

# C++

## Templates

typename T::value

template <int N>

template <typename T>

000

## Exemple de templates

```
template <typename T>
struct vec2
{
    vec2(T x_arg, T y_arg) : x(x_arg), y(y_arg) {}
    T x, y;
};

template <typename T>
std::string to_string(vec2<T> const& v)
{
    std::stringstream str;
    str<<v.x<<" "<<v.y;
    return str.str();
}

int main()
{
    vec2<float> v0(1.2f, 3.3f);
    vec2<double> v1(1.4, 5.5);

    std::cout<<to_string(v0)<<std::endl;
    std::cout<<to_string(v1)<<std::endl;
}
```

Une classe  
template

Une fonction  
template

Utilisation  
de template

001

## Paramètre template entier

- Le paramètre template n'est pas forcément un type.
- Il peut être un entier.

```
template <int N>
std::vector<int> n_premier_entiers()
{
    std::vector<int> v(N);
    for(int k=0; k<N; ++k)
        v[k]=k;
    return v;
}

int main()
{
    std::vector<int> v=n_premier_entiers<10>();
    for(int k=0; k<10; ++k)
        std::cout<<v[k]<<std::endl;
}
```

Paramètre connu  
à la compilation

002

## Paramètre template entier

Intérêt: Calcul réalisé par le compilateur, pas en run-time  
temps de calcul = 0 !

```
template <int N>
int static_square()
{
    return N*N;
}

int main()
{
    const int a=static_square<5>();
    float tableau[static_square<3>()];

    std::cout<<a<<std::endl;
    std::cout<<sizeof(tableau)/sizeof(float)<<std::endl;
}
```

tableau statique  
initialisé à partir d'un  
calcul

003

## Calcul par le compilateur

En C++11, on peut définir des constexpr  
= expressions évalués par le compilateur

```
constexpr int static_square(int N)
{
    return N*N;
}
```

Le compilateur vérifie qu'il peut évaluer l'expression  
Elle ne prendra aucun temps de calcul

004

## Calcul par le compilateur

```
constexpr int static_square(int N){return N*N;}
int square(int N){return N*N;}

template <int N>
void affiche()
{
    std::cout<<N<<std::endl;
}

int main()
{
    affiche<static_square(5)>(); OK
    affiche<square(5)>(); KO
}
```

*expression constante* (pointing to static\_square(5))  
*expression variable* (pointing to square(5))

005

## Meta-programmation

Meta-programmation = réaliser des calculs en amont du code.  
Ici, le compilateur réalise les calculs.

```
constexpr int factoriel(int N)
{
    return N<=1 ? 1 : N*factoriel(N-1);
}

template <typename T,int N>
struct vecN{T data[N];};

int main()
{
    vecN<float,factoriel(4)> v;
    for(int k=0;k<factoriel(4);++k)
        v.data[k]=k;

    std::cout<<v.data[23]<<std::endl;
}
```

Calcul de N!  
Ne prend aucun temps de calcul (pointing to factoriel(4))

Utilisation possible pour  
des paramètres template (pointing to factoriel(4) in the struct definition)

006

## Exemple de classes template

Vecteur de données génériques  
taille générique connue à la compilation.

```
template <typename TYPE,int SIZE>
class vecN
{
public:
    TYPE& operator[](int index);
    TYPE const& operator[](int index) const;

private:
    TYPE data[SIZE];
};

template <typename TYPE,int SIZE>
std::ostream& operator<<(std::ostream& str, vecN<TYPE,SIZE> const& v);
```

007

## Exemple de classes template

```
template <typename TYPE,int SIZE>
class vecN
{
public:
    TYPE& operator[](int index);
    TYPE const& operator[](int index) const;

private:
    TYPE data[SIZE];
};

template <typename TYPE,int SIZE>
std::ostream& operator<<(std::ostream& str, vecN<TYPE,SIZE> const& v);
```

### Implémentation

```
template <typename TYPE,int SIZE>
TYPE& vecN<TYPE,SIZE>::operator[](int index)
{
    if(index<0 || index>=SIZE)
        throw std::out_of_range(__FUNCTION__);
    return data[index];
}

template <typename TYPE,int SIZE>
TYPE const& vecN<TYPE,SIZE>::operator[](int index) const
{
    if(index<0 || index>=SIZE)
        throw std::out_of_range(__FUNCTION__);
    return data[index];
}
```

gestion d'erreur  
par exception

**Attention: Le corps des templates doivent être dans les fichiers d'en-tête.**

008

## Exemple de classes template

```
template <typename TYPE,int SIZE>
class vecN
{
public:
    TYPE& operator[](int index);
    TYPE const& operator[](int index) const;

private:
    TYPE data[SIZE];
};

template <typename TYPE,int SIZE>
std::ostream& operator<<(std::ostream& str, vecN<TYPE,SIZE> const& v);
```

### Implémentation

```
template <typename TYPE,int SIZE>
TYPE& vecN<TYPE,SIZE>::operator[](int index)
{
    if(index<0 || index>=SIZE)
        throw std::out_of_range(__FUNCTION__);
    return data[index];
}

template <typename TYPE,int SIZE>
TYPE const& vecN<TYPE,SIZE>::operator[](int index) const
{
    if(index<0 || index>=SIZE)
        throw std::out_of_range(__FUNCTION__);
    return data[index];
}
```

gestion d'erreur  
par exception

**Attention: Le corps des templates doivent être dans les fichiers d'en-tête.**

009

## Exemple de classes template

```
template <typename TYPE,int SIZE>
class vecN
{
public:
    TYPE& operator[](int index);
    TYPE const& operator[](int index) const;

private:
    TYPE data[SIZE];
};

template <typename TYPE,int SIZE>
std::ostream& operator<<(std::ostream& str, vecN<TYPE,SIZE> const& v);
```

```
template <typename TYPE,int SIZE>
TYPE& vecN<TYPE,SIZE>::operator[](int index)
{
    if(index<0 || index>=SIZE)
        throw std::out_of_range(__FUNCTION__);
    return data[index];
}

template <typename TYPE,int SIZE>
TYPE const& vecN<TYPE,SIZE>::operator[](int index) const
{
    if(index<0 || index>=SIZE)
        throw std::out_of_range(__FUNCTION__);
    return data[index];
}
```

### Affichage par std::cout<<

```
template <typename TYPE,int SIZE>
std::ostream& operator<<(std::ostream& str, vecN<TYPE,SIZE> const& v)
{
    for(int k=0;k<SIZE;++k)
        str<<v[k]<<" ";
    return str;
}
```

010

## Exemple de classes template

```
template <typename TYPE,int SIZE>
class vecN
{
public:
    TYPE& operator[](int index);
    TYPE const& operator[](int index) const;

private:
    TYPE data[SIZE];
};

template <typename TYPE,int SIZE>
std::ostream& operator<<(std::ostream& str, vecN<TYPE,SIZE> const& v);
```

```
template <typename TYPE,int SIZE>
TYPE& vecN<TYPE,SIZE>::operator[](int index)
{
    if(index<0 || index>=SIZE)
        throw std::out_of_range(__FUNCTION__);
    return data[index];
}

template <typename TYPE,int SIZE>
TYPE const& vecN<TYPE,SIZE>::operator[](int index) const
{
    if(index<0 || index>=SIZE)
        throw std::out_of_range(__FUNCTION__);
    return data[index];
}
```

```
template <typename TYPE,int SIZE>
std::ostream& operator<<(std::ostream& str, vecN<TYPE,SIZE> const& v)
{
    for(int k=0;k<SIZE;++k)
        str<<v[k]<<" ";
    return str;
}
```

```
int main()
{
    vecN<float,3> v0;
    v0[0]=1.4f; v0[1]=2.4f; v0[2]=7.6f;

    vecN<long double,4> v1;
    v1[0]=1.78L;v1[1]=4.4L;v1[2]=7.6L;v1[3]=-1.2L;

    vecN<vecN<int,2>,3> v2;
    v2[0][0]=1;v2[0][1]=2;
    v2[1][0]=2;v2[1][1]=5;
    v2[2][0]=3;v2[2][1]=4;

    std::cout<<v0<<std::endl;
    std::cout<<v1<<std::endl;
    std::cout<<v2<<std::endl;
}
```

### Exemple d'utilisation

011

## Méthodes template

Il est possible d'avoir des méthodes avec d'autres paramètres template dans une classe générique.

```
template <typename TYPE,int SIZE>
class vecN
{
public:
    TYPE& operator[](int index);
    const TYPE& operator[](int index) const;

    template <typename TYPE2>
    vecN<TYPE,SIZE>& operator+=(TYPE2 const& vector);

private:
    TYPE data[SIZE];
};
```

### Utilisation

```
vecN<float,3> v0;
v0[0]=1.4f; v0[1]=2.4f; v0[2]=7.6f;

std::vector<float> vec={1.2,0.5,3.1};

v0+=vec;
std::cout<<v0<<std::endl;
```

*vecN peut s'additionner avec tout type de vecteur*

```
template <typename TYPE,int SIZE> 2 niveaux de templates
template <typename TYPE2>
vecN<TYPE,SIZE>& vecN<TYPE,SIZE>::operator+=(TYPE2 const& vector)
{
    for(int k=0;k<SIZE;++k)
        data[k]+=vector[k];
    return *this;
}
```

*implémentation*

012

## Problème de déduction de type

On souhaite créer une fonction de produit scalaire générique

```
template <typename TYPE_INPUT,typename TYPE_OUTPUT,int SIZE>
TYPE_OUTPUT dot(const TYPE_INPUT& arg0,const TYPE_INPUT& arg1)
{
    TYPE_OUTPUT val;
    for(int k=0;k<SIZE;++k)
        val += arg0[k]*arg1[k];

    return val;
}

int main()
{
    vecN<float,3> v0;
    v0[0]=1.4f; v0[1]=2.4f; v0[2]=7.6f;

    vecN<float,3> v1;
    v1[0]=2.8f; v1[1]=4.4f; v1[2]=2.6f;

    float projection=dot<vecN<float,3>,float,3>(v0,v1);

    std::cout<<projection<<std::endl;
}
```

Il serait intéressant de pouvoir déduire les types de sortie et la taille du paramètre d'entrée  
dot(v0,v1) : beaucoup plus lisible

013

## Accès aux paramètres template

On ajoute un accès public aux paramètres template

```
template <typename TYPE,int SIZE>
class vecN
{
public:
    //accès aux paramètres template
    typedef TYPE TYPE_VALEUR;
    static constexpr int size() {return SIZE;}

    TYPE& operator[](int index);
    const TYPE& operator[](int index) const;

private:
    TYPE data[SIZE];
};
```

typedef redefinit un type.

- vecN::TYPE\_VALEUR contient TYPE
- vecN::size() renvoie la taille connue à la compilation

014

## Accès aux paramètres template

```
template <typename TYPE,int SIZE>
class vecN
{
public:
    //accès aux paramètres template
    typedef TYPE TYPE_VALEUR;
    static constexpr int size() {return SIZE;}

    TYPE& operator[](int index);
    const TYPE& operator[](int index) const;

private:
    TYPE data[SIZE];
};
```

```
template <typename T>
typename T::TYPE_VALEUR dot(T const& arg0,T const& arg1)
{
    typename T::TYPE_VALEUR val;
    for(int k=0,N=T::size();k<N;++k)
        val += arg0[k]*arg1[k];

    return val;
}
```

typename obligatoire  
: type défini à partir d'un template

Utilisation très simple  
déduction automatique des types

```
int main()
{
    vecN<float,3> v0;
    v0[0]=1.4f; v0[1]=2.4f; v0[2]=7.6f;

    vecN<float,3> v1;
    v1[0]=2.8f; v1[1]=4.4f; v1[2]=2.6f;

    float projection=dot(v0,v1);

    std::cout<<projection<<std::endl;
}
```

015

# Type traits

## Vocabulaire:

Les caractéristiques associées à une classe template sont appelés les *traits*

Ils peuvent être contenus dans une classe séparée

Ex. La classe traits d'iterator

**iterator\_traits** <iterator>

```
template <class Iterator> class iterator_traits;
template <class T> class iterator_traits<T>;
template <class T> class iterator_traits<T*>;
template <class T> class iterator_traits<const T*>;
```

**iterator\_traits**

Traits class defining properties of iterators.

Standard algorithms determine certain properties of the iterators passed to them and the range they represent by using the members of the corresponding `iterator_traits` instantiation.

For every iterator type, a corresponding specialization of `iterator_traits` class template shall be defined, with at least the following member types defined:

| member                         | description                                                                                                                                                                                                                                                                                                             |
|--------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>difference_type</code>   | Type to express the result of subtracting one iterator from another.                                                                                                                                                                                                                                                    |
| <code>value_type</code>        | The type of the element the iterator can point to.                                                                                                                                                                                                                                                                      |
| <code>pointer</code>           | The type of a pointer to an element the iterator can point to.                                                                                                                                                                                                                                                          |
| <code>reference</code>         | The type of a reference to an element the iterator can point to.                                                                                                                                                                                                                                                        |
| <code>iterator_category</code> | The iterator category. It can be one of these: <ul style="list-style-type: none"> <li><code>input_iterator_tag</code></li> <li><code>output_iterator_tag</code></li> <li><code>forward_iterator_tag</code></li> <li><code>bidirectional_iterator_tag</code></li> <li><code>random_access_iterator_tag</code></li> </ul> |

Note: For output iterators that are not at least forward iterators, any of these member types (except for `iterator_category`) may be defined as void.

The `iterator_traits` class template comes with a default definition that obtains these types from the iterator type itself (see below). It is also specialized for pointers (`T*`) and pointers to const (`const T*`).

Note that any custom class will have a valid instantiation of `iterator_traits` if it publicly inherits the base class `std::iterator`.

**Member types**

| member                         | generic definition                       | <code>T*</code> specialization          | <code>const T*</code> specialization    |
|--------------------------------|------------------------------------------|-----------------------------------------|-----------------------------------------|
| <code>difference_type</code>   | <code>iterator::difference_type</code>   | <code>ptrdiff_t</code>                  | <code>ptrdiff_t</code>                  |
| <code>value_type</code>        | <code>iterator::value_type</code>        | <code>T</code>                          | <code>T</code>                          |
| <code>pointer</code>           | <code>iterator::pointer</code>           | <code>T*</code>                         | <code>const T*</code>                   |
| <code>reference</code>         | <code>iterator::reference</code>         | <code>T&amp;</code>                     | <code>const T&amp;</code>               |
| <code>iterator_category</code> | <code>iterator::iterator_category</code> | <code>random_access_iterator_tag</code> | <code>random_access_iterator_tag</code> |

016

# Specialisation de template

## ■ Définir comportement particulier pour certains paramètres template

```
template <typename TYPE, int SIZE>
class vecN
{
public:
    TYPE& operator[](int index);
    TYPE const& operator[](int index) const;
private:
    TYPE data[SIZ];
};
```

Classe template générale

```
template <typename TYPE>
class vecN<TYPE, 2>
{
public:
    TYPE& operator[](int index);
    TYPE const& operator[](int index) const;

    TYPE const& x() const;
    TYPE const& y() const;
private:
    TYPE x_data, y_data;
};
```

Classe template spécialisée (partiellement)

Comportement différent pour un vecteur de taille 2

Accès direct à x ou y par des méthodes (spécifique pour un vecteur de dimension 2)

017

# Specialisation de template

```
template <typename TYPE>
class vecN<TYPE, 2>
{
public:
    TYPE& operator[](int index);
    TYPE const& operator[](int index) const;

    TYPE const& x() const;
    TYPE const& y() const;
private:
    TYPE x_data, y_data;
};
```

Spécialisation vecN de taille 2

```
template <typename TYPE>
class vecN<TYPE, 3>
{
public:
    TYPE& operator[](int index);
    TYPE const& operator[](int index) const;

    vecN<TYPE, 3> cross(vecN<TYPE, 3> const& vec) const;
private:
    TYPE x_data, y_data, z_data;
};
```

Spécialisation vecN de taille 3

Accès au produit vectoriel

018

# Implémentation des spécialisations

```
template <typename TYPE>
TYPE& vecN<TYPE, 2>::operator[](int index)
{
    switch(index)
    {
        case 0:
            return x_data;
        case 1:
            return y_data;
        default:
            throw std::out_of_range(__FUNCTION__);
    }
}

template <typename TYPE> TYPE const& vecN<TYPE, 2>::x() const {return x_data;}
template <typename TYPE> TYPE const& vecN<TYPE, 2>::y() const {return y_data;}

template <typename TYPE>
vecN<TYPE, 3> vecN<TYPE, 3>::cross(vecN<TYPE, 3> const& vec) const
{
    vecN<TYPE, 3> temp;
    temp[0]=y_data*vec[2]-z_data*vec[1];
    temp[1]=z_data*vec[0]-x_data*vec[2];
    temp[2]=x_data*vec[1]-y_data*vec[0];
    return temp;
}
```

Un seul paramètre template

019

# Utilisation des spécialisations

## Utilisation transparente

Une seule classe visible, comportement s'adaptant aux paramètres

```
int main()
{
    vecN<float,4> v0;
    v0[0]=1.4f; v0[1]=2.4f; v0[2]=7.6f; v0[3]=-2.1f;

    vecN<float,3> v1;
    v1[0]=1.4f; v1[1]=2.4f; v1[2]=7.6f;

    vecN<float,2> v2;
    v2[0]=2.8f; v2[1]=4.4f;

    vecN<float,3> v3=v1.cross(v1);
    std::cout<<v3<<std::endl;

    std::cout<<v2.x()<<" "<<v2.y()<<std::endl;
}
```

spécifique dimension 3

spécifique dimension 2

020

# Spécialisation totale

```
template <>
class vecN<int,2>
{
public:
    vecN(int x_arg,int y_arg);

    int& operator[](int index);
    int operator[](int index) const;

private:
    int x_data,y_data;
};

using pixel_coord=vecN<int,2>;
```

Spécialisation totale d'une classe template

Permet de définir un alias de type  
pixel\_coord est équivalent (autre nom) à vecN<int,2>

021

# Spécialisation totale

L'implémentation n'est plus un template!

Il faut définir le corps des fonctions dans les .cpp à nouveau

```
vecN<int,2>::vecN(int x_arg,int y_arg)
:x_data(x_arg),y_data(y_arg)
{}

int& vecN<int,2>::operator[](int index)
{
    switch(index)
    {
        case 0:
            return x_data;
        case 1:
            return y_data;
        default:
            throw std::out_of_range(__FUNCTION__);
    }
}
```

Dans le .cpp

```
pixel_coord p(4,6);
std::cout<<p<<std::endl;
```

Utilisation transparente

```
template <>
class vecN<int,2>
{
public:
    vecN(int x_arg,int y_arg);

    int& operator[](int index);
    int operator[](int index) const;

private:
    int x_data,y_data;
};

using pixel_coord=vecN<int,2>;
```

022

# Spécialisation des fonctions

- Seule les classes peuvent être partiellement spécialisées.  
Pas les fonctions.
- Une spécialisation de fonction doit être totale.
- Les fonctions fonctionnent plutôt par surcharge, éviter de mélanger spécialisation et surcharge.

023

# Efficacité: template VS polymorphisme

## Template

```
template <typename T>
void scalar_multiply_template(T& vecteur, float s);

template <typename T>
void scalar_multiply_template(T& vecteur, float s)
{
    vecteur.x*=s;
    vecteur.y*=s;
}

struct vec2
{
    vec2(float x_arg, float y_arg):x(x_arg),y(y_arg){}
    float x,y;
};
```

## Polymorphe

```
struct object_with_scalar_multiply
{
    virtual void scalar_multiply_polymorph(float s)=0;
};

struct vec2_polymorphic : public object_with_scalar_multiply
{
    vec2_polymorphic(float x_arg, float y_arg);
    void scalar_multiply_polymorphic(float s);
    int x,y;
};

vec2_polymorphic::vec2_polymorphic(float x_arg, float y_arg)
:x(x_arg),y(y_arg)
{}

void vec2_polymorphic::scalar_multiply_polymorphic(float s)
{
    x*=s;
    y*=s;
}
```

Pour 625 millions d'opérations:

- template: 3.3s (optimisé à 0s si calcul trivial)
- polymorphe: 6.5s (jamais optimisé)