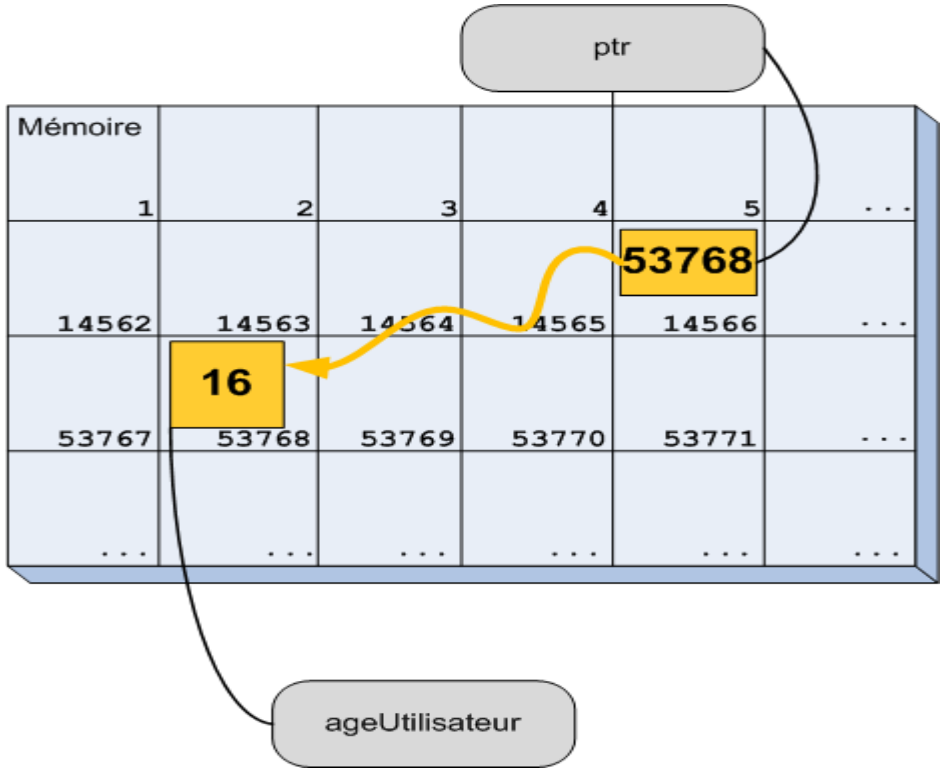
CFI-CIRAS

ALGORITHME AVANCE
Chapitre 2:
Notion de pointeurs

Semestre III
Licence 2 Info



Sommaire

1. Introduction

Nous avons déjà été amené à utiliser l'opérateur & pour désigner l'adresse d'une variable. D'une manière générale, le langage C++ permet de manipuler des adresses par l'intermédiaire de variables nommées pointeurs.

- **Qu'est-ce qu'un pointeur ?**

Un **pointeur** est une **variable** qui contient pour valeur l'adresse d'une autre variable (fonction, objet…). Il possède le même type que cette variable. Il est déclaré précéder de l'opérateur étoile * .

- **Syntaxe de déclaration :**

<Type_Variable> * <Nom_Variable> ;

- **Exemple :**

```
int *x ;  float *a, *b ;  char *t ;
```

x est un pointeur qui va contenir l'adresse d'un entier, a et b sont des pointeurs qui vont contenir chacun l'adresse d'un réel et t contiendra l'adresse d'un caractère ou une chaîne de caractères.

1. Introduction

- **Notation :**

int u ; // déclaration d'un entier u.

int *v = &u ; /* ici en déclare un pointeur nommé v qu'on initialise avec '**&u**'. La notation '**&u**' se lit adresse de la variable u. Elle désigne l'adresse de la place mémoire alloué par le compilateur à la variable u. Le symbole **&** est appelé **opérateur d'adressage**. */

La déclaration précédente est équivaut à :

```
int u, *v ;

v = &u ;
```

Le pointeur **v** n'est pas initialisé en même temps qu'il est déclaré. Il est initialisé par la suite dans le programme. Remarquer qu'il n'y a pas d'étoile devant le **v**. Dans une instruction du programme, < ***v*** > a une autre signification. int u, *v = &u ; *v = 3 ;

1. Introduction

Dans une instruction (pas dans la déclaration), < ***v** > permet d'accéder à la variable dont l'adresse est contenue dans **v**. le symbole ***** est appelé **opérateur d'indirection**. C'est un opérateur qui agit exclusivement sur une variable pointeur.

Puisque le pointeur **v** contient l'adresse de la variable **u**, ***v** signifie donc **u**.

NB : Pour initialiser un pointeur, c'est-à-dire lui donner une valeur par défaut, on n'utilise généralement pas le nombre zéro (**0**) mais le mot-clé **NULL** (en majuscules).

Exemple : `int *Ptr = NULL;`

2. Application des pointeurs en C++

Exemple 1:

```
#include <iostream>

using namespace std;

int main()
{
    cout << "--------------------------" << endl;
    cout << "    NOTION DE POINTEURS   " << endl;
    cout << "--------------------------" << endl;
    int u = 10;
    cout << " VALEUR de u = " << u <<endl;
    cout << " ADRESSE de u = " << &u <<endl;

    return 0;
}
```

Donner et expliquer le résultat.

Exemple 2:

```
#include <iostream>

using namespace std;

int main()
{
    cout << "--------------------------" << endl;
    cout << "    NOTION DE POINTEURS   " << endl;
    cout << "--------------------------" << endl;
    int a = 5;
    int *b;
    b = &a;
    cout << " a = " << a <<endl;
    cout << " *b = " << *b <<endl;
    return 0;
}
```

Donner et expliquer le résultat.

2. Application des pointeurs e

Exemple 3:

```
#include <iostream>

using namespace std;

int main()
{
    cout << "--------------------------" << endl;
    cout << "    NOTION DE POINTEURS   " << endl;
    cout << "--------------------------" << endl;
    int a = 3;
    int *b = &a;
    cout << " a = " << a <<endl;
    *b = 7;
    cout << " a = " << a <<endl;
    return 0;
}
```

Donner et expliquer le résultat.

Exemple 4:

```
#include <iostream>
using namespace std;
int main()
{
    cout << "--------------------------" << endl;
    cout << "    NOTION DE POINTEURS   " << endl;
    cout << "--------------------------" << endl;
    float a = 5;
    float *b = &a;
    float **c = &b;
    float ***d = &c;
    cout << " a = " << a <<endl;
    cout << " &a = " << &a <<endl;
    cout << "-----------------" << endl;
    cout << " b = " << b <<endl;
    cout << " &b = " << &b <<endl;
    cout << " *b = " << *b <<endl;
    cout << "-----------------" << endl;
    cout << " c = " << c <<endl;
    cout << " &c = " << &c <<endl;
    cout << " *c = " << *c <<endl;
    cout << "-----------------" << endl;
    cout << " d = " << d <<endl;
    cout << " &d = " << &d <<endl;
    cout << " *d = " << *d <<endl;
    return 0;
}
```

Donner et expliquer le résultat.

3. Passage de paramètres à un

Lorsqu'on passe un paramètre (argument) par pointeur à une fonction on dit qu'on fait un passage de paramètre par adresse. Contrairement au passage par valeur, les modifications apportées à ces paramètres par la fonction en question seront les nouvelles valeurs de ces paramètres pour la suite du programme.

Passage par valeur

Dans le passage par valeur, la fonction reçoit la valeur de la variable passée en paramètre.

Exemple : Donner et expliquer le résultat

```
#include <iostream>
using namespace std;

void fct(int a)
{
    a = a+7;
    cout <<"Dans la fonction fct(), a = "<< a <<endl;
}

int main()
{
    cout << "-------------------------" << endl;
    cout << "    NOTION DE POINTEURS   " << endl;
    cout << "-------------------------" << endl<<endl;
    int a = 5;
    cout <<"Avant l'appel de la fonction fct(), a = "<< a <<endl;
    fct(a); // Appel de la fonction avec Passage par valeur
    cout <<"Apres l'appel de la fonction fct(), a = "<< a <<endl;
    return 0;
}
```

3. Passage de paramètres à une fonction

Passage par adresse (référence)

Dans le passage par adresse, la fonction ne reçoit pas la valeur de la variable passée en paramètre (comme ci-dessus), mais elle reçoit son adresse.

Exemple :

Donner et expliquer le résultat

```
#include <iostream>
using namespace std;

void fct(int *a)
{
    *a = *a+7;
    cout <<"Dans la fonction fct(), a = "<< *a <<endl;
}

int main()
{
    cout << "--------------------------" << endl;
    cout << "    NOTION DE POINTEURS   " << endl;
    cout << "--------------------------" << endl<<endl;
    int a = 5;
    cout <<"Avant l'appel de la fonction fct(), a = "<< a <<endl;
    fct(&a); // Appel de la fonction avec Passage par adresse
    cout <<"Apres l'appel de la fonction fct(), a = "<< a <<endl;
    return 0;
}
```

3. Passage de paramètres à ur

Utilité du passage par adresse

- Si une fonction reçoit un paramètre et que celui-ci subit des modifications dans la fonction (il prend une nouvelle valeur). Si on souhaite continuer la suite du programme avec cette nouvelle valeur, le passage de ce paramètre par adresse permet de récupérer directement la nouvelle valeur qu'il prend dans la fonction.

- Une fonction ne peut pas retourner plus d'une valeur. Mais grâce au passage de paramètres par adresse, on peut récupérer plusieurs valeurs à la fois, même si la fonction est de type <void>.

```
#include <iostream>
using namespace std;

void calculs(int,int *,int *,int *);

int main()
{
    cout << "--------------------------" << endl;
    cout << "    NOTION DE POINTEURS   " << endl;
    cout << "--------------------------" << endl<<endl;
    int x,Double,Triple,Carre;
    cout <<"Entrez un nombre : ";
    cin >> x;
    calculs(x,&Double,&Triple,&Carre);
    cout <<"Le Double de "<< x <<" est "<< Double <<endl;
    cout <<"Le Triple de "<< x <<" est "<< Triple <<endl;
    cout <<"Le Carre de  "<< x <<" est "<< Carre <<endl;
    return 34;
}

void calculs(int x,int *Double,int *Triple,int *Carre)
{
    *Double = 2*x;
    *Triple = 3*x;
    *Carre  = 2*x;
}
```

3. Passage de paramètres à une

Utilité du passage par adresse

La fonction calculs reçoit en paramètres un entier et trois adresses. L'entier va servir pour les calculs du double, du triple et du carré, et les adresses vont servir à recevoir les résultats de ces calculs.

```cpp
#include <iostream>
using namespace std;

void calculs(int,int *,int *,int *);

int main()
{
    cout << "--------------------------" << endl;
    cout << "    NOTION DE POINTEURS   " << endl;
    cout << "--------------------------" << endl<<endl;
    int x,Double,Triple,Carre;
    cout <<"Entrez un nombre : ";
    cin >> x;
    calculs(x,&Double,&Triple,&Carre);
    cout <<"Le Double de "<< x <<" est "<< Double <<endl;
    cout <<"Le Triple de "<< x <<" est "<< Triple <<endl;
    cout <<"Le Carre de  "<< x <<" est "<< Carre <<endl;
    return 34;
}

void calculs(int x,int *Double,int *Triple,int *Carre)
{
    *Double = 2*x;
    *Triple = 3*x;
    *Carre  = 2*x;
}
```

4. Passage d’un tableau à une fonction

Le passage d’un tableau de données comme paramètre à une fonction est toujours un passage par adresse, car le nom d’un tableau contient en réalité son adresse. Les modifications apportées par la fonction à ce tableau vont apparaître dans le reste du programme.

➢ **Syntaxe :**

- Déclaration : **<Type>** <**Nom Fonction**>**(** **<Type>** <**Nom Tableau**>**[ ])** ;
- Appel : <**Nom Fonction**>**(**<**Nom Tableau**>**)** ;
- Définition : **<Type>** <**Nom Fonction**>**(** **<Type>** <**Nom Tableau**>**[ ]) {**….**}**

➢ **Exemple :**

Ecrire un programme qui permet de saisir un tableau d’entiers, le trie par ordre croissant, puis l’afficher. Le tri se fait dans une fonction qui reçoit en paramètre le tableau saisi. L’affichage se fait dans le main().

5. Retour d'un tableau à une fonction

Comme pour le passage d'un tableau à une fonction, le retour d'un tableau par une fonction est toujours un retour par adresse.

➢ **Syntaxe :**

- Déclaration : **<Type>** * <**Nom Fonction**>**(Paramètres)** ;
- Appel : <**Nom Fonction**>**(Paramètres)** ;
- Définition : **<Type>** * <**Nom Fonction**>**(Paramètres) {…..}**

  Où **<Type>** désigne le type du pointeur retourné par la fonction. Il est le même que celui du tableau.

Lorsqu'une fonction se termine, le lien entre le reste du programme et les variables locales de cette fonction est coupé. Pour maintenir le lien avec l'emplacement mémoire contenant le tableau de données (celui dont on retourne l'adresse), il faut faire une allocation de mémoire en utilisant la fonction **malloc()** contenue dans le fichier entête **«alloc.h»**. Le lien avec cet espace mémoire sera maintenu jusqu'à ce qu'on le libère avec la fonction **free()** contenue dans le même fichier entête.

5. Retour d’un tableau à une fonction

La fonction **malloc()** reçoit en paramètre la taille (en octets) de l’espace mémoire à allouer et retourne l’adresse de cet espace.

Attention !!! : Certain IDE implémente les fonctions **malloc()** et **free()** dans la bibliothèque standard « **stdlib.h** » c’est le cas da CodeBlocks.

Exemple : Programme qui permet de saisir un tableau d’entiers dans une fonction, le retourne au main() qui l’affiche.

5. Retour

```
#include <iostream>
#include <stdlib.h>
using namespace std;

int *Saisie();

int main()
{
    int *t,i;
    t = Saisie();
    cout <<"\n Affichage du tableau: "<<endl;
    for(int i=0;i<5;++i)
    {
        cout <<"  t["<<i<<"] = "<<t[i]<<endl;
    }
    free(t); //Libération de la mémoire
    return 0;
}

int *Saisie()
{
    int *t = (int*)malloc(5*sizeof(int)); //Allocation dynamique de la mémoire
    cout <<" Saisie du tableau :"<<endl;
    for(int i=0;i<5;++i)
    {
        cout <<"  Entrer t["<<i<<"] = ";
        cin >> t[i];
    }
    return t;
}
```

5. Retour d’un tableau à une fonction

Dans la fonction **Saisie()**, on allouer (réserve) un espace mémoire pour cinq (05) entiers avec la fonction **malloc()**. L’adresse de cet espace mémoire (de ce tableau) est affectée au **pointeur t**. Ce pointeur **t** qui est en même temps le nom du tableau (puisque le nom d’un tableau contient en réalité son adresse) est retourné à la fin de la saisie. Ce pointeur est récupéré dans un autre **pointeur t du main()** (on peut l’appeler autrement, ça ne changera rien), puis on affiche les données qui s’y trouvent.

6. Conclusion

Les **pointeurs** sont, comme on l'a vu, très utilisés en C/C++. Il faut donc bien savoir les manipuler.
Mais ils sont très dangereux, car ils permettent d'accéder à n'importe quelle zone mémoire, s'ils ne sont pas correctement initialisés. Dans ce cas, ils pointent n'importe où.

Accéder à la mémoire avec un pointeur non initialisé peut altérer soit :

- **les données du programme;**
- **le code du programme lui-même;**
- **le code d'un autre programme;**
- **ou celui du système d'exploitation.**

Cela conduit dans la majorité des cas au plantage du programme, et parfois au plantage de l'ordinateur si le système ne dispose pas de mécanismes de protection efficaces.

VEILLEZ À TOUJOURS INITIALISER LES POINTEURS QUE VOUS UTILISEZ.

Pour initialiser un pointeur qui ne pointe sur rien (c'est le cas lorsque la variable pointée n'est pas encore créée ou lorsqu'elle est inconnue lors de la déclaration du pointeur), on utilisera le pointeur prédéfini **NULL**.

VÉRIFIEZ QUE TOUTE DEMANDE D'ALLOCATION MÉMOIRE A ÉTÉ SATISFAITE.

Pour plus d'information sur les fonctions **malloc**, **free** et leur version améliorer les opérateurs **new, new[ ]** et **delete,** veuillez vous référer a la littérature de la programmation en langage C/C++ (**gestion dynamique de la mémoire**).