Les Templates en C++ :

Template signifie « **patron** ».

Le mécanisme des templates permet la génération de fonctions et de classes basées sur les types paramétrés.

Les templates sont des patrons de classes ou de fonctions et permettent de créer classes et fonctions dont le **type** de certaines variables, données membres ou arguments sont **parametrables**, choisis lors de l'instanciation ou utilisation de ceux-ci.

Ces mécanismes permettent de développer du code générique réutilisable.

Il y a deux sortes de templates : les templates de fonctions et les templates de classes.

Attention, le code réel utilisé est défini lors de la compilation des sources et ne peut être chargé dynamiquement. Les templates ne permettent pas la ligature dynamique.

A - Templates de fonctions :

```
// Echange des entiers

void SwapInt(int *ia , int *ib)
{
int ic;
ic = *ia; *ia = *ib; *ib = ic;
}
```

Cette fonction manipule des entiers. Supposons que nous voulions permuter des réels. Nous écrirons donc :

```
// Echange des floats

void SwapFloat(float fa, float fb)
{
float fc;
fc = *fa;
*fa = *fb;
*fb = fc;
}
```

Tout ceci implique l'écriture d'une fonction '**SwapNomType'** pour tout type numérique.

Les templates permettent de contourner le problème en offrant la possibilité d'utiliser un type générique **T** qui sera remplacé par un type numérique lors de la compilation de la fonction utilisant le template.

Les mots clés du C++ sont :

template	=>	indique au compilateur l'utilisation d'un type paramétré.
typename ou **class**	=>	doit précédé le nom générique du type paramétré

//exemple : échange de variables

```
//template<typename T>
template <class T>

void SwapT(T *ta, T *tb)
{
T tc;
tc = *ta;
*ta = *tb;
*tb = tc;
}
```

Informe le compilateur de l'utilisation d'un type générique nommé **T** en utilisant les mots clés **template**, **typename** et **class**

Les variables ta et tb sont des pointeurs sur un type **T** non défini ici.
Le type réel des variables sera défini lors de l'utilisation de la fonction.
T pourra être int, double, float …

Utilisation de la fonction.

```
int i  = 5; int j = 6;
cout  << "i " << i << " j " << j << endl;
SwapT(&i,&j);
cout  << " i " << i << " j " << j << endl;

double f = 5.55, g = 6.66;
cout  << " f " << f << " g " << g << endl;
SwapT(&f,&g);
cout  << " f " << f << " g " << g << endl;
```

Le choix du type des données est défini ici par les lignes :

```
int i  = 5; int j = 6;
double f = 5.55, g = 6.66;
```

Questions :

- Que. se passe-t-il si la première ligne devient : double i = 5.1; int j = 6 ; ?
- Comment résoudre le problème?
- Que se passe-t-il si i et j sont de type composite ? par exemple “vecteur”?

B - Templates de classe

Supposons que nous voulions une classe vertex (sommets pour figure 3d => 3 coordonnées x,y,z pouvant être des float, entier ou double) capable de balayer l'ensemble des types de **base**. Ainsi :

```
template < typename T > //typename peut être remplacé par class
class CVertex
{
        T x, y, z;                          //3 données membres de type non défini
public:
CVertex (T inx = 0, T iny = 0, T inz = 0)
        {
                x = inx;
                y = iny;
                z = inz;
        }
void vSetCVertex (T inx, T iny, T inz);
void vDisplayCVertex ();
};
//-------------------------------------------------------------
template < typename T > //à répéter à chaque nouvelle méthode
void CVertex < T >::SetCVertex (T inx, T iny, T inz)
{
        x = inx;
        y = iny;
        z = inz;
}
//-------------------------------------------------------------
template < typename T >
void CVertex < T >::DisplayCVertex ()
{
        cout << "x = " << x << " y = " << y << " z = " << z << endl;
}
```

Maintenant, nous pouvons instancier cette classe :

```
CVertex < int >ivertexa;
ivertexa.DisplayCVertex ();
ivertexa.SetCVertex (1, 2, 3);
ivertexa.DisplayCVertex ();

CVertex < float >ivertexb;
ivertexb.DisplayCVertex ();
ivertexb.SetCVertex (1.111, 2.222, 3.333);
ivertexb.DisplayCVertex ();
```

TD : Réaliser le travail demandé

- Compléter le code suivant afin d'empiler des variables de type quelconque.

```
struct data
{
int a;
int b;
int x[4];
};
//-------------------------------------------------------------------------
template <class T>
class pile
{
public :
        int index;
                            //decalration du tableau de données
        pile();
                            //methode empile à définir
                            //méthode dépile à définir
};
//---------------------------------------
template <class T>
                        //constructeur
{
index=0;
}
//---------------------------------------
template <class T>
                        //prototype de la méthode empile
{
                        //empilage d'une donnée
}
//---------------------------------------
template <class T>
T pile<T>::depile()
{
return tab[--index];
}
//---------------------------------------



int main(int argc, char* argv[])
{
pile<int> p;
p.empile(5555);
printf("taille objet=%d\n",sizeof(p));
pile<char> p1;
p1.empile('a');
printf("taille objet=%d\n",sizeof(p1));
data d;
d.a = 10000;
d.b = 20000;
d.x[0]=30000;
pile<data> p2;
p2.empile(d);
printf("taille objet=%d\n",sizeof(p2));
getch();
return 0;
}
```

- Ecrire le résultat d'exécution de ce programme.

Corrigé du TD :

```
struct data
{
int a;
int b;
int x[4];
};
//---------------------------------------
template <class T>
class pile
{
public :
        int index;
        T tab[2];
        pile();
        void empile(T val);
        T depile();
};
//---------------------------------------
template <class T>
pile < T >::pile()
{
index=0;
}
//---------------------------------------
template <class T>
void pile<T>::empile(T val)
{
tab[index ++]=val;
}
//---------------------------------------
template <class T>
T pile<T>::depile()
{
return tab[--index];
}
//---------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{
pile<int> p;
p.empile(5555);
printf("taille objet=%d\n",sizeof(p));
pile<char> p1;
p1.empile('a');
printf("taille objet=%d\n",sizeof(p1));
data d;
d.a = 10000;
d.b = 20000;
d.x[0]=30000;
pile<data> p2;
p2.empile(d);
printf("taille objet=%d\n",sizeof(p2));
getch();
return 0;
}
```

Résultat :

```
taille objet=12
taille objet=8
taille objet=52
```

TP :

E.1] Vérifier les résultats de l'exercice précédent.

E.2] Un système de chaîne de production utilise l'analyse d'images issues d'une caméra pour contrôler les dimensions des pièces.
Le traitement des images est réalisé à l'aide de classes de gestion d'images notamment pour gérer les pixels .
On rappelle :

- une image est formée par un ensemble de pixels codés de plusieurs façons possibles.

Ecrire une classe template **pixel** permettant de stocker la valeur du pixel ainsi que sa position. Le codage du pixel étant variable selon les formats d'images sera représenté par un type paramètré.
La classe doit pouvoir afficher la taille d'un objet ainsi que la valeur des données membres.
L'opérateur **'='** doit être surchargé afin de mettre d'affecter une nouvelle couleur en écrivant par exemple:

```
pixel pix (1,2,0xfgfgfg) ;
pix = 0   // le pixel devient noir sans changer sa position.
```

Ecrire et tester une fonction qui remet à 0 toutes variables de type numérique et met à zéro (noir) la couleur d'un objet 'pixel' (voir surdéfinition de l'opérateur '=').

corrige du pixel

```
//--------------------------------------
void cimgtest()
{
  CImg< char >  img("im.bmp");//le parametre template indique le nombre format des
pixels
  CImgDisplay disp(img,"CImg Demo Menu",0);
  disp.display(img);
  getch();
}
```