Fonction de rappel (callback)

Sommaire

Thierry Vaira - <tvaira@free.fr> - version v1.0 - 25/05/2021

1. Fonction de rappel (callback)

Une fonction de rappel (*callback*) ou fonction de post-traitement est une fonction qui est passée en argument à une autre fonction. Cette dernière peut alors faire usage de cette fonction de rappel comme de n'importe quelle autre fonction, alors qu'elle ne la connaît pas par avance. (Wikipédia)

La technique de la fonction de rappel s'inspire du principe d'Hollywood (ou Inversion de contrôle) où l'appelant laisse ses coordonnées pour pouvoir être rappelé par la suite : « Ne nous appelez pas, c'est nous qui vous appellerons ».

2. Utilisation

Les fonctions de rappel sont notamment utilisées dans la programmation évènementielle.

Les fonctions de rappel sont très présentes dans les *frameworks* (pour l'inversion de contrôle). Qt utilise par exemple le mécanisme signal/slot et Java/Android les *listeners*.

Elles sont aussi très utilisées en programmation système et notamment en langage C.

En programmation orientée objet (POO), la technique de rappel évolue en pouvant passer en paramètre un objet (qui se conformera à une interface donnée). L'objet pourra alors encapsuler plusieurs fonctions de rappel.

Le fait de passer un objet permet de contextualiser en indiquant sur **quoi** s'effectue le rappel, tandis qu'une fonction de rappel précisait seulement **comment** rappeler. Cependant la technique des fonctions de rappel continue à avoir les faveurs des langages disposant de fermetures, où celles-ci offrent des capacités équivalentes aux objets en termes de contexte. [wikipedia]

3. Adresse d'une fonction

3.1. Langage C

Rappel : En langage C, on manipule les adresses avec des variables (spéciales) de type **pointeur**.

Un **pointeur sur une fonction** correspondra à l'adresse du début du code de la fonction.

Le nom d'une fonction est donc une **constante de type pointeur** :

```
// définition d'une fonction :
int f(int x, int y)
{
    return x+y;
}

// déclaration d'un pointeur sur une fonction :
int (*pf)(int, int);

// pf est un pointeur vers une fonction admettant 2 entiers en paramètres et
retournant un entier, donc
// on peut faire "pointer" pf sur la fonction f
pf = f;

// un appel en utilisant le pointeur pf (qui contient l'adresse de f) :
printf("%d\n", (*pf)(3, 5)); // affiche 8
```

Les pointeurs sur les fonctions sont notamment utilisés dans la fonction `qsort` qui permet le tri des éléments d'un tableau et dans la recherche dichotomique d'un tableau trié avec `bsearch`. Ces deux fonctions sont définies dans la bibliothèque standard :

```
$ man qsort
NOM
     qsort - Trier un tableau

SYNOPSIS
     #include <stdlib.h>

     void qsort(void *base, size_t nmemb, size_t size,
                int (*compar)(const void *, const void *));

DESCRIPTION
     La fonction qsort() trie un tableau contenant nmemb éléments de taille size.
     L'argument base pointe sur le début du tableau.

     Le  contenu du tableau est trié en ordre croissant, en utilisant la fonction de
comparaison pointée par compar, laquelle est appelée avec deux arguments pointant sur
les objets à comparer.

     La fonction de comparaison doit renvoyer un entier inférieur, égal, ou supérieur
à zéro si le premier argument est respectivement considéré comme inférieur, égal ou
supérieur au second. Si la comparaison des deux arguments renvoie une égalité (valeur
de retour nulle), l'ordre des deux éléments est indéfini.
...

$ man bsearch
NOM
     bsearch - Recherche dichotomique dans un tableau trié

SYNOPSIS
     #include <stdlib.h>

     void *bsearch(const void *key, const void *base,
                   size_t nmemb, size_t size,
                   int (*compar)(const void *, const void *));

DESCRIPTION
    La fonction bsearch() recherche l'objet correspondant à key, dans un tableau de
nmemb objets, commençant à l'adresse base. La taille de chaque élément du tableau est
indiquée dans size.

    Le contenu du tableau doit être trié en ordre croissant par rapport à la fonction
de comparaison référencée par compar. La routine compar doit être capable de recevoir
deux arguments, le premier pointant sur l'objet key, et le second sur un élément du
tableau (l'ordre des arguments est toujours respecté par bsearch).

    Cette routine doit retourner une valeur entière respectivement inférieure, égale
ou supérieure à zéro si l'objet key est inférieur, égal, ou supérieur à l'élément du
tableau.
```

Les pointeurs sur les fonctions sont aussi utilisés dans la création d'un *thread* :

```
$ man pthread_create
NOM
    pthread_create - Créer un nouveau thread

SYNOPSIS
    #include <pthread.h>

    int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                       void *(*start_routine) (void *), void *arg);

    Compilez et effectuez l'édition des liens avec l'option -pthread.

DESCRIPTION
    La  fonction pthread_create()  démarre un nouveau thread dans le processus
appelant. Le nouveau thread commence par appeler start_routine() ; arg est passé comme
unique argument de start_routine().
...
```

On retrouve aussi les pointeurs de fonction sur la plateforme Arduino pour installer un gestionnaire d'interruption avec la fonction `attachInterrupt()` (voir FunctionalInterrupt.cpp).

Deux exemples simples :

```c
#include <stdio.h>
#include <stdlib.h>

int somme(int, int);
int produit(int, int);
int operation(int, int, int(*)(int, int));

int somme(int a, int b)
{
    return (a + b);
}

int produit(int a, int b)
{
    return (a * b);
}

int operation(int a, int b, int (*f)(int, int))
{
    return ((*f)(a, b));
}

int main(int argc, char *argv[])
{
    int a = 2, b = 3;

    printf("%d + %d = %d\n", a, b, operation(a, b, somme));
    printf("%d x %d = %d\n", a, b, operation(a, b, produit));

    return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

int rechercher(int t[], size_t taille, bool (*callback)(int element, int valeur), int
valeur)
{
    for(int i = 0; i < taille; i++)
    {
        if(callback(t[i], valeur))
            return i;
    }

    return -1;
}

bool egale(int element, int valeur)
```

```
{
    if(element == valeur)
    {
        return true;
    }

    return false;
}

bool plusGrande(int element, int valeur)
{
    if(element > valeur)
    {
        return true;
    }

    return false;
}

bool plusPetite(int element, int valeur)
{
    if(element < valeur)
    {
        return true;
    }

    return false;
}

int main(int argc, char *argv[])
{
    const int taille = 5;
    int t[] = {1, 2, 3, 4, 5};

    printf("== 2 ? position = %d\n", rechercher(t, taille, egale, 2));
    printf("> 2 ? position = %d\n", rechercher(t, taille, plusGrande, 2));
    printf("< 2 ? position = %d\n", rechercher(t, taille, plusPetite, 2));

    return 0;
}
```

3.2. Langage C++

Créé initialement par **Bjarne Stroustrup** dans les années 1980, le langage C++ est aujourd'hui normalisé par l'ISO. Sa première normalisation date de **1998** (ISO/CEI 14882:1998), ensuite amendée par l'erratum technique de **2003** (ISO/CEI 14882:2003). Une importante mise à jour a été ratifiée et publiée par l'ISO en septembre 2011 sous le nom de ISO/IEC 14882:2011, ou **C++11**. Depuis, des mises à jour sont publiées régulièrement : en 2014 (ISO/CEI 14882:2014 ou C++14) puis en 2017 (ISO/CEI 14882:2017 ou C++17). [wikipedia.org]

En C++, la mise en oeuvre des pointeurs de fonction et de méthodes avec `std::function` et `std::mem_fn` est apparue avec la mise à jour C++11.

En C++, seules les méthodes statiques (les fonctions directement accessible à partir de la classe) ont une adresse mémoire au moment de la compilation.

`std::function` permet d'encapsuler un pointeur de fonction :

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

using namespace std;

void foo(string str)
{
    cout << "message : " << str << endl;
}

int main()
{
    std::function<void(string)> fn_foo = foo;

    fn_foo("Hello world!");

    return 0;
}
```

`std::mem_fn` permet d'encapsuler un pointeur de méthode (une fonction membre) d'une classe :

```cpp
#include <iostream>
#include <string>
#include <functional>

using namespace std;

class Foo
{
    public:
        Foo(const string& str) : str(str) {}
        void print(int n=1) const { for(int i = 0; i<n; ++i) cout << str << '\n'; }
    private:
        string str;
};

int main()
{
    const Foo foo("Hello world!");

    auto fn1 = mem_fn(&Foo::print);
    fn1(foo, 5);

    return 0;
}
```

Compilation :

```
$ g++ -std=c++11 xxxxxxxx.cpp
```

Liens :

- std::function
- std::mem_fn

Voir aussi :

- std::bind

4. Exemples

4.1. Fonction de rappel en C/C++

Un squelette pour une utilisation avec un pointeur de fonction :

```cpp
#include <iostream> // pour std::cout
```

```
class Foo
{
  public:
    Foo();
    ~Foo();

    void traiter();
    void setCallback( void (*f)() );
    void onEvent();

  private:
    void (*_callback)();
};

Foo::Foo() : _callback(NULL)
{
}

Foo::~Foo()
{
}

void Foo::traiter()
{
    // Effectue le traitement et
    // SI condition ALORS on déclenche le rappel ...
    onEvent();
}

void Foo::setCallback( void (*f)() )
{
    _callback = f;
}

// L'évènement Event déclenchera l'appel à callback
void Foo::onEvent()
{
    if ( _callback != NULL)
    {
        _callback();
    }
}

// La fonction de rappel
void fonctionRappel()
{
    // ...
    std::cout << __FUNCTION__ << std::endl;
}

int main()
```

```
{
    // Instancie un objet
    Foo foo;

    // Installe la fonction de rappel
    foo.setCallback(fonctionRappel);

    // Réalise un traitement
    foo.traiter();

    return 0;
}
```

4.2. Fonction de rappel en C++11

Un squelette pour une utilisation avec `std::function` :

```
#include <iostream> // pour std::cout
#include <functional> // pour std::function

class Foo
{
  public:
    Foo();
    ~Foo();

    void traiter();
    void setCallback( std::function<void()> f );
    void onEvent();

  private:
    std::function<void()> _callback;
};

Foo::Foo() : _callback(nullptr)
{
}

Foo::~Foo()
{
}

void Foo::traiter()
{
    // Effectue le traitement et
    // SI condition ALORS on déclenche le rappel ...
    onEvent();
}
```

```
void Foo::setCallback( std::function<void()> f )
{
    _callback = f;
}

// L'évènement Event déclenchera l'appel à callback
void Foo::onEvent()
{
    if ( _callback != nullptr)
    {
        _callback();
    }
}

// La fonction de rappel
void fonctionRappel()
{
    // ...
    std::cout << __FUNCTION__ << std::endl;
}

int main()
{
    // Instancie un objet
    Foo foo;

    // Installe la fonction de rappel
    foo.setCallback(fonctionRappel);

    // Réalise un traitement
    foo.traiter();

    return 0;
}
```

4.3. Méthode de rappel en C++11

Un squelette pour une utilisation avec `std::bind` :

```
#include <iostream> // pour std::cout
#include <functional> // pour std::function et std::bind

class Foo
{
  public:
    Foo();
    ~Foo();

    void traiter();
```

```
    void setCallback( std::function<void()> f );
    void onEvent();

  private:
    std::function<void()> _callback;
};

Foo::Foo() : _callback(nullptr)
{
}

Foo::~Foo()
{
}

void Foo::traiter()
{
    // Effectue le traitement et
    // SI condition ALORS on déclenche le rappel ...
    onEvent();
}

void Foo::setCallback( std::function<void()> f )
{
    _callback = f;
}

// L'évènement Event déclenchera l'appel à callback
void Foo::onEvent()
{
    if ( _callback != nullptr)
    {
        _callback();
    }
}

class Bar
{
    public:
        Bar() { }
        void methodeRappel();
};

// La méthode de rappel
void Bar::methodeRappel()
{
    // ...
    std::cout << __FUNCTION__ << std::endl;
}

int main()
```

```
{
    // Instancie un objet
    Foo foo;

    // Installe le rappel
    foo.setCallback(std::bind(&Bar::methodeRappel, new Bar()));
    // ou :
    //Bar bar;
    //foo.setCallback(std::bind(&Bar::methodeRappel, &bar));

    // Réalise un traitement
    foo.traiter();

    return 0;
}
```

4.4. ESP32 avec la plateforme Arduino

Exemple d'utilisation d'un gestionnaire d'interruption sous forme de méthode d'une classe :

```
#include <Arduino.h>
#include <FunctionalInterrupt.h> // pour std::bind

// Brochage
#define GPIO_SW1 12

class TestInterrupt
{
  private:
    volatile int nb; // volatile si a est utilisé par le gestionnaire d'interruption

  public:
    TestInterrupt() { nb = 0; }
    int getNb() const { return nb; }
    void IRAM_ATTR routine();
};

void IRAM_ATTR TestInterrupt::routine()
{
  ++nb;
}

TestInterrupt testInterrupt;

void setup()
{
  Serial.begin(115200);
  while (!Serial);

  pinMode(GPIO_SW1, INPUT_PULLUP);

  attachInterrupt(digitalPinToInterrupt(GPIO_SW1), std::bind(&TestInterrupt::routine,
&testInterrupt), FALLING);
}

void loop()
{
  if(testInterrupt.getNb() > 0)
  {
    Serial.print("<loop> nb="); Serial.print(testInterrupt.getNb()); Serial.println();
  }
  else
  {
    Serial.print(".");
  }

  delay(1000);
}
```

5. POO

En programmation orientée objet (POO), la technique de rappel peut évoluer en passant en paramètre un objet qui se conformera à une interface donnée. L'objet pourra alors encapsuler plusieurs fonctions de rappel.

5.1. C++

En C++, on pourra utiliser le concept de **classe abstraite** et **encapsuler plusieurs fonctions de rappel** en utilisant les **méthodes virtuelles pures** :

```
#include <iostream>
#include <functional>

class FooAbstractListener
{
    public:
        virtual void onEvent() = 0;
        virtual void onError() = 0;
};

class Foo
{
    public:
        Foo();
        ~Foo();
        void traiter();
        void setCallback(FooAbstractListener *callback);

    private:
        FooAbstractListener *callback; // Pour les fonctions de rappel sur évènement
(onEvent, onError, ...)

};

Foo::Foo() : callback(nullptr)
{
}

Foo::~Foo()
{
}

void Foo::traiter()
{
    // Effectue le traitement et
    // SI condition ALORS on déclenche le rappel ...
    if(callback != nullptr)
    {
```

```
        callback->onEvent();
    }
}

void Foo::setCallback(FooAbstractListener *callback)
{
    this->callback = callback;
}

class FooListener : public FooAbstractListener
{
    public:
        void onEvent();
        void onError();
};

// La méthode de rappel déclenchée sur évènement
void FooListener::onEvent()
{
    // ...
    std::cout << __FUNCTION__ << std::endl;
}

// La méthode de rappel déclenchée sur l'évènement erreur
void FooListener::onError()
{
    // ...
    std::cout << __FUNCTION__ << std::endl;
}

int main()
{
    // Instancie un objet
    Foo foo;

    // Installe les fonctions de rappel
    foo.setCallback(new FooListener());

    // Réalise un traitement
    foo.traiter();

    return 0;
}
```

5.2. Java et Android

En Java, on pourra utiliser le concept d'**interface** et **encapsuler plusieurs fonctions de rappel** :

```
public interface FooListener
{
    public void onEvent();
    public void onError();
}
```

```
public class Foo
{
    private FooListener callback; // Pour les fonctions de rappel sur évènement
(onEvent, onError, ...)

    /**
     * Installe les fonctions de rappel pour les évènements onEvent, ...
     */
    public void setCallback(FooListener callback)
    {
        this.callback = callback;
    }

    public void traiter()
    {
        // Effectue le traitement et
        // SI condition ALORS on déclenche le rappel ...
        if(callback != null)
        {
            callback.onEvent();
        }
    }
}
```

Une utilisation :

```
// Instancie un objet
foo = new Foo();

// Installe les fonctions de rappel
foo.setCallback(new FooListener()
{
    // Déclenchée sur évènement
    public void onEvent()
    {
        // ...
    }

    // Déclenchée sur l'évènement erreur
    public void onError()
    {
        // ...
    }
});

// Réalise un traitement
foo.traiter();
```

6. Les exemples

src-fonction-rappel.zip

7. Voir aussi

- C++ moderne (C++11, ...)

Site : tvaira.free.fr

Thierry Vaira - <tvaira@free.fr> - version v1.0 - 25/05/2021