# Pointeurs sur des fonctions, fonctions objets et adapters par C.G. et M.M.

## Pointeurs sur les fonctions

Le nom des fonctions est automatiquement converti au besoin en adresse des fonctions, un peu à la manière des vecteurs (de base en C et C++) dont le nom est convertible en adresse du premier élément.

Analogie :
```
// VECTEUR
int v[NB_ELEM];
F(v);
```

On doit avoir :
```
void F(int p_v[]);
ou void F(int* p_v);
```

Bref, on peut remplacer le paramètre de style vecteur (`p_v[]`) par un pointeur (`*p_v`) parce que le nom du vecteur (`v`) donnait l'adresse de son premier élément (`v === &v[0]`).

```
// FONCTIONS
void Ecrire(int p_n)
    {
    cout << p_n;
    }
...
Fct(15, Ecrire);
```

On doit avoir :
```
void Fct(int p_nb, void p_F(int p_n));     // Ces trois déclarations
ou  void Fct(int p_nb, void p_F(int)));    //  sont
ou  void Fct(int p_nb, void (*p_F)(int));  //  parfaitement équivalents
```

La troisième déclaration est celle qui fait le mieux voir que le nom de la fonction (`Ecrire`) est l'adresse de la fonction puisqu'on reçoit un pointeur…

Contrairement à ce que l'on pourrait penser dans les 2 premières déclarations, `p_F` n'est pas une fonction, mais un pointeur sur une fonction. En effet, il n'est pas possible en C++ d'avoir des variables ou des paramètres de type « fonction ». Toute tentative de déclarer un tel paramètre est immédiatement convertie en une déclaration de pointeur sur une fonction. Ainsi la première déclaration semble dire qu'on reçoit une fonction retournant `void` et recevant un `int`, alors que la deuxième ne fait qu'enlever le nom du paramètre, ce qu'on peut faire avec tout prototype ou simple déclaration de fonction. Dans ces deux cas, le compilateur interprète ces déclarations de `p_F` de la même façon : c'est un pointeur qui recevra l'adresse d'une fonction respectant ces critères.

La troisième notation, équivalente mais moins simple, est cependant nécessaire lorsqu'il faut déclarer des *variables* de type pointeurs sur des fonctions plutôt que des paramètres, car :

```
{
void p_F(int);
```

indiquerait, dans le code, la déclaration d'une fonction appelée `p_F`... et non pas une variable! On déclarerait donc une variable ainsi :

```
void (*ptrF)(int);  // Ou avec init., par exemple : void (*ptrF)(int)= Ecrire;
```

Comment lire cette déclaration ? Comme toutes les déclarations en C++, on est censé y voir un exemple de son utilisation éventuelle. Par exemple, `int i=0;` signifie qu'on pourra affecter des entiers et donc la variable est un `int`; `int v[10];` signifie qu'avec un indice après `v`, on obtient un `int` donc `v` est un vecteur; `int *p;` (ou `int* p;`) signifie que `*p` est un `int` donc `p` est un pointeur sur un `int`. Finalement `void (*ptrF)(int);` signifie que si on fait `*ptrF,` le résultat peut être utilisé comme une fonction à laquelle on passe un `int` et qui retournera `void`., donc il s'agit d'un pointeur sur une telle fonction. Les parenthèses autour de `*ptrF` sont nécessaires, car sinon, par la priorité des opérateurs, on aurait la déclaration d'une fonction retournant un pointeur `void` (comme `void* ptrF(int);`).

Pour alléger les déclarations, en particulier si la fonction prend plusieurs paramètres et retourne un type complexe, on peut se servir d'un `typedef` :

```
typedef void (*TypePtrSurFctVoidPrenantUnInt)(int);
```

permettant de déclarer :

```
TypePtrSurFctVoidPrenantUnInt ptrF;
```

On aurait alors une quatrième possibilité pour la déclaration ci-haut :

```
void Fct(int p_n, TypePtrSurFctVoidPrenantUnInt p_F);
```

et voici la fonction :

```
void Fct(int p_n, TypePtrSurFctVoidPrenantUnInt p_F)
    {
    p_F(p_n*2);
    }
```

On peut appeler directement une fonction à partir du pointeur ou déréférencer le pointeur d'abord :

```
void Fct(int p_n, TypePtrSurFctVoidPrenantUnInt p_F)
    {
    (*p_F)(p_n*2);
    }
```

C'est une question de goût ![1] Notez d'ailleurs que le `p_` de `p_F` ne voulait pas dire pointeur mais paramètre donc on devrait plutôt faire la version sans déréférencement ou celle-ci :

```
void Fct(int p_n, TypePtrSurFctVoidPrenantUnInt p_ptrF)
    {
    (*p_ptrF)(p_n*2);
    }
```

Autre exemple : si on a la fonction suivante :

```
const char* PosEtQte(const char* p_texte, char p_c, int& p_s_nb)
    {
    const char* pPremier= 0; // Position du premier p_c trouvé ou 0
    p_s_nb= 0;               // Nombre d'occurrences de p_c trouvées

    for ( ; *p_texte != '\0'; ++p_texte)
        {
        if (*p_texte == p_c)
            {
            ++p_s_nb;

            if (pPremier == 0) pPremier= p_texte;
            }
        }

    return pPremier;
    }
```

Dont on se sert ainsi :

```
char lignes[1000][100];
...
Traiter(lignes, PosEtQte);
```

Quelle est la déclaration de `Traiter` ?

```
void Traiter(char p_lignes[][100], const char* p_F(const char*, char, int&))
    {
    ...
    char *p= p_F(p_lignes[i], ' ', nbEspaces);

    // ou
    // char *p= p_F(&p_lignes[i][0],' ', nbEspaces);
    ...
```

[1] Mais pour les pointeurs sur fonctions membres, une autre sorte de bestiole, il faut absolument le déréférencement et la déclaration explicite de pointeur. Par généralité, on peut donc préférer faire la même chose avec les pointeurs sur fonctions ordinaires.

```
        }
```

Dernier exemple :

```
// Plusieurs fonctions (prototypes ou fonctions complètes) :
void AjouterCours(int& p_es_nbCours, int& p_es_nbEleves);
void ListerCoursOfferts(int& p_es_nbCours, int& p_es_nbEleves);
void InscrireEtudiants(int& p_es_nbCours, int& p_es_nbEleves);
void ListerEtudiants(int& p_es_nbCours, int& p_es_nbEleves);
void ImprimerListesDeClasse(int& p_es_nbCours, int& p_es_nbEleves);

// Un vecteur de pointeur sur des fonctions contenant leurs adresses
??? optionsDuMenu[]= { AjouterCours, ListerCoursOfferts, InscrireEtudiants,
                       ListerEtudiants, ImprimerListesDeClasse };
AfficherMenu();
cin >> choix;             // entre 0 et 5, 0==quitter

if (choix != 0)
    optionsDuMenu[choix-1](nbCours, nbEleves);   // appellera la bonne fonction
```

Comment compléter la déclaration de `optionsDuMenu` ?

```
typedef void (*TypePtrSurFctDuMenu)(int&, int&);
TypePtrSurFctDuMenu optionsDuMenu[]= {...};
```

Ou directement

```
void (*(optionsDuMenu[]))(int&, int&)= {...};    // illisible? On verra pire...
```

La traduction de `void (*(optionsDuMenu[]))(int&, int&)= ...` se fait donc ainsi :

```
           ^^^^^^^^^^^^^^^
```

`optionsDuMenu` est une variable de type vecteur, les parenthèses autour de `optionsDuMenu[]` sont facultatives car `[]` a priorité sur *. Quand on met un indice à ce vecteur on obtient une valeur :

```
void (*(  valeurT ))(int&, int&)
```

Les parenthèses autour de `valeurT` étant inutiles on a :

```
void (*  valeurT  )(int&, int&)
      ^^^^^^^^^^^
```

`valeurT` est un pointeur, son déréférencement donne une autre valeur :

```
void (   valeurP  )(int&, int&)
```

qu'on peut appeler comme une fonction prenant deux références à des `int` et ne retournant rien. Donc `optionsDuMenu` était un vecteur de pointeurs sur ce genre de fonctions (ce qu'on avait supposer en regardant son initialisation).

Vous comprenez bien ? Voici deux beaux exemples de la bibliothèque standard. Que signifie ceci :

```
void qsort(void* base, size_t n, size_t width, int (*cmp)(const void*, const void*));
```

Il s'agit de la fonction de tri fournie par la bibliothèque classique en C et disponible aussi en C++ (mais il y a plus commode dans `<algorithm>`). Pour qu'elle soit polymorphique, c'est-à-dire applicable aux vecteurs contenant des éléments de n'importe quel type, sans utiliser de `template`, on doit lui passer plusieurs informations sur les données à trier :

`void* base` : adresse du début de la zone de mémoire, `void*` permet de passer l'adresse de n'importe quoi.
`size_t n` : nombre d'éléments à trier.
`size_t width` : taille de chacun des éléments.
`int (*cmp)(const void*, const void*)` : adresse d'une fonction permettant la comparaison des éléments. Elle reçoit l'adresse de deux éléments et doit renvoyer une valeur `<0`, `>0` ou `==0` selon que l'on considère que la première donnée est plus petite, plus grande ou égale à la deuxième (comme `strcmp`). Puisque `qsort` ne connaît pas le type des éléments, elle travaille avec des pointeurs `void*`, qu'il faut convertir.

Exemple d'utilisation :

```
struct TypeClient
    {
```

```
    int noClient;
    char nom[LARGEUR_NOM+1]; // Je suppose l'exemple en C, où string n'existe pas
    ...
    };

int CompareClient(const void* p_pC1, const void* p_pC2)
    {
    const TypeClient* pC1= (const TypeClient*)p_pC1;  // En C++, on ferait des static_cast
    const TypeClient* pC2= (const TypeClient*)p_pC2;  //  bien entendu.

    int diff= strcmp(pC1->nom, pC2->nom);

    if (diff != 0)
        return diff;                      // Si même nom, on utilise
                                          //  la « comparaison » du
    return pC1->noClient - pC2->noClient;  //  numéro de client.
    }

...
TypeClient clients[NB_MAX_CLIENTS];
...
qsort(clients, NB_MAX_CLIENTS, sizeof(TypeClient), CompareClient);
...
```

La fonction de comparaison est un exemple de ce qu'on appellera une fonction « prédicat ».

Voici un dernier exemple de la bibliothèque standard. Que signifie ceci :

```
void (* signal(int sigACapter, void (*func)(int sigACapter)))(int sigACapter);
```

On voit que

```
void ( la valeur ici est déclarée comme une fct appellable  )(int sigACapter);
      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
      * signal(int sigACapter, void (*func)(int sigACapter))
      * signal(            ses paramètres                  )
```

donc la fonction `signal` renvoie quelque chose qu'on peut appeler comme une fonction… Donc elle renvoie un pointeur sur une fonction. On pourrait déclarer :

```
typedef void (*TypePtrFonction)(int);

TypePtrFonction signal(...);
```

En fait :

```
TypePtrFonction signal(int sigACapter, void (*func)(int));
                                       ^^^^^^^^^^^^^^^^^
```

« clairement » un paramètre de type `TypePtrFonction` aussi

```
TypePtrFonction signal(int sigACapter, TypePtrFonction func);
```

La fonction `signal` s'emploie ainsi (sous Unix) :

```
void CapterKill(int p_code)  // Notre « handler » du Del
    {
    ...
    }

// SIGINT est le signal reçu lors d'un Del sous Unix (style Ctrl+C en Windows en
// mode console). On installe notre fonction pour que le programme ne s'arrête pas
// directement lors d'un Del. On prend en note l'adresse de l'ancienne fonction
// qui s'en occupait, dans la variable ancFctPourSIGINT.
void (*ancFctPourSIGINT)(int)= signal(SIGINT, CapterKill);

...
// traitement pendant lequel un Del appellerait CapterKill
...

// On veut remettre l'ancien «capteur» de Del
signal(SIGINT, ancFctPourSIGINT);  // Renvoie l'adresse de CapterKill
```

---

**Problème exploratoire 1**

Supposons qu'on veuille faire une classe (`template` en fait) de tableau à deux dimensions :

```
template<typename T>
class CMatrice
    {
    public :
    CMatrice(int p_nbLignes, int p_nbColonnes);
    CMatrice();
    ...
    };

CMatrice<int> m;
```

Comment accéder aux éléments ?

`m[3][4]` : le premier `[3]` est un appel à `CMatrice::operator[](int)` qui renverrait quoi, pour qu'on puisse indicer le résultat avec `[4]` ??? C'est possible, mais pas simple...

`m[3, 4]` : on aurait besoin de `CMatrice::operator[](int p_lig, int p_col)` mais les `[ ]` n'acceptent qu'un seul paramètre.

Pourrait-on utiliser une fonction du genre `m.at(lig, col)` ou peut-être utiliser un autre symbole ? Par exemple, plusieurs langages de programmation (COBOL, FORTRAN, BASIC, etc.) utilisent les parenthèses :

```
m(3,4)= 10;
```

Il suffirait que `CMatrice::operator()(int lig, int col)` renvoie une référence à un élément de type `T`. Puisque dans le langage, les parenthèses entourent des listes de paramètres comprenant 0, 1 ou plusieurs paramètres, on peut déclarer `operator()` avec autant de paramètres qu'on veut (on peut même le surcharger).

Donc on ferait la fonction membre :

```
T& operator()(int p_lig, int p_col)
    {
    ...
    }
```

et ce serait une bonne solution...

*(PAUSE... quelques secondes...)*

Mais... ne peut-on pas penser que `m(3,4)` est un appel de fonction ? Sûrement... car c'est le cas... pas un appel de la fonction « `m` » cependant, mais plutôt de la fonction `operator()` qui est une fonction membre de `CMatrice`.

Voilà un truc qui pourrait donc servir ailleurs...

N.B. On a déjà fait ça avant... avec le constructeur :

```
CMatrice<int> m(10,20);   // appel de fonction ? oui le contructeur est appelé !
```

**Problème exploratoire 2**

Supposons qu'on désire rechercher une valeur dans une collection standard. On peut penser à la fonction `template` suivante :

```
template <typename C>
typename C::iterator Trouve(typename C::iterator p_debut, typename C::iterator p_fin,
                            typename C::value_type p_valeur)
    {
    while (p_debut != p_fin && *p_debut != p_valeur)       // On déplace début pour éviter
        ++p_debut;                                         //  une variable supplémentaire

    return p_debut;
    }

// Prendre le temps de comprendre l'utilisation :
```

```
if (col.end() == Trouve(col.begin(), col.end(), x))
    cout << "x n'est pas dans la collection.\n";

it= col.end() - 10;                                 // On ne veut pas chercher
                                                    //  dans les 10 derniers; la
if (it == Trouve(col.begin(), it, x))               //  valeur sentinelle est la
    cout << "x n'est pas dans la collection.\n";    //  limite supérieure...
```

Si on veut trouver plutôt la première valeur respectant un certain critère :

```
template <typename C>
typename C::iterator TrouveSi(typename C::iterator p_debut, tyopename C::iterator p_fin,
                              bool (*p_ValeurDesiree)(C::value_type))
    {
    while (p_debut != p_fin && !p_ValeurDesiree(*p_debut))
        ++p_debut;

    return p_debut;
    }
```

Exemple :

```
    bool PlusGrandQue100(int p_nb)
        {
        return p_nb > 100;
        }

    ...

    vector<int> v...

    vector<int>::iterator it= TrouveSi(v.begin(), v.end(), PlusGrandQue100);

    if (it == v.end())
        cout << "Aucune valeur > 100.\n"
    else
        cout << *it << " est > 100.\n";
```

Mais si on veut `>1000`, `>500`, `>variable` ?

Il est facile (mais *bien désolant*) d'écrire `PlusGrandQue1000`, `PlusGrandQue500`, mais que faire avec `>variable` :

```
    bool PlusGrandQueN(int v)
        {
        return v > N; // C'est quoi N ??? une globale ? ouache !
        }
```

Solution ? Faisons du C++...

```
    class CPlusGrandQue
        {
        public :
        CPlusGrandQue(int p_borne)
            : m_borne(p_borne)
            {}

        bool operator()(int p_nb) const
            {
            return p_nb > m_borne;
            }

        private :
        int m_borne;
        };
```

Ça donne quoi ? Par exemple :

```
    CPlusGrandQue f(10);  // Objet de type CPlusGrandQue contenant m_borne==10

    int n= ...;

    if (f(n))             // ==  f.operator()(n)  ==  n > m_borne  ==  n > 10
        cout << "n est plus grand que 10.\n";
```

Le code généré sera probablement bel et bien simplement `if (n > 10)` car tout est *inline*... Donc l'objet de type `CPlusGrandQue` peut être utilisé comme une fonction pour comparer le paramètre avec la borne fournie dans la classe. On utilise donc un objet de cette classe comme s'il s'agissait d'une fonction. On appelle ce type d'objet « function object » qu'on traduira, tant bien que mal, par fonction objet (en réalité ce serait plutôt un objet fonction, où « fonction » est interprété comme un adjectif).
On peut aussi faire bien sûr :

```
int max= ...;
CPlusGrandQue depasse(max);
int n= ...;

if (depasse(n))
    cout << "n depasse le maximum.\n";
```

( Rappel : création d'un objet ou temporaire... plusieurs possibilités :

```
     // Ces trois déclarations donnent un résultat équivalent...
    string s= "Allo";          // Forme où on s'imagine que "Allo" est de type string
    string s= string("Allo");  // Celle-ci est la plus facile à expliquer de façon
                               //  formelle, car on peut penser à
                               //  int i= int(3.2); // int i= static_cast<int>(3.2);
    string s("Allo");          // Utilisation explicite du constructeur

    CQqc q(p1, p2);

    if (q == CQqc(p1, p2))     // Le constructeur sert à créer un objet temporaire qui
        ...                    //  qui servira simplement pendant la comparaison, son
                               //  destructeur sera appelé ensuite

    // Ces deux appels sont donc équivalents
    Fct(q);
    Fct(CQqc(p1, p2)); /// l'objet temporaire sera détruit ensuite  )
```

Donc `if (ClPlusGrandQue(10)(n))`
`\----------------------/  \--`appel de `operator()` de l'objet tempo créé
objet tempo de type `ClPlusGrandQue`

est correct et aussi

```
if (ClPlusGrandQue(max)(n))
```

On peut faire maintenant faire un `TrouveSi` vraiment plus général :

```
template <typename Iter, typename Pred>
typename Iter TrouveSi(Iter p_debut, Iter p_fin, const Pred& p_ValeurDesiree)
    {
    while (p_debut != p_fin && !p_ValeurDesiree(*p_debut))
        ++p_debut;

    return p_debut;
    }
```

Et on pourra ensuite faire :

```
/*1*/     vector<int>::iterator it= TrouveSi(v.begin(), v.end(), ClPlusGrandQue(99));
```

mais aussi toujours

```
/*2*/     vector<int>::iterator it= TrouveSi(v.begin(), v.end(), ClPlusGrandQue100);
```

et même :

```
          int vec[1000];
/*3*/     int* p= TrouveSi(vec, vec+1000, ClPlusGrandQue(101));
```

Regardons les paramètres du `template` dans chacun des cas :

`/*1*/` `Iter` est le type `vector<int>::iterator`
`Pred` est le type `CPlusGrandQue`

/*2*/ `Iter` est le type `vector<int>::iterator`
`Pred` est le type pointeur sur fonction retournant `bool` et prenant `int` c'est-à-dire `bool (*)(int)`

/*3*/ `Iter` est `int*`
`Pred` est le type `ClPlusGrandQue`

Dans la fonction `TrouveSi` générée :

/*1*/ `p_ValeurDesiree` est un objet temporaire de type `ClPlusGrandQue` avec `m_borne==99`
Donc `p_ValeurDesiree(*p_debut)` appellera `tempo.operator()(*p_debut)` qui
est équivalent à `*p_debut > 99`... (`*p_debut` donne le `int` indiqué par l'itérateur.)

/*2*/ `p_ValeurDesiree` est un pointeur sur la fonction `PlusGrandQue100`...
Donc `p_ValeurDesiree(*p_debut)` appellera cette fonction qui compare avec 100.
(`*p_debut` donne le `int` indiqué par l'itérateur.)

/*3*/ `p_ValeurDesiree` est un objet temporaire de type `ClPlusGrandQue` avec `m_borne==101`
Donc `p_ValeurDesiree(*p_debut)` appellera *`tempo`*`.operator()(*p_debut)` qui
est équivalent à `*p_debut > 101`... (`*p_debut` donne le `int` pointé par ce pointeur)

Ça marche et ça permet aussi :

```
int max= ...;
vector<int>::iterator it= TrouveSi(v.begin(), v.end(), ClPlusGrandQue(max));
```

Finalement on pourrait remplacer la classe `ClPlusGrandQue` par ce template plus général :

```
template <typename T>
class ClGreaterThan
    {
    public :
    ClGreaterThan(const T& p_borne) : m_borne(p_borne) {}
    bool operator()(const T& p_n) const
        {
        return m_borne < p_n;        // test dans autre sens pour utiliser <
        }

    private :
    const T& m_borne;
    };
```

Et alors :

```
int max= ...;
vector<int>::iterator it= TrouveSi(v.begin(), v.end(), ClGreaterThan<int>(max));

double maxD= ...;
vector<double>::iterator itD= TrouveSi(vd.begin(), vd.end(), ClGreaterThan<double>(maxD));
```

Et même :

```
    ClClient c= ...;  // faut que ClClient ait défini un opérateur <
    typedef list<ClClient>::iterator ClIterClient;
    list<ClClient> clients;
    ...
    ClIterClient itC= TrouveSi(clients.begin(), clients.end(), GreaterThan<ClClient>(c));
```

La fonction `TrouveSi` existe déjà, dans `<algorithm>` et s'appelle `find_if` :

```
    int max= ...;
    deque<int> deq;
    ...
    deque<int>::iterator it= find_if(deq.begin(), deq.end(), GreaterThan<int>(max));
```

L'équivalent de `GreaterThan` existe aussi dans la bibliothèque standard... On y revient...

## Fonction objet

On appelle donc fonctions objet  les objets qu'on peut appeler comme des fonctions. Ces objets proviennent de classes, prévues à cette fin, qui fournissent la fonction `operator()` et, souvent, presque rien d'autres. On peut programmer ces classes au besoin ou les obtenir à partir de template de classes. Il existe d'ailleurs des templates de ce genre dans la bibliothèque standard, qu'on peut donc utiliser directement.

Les fonctions objets peuvent recevoir plusieurs paramètres, il faut que l'utilisation corresponde à ce qu'on a prévu. Par exemple, voici un template de classe très simple :

```
template <typename T>
class ClPlusGrand
    {
    public :
    // Le constructeur par défaut (qui ne fait rien) suffira

    bool operator()(const T& p_1, const T& p_2)
        {
        return p_2 < p_1;
        }
    };


EcrireMessageSi(10, 20, ClPlusGrand<int>());    // On crée un objet tempo du type
                                                //  ClPlusGrand, le constructeur (par
                                                //  défaut) n'a pas de paramètre
```

Avec :

```
template <typename Pred>
void EcrireMessageSi(int p_v1, int p_v2, Pred p_Condition)
    {
    if (p_Condition(p_v1, p_v2))
        cout << "Les paramètres respectent la condition.\n";
    else
        cout << "Condition non respectée.\n";
    }
```

L'avantage de cette approche, c'est qu'on peut changer le comportement de la fonction selon un « opérateur relationnel ». Par exemple, si on faisait une classe `ClPlusPetit`, on pourrait faire

```
EcrireMessageSi(10, 20, ClPlusPetit<int>());
```

On peut parfois passer un opérateur presque directement, si on a une fonction `operator`. Par exemple, si on fait une autre fonction `EcrireMessageSi`, plus générale et appelé `EcrireMessageSelonCondition` :

```
template <typename T, typename Pred>
void EcrireMessageSelonCondition(const T& p_v1, const T& p_v2, Pred p_Condition)
    {
    if (p_Condition(p_v1, p_v2))
        cout << "Les paramètres respectent la condition.\n";
    else
        cout << "Condition non respectée.\n";
    }
```

Elle pourra accepter des `ClClient, par exemple :`

```
EcrireMessageSelonCondition(client1, client2, operator<); // client1 et client2 sont des ClClient
```

Pour que ça fonctionne, il faut que l'opérateur < ait été défini en fonction globale pour `ClClient`. Si c'est une fonction membre, on ne peut pas passer simplement `operator<`. Même si l'on passait `&ClClient::operator<`, ça ne fonctionnerait pas, car il s'agirait d'une fonction à un seul opérande alors que la fonction veut lui en passer deux.

De toute façon, il n'est pas possible de passer directement l'opérateur < ou > des types de base,  `operator<` des `int` par exemple, pour la fonction `EcrireMessageSi`. Des templates de classes semblables à `ClPlusPetit`, mais pour tous les opérateurs relationnels, existent donc déjà dans la bibliothèque standard et permettent de faire ce qu'on veut, autant avec les types de base qu'avec les types définis par le programmeur.

Ces templates de classes s'appellent `equal_to`, `not_equal_to`, `greater`, `less`, `greater_equal` et `less_equal`. Par exemple, on peut écrire directement :

```
EcrireMessageSi(10, 20, not_equal_to<int>());
```

Même si `operator<` est un membre de `ClClient`, on pourra faire :

```
EcrireMessageSelonCondition(client1, client2, less<ClClient>());
```

Ici `less<ClClient>()` génère une comparaison utilisant directement `<` entre les deux clients, ce que l'opérateur membre permet de faire aussi.

Pour notre `TrouveSi`, on avait utilisé des fonctions objets en tant que condition simple sur une donnée (l'`operator()` prenait un paramètre). Ici on a des conditions entre deux données (l'`operator()` prend deux paramètres). Pour nous rendre service, la bibliothèque standard devrait nous fournir ce qu'il faut pour les conditions des deux types. Or, les templates `equal_to`, etc., prennent deux paramètres. Peut-on éviter d'avoir deux types de template de classe pour « plus grand », c'est-à-dire utiliser le « plus grand » à deux paramètres quand on a besoin de celui à un paramètre ?

Il faut comprendre qu'on ne peut pas faire :

```
it= find_if(v.begin(), v.end(), greater<int>(max));
```

car `greater` (comme notre `ClPlusGrand`) n'a pas de constructeur avec paramètre, ni de variable membre pour conserver la donnée ! De plus, l'appel de la fonction objet dans `find_if` ne passe qu'un paramètre à son `operator()` alors qu'il en faudrait deux (voir l'appel de `p_ValeurDesiree` dans notre `TrouveSi`). Ce n'est donc pas directement compatible.

Par contre, on peut fabriquer un convertisseur. D'abord une classe qui conservera une donnée et une fonction objet à deux paramètres. Elle fournira un `operator()` à un paramètre qui appellera la fonction objet avec ce paramètre et la donnée conservée (ce qui donne les deux paramètres nécessaires).

Par exemple :

```
template <typename T, typename Pred>
class ClPasser2emeParam
    {
    public :

    ClPasser2emeParam(const Pred& p_Pred, const T& p_deuxiemeParam)
        : m_Pred(p_Pred), m_valeurDu2emeParam(p_deuxiemeParam)
        {}

    bool operator()(const T& p_v)
        {
        return m_Pred(p_v, m_valeurDu2emeParam);
        }

    private :

    Pred m_Pred;
    T m_valeurDu2emeParam;
    };
```

Par la suite, on pourra déclarer :

```
ClPasser2emeParam<int, greater<int> > plusGrandQue10(greater<int>(), 10);
```

et l'utiliser ainsi :

```
it= find_if(v.begin(), v.end(), plusGrandQue10);
```

On peut aussi utiliser directement :

```
        it= find_if(v.begin(), v.end(),
                    ClPasser2emeParam<int, greater<int> >(greater<int>(), 10));
                    // c'est-à-dire qu'on passe un temporaire
```

C'est lourd et répétitif parce qu'on doit passer manuellement les types au template de la classe `ClPasser2emeParam`. On peut simplifier l'utilisation en créant un template de fonction, car ceux-ci peuvent déduire leur paramètre :

```
template <typename T, typename Pred>
ClPasser2emeParam<T, Pred> Passer2emeParam(Pred p_Pred, T p_val)
    {
    return ClPasser2emeParam<T, Pred>(p_Pred, p_val);
    }
```

Par la suite, finalement :

```
 it= find_if(v.begin(), v.end(), Passer2emeParam(greater<int>(), 10));
```

Mais `Passer2emeParam` existe déjà dans la bibliothèque standard, dans `<functional>`, et on utilisera donc plutôt :

```
it= find_if(v.begin(), v.end(), bind2nd(greater<int>(), 10));
                               // style p_ValeurDesiree(*p_debut)
                               // == greater<int>.operator()(*p_debut, 10)
                               // == *p_debut > 10, ce qu'on veut trouver
```

Aussi :

```
it= find_if(v.begin(), v.end(), bind1st(less<int>(), valeurCible));
                                // style p_ValeurDesiree(*p_debut)
                                // == less<int>.operator()(valeurCible, *p_debut)
                                // == valeurCible < *p_debut
```

Ces possibilités sont aussi utiles dans d'autres fonctions de la bibliothèque standard. Par exemple, une des deux fonctions (template) `transform` applique une opération sur chacune des données d'une étendue de données et store le résultat dans la collection spécifiée. Par exemple :

```
double DiviserParTrois(double p_valeur)
    {
    return p_valeur/3.0;
    }

vector<double> v; //...
transform(v.begin(), v.end(), v.begin(), DiviserParTrois);
                              // on remet les résultats dans le vecteur
```

Voici ce que pourrait être la fonction `transform` :

```
template <typename It1, typename It2, typename Pred>
It2 transform(It1 p_debut, It1 p_fin, It2 p_dest, Pred p_Operation)
    {
    for ( ; p_debut != p_fin; ++p_debut, ++p_dest)
        *p_dest= p_Operation(*p_debut);

    return p_dest;// peut être utile
    }
```

Mais la force de la bibliothèque de C++ est qu'on puisse faire directement :

```
transform(v.begin(), v.end(), v.begin(), bind1st(divides<double>(), 1000.0));
transform(v.begin(), v.end(), v.begin(), negate<double>());
transform(v.begin(), v.end(), v.begin(), bind2nd(plus<double>(), 5.0));
```

Les données contiendront ensuite le résultat de `-(1000/val) + 5` pour chaque valeur de la collection. On n'a pas eu besoin de « bind » avec `negate` car il prend un seul paramètre (on aurait pu mettre `bind1st(minus<double>(), 0.0)` à la place). Évidemment comme on passe par tous les éléments de la collection trois fois, il pourrait être plus rapide d'écrire une fonction ou une classe de fonction objet spécialisée et parcourir la collection une seule fois en appliquant celle-ci :

```
double CalculBizarre(double p_val)
    {
    return -(1000.0/p_val) + 5.0;
    }
...
transform(v.begin(), v.end(), v.begin(), CalculBizarre);
```

ou

```
class ClCalculBizarre
    {
    public :
    double operator()(double p_val)
        {
        return -(1000.0/p_val) + 5.0;
        }
    };

transform(v.begin(), v.end(), v.begin(), ClCalculBizarre());
```

ou (solution plus flexible)

```
class ClCalculCurieux
    {
    public :
    ClCalculCurieux(double p_dividende, double p_ajout)
        : m_dividende(p_dividende), m_ajout(p_ajout)
        {}

    double operator()(double p_val)
        {
        if (p_val == 0.0) throw "Division par zéro";
        return -(m_dividende/p_val) + m_ajout;
        }

    private :
    double m_dividende;
    double m_ajout;
    };

transform(v.begin(), v.end(), v.begin(), ClCalculCurieux(1000.0, 5.0));
```

Les `bind2nd` et `bind1st` sont appelés des *binders*. Ils permettent de créer des fonctions objets unaires à partir de fonctions objets binaires en passant un des deux paramètres selon la valeur fournie. Il existe aussi d'autres templates de modification de prédicats.

Les *negaters* `not1` et `not2`, permettent d'obtenir le résultat logique inverse d'une opération unaire ou binaire respectivement. Par exemple, pour utiliser la fonction template `count_if` :

```
class ClCommencePar // Pas tout à fait correct pour not1...
    {
    public :
    ClCommencePar(char p_c) : m_c(p_c) {}

    bool operator()(string p_s)
        {
        return p_s[0] == m_c;
        }

    private :
    char m_c;
    };


list<string> mots;
...
size_t nbMotsCommencantParI= count_if(mots.begin(), mots.end(), ClCommencePar('I'));
size_t nbMotsNeCommencantPasParM
       = count_if(mots.begin(), mots.end(), not1(ClCommencePar('M')));
```

En fait, il faut un type `argument_type` [...] dans la fonction objet pour que `not1`, puisse être appliquée [...]. Le plus simple est de dériver la classe de unary_function<...,...> [...] ou binary_function [...]:

```
class ClCommencePar : public unary_function<char, bool>
```

[...]
mem_fun ?? …
[…]