu d'aller afficher une balle, mais le jeu peut demander à la balle de s'afficher et cette dernière explique comment elle s'y prend pour le faire.

2.3 Exécuter les actions

La méthode **exécuter** met à jour les états du jeu, en fonction des entrées du clavier. Elle est effectuée à intervalles réguliers et décide du démarrage et de l'arrêt du jeu.

Nous trouverons donc deux parties dans cette fonction : le traitement de l'action reçue et la mise à jour périodique des objets. Voici la première partie :

Fichier

```
01: bool Jeu::executer(Action action)
02: {
03:   std::list<Sprite*> collisions;
04:   Vecteur<double> mouvement;
05:    /* on exécute l'action qu'on a trouvé pendant la période */
06:    switch (action) {
07:       case Action::DROITE:
08:           m_sprite_raquette->nouveauMouvement(Vecteur<double>(1,0));
09:           mouvement = faireDeplacement(m_physique, m_sprite_raquette,
collisions);
10:           m_sprite_raquette->nouveauMouvement(mouvement);
11:           break;
12:
13:     case Action::GAUCHE:
14:           m_sprite_raquette->nouveauMouvement(Vecteur<double>(-1,0));
15:           mouvement = faireDeplacement(m_physique, m_sprite_raquette,
collisions);
16:           m_sprite_raquette->nouveauMouvement(mouvement);
17:           break;
18:
19:     case Action::DEMARRER:
20:           /* lance la balle : démarre le jeu */
21:           m_jeu_en_cours = true;
22:           break;
23:
24:     case Action::QUITTER:
25:           /* quitte le jeu */
26:           return false;
27:
28:     case Action::AUCUNE:
29:           break;
30:         }
31:
```

Cette première partie est constituée d'un simple **switch** sur la liste des différentes actions possibles. Les actions **DROITE** et **GAUCHE** initient le déplacement de la raquette avec un vecteur mouvement dépendant du sens. La barre d'espacement démarre le jeu en mettant à jour une variable de classe et l'action **QUITTER** retourne **false** pour arrêter la boucle principale et ainsi quitter l'application. La deuxième partie de la fonction déplace périodiquement la balle, à partir du moment où le jeu a été démarré :

Fichier

```
32:     /* on déplace la balle à chaque itération */
33:     if (m_jeu_en_cours) {
34:       Vecteur<double> mouvement = faireDeplacement(m_physique,
m_sprite_balle, collisions);
35:       m_sprite_balle->nouveauMouvement(mouvement);
36:
37:       if (collisions.size()) {
38:         /* on a eu une collision */
39:         for (auto sprite : collisions) {
40:           collision(sprite);
41:         }
42:
43:         if (m_niveau->estTermine() == true) {
44:           return false;
45:         }
46:       }
47:     }
48:
49:     return true;
50: }
```

2.4 Déplacement de la balle et collision

Comme nous l'avons vu dans le code ci-dessus, nous utilisons la méthode **faireDeplacement** aux lignes 9, 15 et 34 pour réaliser le déplacement de la raquette et de la balle. La liste des collisions est remplie par cette fonction lors de l'exécution sur le calque de la balle avec les objets qu'elle a pu rencontrer : brique, canevas ou raquette. Ensuite, la méthode **collision** de la classe **Jeu** est exécutée à la ligne 40 sur chaque élément de la liste pour résoudre la collision. Pour gérer la différence de traitement entre les différents types d'objets, nous utilisons simplement le mécanisme des fonctions virtuelles vu en jour 4 : nous demandons à l'élément rencontré s'il se détruit ou non. Pour cela, nous appelons la méthode virtuelle **collision** de cet objet.

Fichier

```
bool Jeu::collision(Sprite *rencontre)
{
    switch (rencontre->element().collision()) {
        case ObstacleResultat::INTACT:
            return false;

        case ObstacleResultat::DETRUIT:
            /* l'objet doit être détruit */
            m_sprites.remove(rencontre); /* de la liste des choses
affichées */
            m_physique.detruire(*rencontre); /* du moteur de collision */
            m_niveau->detruire(rencontre->element()); /* maj du nombre de
briques restantes */
            return true;
    }
    return false;
}
```

CONCLUSION

Ainsi se conclut ce hors-série sur le langage C++. Nous vous invitons bien entendu à lire le code qui est fourni avec (et qui contient une interface Curses) et à le tester (en jouant, ce qui n'est pas la pire chose que l'on puisse faire), mais surtout à le modifier, de plusieurs manières possibles. En effet, vous pouvez créer vos propres niveaux, tester plusieurs raquettes ou plusieurs balles, faire varier leurs tailles, avoir des canevas tordus, des briques disposées de diverses manières, plus ou moins cachées derrière des briques incassables, etc.

À ce stade, nous avons déjà vu beaucoup de concepts du C++ et vous devriez maîtriser les fondamentaux du langage. Ceci étant dit, la force de C++ réside aussi dans le fait qu'il existe de magnifiques bibliothèques l'enrichissant. Aussi, nous reviendrons sans doute pour présenter de nouvelles choses et vous permettre de vous amuser avec C++ en faisant des programmes d'un niveau professionnel (pour l'instant, nous n'avons fait qu'un programme terminal, même s'il est relativement avancé).

Entraînez-vous donc et à bientôt ! ▮

SYNTAXE

Directives pré-processeur

`#include <bibliotheque>`	Inclusion d'un entête standard C++
`#include <bibliotheque.h>`	Inclusion d'un entête standard C
`#include "bibliotheque"`	Inclusion d'un entête local
`#pragma`	Permet de spécifier une option pour le compilateur
`#pragma once`	Permet de spécifier que le fichier entête doit être lu une seule fois

Chaînes de caractères

Création	`string s1, s2 = "hello", s3 {"world"}`	`O(n)`
Taille	`s1.size(), s2.size(), s3.size()  // 0, 5, 5`	`O(1)`
Concaténation	`s1 += s2 + ' ' + s3 + "!"  // "hello world!"`	`O(n)`
Accès à un caractère	`s1[0], s2.at(0), s3.front(), s3.back()  // h, h, w, d`	`O(1)`
Extraction d'une sous-chaîne	`s1.substr(debut, taille)`	`O(n)`
Recherche	`s1.find(chaine), s2.find(caractere)`	`O(n)`
Recherche depuis la fin	`s1.rfind(chaine), s2.rfind(caractere)`	`O(n)`
Recherche de plusieurs caractères	`s1.find_first_of(chaine_liste_blanche)`	`O(n)`
Recherche d'autres caractères	`s1.find_first_not_of(chaine_liste_noire)`	`O(n)`
Remplacement de caractères	`s1.replace(position_depart, longueur_a_enlever, s2)`	`O(n)`

Déclarations et classes de stockage

`int a`	N'existe que dans le bloc dans lequel elle est déclarée Supprimée à la fin de chaque exécution de ce bloc
`static int a`	N'existe que dans le bloc dans lequel elle est déclarée Conservée en l'état à la fin de chaque exécution de ce bloc
`extern int a`	Variable déclarée dans un autre fichier cpp
`const a`	Variable constante, doit être initialisée et déclarée en même temps
`Int* p=&a`	La variable p est un pointeur vers l'entier a
`int& r=a`	La variable r est une référence (alias) de l'entier a
`const int *p=&a`	Le contenu pointé a est constant
`int* const p=&a`	Le pointeur p (mais non le contenu a) est constant
`const int* const p=&a`	Le pointeur p et le contenu a sont constants
`const int& r=a`	Il n'est pas possible de faire des modifications en utilisant r Cela reste cependant possible en utilisant a, selon sa déclaration

Énumérations

Fichier

```
enum nom_enum {
    NOM_CONTENU_1,      // vaudra 0
    NOM_CONTENU_2 = 3, // vaudra 3
    NOM_CONTENU_3,      // vaudra 4
};
enum class NOM_ENUM {};
```

Fonctions

Fichier

```
type_retour nom_fonction(type_1 a,  // argument passé par valeur
                         type_2 &b)  // argument passé par référence
{
    // Bloc de la fonction
    return value;  // Renvoi du résultat (de type type_retour)
}
```

Classe

Fichier

```
class NomDeLaClasse {
    private:
        // Seules les méthodes de la classe y accèdent
        type_attribut attribut_private;
    protected:
        // Seules les méthodes de la classe et des classes dérivées y
accèdent
        type_attribut attribut_protected;
        type_retour methode1(type_1 arg1, type_2 arg2);
    public:
        // Tout le monde y accède
        // Constructeur
        NomDeLaClasse(type_attribut argument) : attribut_private= argument
{};
        // Destructeur
        ~NomDeLaClasse();
        // Méthodes publiques visibles de l'extérieur
        type_retour methode2(type_1 arg1, type_2 arg2);
        void attribut_setter(type_attribut arg);
        type_attribut attribut_getter() const; // ne modifie pas l'objet
};
// Classe d'héritage public
class Derivee1 : public NomDeLaClasse {}
// Classe d'héritage private
class Derivee2 : private NomDeLaClasse {}
```

`MaClasse::MaClasse()`	Constructeur
`MaClasse(type1 val1) : attr1(val1) {}`	Liste d'initialisations du constructeur
`MaClasse::~MaClasse()`	Destructeur
`type MaClasse::Getter() const;`	Méthode ne modifiant pas l'objet
`virtual type MaClasse::Methode() { }`	Autorise la surcharge de la fonction par une méthode d'une classe fille
`virtual type MaClasse::Methode() = 0;`	Méthode virtuelle pure (empêche la classe d'être instanciée). Une classe fille peut l'être si elle l'implémente.
`type MaClasse::Methode() override;`	Oblige cette méthode à surcharger une fonction virtuelle d'une classe mère
`Class Fille : public Mere {... };`	Déclare une classe fille héritant d'une classe mère avec un héritage public
`private:`	Seules les méthodes de la classe y accèdent
`protected:`	Seules les méthodes de la classe et des classes dérivées y accèdent
`public:`	Tout le monde y accède
`*this`	Objet courant
`Mere &objet = Fille();`	Upcast, polymorphisme : objet fille dans une référence mère
`MaClasse& MaClasse::operator+ (const MaClasse &)`	Méthode surchargeant l'opérateur + pour les objets de la classe

Template

Modèle de fonction

Fichier

```
template <typename T1, typename T2>
T1 maFonction(T2);
```

Modèle de classe

Fichier

```
template <typename T1, typename T2>
class MaClasse {
 public:
  T1 maMethode(T2);
}
```

Bloc conditionnel

Fichier

```
if (condition1) { ... }
[elif (condition2) { ... }, ...]
[else { ... }]
```

Branchement

Fichier

```
switch(variable)
{
    case valeur: { ... }
    [case auwtre_valeur: { ... } , ...]
    [default: { ... }]
}
```

Bloc itératif indénombrable

Fichier

```
while (condition)
{                    // ce qui suit fonctionne aussi pour for
    [...] break;       // permet de sortir de la boucle
    [...] continue;    // permet de passer à l'itération suivante
}
while(true) { ... }    // boucle infinie (doit être cassée)
```

Bloc itératif dénombrable

Fichier

```
for (initialisation; condition_de_fin; commande_de_fin_d_iteration) { ...
}
for (int i=0; i < nombre_de_boucles; i++) { ... }
for (type_iter iter=conteneur.begin(); iter=conteneur.end(); iter++) { ...
}
```

CONTENEURS

Vecteurs

Création	`vector<int> v1, v2 = {0, 1, 2, 3, 4}`	`O(1), O(n)`
Taille	`v1.size(), v2.size()  // 0, 5`	`O(1)`
Redimensionnement	`v1.resize(), v2.shrink_to_fit`	`O(n)`
Accès à un élément	`v2[0], v2.at(1), v2.front(), v2.back()  // 0, 0, 0, 4`	`O(1)`
Ajout d'un élément à la fin	`v1.push_back(5)`	`O(1)`
Suppression du dernier élément	`v1.pop_back()`	`O(1)`
Insertion d'un (ou +) élément(s)	`v1.insert(it, 6), v2.insert(v1.begin(), v1.end())`	`O(n)`
Suppression d'éléments	`v1.erase(v1.begin()+1), v2.erase(v2.begin()+2, v2.end()-2)`	`O(n)`
Ré-allocation	`// Automatique lorsque la capacité doit changer`	`O(n)`

Liste et Deque (double ended queue)

Ajout d'un élément au début	`d.push_front(42)`	`O(1)`
Ajout d'un élément à la fin	`d.push_back(42)`	`O(1)`
Ajout d'un élément n'importe où	`d.insert(it, 6), d.insert(d.begin(), d.end())`	`O(n)`
Suppression d'un élément au début	`d.pop_front()`	`O(1)`
Suppression d'un élément à la fin	`d.pop_back()`	`O(1)`
Suppression d'un élément n'importe où	`d.erase(d.begin()+1), d.erase(d.begin()+2, d.end()-2)`	`O(n)`

Pile (stack) ou file

Ajout d'un élément	`p.push(42)  // Equivalent de deque.push_back`	`O(1)`
Suppression d'un élément	`p.pop()  // Equivalent de deque.pop_back`	`O(1)`

File (queue)

Ajout d'un élément	`f.push_back(42)  // Equivalent de deque.push_back`	`O(1)`
Suppression d'un élément	`f.pop_front()  // Equivalent de deque.pop_front`	`O(1)`

Associations

Création	`map<int, int> m`	`O(1),` `O(n)`
Taille	`m.size()`	`O(1)`
Accès à un élément	`m[0], m.at(1)`	`O(ln(n))`
Ajout d'un élément	`m.insert(std::pair<int, int>(4, 16))` `m.insert(it, std::pair<int, int>(4, 16))`	`O(ln(n))` `O(1)`

Suppression d'un élément	`m.erase(it)`
	`m.erase(42)`
	`m.erase(it1, it2)`
		`O(1)`
		`O(ln(n))`
		`O(n)`

Ensembles

Création	`set<int, int> s`	`O(1),` `O(n)`
Taille	`s.size()`	`O(1)`
Ajout d'un élément	`s.insert(42)` `s.insert(it, 42)`	`O(ln(n))` `O(1)`

Suppression d'un élément	`s.erase(it)`
	`s.erase(42)`
	`s.erase(it1, it2)`
		`O(1)`
		`O(ln(n))`
		`O(n)`

VISITEZ NOTRE BOUTIQUE ET DÉCOUVREZ NOS GUIDES !

LINUX MAGAZINE / FRANCE

GNU LINUX PRATIQUE — COMPRENDRE, UTILISER ET ADMINISTRER LINUX — ESSENTIEL

MISC — Multi-System & Internet Security Cookbook

HACKABLE MAGAZINE

Open Silicium

ET VOUS ?
COMMENT LISEZ-VOUS VOS MAGAZINES PRÉFÉRÉS ?

« Moi, je les lis en version PAPIER ! »

« Moi, je les lis en version PDF ! »

« Moi, je consulte la BASE DOCUMENTAIRE ! »

RENDEZ-VOUS SUR **www.ed-diamond.com** POUR DÉCOUVRIR TOUTES LES MANIÈRES DE LIRE VOS MAGAZINES PRÉFÉRÉS !

AVANT DE COMMENCER

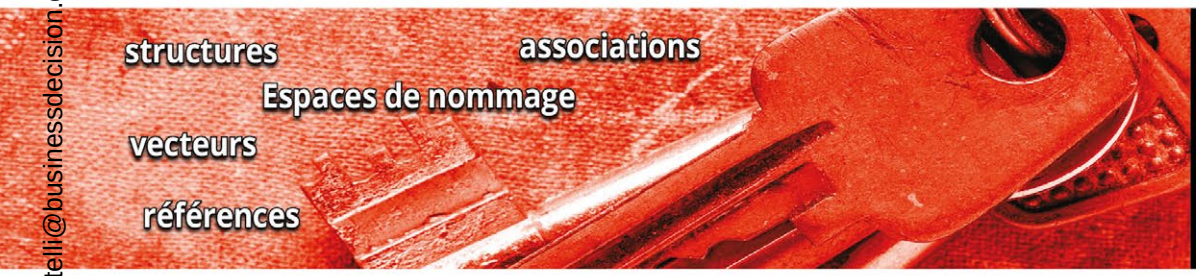
Une rapide présentation du langage C++

JOUR 1 :

Installez vos outils et découvrez les bases du C++

JOUR 2 :

Débutez votre projet



JOUR 3 :

Abordez les types avancés et les pointeurs

JOUR 4 :

Modélisez des objets

JOUR 5 :

Écrivez des classes génériques

JOUR 6 :

Finalisez votre programme