# Exercices pratiques de passage de paramètres aux fonctions

## Notions abordées :

* Passage de tableaux : par référence
* Passage de valeurs
* Références
* Pointeurs
* Allocations mémoire
* Passage d’objets
* La classe Array
* Les « vector »

## Passage de tableaux à une fonction

Voici trois fonctions : la fonction *main()*  et deux fonctions de traitement *modifierTableau()* et *modifierElement()*

#include <iostream>

#include <iomanip>

using namespace std;

void **modifierTableau**(int Tab[],int taille)

{

 for (int i = 0; i < taille; i++)

 {

 Tab[i] \*= 2;

 }

}

void **modifierElement**(int e)

{

 e \*= 2;

}

int **main**()

{

 const int tailleTableau = 5;

 int Tableau[tailleTableau] = {0,1,2,3,4};

 int indice;

 cout << "Modification du tableau" << endl;

 cout << "Valeurs d'origine" << endl;

 for (int j = 0; j < tailleTableau;j++)

 {

 cout << setw(3) << Tableau[j];

 }

 cout << endl<< endl;

 modifierTableau(Tableau, tailleTableau);

 cout << "Valeurs modifiees" << endl;

 for (int j = 0; j < tailleTableau;j++)

 {

 cout << setw(3) << Tableau[j];

 }

 cout << endl<< endl;

 cout << "Modifier un element; quel element voulez-vous modifier ? entre 0 et " << tailleTableau-1 << endl;

 cin >> indice;

 cout << "La valeur de Tableau["<< indice<< "] etait de : " << Tableau[indice] << endl;

 modifierElement( Tableau[indice]);

 cout << "La valeur de Tableau["<< indice<< "] est de : " << Tableau[indice] << endl;

 return 0;

}

Saisir le code précédent dans un *« non Qt Project »* *« Plain C++ Project »*

Compiler et exécuter ce programme. Vous obtenez ceci :



### Pourquoi Tableau[3] n’a-t-il pas été modifié ?

* La fonction chargée de modifier l’élément d’indice 3 du tableau (le 4ème élément) est *modifierElement()*

void **modifierElement**(int e)

{

 e \*= 2;

}

e est passée par valeur : le compilateur prend le contenu de la variable à l’appel et la copie dans e.

L’appel de la fonction est :

modifierElement( Tableau[indice]);

Donc le compilateur prend ce qui se troue dans le tableau à l’indice « indice » (ici 3), par conséquent la valeur 6 et la copie dans e.

e n’est qu’une varaible locale à la fonction *modifierElement()*; elle n’existe que pendant la durée de l’exécution de la fonction. On a beau modifier le contenu de e ( e \*= 2) ça ne change pas le contenu de Tableau[indice]

### Pourquoi les valeurs du tableau sont modifiées par la fonction *modifierTableau*?

* Tout simplement parce qu’on passe à la fonction, non pas les valeurs du tableau (copie des valeurs comme pour e) mais l’adresse du premier élément du tableau. Par conséquent, quand on modifie un élément du tableau, en fait, on accède à l’emplacement mémoire connu de toutes les fonctions (et notamment la fonction appelante)

void **modifierTableau**(int Tab[],int taille)

{

 for (int i = 0; i < taille; i++)

 {

 Tab[i] \*= 2;

 }

}

L’écriture « Tab[i] \*= 2 » pourrait s’écrire « \*(Tab+i) \*= 2 » ; c’est-à-dire « Tab+i » c’est l’adresse Tab incrémentée de i. « \*(Tab+i) » signifie le contenu mémoire pointé par l’adresse (Tab+i)

Remarque : on aurait pu écrire aussi void **modifierTableau**(int \*Tab,int taille) En effet, l’écriture Tab[] et \*Tab sont équivalente dans la définition de la fonction.

### Comment permettre la modification d’un élément du tableau dans la fonction *modifierElement() ?*

### Il suffit de passer l’adresse de la variable dont on veut modifier son contenu :

void **modifierElement**(int \*e)

{

 \*e \*= 2;

}

« \*e \*= 2 » signifie le contenu pointé par e

On aurait pu écrire : « \*e = \*e\*2 ; » mais ça n’est pas très lisible !

Pour utiliser cette fonction, il faut l’appeler en lui passant l’adresse de la variable dont on veut modifier son contenu :

modifierElement( &Tableau[indice]);

On spécifie ici, qu’on passe l’adresse mémoire de l’élément Tableau[indice] à l’aide de l’opérateur &

**Faites les modifications nécessaires pour obtenir le résultat suivant :**



Le 4ème élément a bien été modifié par la fonction *modifierElement()*

## Passage de valeurs VS passage par référence

Le passage de valeurs est le seul existant en C : il s’agit de passer aux fonctions les valeurs des variables. Si on a besoin que la fonction modifie le contenu d’une variable alors il faut lui passer l’adresse de la variable. Comme ce mode de passage n’existe pas en C alors il faut passer l’adresse par valeur : on passe à la fonction la valeur de l’adresse ! (cf le début du TP)

En C++, il est possible de passer des variables par référence. Le compilateur C++ crée une référence pour le passage d’adresse. La référence permet (comme un alias) de préciser qu’on veut utiliser l’adresse d’une variable. Il est à noter que contrairement au passage par valeur, la référence n’induit aucune variable locale.

En résumé :

Si on utilise un compilateur C (par ex gcc), on ne peut que passer des valeurs aux fonctions. Si on a besoin de passer une adresse alors on passe la valeur de l’adresse. Dans tous les cas, le compilateur crée une variable locale à la fonction. On ne travaille que sur la copie de la valeur. Si la valeur est une adresse alors on modifie le contenu mémoire de la variable d’origine.

Si on utilise un compilateur C++ (par ex g++), et qu’on indique au compilateur qu’on veut compiler du c++ alors on peut passer des paramètres par valeur ou par référence.

Avantage du passage par référence :

Le compilateur vérifie l’existence réelle de la variable à l’appel. En effet, en C avec un simple pointeur, le compilateur ne vérifie pas l’existence de la variable.

# Modélisation d’un grand prix de Formule 1

# Passage d’objets

On se propose de créer une structure de données permettant de modéliser une course automobile.

Voici la classe UML Bolide :



Les fonctions *get…* et *set..* sont appelées des fonctions **accesseurs**. On les appelle aussi les fonctions « getter » et « setter » (pas très jolis !)

Les fonctions *get* renvoient la valeur d’un attribut privé : c’est pourquoi il y a un type pour la valeur de retour

Exemple :

string Bolide::**getModeleVoiture**()

{

 return modeleVoiture;

}

Les fonctions *set* permettent de modifier un attribut privé : c’est pourquoi il y a un type pour le paramètre.

Exemple :

void Bolide::**setModeleVoiture**(string modele)

{

 modeleVoiture = modele;

}

Coder la classe Bolide et ajouter le code de toutes les méthodes.

On propose la fonction main() de test de la classe Bolide :

#include "bolide.h"

int **main**()

{

 cout << "Course de voitures!" << endl;

 Bolide Voiture1, Voiture2;

 Voiture1.setModeleVoiture("Ferrari");

 Voiture2.setModeleVoiture("Mercedes");

 Voiture1.setNomPilote("Vettel");

 Voiture2.setNomPilote("Hamilton");

 Voiture1.setNumeroVoiture(5);

 Voiture2.setNumeroVoiture(44);

 cout << "La " << Voiture1.getModeleVoiture() << " numero " << Voiture1.getNumeroVoiture() << " pilotee par " << Voiture1.getNomPilote() << " est en tete devant ";

 cout << "la " << Voiture2.getModeleVoiture() << " numero " << Voiture2.getNumeroVoiture() << " pilotee par " << Voiture2.getNomPilote() << endl;

 return 0;

}

Vérifier que votre classe fonctionne.

## Tableau de bolides

Dans la fonction main() précédente, on ne crée que deux voitures et rien ne relie la voiture « Voiture1 » à la voiture « Voiture2 ». Pourtant, elles devraient appartenir à la même course !

Et puis, il est contraignant de créer autant de variables Bolide avec des noms différents.

On va donc créer un tableau de bolides. On a alors qu’une seule variable pour sauvegarder toutes les voitures. De plus, elles font parties de la même variable ; cette variable symbolisera la course.

### Comment créer un tableau d’objets « Bolide » ?

* Bolide Course[nombreVoitures];

En reprenant la fonction main() précédente et en la modifiant comme ceci :

#include "bolide.h"

int **main**()

{

 const int nombreVoitures = 22;

 cout << "Course de voitures!" << endl;

 Bolide Course[nombreVoitures];

 Course[0].setModeleVoiture("Ferrari");

 Course[1].setModeleVoiture("Mercedes");

 Course[0].setNomPilote("Vettel");

 Course[1].setNomPilote("Hamilton");

 Course[0].setNumeroVoiture(5);

 Course[1].setNumeroVoiture(44);

 cout << "La " << Course[0].getModeleVoiture() << " numero " << Course[0].getNumeroVoiture() << " pilotee par " << Course[0].getNomPilote() << " est en tete devant ";

 cout << "la " << Course[1].getModeleVoiture() << " numero " << Course[1].getNumeroVoiture() << " pilotee par " << Course[1].getNomPilote() << endl;

 return 0;

}

On obtient le même résultat. Sauf que, maintenant on peut gérer la position des voitures dans la course comme par exemple inverser la position 0 et 1.

**Ajouter une fonction permettant d’inverser la position de la voiture i et la voiture i+1**

Remarque :

La méthode Bolide() est le constructeur de la classe Bolide. Son rôle est de construire l’objet c’est-à-dire réserver un emplacement mémoire pour y sauvegarder un objet.

**Ajouter dans le constructeur, la ligne suivante :**

cout << "objet cree"<<endl;

Relancer votre programme. Que constatez-vous ?

Pour bien comprendre ce qui se passe, on peut afficher les tailles en octets des éléments que nous manipulons ; **ajouter dans la fonction main() les lignes suivantes :**

cout << "Taille de la classe Bolide : " <<sizeof(Bolide) << " octets" << endl;

cout << "Taille du tableau Course : " << sizeof(Course) << " octets" << endl;

Les affichages sont-ils cohérents ?

## La classe GrandPrix

On décide de créer une classe modélisant un grand prix automobile. Ce grand prix est composé d’un tableau « Course » d’objets « Bolide ».

Voici le diagramme de classes UML :



En C++, cela donne la classe suivante :

#ifndef GRANDPRIX\_H

#define GRANDPRIX\_H

#include "bolide.h"

const int nombreVoitures = 22;

class GrandPrix

{

 Bolide \*Course;

public:

 GrandPrix();

};

#endif // GRANDPRIX\_H

Et le constructeur de la classe GrandPrix :

#include "grandprix.h"

GrandPrix::**GrandPrix**()

{

 Course = new Bolide[nombreVoitures];

}

Explications : *Course* est un pointeur sur des « Bolide ». A la création d’un objet « GrandPrix », seul le pointeur Course est créé. La création du tableau n’est pas effectuée. Il faut la faire à la création de l’objet « GrandPrix » dans le constructeur.

### Liste d’initialisations

Il y a une méthode plus élégante et plus performante surtout, d’effectuer la création du tableau Course ; on peut utiliser la liste d’initialisation.

Il s’agit de demander au constructeur d’effectuer une initialisation de certains attributs. Le gros avantage c’est que ça n’est pas considéré comme du code. Cela s’écrit comme ça :

GrandPrix::**GrandPrix**():Course(new Bolide[nombreVoitures])

{

}

**Vérifier que la création d’un objet GrandPrix (dans la fonction main() )crée bien automatiquement les 22 objets « Bolide » :**

GrandPrix gpdefrance;

### Méthode void **ajouterConcurrent**(Bolide &unbolide) de la classe GrandPrix

Elle consiste à ajouter un bolide dans la course. Elle prend en paramètre une référence sur un objet de la classe Bolide

**Donner le code cette méthode – il sera sans doute nécessaire d’ajouter un attirbut gérant le nombre de concurrents actuellement enregistrés – cet attribut sera initialisé à 0 grâce à la liste d’initialisation du constructeur.**

En fait, on s’aperçoit que cette méthode ne modifie pas l’objet Bolide passé en paramètre. Par conséquent, il est préférable de restreindre l’accès et ne pas permettre à celle-ci de modifier son contenu en ajoutant le modificateur *const* Cela donne :

void **ajouterConcurrent**(const Bolide &unbolide);

### Méthode Bolide **getConcurrent**(int position)

Elle consiste à renvoyer le bolide occupant la position « position ».

**Donner le code de cette méthode.**

La fonction main() pourrait ressembler à :

int **main**()

{

 cout << "Course de voitures!" << endl;

 GrandPrix gpdefrance;

 Bolide Voiture;

 Voiture.setModeleVoiture("Ferrari");

 Voiture.setNomPilote("Vettel");

 Voiture.setNumeroVoiture(5) ;

 gpdefrance.ajouterConcurrent(Voiture);

 cout << "pilote en premiere position : " << gpdefrance.getConcurrent(0).getNomPilote() << endl;

 return 0;

}

**Modifier la pour insérer plusieurs concurrents et afficher tous les concurrents de la course.**