Travaux pratiques

1. Utilisation de l'héritage de classes dans l'interprète tiny-lisp

Dans l'interprète tiny-lisp, des classes dérivant de `ScriptBox` sont chargées de faire fonctionner le parser dans autant d'environnements :

- La boîte à sable ou `Sandbox` est un environnement isolé où les entrées-sorties sont neutralisées (elles ne sont pas activées).
- L'environnement `ConsoleBox` assure des entrées-sorties sur la console système.

La classe de base est appelée `ScriptBox` et elle définit une méthode virtuelle `init_events()` :

```
/*
    Environnement d'exécution générique
*/
class ScriptBox
{
public:
    LexScan*lexScan;
    Evaluator*parser;

    ScriptBox();
    ~ScriptBox();

    virtual void init_events();

    void new_text();
    void set_text(string text);
    void parse();
    bool has_errors();
    string get_errors();
    string get_text();
```

```
        bool set_debug();
};
```

Dans la version neutre, `Sandbox`, cette méthode n'est pas surchargée.

```
/*
        Environnement d'exécution neutre
*/
class Sandbox :
        public ScriptBox
{
public:
        Sandbox();
        ~Sandbox();
};
```

Dans le cas de l'environnement `Consolebox`, la méthode est au contraire surchargée pour déclarer des événements de type entrées-sorties sur la console (écran, clavier).

```
/*
        Environnement d'exécution Console
*/
class Consolebox :
        public ScriptBox
{
public:
        Consolebox();
        ~Consolebox();

        virtual void init_events();
};
```

Voyons maintenant comment la fonction `main()` initialise les instances de ces environnements d'exécution :

```
int main()
{
      sandbox = new Sandbox;
      sandbox->init_events();

      consoleBox = new Consolebox;
      consoleBox->init_events();

      show_hello();
      show_prompt(LIGNE_NUM::RESET);
      eval_loop();
  return 0;
}
```

La méthode virtuelle `init_events()` étant surchargée dans la classe `Consolebox`, c'est elle qui est appelée pour l'instance `consoleBox`.

2. Des pointeurs de membres pour des fonctions callback

Passons à présent à la méthode `init_events()` ; elle réalise l'association entre l'analyseur syntaxique (le parseur) et l'environnement d'exécution. Le parseur interagit en deux occasions : l'affichage à l'écran et la lecture de données du clavier.

Cependant, le code du parseur doit rester aussi neutre et générique que possible, aucune référence explicite à des classes d'exécution ne doit être effectuée. C'est là que les pointeurs de membres entrent en jeu.

Considérons la classe `GenericDelegate` qui définit les modalités d'appel d'une méthode prenant deux pointeurs `void*` comme paramètres et renvoyant `void` (en fait, le premier paramètre représente l'objet d'entrée, le second l'objet résultat). La méthode `do_handler` est virtuelle pure, puisque la classe est totalement générique (on ne peut définir aucune implémentation par défaut).

```
/*
    Définit un 'contrat' pour appeler une méthode générique
*/
class GenericDelegate
{
public:
    // méthode à appeler, évidemment virtuelle pure
    virtual void do_handler(void*inp, void*outp) = 0;

    // type pointeur de méthode vers une fonction
    // ayant la même signature
    typedef void (GenericDelegate::*pdo_handler)(void*, void*);
};
```

La classe `GenericEvent` réalise l'association entre le demandeur et le gestionnaire :

```
/*
   Association entre un événement (le demandeur d'un service) et
   un délégué qui rend le service (le gestionnaire de l'événement).
*/
class GenericEvent
{
public:
    // delegué
    GenericDelegate * rhandler;      // instance
    GenericDelegate::pdo_handler doo_handler; // méthode gestionnaire

    // méthodes d'association du gestionnaire
    void AddHandler(GenericDelegate *handlerInstance,
      GenericDelegate::pdo_handler handlerMethod);

    // déclenche l'événement => appelle le gestionnaire
    void RaiseEvent(void*inp, void*outp);
};
```

Voyons maintenant comment la classe `Procs` déclare un événement `OnReadLine` :

```
#pragma region Procs : primitives intégrées
```

```
class Procs
{
public:
  static GenericEvent OnReadLine;
}
```

Un peu plus loin, l'analyseur souhaite recourir aux services de `OnReadLine`, il ne se préoccupe pas de l'éventuelle implémentation ni de la classe qui porte le code du gestionnaire :

```
#pragma region read_line
Variant Procs::read_line(const variants& c)
{
  // retourner vide si aucun gestionnaire n'est prévu
  if (OnReadLine.rhandler == nullptr)
    return Variant();

  // déclenche l'événement en passant data en paramètre de sortie
  std::string data;
  OnReadLine.RaiseEvent(nullptr, &data);

  // retour dans le flux d'évaluation
  return Variant(Chaine, data);
}

#pragma endregion
```

C'est à la méthode virtuelle `ConsoleBox::init_event()` que revient la mission de définir le gestionnaire de l'événement, c'est-à-dire la méthode qui sera appelée au déclenchement de l'événement `OnReadLine`.

```
/**
 init_events()
 appelé pour définir les gestionnaires d'évenements de ConsoleBox
*/
void Consolebox::init_events()
{
```

```
  // TODO:
  // Ajouter ici les gestionnaires d'événements pris en charge
par la ScriptBox
  // Evenement.AddHandler(handlerInstance,handlerMethod)

  Procs::OnReadLine.AddHandler(
      FxConsole::get_instance()->lire_handler /* handlerInstance */,
      &GenericDelegate::do_handler /* handlerMethod */ );
}
```

Par simplicité, `get_instance()` est un singleton de la classe `FxConsole` (une instance unique). La classe `FxConsole` porte un champ `lire_handler` de type `LireHandler*`. Cet objet comporte la méthode `do_handler()` elle-même appelée lorsque l'événement se déclenche.

```
/**
*  Fournit les E/S console
*/
class FxConsole
{
private:
  static FxConsole* instance;
  FxConsole();
public:
  ~FxConsole();
  static FxConsole* get_instance();

  /*
  * Evénement OnLire
  */
  class LireHandler : public GenericDelegate
  {
  public:
    void do_handler(void*inp, void*outp)
    {
      string data;
      cin >> data;
```

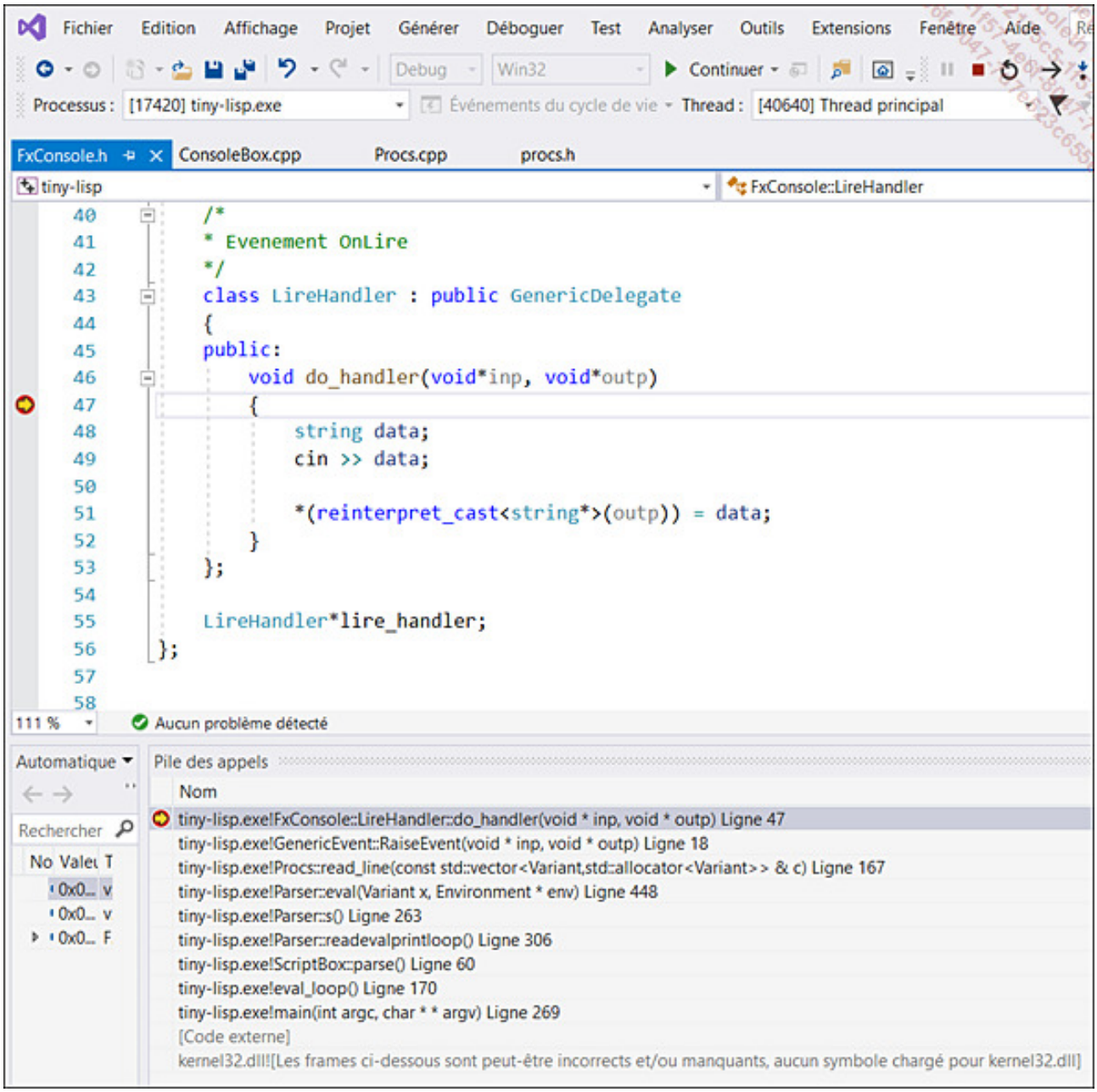
```
        *(reinterpret_cast<string*>(outp)) = data;
      }
    };

    LireHandler*lire_handler;
};
```

Pour tester ce mécanisme d'événement, il faut placer un point d'arrêt au niveau de la méthode `do_handler()`, lancer l'interprète puis indiquer les instructions tiny-lisp suivantes :

```
(read-line)
go.
```

À l'exécution, la méthode `Procs::read_line()` déclenche l'événement et appelle indirectement la méthode `LireHandler::do_handler()`.



L'interprète tiny-lisp fait l'écho de la saisie clavier :

En conclusion, cette approche C++ basée sur les pointeurs de membres rend possible la

définition d'événements et de gestionnaires, sans que la classe demandeuse ait la connaissance explicite de la classe qui réalisera le travail. Cette construction est particulièrement appréciée voire nécessaire dans les environnements asynchrones, les interfaces graphiques, les programmations à base de composants… À tel point que les langages C++ CLI, C# ou Java l'ont intégrée plus ou moins directement dans leur syntaxe.