La généricité permet d'avoir des fonctions et des classes paramétrables, c'est-à-dire que, au moment où nous en avons besoin, nous précisons le type à utiliser pour ladite fonction ou ladite classe. C'est le type qui est paramétrable. Ce concept nous permettra d'avoir des écritures plus concises et ainsi d'éviter de nombreuses surdéfinitions. La généricité est souvent appelée « template » (patron – évocation de la haute couture), ou également « modèle ».

Fonctions génériques

Prenons l'exemple de la fonction qui retourne la valeur minimum de deux nombres.

```
1  double min(double x, double y)
2  {
3    return x<=y ? x : y;
4  }
5  //-----------------------------
6  int main()
7  {
8    double y = min(2.0, -3.5);
9    int z = min(2, 18);
10   return 0;
11 }
```

La difficulté, dans ce genre de situation, est de proposer les bons types à la fois pour les paramètres et pour la valeur de retour. A priori, il semble que se soit le type réel qui soit le plus adapté. En effet, le compilateur accepte les écritures prévues pour « **y** » et pour « **z** ». En prenant le type le plus haut dans la hiérarchie, nous sommes sûrs que le compilateur acceptera des écritures proposant des types dits « plus petit » puisque le langage effectue le changement de type automatique. De plus, en prenant le type le plus fort, nous n'obtenons aucune perte d'informations.

Dans le premier cas, l'utilisation est parfaitement adaptée puisque les arguments placés lors de l'appel de la fonction sont bien de type réel. Le résultat de cette fonction est compatible avec la variable « **y** » qui est elle-même de type réel.

Dans le deuxième cas, l'adéquation n'est pas parfaite puisque les arguments sont des entiers. Au moment de l'exécution, nous aurons des changements de type pour passer des types entiers vers les types réels. De même, la fonction renvoie un réel qui, lui aussi doit être transformer pour s'adapter à la variable « **z** » qui est de type entier.

Ce code fonctionne bien, mais il n'est pas très optimisé puisque nous constatons une perte de temps au moment des changements de type automatiques. Il serait alors souhaitable de proposer des fonctions qui soient adaptées à chaque situation particulière. Puisque le langage le permet, nous pouvons surdéfinir la fonction « min » en proposant une fonction spéciale pour les réels et une fonction spéciale pour les entiers.

```
1  double min(double x, double y)
2  {
3    return x<=y ? x : y;
4  }
5  //-----------------------------   Code identique
6  int min(int x, int y)
7  {
8    return x<=y ? x : y;
9  }
10 //-----------------------------
11 int main()
12 {
13   double y = min(2.0, -3.5);
14   int z = min(2, 18);
15   return 0;
16 }
```

Cette fois-ci, chaque fonction est adaptée à la situation proposée. En effet, « **y** » appelle la première fonction alors que « **z** » appelle la deuxième fonction. Si nous regardons bien, nous découvrons que le code est identique, c'est juste la signature qui diffère.

Dans le même ordre d'idée, nous pourrions écrire des fonctions pour des « long », des « long double », etc. Pour éviter tous ces cas de figure, et vu que le code interne est identique, il serait peut-être plus judicieux de proposer une fonction qui soit un modèle pour toutes les autres.

```
1  template <class Type>
2  Type min(Type x, Type y)
3  {
4    return x<=y ? x : y;
5  }
6  //-----------------------------
7  int main()
8  {
9    double y = min(2.0, -3.5);
10   int z = min(2, 18);
11   return 0;
12 }
```

« **template** » indique que nous mettons en place un nouveau modèle. Après ce mot réservé nous trouvons systématiquement les opérateurs de séparations « **< >** » à l'intérieur desquels nous spécifions les paramètres.

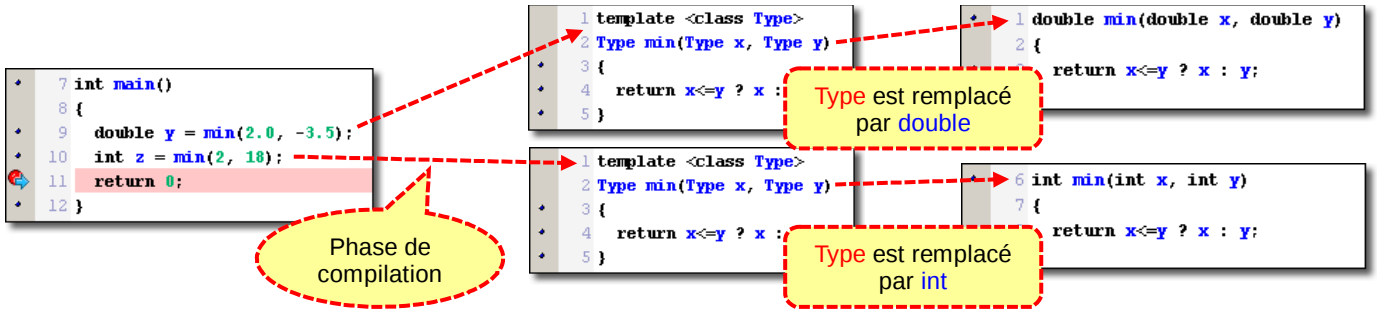
Si nous avons une liste de paramètres, nous devons utiliser l'opérateur virgule « **,** » pour les séparer. Il existe deux sortes de paramètres :

Paramètre de type : c'est le cas le plus fréquent, et c'est justement l'intérêt des modèles. Lorsque nous utilisons un tel modèle, le compilateur détermine le type et fabrique la fonction en conséquence. Dans ce cas, c'est bien le type qui est paramétrable. Pour identifier un paramètre de type, vous devez utiliser le mot réservé « **class** » (class indique qu'il s'agit d'un type) suivi du nom du paramètre qui représentera le type. Ce nom est un identificateur classique, et c'est vous qui décidez de son appellation.

Paramètre non type : qui représente une expression constante et qui est une déclaration de paramètre ordinaire du style : « **int taille** ». Nous exploiterons un exemple lorsque nous aborderons les classes génériques.

Le paramètre est utilisé dans le modèle et remplace les types prédéfinis « **double** » et « **int** » que nous avions au préalable.

Que se passe t-il réellement lorsque nous utilisons les modèles ?



Il faut bien comprendre que la transformation du modèle vers de véritables fonctions, s'effectue durant la phase de compilation, et dans le scénario proposé, le compilateur fabrique bien deux fonctions différentes. Lorsque nous lancerons le programme, c'est bien ces deux fonctions qui seront directement appelées, le modèle, à ce moment là, n'existe plus. Le modèle nous sert à composer, une fois pour toute, les lignes de codes nécessaires à l'élaboration de toutes les fonctions surdéfinies et c'est le compilateur qui finalement travaille pour nous. Cela nous évite d'écrire des lignes de codes identiques.

Remarque

Encore une fois, nous sommes en présence de l'élaboration d'un texte (paramétrable) par rapport à un autre et pour être plus précis, cette mise en place du texte s'effectue durant la phase du préprocesseur ce qui sous-entend que ***les fonctions génériques ainsi que les classes génériques devront se trouver dans un fichier en-tête***.

Fonction générique « inline »

Nous pouvons modéliser n'importe quel type de fonction et le fait qu'elle puisse être « inline » ne change absolument rien. Dans le cas de la fonction « **min** », il serait d'ailleurs judicieux de la qualifier « inline » puisque le code est extrêmement réduit.

```
2  template <class T> inline T min(T x, T y)
3  {
4    return x<=y ? x : y;
5  }
6  //-----------------------------------------
7  int main()
8  {
9    double y = min(2.0, -3.5);
10   int z = min(2, 18);
11   return 0;
12 }
```

Cette fois-ci, nous avons écris la signature du modèle sur une seule ligne. Remarquez également le changement du nom de l'identificateur du paramètre de type. C'est le programmeur qui décide du nom.

Transformation après la phase du préprocesseur.

```
7  int main()
8  {
9    double y = 2.0 <= -3.5 ? 2.0 : -3.5;
10   int z = 2 <= 18 ? 2 : 18;
11   return 0;
12 }
```

Surdéfinition de fonctions génériques

L'avantage de ces modèles c'est d'écrire très peu de ligne de codes. De plus, le modèle proposé pour trouver une valeur minimale répond à des situations très différentes. Il existe toutefois, des cas où le modèle proposé ne suffit plus. Nous pouvons avoir besoin, par exemple, de récupérer la valeur minimale d'un tableau d'entiers passé en argument, auquel cas, la définition proposée par le modèle n'est plus du tout adaptée à la situation. Il est alors nécessaire de proposer une définition particulière à ce cas d'utilisation.

Nous pouvons donc faire coexister des fonctions génériques avec des fonctions classiques. Le tout, c'est de proposer des signatures différentes pour que le compilateur soit à même de comprendre le souhait du programmeur et donc de résoudre la sur-définition.

```
2  template <class T> inline T min(T x, T y)
3  {
4    return x<=y ? x : y;
5  }
6  //-----------------------------------------
7  int min(const int tab[], unsigned taille)
8  {
9    int minimum = 2147483647;
10   for (int i=0; i<taille; i++)
11     if (tab[i]<minimum)  minimum = tab[i];
12   return minimum;
13 }
14 //-----------------------------------------
15 int main()
16 {
17   double y = min(2.0, -3.5);
18   int z = min(2, 18);
19   int tableau[] = {1, 2, 3, 4};
20   int k = min(tableau, 4);
21   return 0;
22 }
```

Utilisation de la définition particulière

Il est même possible de surcharger des fonctions génériques entre elles. Nous avons pris l'exemple d'un tableau d'entier, mais nous imaginons bien que le codage proposer puisse fonctionner tout aussi bien pour un tableau de réels, de caractères, etc. Il vient donc à l'esprit que nous pouvons finalement en faire une fonction générique plutôt que de figer le codage pour un seul cas particulier qui traite uniquement des entiers.

```
2  template <class T> inline T min(T x, T y)
3  {
4    return x<=y ? x : y;
5  }
6  //-----------------------------------------
7  template <class T>
8  T min(const T tab[], unsigned taille)
9  {
10   T minimum = 2147483647;
11   for (int i=0; i<taille; i++)
12     if (tab[i]<minimum)  minimum = tab[i];
13   return minimum;
14 }
15 //-----------------------------------------
16 int main()
17 {
18   double y = min(2.0, -3.5);
19   int z = min(2, 18);
20   int tableau[] = {1, 2, 3, 4};
21   int k = min(tableau, 4);
22   return 0;
23 }
```

Nous transformons la fonction pour la rendre générique. Remarquez que l'identificateur de type porte le même nom que la première fonction générique. Il n'existe pas de conflit de nom puisqu'il s'agit d'un paramètre local à la fonction et dont la portée est limitée à cette dernière.

Partout, nous avons remplacé le type « int » par le type paramétrable.

Avec très peu de lignes de code, nous gérons déjà une très grande diversité de situations.

Depuis le C++11, nous pouvons utiliser le terme typename (nom du type) à la place du terme class. Ces deux appellations sont similaires.

Gestion de types différents pour une fonction avec plusieurs paramètres

Jusqu'à présents, pour cette fonction ***min()***, tous les arguments étaient de la même nature, test sur des entiers, test sur des réels, etc. Que se passe-t-il lorsque nous proposons deux arguments de types différents ? Testez le code suivant :

principal.cpp

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

template <typename Type>
inline Type minimum(const Type& x, const Type& y) { return x<=y ? x : y; }

int main()
{
  cout << minimum(3, -2) << endl;
  cout << minimum(3, 2.5) << endl; // erreur de compilation
  return 0;
}
```

Une erreur de compilation se produit lorsque nous tentons de connaître le minimum entre une valeur entière et une valeur réelle. Le compilateur est incapable de choisir entre fabriquer une fonction qu'avec des entiers et une autre fonction qu'avec des réels. Ce n'est pas à lui de choisir. C'est le programmeur qui doit le faire.

Cette écriture n'est donc pas correcte puisqu'elle ne prend pas en compte toutes les situations. Dans la fabrication de ce modèle de fonction, il est donc nécessaire de se dire qu'il peut y avoir des arguments de type différents (ils peuvent aussi être identiques).

principal.cpp

```
template <typename Tx, typename Ty>
inline Tx minimum(const Tx& x, const Ty& y) { return x<=y ? x : y; }

int main()
{
  cout << minimum(3, -2) << endl;
  cout << minimum(3, 2.5) << endl; // plus d'erreur de compilation, mais résultat erroné
  return 0;
}
```

*Cette fois-ci nous envisageons d'utiliser deux paramètres de type qui vont nous permettre de résoudre le problème rencontré. Le problème dans ce cas est de choisir le type de retour. Dans l'exemple ci-dessus, j'ai pris le paramètre **Tx**, mais nous aurions pu choisir l'autre paramètre. Du coup, la deuxième fonction **minimum()** renvoie la valeur « 2 », et non « 2,5 », puisque le type de retour correspond au type du premier argument qui est un entier (transtypage d'un réel vers un entier).*

*Finalement, notre solution n'en est pas une. Il faudrait que le compilateur détermine automatiquement le type de retour suivant le résultat du test ; dans notre exemple soit le type **int**, soit le type **double**.*

L'inférence de type : mot clé auto – C++11 et C++14

Jusqu'à présent, pour déclarer une variable en C++, il était obligatoire de spécifier son type. Cependant, avec l'arrivée des templates, cela n'est pas toujours aisé, notamment pour les valeurs de retour des fonctions templates.

*Le C++11 a introduit la déduction de types (inférence de types) sous deux formes : **auto** et **decltype()** (que nous aborderons utltérieurement). Cette fonctionnalité permet ainsi de laisser le compilateur choisir le type d'un élément en fonction de ce qui lui est assigné. Nous pouvons maintenant déclarer une variable sans en connaître le type, à condition toutefois de l'initialiser explicitement :*

```
  int i = 12;
  auto j = i;       // j est entier puisque i est entier
  auto k = -15;     // k est également entier vu la constante littérale entière
  auto l = i + 2.3; // l est réel vu la constante littérale réelle
```

*Dans le code ci-dessus, ce mot réservé possède très peu d'intérêt. Il faut d'ailleurs éviter de l'utiliser pour ces cas de figure. Il vaut mieux connaître exactement les types que nous manipulons sans avoir besoin de réfléchir ou de calculer les expressions. Par contre ce mot clé va nous permettre de résoudre notre problématique sur cette fonction générique **minimum()** :*

principal.pp

```
template <typename Tx, typename Ty>
inline auto minimum(const Tx& x, const Ty& y) { return x<=y ? x : y; }

int main()
{
  cout << minimum(3.7, -2) << endl; // résultat correct
  cout << minimum(3, 2.5) << endl;  // résultat correct
  return 0;
}
```

*auto agit de la même façon que les templates, Il doit résoudre le type durant la phase de compilation. Ainsi, nous pouvons encore simplifier notre écriture en proposant le mot clé auto en paramètres de la fonction **minimum()**. Comme pour les templates, autant de fonctions seront générées suivant le nombre de cas différents d'utilisation : **(Il faut le compilateur G++6 pour que cela fonctionne)***

```
inline auto minimum(const auto& x, const auto& y) { return x<=y ? x : y; }  // Il faut le compilateur G++6

int main()
{
  cout << minimum(3.7, -2) << endl; // résultat correct
  cout << minimum(3, 2.5) << endl;  // résultat correct
}
```

Classes génériques - Conception

```
4  class TabInt
5  {
6    int *tableau;
7    unsigned taille;
8  public:
9    TabInt(unsigned taille=10);
10   ~TabInt();
11   TabInt(const TabInt &);
12   TabInt& operator= (const TabInt &);
13   int& operator[] (unsigned indice);
14 };
```

Tableau d'entiers

Les classes aussi peuvent être génériques. Dans la leçon précédente, nous avons mis en œuvre une classe représentant les tableaux d'entiers. Cette classe nous a permis de créer un véritable tableau qui tenait compte à la fois de l'affectation directe de tout le contenu et également d'un accès particulier à l'une de ses cases. Malgré tout, l'effort que nous avons fourni n'est pas suffisamment récompenser puisque ce tableau est codé seulement pour les entiers.

Lorsque nous consultons le code interne, nous remarquons que nous avons très peu de référence au type entier, et surtout, nous pouvons le remplacer par un autre type pour retrouver un fonctionnement identique pour d'autres types de tableau. Dans ce cas, il est plus avantageux de proposer un tableau générique, et c'est l'utilisateur qui décidera le type qu'il désire.

```
4  template <class Type> class Tableau
5  {
6    Type *contenu;
7    unsigned taille;
8  public:
9    Tableau(unsigned taille=10);
10   ~Tableau();
11   Tableau(const Tableau &);
12   Tableau& operator= (const Tableau &);
13   Type& operator[] (unsigned indice);
14 };
```

Tableau de types quelconques

La syntaxe demeure identique à l'écriture des fonctions génériques. Partout où le type « int » est utilisé, vous le remplacez par le paramètre « Type » puisque c'est lui qui est défini dans le modèle.

Mise à part l'écriture de la partie paramétrable, les classes génériques demandent très peu d'investissement supplémentaire. Il ne faut pas hésiter à utiliser cette technique.

Classe générique - Utilisation

```
5  int main()
6  {
7    Tableau<int> tab1(3); // tableau de 3 entiers
8    tab1[0] = -5;    // changement de la première case
9    Tableau<double> tab2; // tableau de 10 réels
10   Tableau<int> tab3;    // tableau de 10 entiers
11   Tableau<Complexe> tab4(5);    // tableau de 5 complexes
12   return 0;
13 }
```

Il existe toutefois une petite différence dans l'utilisation des classes génériques par rapport aux fonctions génériques.

Le compilateur, au moment de l'appel d'une fonction générique, contrôle la signature proposée et détermine le type demandé pour effectivement fabriquer la fonction avec le type désiré. Sans spécification supplémentaire, le compilateur arrive à connaître le type demandé.

Dans le cas d'un objet, c'est différent. Lorsque nous déclarons cet objet, le type de certains attributs n'apparaît pas au moment de la déclaration puisque seul le nom de l'objet est visible. En conséquence, il est nécessaire, durant la création d'indiquer le ou les types voulus. La syntaxe est simple d'utilisation. Après le nom de la classe, vous devez préciser le type voulu entre « <> ». Le nom de votre type devient alors l'argument de votre classe générique. Du coup, la syntaxe de votre code est très lisible, en ce sens que nous voyons bien qu'il s'agit, par exemple, d'un tableau d'entiers ou d'un tableau de complexes.

Remarque

L'utilisation des « < > » est impérative dans le cas des classes génériques au même titre que l'utilisation des parenthèses est impérative pour les fonctions et les méthodes. C'est ce qui permet d'ailleurs de les reconnaître.

Classe générique – Définition des méthodes

```
5  int main()
6  {
7    Tableau<int> tab1(3); // tableau de 3 entiers
8    tab1[0] = -5;    // changement de la première case
9    Tableau<double> tab2; // tableau de 10 réels
10   Tableau<int> tab3;    // tableau de 10 entiers
11   Tableau<Complexe> tab4(5);    // tableau de 5 complexes
12   return 0;
13 }
```

```
15 //----------------------------------------------------
16 template <class Type>
17 inline Type& Tableau<Type>::operator[] (unsigned indice)
18 {
19   return contenu[indice];
20 }
21 //----------------------------------------------------
22 template <class Type>
23 inline Tableau<Type>::~Tableau ()
24 {
25   delete[] contenu;
26 }
27 //----------------------------------------------------
28 template <class Type>
29 inline Tableau<Type>::Tableau(unsigned taille)
30 {
31   contenu = new Type[this->taille = taille];
32 }
33 //----------------------------------------------------
```

```
27 //----------------------------------------------------
28 inline Tableau<int>::Tableau(unsigned taille)
29 {
30   contenu = new int[this->taille = taille];
31 }
32 //----------------------------------------------------
```

```
27 //----------------------------------------------------
28 inline Tableau<double>::Tableau(unsigned taille)
29 {
30   contenu = new double[this->taille = taille];
31 }
32 //----------------------------------------------------
```

D'habitude, lorsque nous développons des classes, nous plaçons la déclaration de la classe dans un fichier en-tête alors que la définition de ses méthodes se trouve dans le fichier source correspondant qui porte le même nom, mais dont l'extension de fichier est « *.cpp ».

Dans le cas d'une classe générique, c'est différent. N'oubliez pas qu'il s'agit d'un prototype (d'un patron) qui servira à la fabrication (réelle) de plusieurs classes de types différents. Dans ce contexte, la déclaration de la classe ainsi que la définition des méthodes doivent se trouver entièrement dans le fichier en-tête.

En effet, tout ce qui est générique est utilisé uniquement par le « préprocesseur » du compilateur, et n'oubliez pas que cette phase particulière de la compilation propose de transformer un texte par un autre. Ce n'est que lorsque le texte a été mis en place que la compilation réelle s'effectue.

En revenant sur notre exemple, à l'utilisation, nous avons besoin d'avoir d'une part un tableau d'entiers et d'autre par un tableau de réels. Bien que le code interne soit identique, il existe tout de même des différences. Nous ne traitons pas des réels comme des entiers. Il faut donc qu'il y ait, par exemple, un constructeur pour un tableau d'entier et un constructeur pour un tableau de réel puisque la réservation mémoire dynamique est totalement différente. Les réels prennent plus de place en mémoire. Pour un tableau de complexe, c'est encore pire.

Du coup, la généricité de la définition des méthodes doit être pris en compte au même titre que la généricité de la déclaration de la classe dans son entier. En effet, de même que nous devons qualifier la méthode par le nom de la classe à laquelle elle appartient. De même, nous devons spécifier qu'il s'agit bien d'une méthode générique qui va manipuler des types différents. Finalement, les méthodes sont également paramétrées. Dans ce contexte, à la définition de chaque méthode, vous devez impérativement utiliser la syntaxe complète des « templates ».

Une fois que nous sommes dans la méthode, il n'est plus nécessaire de spécifier le paramètre de type lorsque nous faisons référence à la classe. C'est le cas du paramètre de la méthode puisqu'il s'agit d'une variable locale. Par contre, le retour de la méthode ne fait pas parti de la signature. Dans ce cas, nous devons spécifier le paramètre de type.

```
33 //-----------------------------------------------------------
34 template <class Type>
35 Tableau<Type>::Tableau(const Tableau &tab)
36 {
37   contenu = new Type[taille = tab.taille];
38   for (unsigned i=0; i<taille; i++)
39     contenu[i] = tab.contenu[i];
40 }
41 //-----------------------------------------------------------
42 template <class Type>
43 Tableau<Type>& Tableau<Type>::operator= (const Tableau &tab)
44 {
45   delete[] contenu;
46   contenu = new Type[taille = tab.taille];
47   for (unsigned i=0; i<taille; i++)
48     contenu[i] = tab.contenu[i];
49   return *this;
50 }
51 //-----------------------------------------------------------
```

Reste des définitions des méthodes de la classe "Tableau" dans le fichier en-tête

Classe générique – Paramètre non type

Nous avons vu qu'il existait deux types de paramètres pour les fonctions et les classes génériques : les paramètres de type, et les paramètres non type.

Dans le cas des classes génériques, le paramètre « non type » peut s'avérer particulièrement utile, notamment pour implémenter un tableau de type quelconque en spécifiant, dès le départ, la dimension du tableau, et c'est ce dernier élément qui servira de paramètre non type. En effet, dans ce cas là, le paramètre attend une valeur entière non signée. Il ne s'agit en aucun cas de proposer un type mais bien une valeur.

```
1 #ifndef Tableau_H
2 #define Tableau_H
3 //-----------------------------------------------------------
4 template <class Type, unsigned taille> class Tableau
5 {
6   Type contenu[taille];
7 public:
8   Type& operator[] (unsigned indice) { return contenu[indice]; }
9 };
10 //-----------------------------------------------------------
11 #endif
```

Paramètre non type qui sert de dimension au tableau, donc sans variable dynamique

```
3  int main()
4  {
5    Tableau<int, 3> tab1; // tableau de 3 entiers
6    tab1[0] = -5;    // changement de la première case
7    Tableau<int, 3> tab2 = tab1; // construction de copie
8    Tableau<int, 3> tab3;    // tableau de 3 entiers
9    tab3 = tab1;     // affectation
10   tab3[2] = 3;     // changement de la dernière case
11   return 0;
12 }
```

```
(Tableau<int, 3>) tab1 = { { -5, 256, 1 } }
(Tableau<int, 3>) tab2 = { { -5, 256, 1 } }
(Tableau<int, 3>) tab3 = { { -5, 256, 3 } }
```

En prenant un paramètre non type pour spécifier la dimension, le code de la classe devient extrêmement simple. En effet, comme la dimension est connue en moment de la création de la classe, il est possible de mettre comme attribut un tableau et non plus un pointeur vers une variable dynamique. Du coup, nous n'avons plus besoin de redéfinir tous les comportements par défaut. Il suffit de redéfinir l'opérateur de crochets « [ ] ».

```
3  //-----------------------------------------------------------
4  template <class Type=int, unsigned taille=10> class Tableau
5  {
6    Type contenu[taille];
7  public:
8    Type& operator[] (unsigned indice) { return contenu[indice]; }
9  };
10 //-----------------------------------------------------------
```

Valeurs par défaut

```
5  int main()
6  {
7    Tableau<int, 3> tab1;       // tableau de 3 entiers
8    Tableau<Complexe, 7> tab2;  // tableau de 7 compexes
9    Tableau<double> tab3;       // tableau de 10 réels
10   Tableau<> tab4;             // tableau de 10 entiers
11   return 0;
12 }
```